Modélisation du comportement temporel du pipeline pour le calcul de WCET

Zhenyu Bai

THÈSE

En vue de l'obtention du
DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par l'Université Toulouse 3 - Paul Sabatier

Présentée et soutenue par

Zhenyu BAI

Le 12 mai 2023

Modélisation du Comportement Temporel du Pipeline pour le Calcul de WCET

Ecole doctorale : **EDMITT - Ecole Doctorale Mathématiques, Informatique et Télécommunications de Toulouse**

Spécialité : **Informatique et Télécommunications**

Unité de recherche :
IRIT : Institut de Recherche en Informatique de Toulouse

Thèse dirigée par
Christine ROCHANGE et Hugues CASSE

Jury

Mme Isabelle PUAUT, Rapporteure
M. Mathieu JAN, Rapporteur
M. Jean-Paul BODEVEIX, Examinateur
M. Pascal RAYMOND, Examinateur
Mme Christine ROCHANGE, Directrice de thèse
M. Hugues CASSé, Co-directeur de thèse

Modélisation du comportement temporel du pipeline pour le calcul de WCET

Zhenyu Bai

16 mai 2023

Résumé

Le calcul du pire temps d'exécution (WCET, *Worst Case Execution Time*) est une phase indispensable pour la vérification et la certification des systèmes embarqués strictement temps-réel. Le temps d'exécution des programmes, dont l'évaluation nécessitant une précision au niveau du cycle machine, est le produit du fonctionnement de la micro-architecture du processeur qui exécute le programme. Cependant, les processeurs modernes tendent à être de plus en plus complexes : ils sont équipés de *pipelines* et de *mécanismes d'accélération* comme les mémoires caches, la prédiction de branchement, etc.

Ces mécanismes introduisent des *variations temporelles* qu'il est nécessaire de prendre en compte lors du calcul du temps d'exécution. Par exemple, la latence d'accès à une mémoire cache est différente qu'elle contient la donnée (*Hit*) ou non (*Miss*).

En combinant le pipeline et les mécanismes d'accélération, le temps d'exécution devient une fonction complexe des instructions, de la structure du pipeline et des variations temporelles. Dans cette thèse, nous proposons une nouvelle structure de données nommée XDD (*eXecution Decision Diagram*) qui permet de représenter efficacement et précisément la relation entre le temps d'exécution et les variations temporelles. Comme son nom l'indique, cette représentation est inspirée des BDD (*Binary Decision Diagrams*).

À l'aide des XDD, nous proposons un modèle de calcul du temps dans le pipeline qui représente de manière compacte cette fonction temporelle complexe sous forme d'un ensemble de matrices associées aux blocs d'instructions machine composant le programme. Il est alors possible de calculer l'ensemble des états temporels tout au long des chemins d'exécution du programme de manière efficace et d'en déduire le WCET.

Nous avons expérimenté ce modèle sur des benchmarks classiques (TACLe) pour des processeurs avec exécution dans l'ordre du programme, ainsi que pour des processeurs contenant des ressources allouées dans le désordre (bus mémoire). Les résultats expérimentaux montrent une amélioration significative de la précision du WCET calculé par rapport à la méthode originale utilisée dans l'outil OTAWA, ainsi qu'une amélioration importante des performances de calcul.

Abstract

The computation of the Worst-Case Execution Time (WCET) of tasks is an essential step for the verification and certification of critical real-time embedded systems. The execution time at the precision of the processor cycle is determined by the micro-architecture of processor executing the programs. However, modern processors tend to be more and more complex : they are pipelined and equipped with several *acceleration mechanisms* such as caches, branch predictors, etc.

These mechanisms introduce *timing variations* that must be taken into account when computing the execution time. For example, the latency of a cache access may result in a *Hit* if the data is present in the cache, or in a *Miss* otherwise.

By combining the pipeline and the acceleration mechanisms, the execution time becomes a complex function of instructions, pipeline structure, and timing variations. In this thesis, we present a new data structure named XDD (*eXecution Decision Diagram*) which allows to efficiently and accurately represent the relationship between execution time and timing variations. As its name suggests, it is inspired by BDDs (*Binary Decision Diagrams*).

Using XDDs, we present a model for computing the execution time in the processor pipeline, which represents efficiently this complex function of execution time by a set of matrices associated with the blocks of machine instructions composing the program. It is then possible to compute efficiently the set of timing states along the execution paths of the program and to deduce the WCET therefrom.

We experimented this model on classical benchmarks (TACLe), for in-order processors, as well as for processors containing out-of-order resources (memory bus). The experimental results show a significant improvement in the accuracy of the computed WCET compared to the original method used in the OTAWA tool as well as a significant improvement in terms of analysis time.

Remerciements

Je tiens à remercier d'abord les trois personnes qui m'ont énormément aidé et m'ont accompagné avec gentillesse tout au long de ma thèse : Docteur Hugues Cassé, Professeur Christine Rochange et Docteur Thomas Carle. Ils ont toujours été là pour m'aider dans la recherche, dans la rédaction des articles, dans l'enseignement. Ils m'ont transmis, en se servant d'exemples, leurs valeurs, leur goût, dans l'enseignement et dans la recherche. Je me sens très heureux de commencer ma carrière académique, avec de la passion, grâce à eux. D'autre part, ils sont non seulement des encadrants et des collègues qui m'ont appris les compétences professionnelles, mais ils sont toujours disponibles pour discuter, pour m'écouter comme des amis, comme de la famille.

Tout d'abord, Hugues Cassé m'a encadré tout au long de ma thèse. Étant mon co-codirecteur, il a toujours été là pour m'apprendre, pour m'aider, pour m'encourager, pour me transmettre sans réserve ses expériences et ses savoir-faire. Il s'est toujours montré gentil et patient pour m'aider dans la rédaction des articles, spécialement cette thèse. Cela a été plaisant de discuter de la technique, passionnant de faire de la recherche avec lui et amusant d'encadrer des projets d'enseignement avec lui.

Christine Rochange, ma directrice de thèse, a toujours été disponible pour m'aider dans les affaires administratives, pour me donner des conseils d'enseignement, de recherche et pour ma carrière professionnelle.

Je dois absolument remercier également mon collègue, Thomas Carle, qui est toujours présent pour m'aider dans la rédaction des articles, pour m'écouter et pour m'encourager avec ses magnifiques blagues et plaisanteries.

Je voudrais aussi remercier les autres membres de l'équipe TRACES pour leur gentillesse. Ce n'était que du plaisir de travailler avec eux.

Je profite de cette opportunité pour également remercier les rapporteurs, Monsieur Mathieu Jan et Madame Isabelle Puaut, ainsi que tous les examinateurs du jury : Monsieur Jean-Paul Bodeveix et Monsieur Pascal Raymond, de prendre votre temps précieux pour lire ma thèse et pour d'assister à ma soutenance.

Table des matières

Liste des Abréviations

WCET	**W**orst **C**ase **E**xecution **T**ime
ISA	**I**nstruction **S**et **A**rchitecture
FCFS	**F**irst-**C**ome-**F**irst-**S**erved
CFG	**C**ontrol **F**low **G**raph
BB	**B**asic **B**lock
IPET	**I**mplicit **P**ath **E**numeration **T**echniques
ILP	**I**nteger **L**inear **P**rogramming
BDD	**B**inary **D**ecision **D**iagram
FIFO	**F**irst-**I**n-**F**irst-**O**ut
DAG	**D**irected **A**cyclic **G**raph
XDD	e**X**ecution **D**ecision **D**iagram
ROBDD	**R**educed and **O**rdered BDD
BFS	**B**readth-**F**irst **S**earch

Liste des Symboles

$e \in \mathcal{E}$	Évènement et l'ensemble des évènements
$a \in V_{\text{CFG}}$	Bloc de base et l'ensemble des blocs de base d'un CFG.
$a \to b \in E_{\text{CFG}}$	Arc entre deux blocs de bases de l'ensemble des arcs d'un CFG.
$\mathcal{I}$	Ensemble des instructions machine.
$\mathcal{I}^*$	Séquence d'instructions
$\mathcal{D}$	Domaine (en interprétation abstraite).
$X^\#$	Marque du symbole X dans la sémantique abstraite.
$\wp(S)$	Ensemble des parties de l'ensemble S.
ρ	Date de démarrage.
ρ^*	Date de fin.
α	Fonction d'abstraction (en interprétation abstraite)
γ	Fonction de concrétisation (en interprétation abstraite)
$\preceq$	Ordre partiel (pour un treillis ou un ordre partiel complet)
$\sqcup$	Fonction de jonction (pour un treillis ou un ordre partiel complet)
$\top$	Borne supérieure d'un treillis ou un ordre partiel
$\bot$	Borne inférieure d'un treillis ou un ordre partiel
$\mathsf{s} \in \mathsf{S}$	Ensemble d'étage d'un pipeline
$\mathbb{N}$	Ensemble de nombres naturels
$\wp(\mathcal{D})$	Ensemble des parties de l'ensemble $\mathcal{D}$
ρ	Date de démarrage (d'un nœud du graphe d'exécution)
ρ^*	Date de fin (d'un nœud du graphe d'exécution)
$\emptyset$	Ensemble vide
$\mathbb{Z}^\infty$	Ensemble des entiers étendu des infinies
$f^\# \in \mathcal{XDD}$	$\mathcal{XDD}$ dans le domaine des XDD.
$\text{NODE}(v, \overline{f^\#}, f^\#)$	Nœud XDD
$\text{LEAF}(k)$	Feuille XDD
$\mathbb{B}$	Ensemble des variables booléennes.
$\mathbb{B}^n$	Ensemble des vecteurs de n variables booléennes.
$\mathcal{XDD}$	Domaine des XDD
$f^\#[\gamma]$	Évaluation d'un XDD avec une configuration γ
$\mathbb{Z}^\infty$	$\mathbb{Z} \cup \{-\infty, +\infty\}$
$\bullet$	Opérateur binaire dans le domaine Z^∞
$\boxdot$	$\bullet$ transféré dans le domaine $\mathbb{D}$
$\odot$	$\bullet$ ou $\boxdot$ transféré dans le domaine $\mathcal{XDD}$
$\oplus$	Maximum transféré dans le domaine $\mathcal{XDD}$.
$\ominus$	Minimum transféré dans le domaine $\mathcal{XDD}$.
$\otimes$	Addition transférée dans le domaine $\mathcal{XDD}$.
$\oslash$	Soustraction transférée dans le domaine $\mathcal{XDD}$.
$\mathbb{0}_{\boxplus}$	Élément absorbant de $\oplus$ (pour le demi-anneau sur $\mathbb{D}$).
$\mathbb{1}_{\boxtimes}$	Élément neutre de $\otimes$ (pour le demi-anneau $\mathbb{D}$).

$\mathbb{0}_{\oplus}$	Élément absorbant de $\oplus$ (pour le demi-anneau sur $\mathcal{XDD}$).
$\mathbb{1}_{\otimes}$	Élément neutre de $\otimes$ (pour le demi-anneau sur $\mathcal{XDD}$).
$\mathbb{D} : \mathbb{B}^n \to \mathbb{Z}^\infty$	Représentation explicite de la relation entre les variables binaires et des entiers
$\mathcal{XDD} \underset{\beta}{\overset{\alpha}{\leftrightarrows}} \mathbb{D}$	Isomorphisme entre le domaine des XDD et la représentation explicite
$\mathcal{M}$	Matrice de XDD
$[I, \mathsf{s}] \in V_{\mathrm{XG}}$	Nœud du graphe d'exécution
$[I_0, \mathsf{s}_0] \to [I_1, \mathsf{s}_1] \in E_{\mathrm{XG}}$	Arc du graphe d'exécution
$\mathcal{I}^{\lvert N_I \rvert}$	Séquence de N_I instructions
λ_v	Latence d'un nœud du graphe d'exécution
$\delta(v \to w)$	Type d'arc du graphe d'exécution (solide ou pointillé)
$\lvert \mathsf{s}_k \rvert$	Capacité de l'étage s_k
$\mathcal{R}eg$	Ensemble des registres du processeur
$q \in \mathcal{Q}$	Ensemble des files du processeur
s_q^p	Étage producteur de la file q
s_q^c	Étage consommateur de la file q
$\gamma \in \Gamma$	Configuration et l'ensemble de configurations (d'évènements)
$X_e \geq x_e$	Nombre d'occurrences maximal (i.e. la borne) d'un évènement
Γ^*	Partie *nullifiée*
γ^*	Configuration *nullifiée*
$\mathcal{E}_\Gamma^{Entire}$	Ensemble d'évènements bornant entièrement une partie
$\mathcal{E}_\Gamma^{Partial}$	Ensemble d'évènements bornant partiellement une partie
$\overrightarrow{S} \in \mathcal{S}$	État temporel (vecteur de XDD) de l'ensemble d'états temporels
$\mathcal{R}$	Ensemble des ressources temporelles
$r \in \mathcal{R}, i_r$	Index de la ressource r dans l'état temporel
$\mathcal{D}$	Ensemble des dépendances temporelles
$\tau_{reset/wait/move/consume}$	Quatre transitions basiques de l'état temporel
ϱ	Pointeur de temps
$\hat{\rho}$	Ordonnancement définitif des *points de contention*.
ρ_{sched}	Ordonnancement dans une itération.
ρ_{rel}	Dates de libération du bus
$\blacktriangleleft_{ME}, \blacktriangleleft_{FE}$	Opérateurs de comparaison (pour déterminer la disponibilité des *points de contention*)

Chapitre 1

Introduction

Les systèmes embarqués, dits *Temps Réel*, interagissent avec l'environnement et doivent fournir une réponse aux stimuli extérieurs en un temps inférieur à un certain seuil. Une réaction trop longue est considérée comme un échec pouvant causer la perte du système ou comme un facteur dégradant son fonctionnement. Ce type de système est donc soumis à des contraintes sur le temps de réponse en plus de contraintes de correction du résultat. Dans certains cas, l'échec temporel du système peut engendrer des catastrophes (risques pour la sécurité des personnes, impact économique important, etc). De tels systèmes, dont le comportement temporel doit être vérifié rigoureusement, sont appelés des systèmes temps-réel critiques et constituent l'objet principal de cette thèse. Les exemples ci-dessous sont typiques de ce genre de système :

1. Le système qui contrôle les airbags d'une voiture doit déclencher les airbags dans un temps limité après avoir détecté une collision à l'aide de capteurs.
2. Le système de commandes de vol électrique (*fly-by-wire system*) des avions doit correctement contrôler les organes de pilotage dans un temps limité après avoir reçu des commandes des pilotes ou des capteurs pour garantir la stabilité de l'avion.

Dans les applications industrielles, ce genre de système est soumis à des processus de certification qui reposent sur une vérification du comportement fonctionnel – est-ce que les programmes calculent bien ce qu'ils doivent calculer ? Et du comportement temporel – est-ce que les programmes calculent le résultat dans un délai acceptable ?

En ce qui concerne le comportement temporel, si nous considérons que le système est composé de plusieurs tâches, la vérification porte sur le pire temps de réponse du système qui résulte de l'ordonnancement des tâches BRIL et al., 2009 ; GRAILLAT et al., 2019. Cet ordonnancement est généralement calculé à partir de leur pire temps d'exécution (WCET, Worst-Case Execution Time WILHELM et al., 2008).

Le calcul du WCET peut être fait selon deux approches majeures : les analyses statiques et les analyses dynamiques. Le principe des analyses dynamiques est d'exécuter le programme dans des situations variées (i.e. avec des entrées différentes) et de mesurer le temps d'exécution. Comme il est généralement impossible d'exécuter le programme avec toutes les entrées possibles, cette approche est souvent combinée avec la théorie des probabilités et fournit un WCET probabiliste (BERNAT et al., 2002 ; DAVID et al., 2004). Les analyses probabilistes peuvent être combinées avec des analyses statiques (BETTS et al., 2010) et éventuellement d'autres techniques comme de l'apprentissage automatique (AMALOU et al., 2021) : on parle alors d'analyses *hybrides*. Mais les approches fondées sur des techniques d'analyse dynamiques ne peuvent garantir une sûreté absolue du WCET que si la couverture des tests est suffisante (ABELLA et al., 2014).

L'analyse statique des programmes propose plusieurs approches. L'analyse de programme par exécution symbolique (BIERE et al., 2013 ; BENHAMAMOUCH et al., 2008 ; KEBBAL et al., 2006 ; BENHAMAMOUCH et al., 2009) a pour principe d'exécuter le programme "symboliquement" en suivants les différents chemins d'exécution qui composent le programme. La différence entre l'approche par l'exécution symbolique et les approches dynamiques est que le premier n'exécute (simule) pas le programme concrètement, mais avec un état symbolique et synthétique du programme. Les approches par exécution symbolique doivent faire face au problème d'explosion combinatoire du nombre de chemins : en pratique, il est généralement impossible de considérer tous les chemins d'un programme. C'est la raison pour laquelle cette technique est spécialement utilisée dans le domaine du **test** des programmes où le but est de trouver des cas (des entrées) pour lesquels certaines contraintes ne sont pas satisfaites par le programme plutôt que de prouver que le programme est valide pour toute entrée possible. Dans le cas du problème du WCET, quand la sûreté est exigée et quand ce dernier doit être valide pour toute entrée possible, les approches par exécution symbolique souffrent donc particulièrement du problème de performance.

Contrairement aux approches présentées ci-dessus, nous nous intéressons aux approches purement statiques. Elles ont pour principe d'examiner le programme sans vraiment l'exécuter, mais calculent une surestimation des propriétés que nous voulons vérifier. Pour le problème du WCET, la propriété visée est le temps d'exécution. Dans des situations extrêmement critiques, il faut pouvoir garantir un WCET avec une sûreté absolue. Donc le calcul du WCET par analyse statique vise à déterminer une borne garantie supérieure au pire temps réel d'exécution du programme, quelle que soit l'entrée du programme. On accepte que les approches purement statiques introduisent de la surestimation, ce qui donne la possibilité d'atténuer le problème de complexité combinatoire rencontré lors de l'analyse des programmes.

1.1 Pourquoi le calcul du WCET est-il difficile ?

Le temps d'exécution d'un programme en nombre de cycles du processeur dépend fortement de sa micro-architecture. Cependant, pour la plupart des processeurs d'aujourd'hui, cette information temporelle n'est accessible ni dans l'ISA (*Instruction Set Architecture*), ni dans les langages de programmation de plus haut niveau. En effet, le temps d'exécution précis n'est pas utile pour la plupart des applications.

Ainsi beaucoup d'analyses statiques se basent sur le code source écrit dans un langage haut niveau (comme C ou Java) ou sur un langage intermédiaire du compilateur, tel que l'IR (*Intermediate Representation*) du compilateur LLVM (LATTNER et al., 2004) qui est une abstraction de l'ISA. Ces langages ne sont pas donc suffisants pour spécifier le comportement temporel du programme et du processeur et l'analyse temporelle est par conséquent incapable de calculer un WCET précis sur ces représentations. Ainsi, les outils existants pour le calcul du WCET par analyse statique utilisent plutôt une représentation intermédiaire ad-hoc permettant de fournir l'information temporelle liée au processeur. Dans la plupart des cas, ils travaillent directement sur des programmes binaires, et dérivent les informations temporelles à partir d'une description additionnelle de la micro-architecture, comme le font par exemple *chronos* (LI et al., 2007), *Bound-T* (HOLSTI et al., 2000) ou *aiT* (FERDINAND et al., 2004). Cela est également vrai pour l'outil que nous utilisons, OTAWA (BALLABRIGA

et al., 2010). Il existe aussi des outils qui se basent sur des représentations intermédiaires du compilateur, comme *llvmta* (HAHN et al., 2022), mais en se basant sur une représentation bas niveau (plus bas que l'IR) dans le compilateur LLVM, et en rajoutant des spécifications complémentaires afin de considérer plus précisément le comportement temporel.

Pour calculer un WCET précis, la micro-architecture du processeur doit être modélisée. Or, la micro-architecture des processeurs modernes est complexe : ils sont souvent pipelinés, équipés de mémoires caches ; ils mettent en œuvre la prédiction de branchement, l'exécution dans le désordre et d'autres mécanismes d'accélération. Ces mécanismes introduisent des *variations temporelles*, c'est-à-dire que le comportement temporel de ces mécanismes est variable, tout en dépendant de l'état de la micro-architecture du processeur **durant l'exécution**. Par conséquent, l'analyse temporelle doit considérer l'état du processeur plus précisément pour déterminer des temps d'exécution précis – ce qui augmente la complexité. Cette caractéristique permet de classifier les processeurs par leur *prédictibilité* – le niveau de difficulté qu'une micro-architecture impose sur ses analyses. Typiquement, les processeurs utilisés dans les ordinateurs personnels (cœurs Intel, AMD) ont des micro-architectures beaucoup trop complexes et donc moins *prédictibles* que celles des micro-processeurs ou des micro-contrôleurs utilisés dans les systèmes embarqués. Cependant, avec les technologies de conception et de fabrication de processeur actuelles, et avec la mode de l'*edge computing*(VARGHESE et al., 2016), les processeurs embarqués ont des structures de plus en plus complexes pour supporter une forte demande de calcul provenant du transfert de responsabilité des serveurs centraux vers les dispositifs embarqués. Les techniques de modélisation doivent aussi évoluer pour supporter ces situations.

La *prédictibilité* des architectures a une importance majeure pour le calcul de WCET mais ne constitue pas le sujet principal de cette thèse. SCHOEBERL et al., 2015 ; SAINRAT et al., 2014 ; PELLIZZONI et al., 2011 ; BAK et al., 2012 ; SCHUH et al., 2020 abordent ce domaine en examinant différents aspects : pipeline, multi-cœur, ordonnancement, etc. Dans ce document, nous nous intéressons plutôt au pipeline des processeurs mono-cœur et aux mécanismes d'accélération dont le comportement temporel peut être représenté par des *évènements*.

1.1.1 Les évènements

Des *variations dans le temps d'exécution* sont introduites par les mécanismes qui ont un comportement dynamique lors de l'exécution. Un exemple typique d'un tel mécanisme est la mémoire cache. Pour un processeur équipé d'une mémoire cache d'instruction par exemple, le temps de chargement d'une instruction est variable : si l'instruction est déjà présente dans le cache, la latence du chargement est courte ; sinon, l'instruction est chargée depuis la mémoire principale ou depuis un cache de niveau plus haut ce qui prend plus de temps.

Un constat similaire peut-être fait au sujet de la prédiction de branchement : si le branchement est bien prédit, l'exécution continue normalement ; si le branchement est mal prédit, le pipeline est vidé pour charger les instructions du chemin correct – ce qui prend plus de temps et conduit à un état de pipeline totalement différent.

Pour le cache et la prédiction de branchement, les variations sont binaires : il existe deux situations possibles (*Miss / Hit*, bien prédit/mal prédit). Nous appelons ces variations binaires des *évènements* (on note $e \in \mathcal{E}$ un évènement e dans l'ensemble

$\mathcal{E}$ des évènements du programme). Le cas où une latence supplémentaire est introduite (*Miss* ou mal prédit) est qualifié d'*activation* de l'évènement, et le cas contraire est qualifié d'*inactivation* de l'évènement.

Dans cette thèse, nous nous intéressons principalement aux *variations temporelles* binaires – les *évènements*. Néanmoins, les processeurs réels peuvent avoir des composants avec des latences plus complexes. Par exemple, la division implémentée matériellement présente souvent plusieurs latences possibles en fonction des valeurs des opérandes. Dans cas, il est facile de composer des variables binaires pour sélectionner différents comportements temporels, même si nos expérimentations n'ont pas porté sur ce genre de situation.

1.1.2 La source des évènements

Un processeur qui n'a pas de mécanisme aléatoire est déterministe, c'est-à-dire qu'étant donné une entrée du programme et l'état initial du cache, il est tout à fait possible de calculer le comportement **exact** du cache tout au long de l'exécution. Néanmoins, l'entrée du programme est souvent inconnue en réalité. À cause du nombre très important des chemins d'exécution, l'analyse statique est généralement incapable de réaliser le calcul, en un temps raisonnable, du comportement du cache pour toute exécution possible.

Des *abstractions* sont alors nécessaires pour diminuer la complexité. Nous nous étendrons plus en détail sur la notion d'abstraction dans la section 2.1.2 mais ici, on peut qualifier simplement l'approche comme étant "des techniques d'analyse statique qui font de l'approximation lors du calcul contre un gain en performance". Pour les analyses de cache, l'abstraction est généralement faite d'abord sur le chemin d'exécution : le comportement est déterminé typiquement par les analyses *may* et *must* qui classifient les accès mémoire en toujours *Miss* (*always miss*) ou toujours *Hit* (*always hit*) (THEILING et al., 2000) ; les accès qui sont ni *always miss*, ni *always hit* sont qualifiés (*not classified*– *Not Classified*). Ici, un accès mémoire signifie "une instruction qui accède à la mémoire via le cache **en prenant en compte tous les chemins d'exécution passant par cette instruction**". En effet, *always hit*, respectivement *always miss*, signifie "cet accès mémoire fait toujours un *Hit*, respectivement un *Miss*, **quel que soit le chemin d'exécution**.

Pour les accès classifiés en *always miss* ou *always hit*, même si le chemin d'exécution n'est pas explicite, le comportement est valide pour tout chemin, donc il n'y a pas de surestimation introduite. Cependant, les accès *not classified* peuvent faire soit un *Miss* (sur certains chemins), soit un *Hit* (sur les autres chemins), mais la relation entre le comportement et le chemin est perdue – une abstraction (ou une surestimation) est faite. De plus, il existe un écart entre les catégories existantes et la capacité des analyses de cache qui classifient précisément les accès avec ces catégories : en fonction de la précision de l'analyse *may* (qui détermine les *always miss*) et de l'analyse *must* (qui détermine les *always hit*), il est possible que certains accès soient effectivement *always miss* ou *always hit*, mais sont classifiés *not classified*. Même si dans certains cas, les accès *not classified* sont le résultat de l'approximation de l'algorithme d'analyse, il existe des cas où les *not classified* sont définitifs quelque soit l'algorithme d'analyse (TOUZEAU et al., 2017).

La conséquence des abstractions faites par les analyses statiques des mécanismes d'accélération est que le comportement de ces mécanismes devient variable du point de vue de l'analyse temporelle – donc représenté par des *évènements*.

1.1.3 Les évènements dans le pipeline

Pour avoir une meilleure performance en nombre d'instructions exécutées par cycle, les processeurs modernes utilisent des pipelines. Cela signifie que les instructions ne sont pas exécutées indépendamment les unes après les autres, mais **parallèlement** en partageant certaines ressources du pipeline. Or, le nombre d'instructions exécutées en parallèle est dynamique (autrement dit, la bande passante du pipeline est variable) en fonction de l'opération réalisée par chaque instruction, des dépendances entre les instructions et de la capacité des ressources du pipeline. La conséquence de ce fonctionnement est que le temps d'exécution d'une séquence d'instructions n'est pas la somme linéaire du temps d'exécution de chaque instruction, mais une fonction complexe qui dépend des instructions exécutées et de l'état de la micro-architecture. *L'analyse de pipeline* est la phase de l'analyse du WCET qui crée un *modèle de pipeline* pour calculer le temps d'exécution des instructions tout en considérant les effets du pipeline et de l'état de la micro-architecture.

En présence des évènements, le temps d'exécution dépend en plus de leur activation. Intuitivement, l'analyse de pipeline a deux choix possibles lors de la rencontre d'un évènement : (a) comme seul le WCET du programme est intéressant, nous pouvons considérer uniquement le pire cas pour cet évènement, ou (b) nous considérons les deux possibilités.

L'option (a) présente deux inconvénients majeurs. Tout d'abord, elle engendre un problème de précision : parmi tous les chemins d'exécution passant par une séquence d'instructions, le pire cas ne se produit généralement pas à chaque coup. Donc le WCET calculé en considérant toujours le pire cas est surestimé. Un exemple d'une telle situation est le comportement typique du cache dans les boucles, appelé la *persistance de cache*. Au sein d'une boucle, un accès à la mémoire produit souvent un *Miss* à la première itération, pour charger les instructions/données dans le cache, et puis uniquement des *Hit* dans les itérations restantes car les instructions/données ont été déjà chargées à la première itération. Dans ce scénario, l'évènement de cache existe effectivement (*Hit* et *Miss* sont tous les deux possibles), mais si nous considérons un *Miss* à chaque itération, la surestimation devient énorme car le pire cas qui n'arrive qu'une seule fois est compté pour chaque itération. L'autre inconvénient de l'option (a) concerne les *anomalies temporelles* – pour certaines micro-architectures, le pire cas local n'est pas forcément le pire cas global. Autrement dit, considérer l'activation de chaque évènement ne produit pas forcément un WCET valide. Nous allons donner des explications plus détaillées concernant les *anomalies temporelles* dans la section suivante.

L'option (b) pose simplement un problème de performance. Considérer les deux possibilités lors d'un évènement cause une divergence d'état du pipeline. Par conséquent, l'analyse de pipeline qui considère toutes les combinaisons d'activation ou d'inactivation des évènements a une complexité combinatoire en fonction de leur nombre. Malgré ce problème, cette option fournit une meilleure précision et ne souffre pas du problème de validité induit par les *anomalies temporelles*, deux points qui sont précisément les défauts de l'option (a). Par conséquent, dans les approches proposées dans la suite de ce document, nous visons l'option (b) mais nous allons créer de nouvelles techniques pour atténuer le problème de performance de l'analyse de pipeline et, ainsi, rendre l'approche utilisable dans des situations réalistes.

1.1.4 Les ressources allouées dans le désordre et les anomalies temporelles

La notion d'*anomalie temporelle* a d'abord été introduite dans le domaine du calcul parallèle (GRAHAM, 1969). Dans ce contexte, des tâches sont exécutées en parallèle sur plusieurs processeurs avec des dépendances potentielles entre elles. Les processeurs exécutent les tâches en suivant une politique du FCFS (*First-Come-First-Served*) en respectant les dépendances entre les tâches. Dans cette configuration, des cas "anormaux" (contre-intuitifs) se présentent où (a) l'augmentation du nombre de processeurs augmente le temps d'exécution total, ou (b) la diminution du temps d'exécution de certaines tâches augmente le temps d'exécution total.

Ce phénomène est causé par le nombre limité de ressources partagées (dans ce cas, les processeurs), et par les dépendances qui empêchent la libre exécution des tâches. Les deux contraintes combinées font que le vrai chemin critique (la chaîne de dépendances la plus longue) ne peut pas être ordonnancé idéalement sur les ressources disponibles.

Ce modèle d'exécution de tâches concurrentes sur des ressources partagées est très similaire à l'exécution des instructions dans le pipeline, où les instructions peuvent être considérées comme des tâches et les ressources sont celles du pipeline partagées par les instructions exécutées en parallèle. Par conséquent, l'ordonnancement des instructions dans le pipeline peut également souffrir d'*anomalies temporelles*. La figure 1.1 montre un exemple de l'occurrence d'une *anomalie temporelle*. Dans cet exemple, deux instructions (1 et 2) sont exécutées sur les ressources 1 et 2. La ressource 2 du pipeline est allouée avec la politique FCFS. L'instruction 2 entre dans le pipeline plus tard que l'instruction 1, à l'instant indiqué par la ligne pointillée. L'instruction 1 utilise d'abord la ressource 1 puis la ressource 2, inversement pour l'instruction 2. Les dépendances entre l'exécution des instructions sont représentées par les flèches solides. Dans le premier scénario (celui du haut), la ressource 2 est d'abord allouée à l'instruction 2 car elle est la première instruction prête (l'instruction 1 doit d'abord finir d'utiliser la ressource 1). Dans le deuxième scénario, l'instruction 1 prend moins de temps dans la ressource 1 et donc la ressource 2 est d'abord allouée à l'instruction 1. Finalement, on peut constater que la réduction du temps d'exécution de l'instruction 1 sur la ressource 1 cause contre-intuitivement l'augmentation du temps d'exécution total. Une explication simple est que, dans le deuxième scénario, l'allocation de la ressource 2 à l'instruction 1 empêche le démarrage de l'instruction 2 et donc limite le parallélisme entre les deux instructions.

Plusieurs définitions de *anomalies temporelles* existent en fonction du modèle de pipeline (EISINGER et al., 2006 ; CASSEZ et al., 2012 ; REINEKE et al., 2006 ; BINDER et al., 2022 ; KIRNER et al., 2009 ; BINDER et al., 2021) mais le problème causé par les *anomalies temporelles* reste le même : lors d'évènements, le temps d'exécution d'une instruction (dans une ressource, typiquement un étage) est variable. Mais si on considère l'activation des évènements comme le pire cas, le WCET final calculé risque d'être invalide, comme dans l'exemple de la figure 1.1 : une durée plus courte de l'exécution de certaines instructions peut produire un temps d'exécution total plus long. L'analyse de WCET peut toujours essayer de couvrir la pire latence induite par un évènement (qui est plus longue que la latence d'activation en considérant les *anomalies temporelles*), mais une telle latence peut introduire une surestimation considérable du WCET final. De plus, l'article GEBHARD, 2010 a montré que l'effet des *anomalies temporelles* ne peut pas être couvert par une simple constante $k \in \mathbb{N}$ qui serait ajoutée au WCET calculé sans compter les *anomalies temporelles*, mais par une fonction plus complexe dépendant du programme.

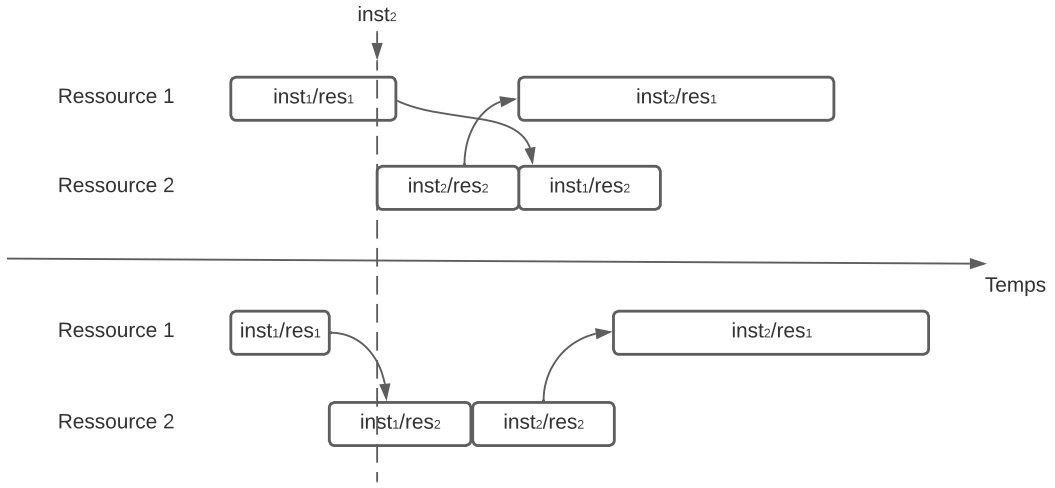


FIGURE 1.1 – Un exemple de l'*anomalie temporelle*.

Pour éviter le problème de validité causé par les *anomalies temporelles*, il est possible d'utiliser des processeurs garantissant leur absence (HAHN et al., 2020) : l'article WENZEL et al., 2005 montre que la présence des ressources allouées dans le désordre dans le pipeline est une condition nécessaire pour la présence d'anomalies temporelles. Donc, on peut montrer qu'un pipeline sans ce type de ressources est insensible aux *anomalies temporelles*. Au contraire, dans le cas où des *anomalies temporelles* peuvent se présenter, considérer explicitement la divergence d'état de pipeline (à cause des évènements) nous assure un WCET valide et peut améliorer la précision du WCET calculé.

1.2 Objectif de la thèse

Cette thèse se focalise sur l'*analyse de pipeline*, la phase où le temps d'exécution des instructions est déterminé en considérant les détails du pipeline et de la micro-architecture.

Comme discuté précédemment, pour supporter les *anomalies temporelles* et préserver la précision de l'analyse de WCET, nous choisissons de traiter explicitement la divergence d'état provoquée par des évènements. Le problème principal de ce choix est la performance puisque le nombre d'états du pipeline est combinatoire en fonction du nombre d'évènements. Par conséquent, nous avons exploré des modèles de pipeline efficaces et des techniques spécifiques pour atténuer l'explosion combinatoire de cette analyse. Ces modèles doivent supporter des processeurs à exécution dans l'ordre mais également des ressources allouées dans le désordre (ce qui rend le processeur sensible aux *anomalies temporelles*).

1.3 Plan

Après ce chapitre d'introduction, nous présentons, dans le chapitre 2, les notions et les théories de base qui seront utilisées dans cette thèse. Ensuite, nous discutons de l'état de l'art sur le calcul de WCET par analyse statique et spécialement des techniques utilisées dans l'analyse de pipeline.

Dans le chapitre 3 est introduite une nouvelle structure de données, appelée XDD, qui permet de représenter efficacement la relation entre les temps d'exécution et les évènements. L'utilisation des XDD dans l'analyse de pipeline permet de réaliser une analyse **exacte** du comportement temporel du pipeline tout en gardant une performance acceptable.

Dans le chapitre 4, nous présentons notre nouvelle analyse de pipeline qui intègre l'utilisation des XDD. Cette analyse est capable de calculer **tous** les temps d'exécution possibles et exacts d'une séquence d'instructions quelconque en fonction des évènements. Cette analyse est seulement compatible avec les processeurs à exécution dans l'ordre.

La nouvelle analyse de pipeline calcule **tous** les temps d'exécution possibles. Cependant, dans des approches plus anciennes, seul le pire temps d'exécution des séquences d'instructions était pris en compte. Pour profiter du gain de précision de notre nouvelle analyse, une amélioration est proposée pour augmenter la précision du WCET produit. Notre analyse calcule séparément les temps d'exécution des séquences d'instructions du programme mais, lorsqu'on considère le programme en entier, le temps d'exécution de chaque morceau de code dépend aussi de son *contexte d'exécution*, c'est-à-dire de l'historique d'exécution du programme avant ce morceau code. Pour mieux profiter du *contexte d'exécution*, notre analyse est étendue dans le chapitre 6 pour s'appliquer globalement au programme entier et non à chaque séquence d'instructions séparément.

La structure de données XDD apporte des propriétés algébriques intéressantes : nous montrons, dans le chapitre 3, que les XDD forment un demi-anneau tropical (aussi appelé *algèbre max-plus*) (COHEN et al., 1999 ; HEIDERGOTT et al., 2006). Dans la version étendue de notre analyse de pipeline présentée dans le chapitre 6, les états du pipeline sont alors représentés sous forme de *vecteurs* de XDD et les transitions se révèlent être finalement des fonctions affines sur le demi-anneau des XDD. Ces fonctions affines sont analogues aux applications linéaires sur des espaces vectoriels *de nombres réels*, c'est-à-dire que les transitions d'état de pipeline peuvent être représentées par une multiplication de matrices de XDD. Comme le demi-anneau des XDD garantit l'associativité de la multiplication matricielle, il devient possible de résumer les transitions induites par d'une séquence d'instructions par *une matrice unique résultat de la multiplication des matrices de chaque étape d'exécution* ce qui, finalement, permet d'accélérer l'analyse.

Dans le chapitre 7, nous étendons notre modèle pour supporter des ressources allouées dans le désordre. Nous nous concentrons sur un exemple typique pour des petits microprocesseurs utilisés dans l'embarqué : le bus de mémoire FCFS partagé par le cache d'instructions et le cache de données. Nous présentons une nouvelle méthode pour calculer la latence supplémentaire causée par les contentions lors d'accès à ce bus. Nous montrons que, à l'aide des XDD, l'analyse est capable de calculer les temps d'exécution **exacts** avec un temps d'analyse raisonnable. Finalement, nous concluons cette thèse et proposons des perspectives de recherche dans le chapitre 8.

Chapitre 2

État de l'art

Le calcul de WCET reste un domaine très actif dans la communauté scientifique de la vérification, des systèmes embarqués et de l'analyse statique. Pour les processeurs modernes, les évènements [1], la divergence d'état qu'ils induisent et la complexité combinatoire qui en résulte est une problématique importante de l'analyse du WCET, spécialement dans l'analyse de pipeline. La compréhension de ces analyses nécessite d'abord d'exposer, dans ce chapitre, plusieurs concepts et théories plus largement utilisés dans le domaine de l'analyse statique. Dans une seconde partie, nous discutons des approches statiques existantes traitant du problème de calcul de WCET, spécialement en examinant le modèle de pipeline et comment les *évènements* y sont traités.

2.1 Concepts pour l'analyse statique

2.1.1 Le Graphe de Flot de Contrôle

Dans le calcul de WCET par analyse statique, la représentation quasi-standard d'un programme est le *Graphe de Flot de Contrôle* (CFG, *Control-Flow Graph*). Un CFG est un graphe $CFG : V_{\text{CFG}} \times E_{\text{CFG}}$, dont l'ensemble des sommets (V_{CFG}) représente des blocs de base (BB, *Basic Block*). Un BB dans un CFG $a \in V_{\text{CFG}}$ est une séquence d'instructions machine ($a \in \mathcal{I}^*$) exécutées séquentiellement de manière unitaire : seule la dernière instruction peut être un branchement et seule la première instruction peut être la cible d'un branchement. L'ensemble des arcs $E_{\text{CFG}} : V_{\text{CFG}} \times V_{\text{CFG}}$ du CFG représentent l'ordre d'exécution entre les BB. Un arc entre deux BB $a \to b \in E_{\text{CFG}}$ indique que b peut s'exécuter après a.

2.1.2 Abstraction

Le terme abstraction, dans le domaine de l'analyse statique, désigne une représentation partielle ou approximative d'une certaine sémantique du programme. Elle est nécessaire car il est généralement impossible de simuler toutes les exécutions concrètes : il existe trop de chemins d'exécution possibles. L'analyse d'une abstraction permet d'extraire une propriété partielle du programme vis-à-vis du comportement qui nous intéresse. Elle omet certaines informations afin de diminuer la complexité de l'analyse. Une bonne abstraction est une abstraction qui cache le plus d'informations possible tout en supportant une analyse fournissant des résultats suffisamment précis pour décrire et exploiter le comportement qui nous intéresse.

1. Même si cette terminologie n'est pas communément adoptée, on retrouve des concepts similaires.

Exemple. On veut vérifier que la valeur des variables d'un programme n'est jamais négative. Une méthode sans abstraction consiste à simuler le programme pour toute entrée possible et à vérifier si les variables ont été négatives le long de chaque exécution. Intuitivement, cette analyse a une complexité élevée, d'une part si le nombre d'entrées possibles peut être très grand et d'autre part si le programme est lui-même complexe : le temps d'analyse peut alors devenir rédhibitoire. Une abstraction de la propriété prise en exemple consiste à se concentrer uniquement sur le signe des variables en représentant leur valeur par l'abstraction $\{+, -, 0, ?\}$, positive, négative, nulle ou inconnue. De plus, on ne s'intéresse pas au signe des variables pour chaque chemin d'exécution individuel, mais au(x) signe(s) possible(s) pour tous les chemins d'exécution. Intuitivement, une simulation qui ne représente les valeurs numériques que par $\{+, -, 0, ?\}$ doit être beaucoup plus simple à produire qu'une simulation complète car elle ne calcule qu'une sous-propriété (signe) par rapport à la sémantique concrète (valeur exacte).

Cependant, cette abstraction a un prix : le résultat "?" – parfois, on ne sait pas déterminer le signe d'une variable (de manière analogue à la catégorie *not classified* pour l'analyse du cache). Ce "?" est engendré par deux abstractions : (a) l'abstraction du chemin d'exécution – lorsque la variable est positive sur un chemin, mais négative sur un autre, et (b) l'abstraction de la valeur concrète : les variables sont représentées par leur signe, et la somme de deux variables de signes différents ou la soustraction de variables de même signe a un signe imprévisible. Par exemple, pour la séquence $x = 4; y = 3; x = x - y;$, x reste positif. Mais si on ne connaît que les signes, le calcul devient : $x = +; y = +$, et le signe de $x - y$ est inconnu.

L'abstraction (b) peut être réparée si l'**ensemble** des valeurs **exactes** des variables est connu – le signe peut être correctement calculé. Cependant, quand on utilise le terme **d'ensemble des valeurs**, une abstraction sur les chemins d'exécution est sous-entendue. Les vraies valeurs concrètes des variables sont en fait la valeur exacte par chemin d'exécution depuis le début du programme. Cependant, calculer la valeur exacte pour tout chemin est très proche d'une simulation complète du programme pour toute entrée possible et de complexité combinatoire. L'abstraction de la valeur exacte par le signe introduit de la perte de précision mais est indispensable pour réduire la complexité de l'analyse.

2.2 Interprétation Abstraite

L'abstraction est un concept central de l'analyse statique par interprétation abstraite. Généralement, on veut que l'abstraction soit valide. La définition de la validité (*soundness* en anglais) d'une analyse est dérivée de sa définition en logique [2] où "un système logique est valide si et seulement si toutes les formules qui peuvent être prouvées sont valides par rapport à la sémantique du système". Autrement dit, tout ce qui peut être prouvé par les formules (la syntaxe) est forcément vrai selon la sémantique, mais ce qui ne peut pas être prouvé par les formules n'est pas forcément faux selon la sémantique. Dans le contexte de l'analyse de programme, la validité est souvent exprimée par "tout ce que l'analyse affirme est forcément vrai, mais l'analyse a le droit de dire qu'elle ne sait pas.".

2. En anglais, le terme *soundness* est utilisé à la fois dans le domaine de logique et dans le domaine de l'analyse de programme. Mais en français, le terme correspondant est *cohérence* ou *validité* dans le domaine d'analyse de programme et *correction* dans le domaine de la logique. Pour ne pas confondre ces notions, dans cette thèse, le terme *validité* désigne la *soundness* et le terme *correction* correspond à la notion de *correctness* en anglais.

L'interprétation abstraite (COUSOT et al., 1977 ; COUSOT et al., 1992 ; COUSOT, 1996) est une base théorique qui fournit un cadre pour la conception d'abstractions. En suivant ce cadre, les analyses conçues sont garanties valides.

L'exemple donné dans la section 2.1.2 est un cas typique d'interprétation abstraite qui s'intéresse uniquement au signe des variables – une abstraction des valeurs exactes. De la même manière, nous pouvons concevoir une analyse du comportement d'une mémoire cache, du comportement d'un prédicteur de branchements et nombreuses autres analyses qui fournissent des informations utiles à l'analyse de WCET.

L'interprétation abstraite permet de prouver la validité d'une abstraction mais elle nécessite d'abord de trouver artificiellement une telle abstraction. La performance de l'abstraction (typiquement le temps d'analyse) n'est pas garantie par le cadre de l'interprétation abstraite mais dépend du choix artificiel de l'abstraction. Ceci étant dit, trouver une bonne abstraction par rapport à une sémantique intéressante est loin d'être une tâche aisée.

Le but de cette thèse n'est pas d'aborder en détails l'interprétation abstraite. Nous allons seulement présenter son principe et les notions importantes, ce qui va permettre de simplifier la lecture des chapitres suivants.

2.2.1 La sémantique

Informellement, la sémantique d'un programme est une propriété du programme. Dans l'interprétation abstraite, on s'intéresse à l'abstraction des sémantiques. Si une sémantique est une représentation abstraite (ou approximative) d'une autre sémantique, cette dernière est appelée "sémantique concrète" et la première est appelée "sémantique abstraite".

Une sémantique opérationnelle, concrète ou abstraite, est définie par un domaine d'états et une fonction de transition, notée *Update*. Le domaine définit la représentation des états et la fonction de transition définit comment ces états évoluent tout au long de l'exécution.

Dans l'exemple de la sémantique du signe des variables proposée précédemment, le domaine est $\{+, -, 0, ?\}$ et la fonction de transition est définie par rapport à la sémantique du langage de programmation, typiquement, dans un langage impératif :

— L'affectation remplace la valeur de la variable à gauche par la valeur de l'expression à droite, donc le signe de la variable après l'affectation est égal au signe de l'expression.

— Le signe d'une expression est calculé à partir des opérations réalisées dans l'expression. Par exemple, la somme de deux opérandes de même signe ou nuls garde le signe, sinon le signe est inconnu. On procède de manière similaire pour la soustraction, la multiplication, etc., en fonction de la nature de l'opération.

2.2.2 La sémantique collectrice

Dans un programme à analyser, nous pouvons imaginer des *points de programme*, par exemple, à la fin de chaque instruction. En considérant la sémantique concrète (par exemple, la valeur des variables lors d'une exécution réelle), en un même point du programme, correspondent en général un ensemble d'états concrets possibles (par exemple, la valeur des variables à la fin de chaque instruction peut être totalement différente en fonction du chemin d'exécution du programme).

La *sémantique collectrice* est définie comme l'ensemble de tous les états concrets en chaque point du programme. Par conséquent, le domaine de la *sémantique collectrice* est défini par l'ensemble des parties de l'ensemble des états concrets. Nous pouvons utiliser la *sémantique collectrice* comme un outil théorique pour tester la *validité* de certaines propriétés du programme mais elle est rarement calculable. Par exemple, si le **sur-ensemble** des valeurs d'une variable booléenne dans la condition d'un *IF* (en certain point du programme) est $\{TRUE\}$, on peut être sûr que la condition n'est jamais $FALSE$ et donc que le *IF* est **toujours pris** quelle que soit l'exécution du programme.

La fonction *Update* de la *sémantique collectrice* est simplement obtenue en appliquant *Update* de la sémantique concrète à chaque élément de l'ensemble d'états concrets produits par la *sémantique collectrice*.

De plus, les programmes sont rarement linéaires et contiennent des instructions de contrôle de flot. Pour **collecter** tous les états concrets en chaque point du programme, une fonction *Join* doit être définie pour décrire la fusion des états abstraits aux points de jonction du flot de contrôle. Le *Join* de la *sémantique collectrice* est simplement l'union des ensembles des états concrets. La *sémantique collectrice* est une sémantique basique qui fait uniquement l'abstraction sur le chemin d'exécution (car les états concrets sont groupés dans des ensembles en chaque point du programme sans prendre en compte la relation avec les chemins d'exécution). En général, pour concevoir une analyse, on commence par définir une sémantique concrète. Ensuite, la *sémantique collectrice* peut être directement déduite de la sémantique concrète. Les abstractions plus avancées sont conçues plus tard en se basant sur la *sémantique collectrice*, c'est-à-dire en considérant la sémantique collectrice comme la sémantique concrète.

Lorsque l'on conçoit une abstraction, il faut que l'abstraction finale (la plus abstraite) soit *valide* par rapport à la sémantique concrète initiale. Cela signifie que l'état abstrait calculé en chaque point du programme "inclut", au moins, tous les états concrets en ce point. Cette inclusion peut être implicite ou explicite. Dans le cas de la *sémantique collectrice*, "l'inclusion" désigne une inclusion explicite – tout état concret possible en chaque point du programme est dans l'ensemble d'états concrets calculé par la *sémantique collectrice*. Un exemple "d'inclusion" implicite est la représentation $\{+, -, 0, ?\}$ du signe des variables – le "?" représente tous les signes possibles : on a donc '?' qui inclut toutes les autres valeurs '$+$', '$-$' et '0'.

2.2.3 La conception des abstractions

L'interprétation abstraite fournit un cadre pour concevoir les abstractions et prouver leur validité. En général, on exige que tout domaine $\mathcal{D}$, abstrait ou concret, soit au moins un *Ordre Partiel Complet*, $\langle \mathcal{D}, \preceq, \top \rangle$: un ensemble $\mathcal{D}$, un ordre partiel $\preceq$ défini sur les éléments de $\mathcal{D}$, et un élément $\top$ qui est supérieur à tous les éléments de $\mathcal{D}$. Pour des raisons pratiques, souvent un plus petit élément $\bot$ est ajouté et permet de former un *treillis complet* (ce qui est aussi le cas de la *sémantique collectrice*, pour qui $\bot$ est l'ensemble vide).

Pour former une abstraction valide, nous devons assurer que tout état abstrait "inclut" au moins tous les états concrets correspondants. C'est-à-dire que le domaine concret $\langle \mathcal{D}, \preceq, \top \rangle$ et le domaine abstrait $\langle \mathcal{D}^\#, \preceq^\#, \top^\# \rangle$ forment une connexion de Galois :

$$\langle \mathcal{D}, \preceq, \top \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{D}^\#, \preceq^\#, \top^\# \rangle \tag{2.1}$$

avec l'opérateur d'abstraction $\alpha : \mathcal{D} \to \mathcal{D}^\#$ et l'opérateur de concrétisation $\gamma : \mathcal{D}^\# \to \mathcal{D}$ tels que :

$$\forall d \in \mathcal{D}, d^\# \in \mathcal{D}^\#, \alpha(d) \preceq^\# d^\# \Leftrightarrow d \preceq \gamma(d^\#) \tag{2.2}$$

Donc l'état abstrait obtenu par l'abstraction d'un état concret *inclut* forcément l'état concret dans le sens où la concrétisation (γ) de l'état abstrait est forcément supérieure ($\preceq$) à l'état concret initial.

Soit une fonction de transition dans le domaine concret, $inter : \mathcal{D} \to \mathcal{D}$ qui interprète le programme (ou une partie du programme) dans la sémantique concrète, et la fonction correspondante $inter^\# : \mathcal{D}^\# \to \mathcal{D}^\#$ qui interprète le programme dans la sémantique abstraite. Grâce à la connexion de Galois, nous savons que la relation ci-dessous tient si α, γ, $inter$ et $inter^\#$ sont monotones sur leurs domaines respectifs et donc que notre abstraction est valide :

$$inter(d) \preceq \gamma \circ inter^\# \circ \alpha(d) \tag{2.3}$$

Dans la pratique, la fonction $inter$ est souvent exprimée par deux fonctions : $Update$ pour interpréter l'exécution d'une séquence de programme et $Join$ pour exprimer la fusion d'états lors de la jonction du flot de contrôle. Nous pouvons ainsi dériver les propriétés ci-dessous qui sont exigées pour former une abstraction valide :

$$\begin{aligned} Update(d) &\preceq \gamma \circ Update^\# \circ \alpha(d) \\ Join(a,b) &\preceq \gamma \circ Join^\#(\alpha(a), \alpha(b)) \end{aligned} \tag{2.4}$$

Enfin, la relation entre états est représentée sur la figure 2.1 : il y a deux états d'entrée, l'état concret d, et l'état abstrait $d^\# = \alpha(d)$. Ces deux états subissent les mêmes transformations : les $Update$, $Join$ du domaine concret et les transformations correspondantes dans le domaine abstrait $Update^\#$ $Join^\#$.

$$\begin{array}{ccc} d^\# \in \mathcal{D}^\# & \xrightarrow{Update^\#, Join^\#} & d'^\# \in \mathcal{D}^\# \\ \uparrow \alpha & & \downarrow d' \preceq \gamma(d'^\#) \\ d \in \mathcal{D} & \xrightarrow[Update, Join]{} & d' \in \mathcal{D} \end{array}$$

FIGURE 2.1 – Connexion de Galois entre les sémantiques concrètes et abstraites.

Comme toutes les fonctions $Update$ et $Join$ sont consistantes par rapport à la connexion de Galois (équation 2.4), on est donc sûr qu'après des transitions ($Update^\#$, $Join^\#$) dans le domaine abstrait, l'état $d'^\#$ obtenu est valide par rapport au résultat des transitions correspondantes dans le domaine concret avec l'état final d', car la concrétisation de $d'^\#$ donne un état supérieur à d' – un état qui inclut le résultat concret.

Finalement, pour calculer les propriétés du programme, l'analyse commence par le début du programme et applique des $Update$ et des $Join$ en suivant le flot de contrôle dans le CFG pour mettre à jour l'état en chaque point du programme. Comme un programme peut contenir des boucles, un point du programme peut être visité plusieurs fois. Le caractère croissant de ces opérateurs, ainsi que l'existence de l'élément maximal $\top$ assure l'existence d'un point fixe des états. Il faut remarquer

que l'existence du point fixe assure seulement une terminaison **théorique** de l'analyse. Dans des cas réalistes, beaucoup de sémantiques ne sont pas calculables en un temps raisonnable.

De la même manière, nous pouvons appliquer $Update^{\#}$ et $Join^{\#}$ pour travailler dans le domaine abstrait. En général, on travaille dans le domaine abstrait pour avoir une meilleure performance parce que le domaine est plus petit et les fonctions de transitions sont croissantes : l'analyse converge donc plus vite. L'abstraction étant valide, le résultat obtenu est lui-même automatiquement valide par rapport à la sémantique concrète.

2.3 La structure de l'analyse de WCET

L'analyse de WCET est influencée par deux composantes principales : le programme lui-même et la micro-architecture du processeur sur lequel le programme est exécuté. La micro-architecture des processeurs modernes est complexe en raison de la présence de mécanismes d'accélération avancés comme les mémoires caches, le pipeline, la prédiction de branchement, etc. Les combiner augmente la combinatoire de l'analyse : si les domaines d'états de l'analyse des deux mécanismes sont respectivement A et B, le domaine d'états qui considère les deux composants simultanément est $A \times B$ – le produit cartésien des domaines. Le nombre d'états de l'analyse sur $A \times B$ est ainsi beaucoup plus grand que celui sur A **plus** celui sur B. C'est pour cette raison que l'analyse de WCET est généralement divisée en plusieurs sous-analyses, présentées ci-après. Il est important de noter que cette structure n'est pas la même pour tous les outils de WCET. Par exemple, les approches par vérification de modèle (que nous allons présenter plus tard dans ce chapitre) ont des structures assez différentes puisqu'elles font peu ou pas d'abstraction. Cependant les autres outils par analyse statique (aiT (FERDINAND et al., 2004), Chronos (LI et al., 2007), OTAWA (BALLABRIGA et al., 2010), etc.) ont des structures assez proches.

Dans cette section, nous présentons uniquement la structure de l'outil OTAWA illustrée par la figure 2.2. Les détails sont exposés en quatre parties : la construction des chemins d'exécution, les analyses globales, le calcul de WCET selon la méthode IPET et l'analyse du pipeline pour obtenir le temps des blocs.

2.3.1 Construction du CFG

Du point de vue du développement des systèmes temps-réel embarqués, les logiciels sont souvent développés avec des langages de programmation haut niveau (comme du C, C++, Ada, etc). Cependant, le code source est compilé sous forme de fichiers binaires exécutables sur lesquels se base le calcul de WCET. L'outil OTAWA commence par lire et décoder ce fichier binaire grâce à une description du jeu d'instructions (ISA, *Instruction Set Architecture*) exprimée à l'aide du langage de description *Sim-NML* (RAJESH et al., 1999 ; HERBEGUE et al., 2014 ; RATSIAMBAHOTRA et al., 2009). De nombreuses ISAs sont actuellement supportées par OTAWA : ARM, RISC-V, PowerPC, TriCore, etc. Nous pouvons faire une analogie entre cette phase dans le calcul du WCET et le front-end d'un compilateur : ils ont tous les deux comme but de générer une représentation intermédiaire des instructions. Concrètement, cela consiste à annoter les instructions avec des informations spécifiées par l'ISA, typiquement :

— le type d'instruction / l'opération réalisée (branchement / arithmétique / multiplication /...),

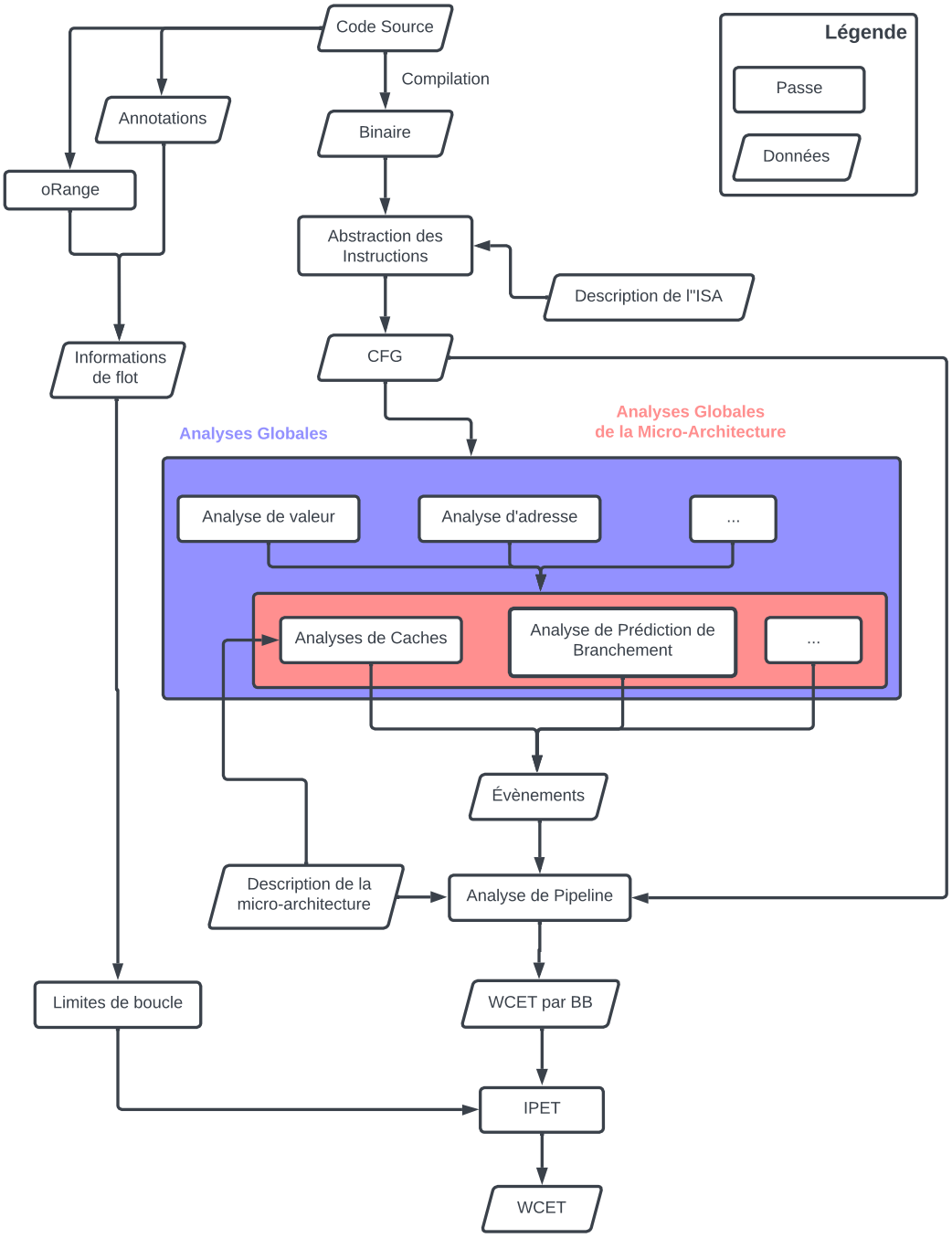


FIGURE 2.2 – Structure de l'analyse de WCET

— les registres lus et écrits par l'instruction,
— l'adresse de l'instruction,
— l'adresse cible pour une instruction de branchement,
— la sémantique de l'instruction par rapport au pipeline, c'est-à-dire la liste et l'ordre des ressources du pipeline utilisées par l'instruction,
— etc.

Cette interface permet aux analyses suivantes de travailler dans un environnement indépendant de l'ISA.

Ensuite, le graphe de flot de contrôle (CFG, *Control Flow Graph*) du programme peut être construit.

2.3.2 Les analyses globales

Commence ensuite la phase des analyses globales. Dans un premier temps, des analyses de valeur et d'adresse sont appliquées pour déterminer les valeurs possibles des registres et des adresses accédées en mémoire. Puis, les analyses des mécanismes d'accélération de la micro-architecture sont réalisées pour déterminer leur comportement temporel lors de l'exécution. Il s'agit typiquement de l'analyse du cache d'instructions et du cache de données ainsi que des analyses du prédicteur de branchement, du buffer d'écriture, etc. Ces analyses prennent en compte les informations fournies par les analyses de valeur et d'adresse ainsi que la description de la micro-architecture (qui contient par exemple les caractéristiques des caches). En sortie de ces analyses globales, les *évènements* (que nous avons présentés dans la section 1.1.1) sont générés pour décrire les variations temporelles engendrées par ces mécanismes. Par exemple, les évènements de cache concernent les accès mémoire pouvant donner lieu parfois à des *Hit*, ou parfois à des *Miss*.

2.3.3 La méthode IPET

Généralement, le nombre de chemins d'exécution possibles d'un programme est trop élevé pour les représenter explicitement. Il est impossible de calculer le temps d'exécution de chaque chemin et d'en prendre le maximum pour déterminer le WCET. Une solution classique à ce problème est la méthode IPET (Implicit Path Enumeration Technique ou *technique d'énumération implicite des chemins*) (Li et al., 1995).

Cette méthode représente le temps d'exécution du programme par la somme des WCETs de chaque bloc de base (t_a) pondérés par leurs nombres d'exécution (x_a) :

$$WCET = \sum_{a \in V_{\text{CFG}}} (t_a \times x_a) \tag{2.5}$$

Les compteurs x_a représentent le nombre d'exécutions d'un bloc sur le pire chemin "implicite" d'exécution. Ils sont soumis aux contraintes de flots. Par exemple, pour chaque BB du CFG, le flot entrant est égal au flot sortant :

$$\forall a \in V_{\text{CFG}}, \sum_{b \to a \in IN(a)} x_{b \to a} = \sum_{a \to b \in OUT(a)} x_{a \to b} = x_a \tag{2.6}$$

où les fonctions IN et OUT renvoient l'ensemble des arcs entrants et sortants d'un bloc de base.

De plus, les programmes peuvent contenir des boucles, ce qui créent également des boucles dans le CFG correspondant. Les contraintes de flot décrites précédemment ne sont alors pas suffisantes car il faut aussi donner une borne supérieure aux compteurs des arcs retour de ces boucles : sans cela, une boucle est considérée comme s'exécutant un nombre infini de fois. Par conséquent, des contraintes supplémentaires sont obligatoires pour limiter les compteurs concernés. Ces contraintes sont appelées des *contraintes de limite de boucle* :

$$\forall loop \in LOOPS(\text{CFG}), \sum_{a \to b \in BACK(loop)} x_{a \to b} \leq LOOP\_BOUND(loop) \tag{2.7}$$

La fonction *LOOPS* renvoie l'ensemble des boucles du CFG, *BACK* renvoie l'ensemble des arcs retour d'une boucle donnée et la fonction *LOOP_BOUND* donne la limite d'une boucle : une borne supérieure de son nombre d'itérations.

La fonction objectif, qui maximise l'expression du WCET (équation 2.5) et les contraintes sur les flots forment un programme linéaire en nombres entiers (ILP, *Integer Linear Programming*). Par conséquent, la méthode IPET est utilisée dans la dernière étape de notre schéma de l'analyse de WCET (figure 2.2). Elle reçoit en entrée le WCET de chaque BB (calculé par l'analyse de pipeline que nous allons présenter dans la section suivante) et les bornes de boucles.

Pour que le système ILP converge, une borne doit être déterminée pour chaque boucle (les *limites de boucle* dans la figure 2.2). Cette information est obtenue avec *oRange* (MICHIEL et al., 2010), un outil qui calcule les bornes de boucle à partir du code source. Dans le cas où l'outil est incapable de calculer les bornes de certaines boucles, ces informations peuvent être fournies par des annotations (dans un fichier indépendant), au format de *FFX* (BONENFANT et al., 2012). Dans l'équation 2.7, nous exploitons les bornes de boucle pour créer des contraintes sur les arcs retour des boucles, mais ces contraintes sont automatiquement transférées aux arcs entrants par l'égalité des flots (l'équation 2.6).

Nous pouvons voir le système ILP comme une abstraction des chemins d'exécution par rapport à la sémantique temporelle du programme parce que nous ne nous intéressons pas explicitement aux chemins d'exécution mais au nombre d'exécutions de chaque bloc (d'où le nom *énumération implicite des chemins*).

Une autre source d'imprécision introduite par la méthode IPET est qu'elle ne considère qu'un seul temps par BB alors que ce temps peut varier en fonction du chemin d'exécution. Il existe des cas où cette abstraction introduit une surestimation considérable. Un cas typique est la *persistance de cache* que nous avons déjà présentée dans la section 1.1.3 : il s'agit du comportement d'un cache pour lequel, dans une boucle, les accès font des *Miss* à la première itération et ensuite seulement des *Hits* dans les itérations suivantes parce que les données ou instructions sont déjà chargées dans le cache. Le pire cas correspond généralement aux Miss (si on ignore les anomalies temporelles) mais ce pire cas n'arrive qu'une seule fois pour chaque exécution de la boucle. Cependant, l'approche IPET classique considère un seul temps – le WCET de chaque bloc, et donc il considère le pire cas pour chaque itération, ce qui peut introduire énormément de surestimation. Mais comme discuté dans la section 1.1.3, la méthode IPET n'est pas la seule cause de cette surestimation. Pour bien gérer la *persistance de cache*, l'analyse de pipeline doit aussi être capable de fournir plusieurs temps et d'expliciter la relation entre ces temps et les évènements du cache.

Dans notre nouvelle analyse de pipeline, présentée plus loin dans cette thèse,

nous arrivons à calculer explicitement les états du pipeline ce qui permet de fournir plusieurs temps (en fait tous) ainsi que leur relation avec les évènements. La méthode IPET classique n'est alors plus adaptée car elle ignore simplement ces informations supplémentaires, en ne considérant que le pire temps. Par conséquent, pour améliorer le WCET final, nous proposons une amélioration (BAI et al., 2022) présentée dans le chapitre 5.

2.3.4 Analyse de pipeline

Grâce à la méthode IPET, nous pouvons nous concentrer sur le WCET de chaque BB. Comme discuté dans le chapitre 1, le temps d'exécution dépend fortement de la structure du pipeline du processeur. C'est pour cette raison que cette phase est appelée *analyse de pipeline*. L'analyse de pipeline reçoit en entrée une séquence d'instructions (un BB), la description du pipeline et les évènements produits par les analyses globales et calcule le WCET du BB, comme montré sur la figure 2.2. L'analyse de pipeline joue un rôle très important pour la précision et la performance du calcul du WCET, ce qui constitue le sujet central de cette thèse – comment calculer de manière efficace et précise du WCET des BB en considérant à la fois les évènements et la structure du pipeline ?

2.3.5 Discussion

L'avantage principal de la structure de l'outil d'analyse de WCET présentée dans cette section est la décomposition du problème initial en sous-problèmes. En fait, dans cette structure, nous décomposons la sémantique temporelle du programme en plusieurs éléments :

— le CFG, qui représente les chemins d'exécution et les flots de contrôle, dont la sémantique temporelle est modélisée avec la méthode IPET ;
— la sémantique temporelle du pipeline, modélisée par l'analyse du pipeline ;
— la sémantique temporelle des caches et du prédicteur de branchement et d'autres composants indépendants du pipeline, modélisée par l'analyse globale de chaque composant.

La décomposition de la sémantique temporelle en sous-sémantiques est en effet une abstraction parce que les relations entre les sous-sémantiques sont ignorées. Nous allons illustrer cet effet par un exemple.

Soit l'ensemble d'états concrets possibles du composant A en un point du programme $\{a_1, a_2\}$; soit l'ensemble d'états concrets possibles du composant B au même point du programme $\{b_1, b_2\}$. Comme les deux composants sont analysés indépendamment, lors de la prise en charge des deux composants dans l'analyse de pipeline, leurs états sont combinés pour former l'état du pipeline. Considérés séparément, l'état est défini dans $A \times B = \{(a_1, b_1), (a_1, b_2), (a_2, b_1), (a_2, b_2)\}$ – toutes les combinaisons possibles de l'état de A et l'état de B, puisqu'on ne connaît pas la relation entre ces états. Cependant, si on considère le scénario suivant : deux chemins d'exécutions passent par ce point du programme. Par un chemin, l'état de A et B est a_1 et b_1 ; par l'autre chemin, l'état est a_2 et b_2. Donc, l'ensemble des états possibles de A et B à ce point est $\{(a_1, b_1), (a_2, b_2)\}$, qui est seulement un sous-ensemble de $A \times B$ quand on considère A et B indépendamment. Cela signifie que l'analyse indépendante de A et de B introduit, potentiellement, une surestimation. Du point de vue de l'interprétation abstraite, nous pouvons visualiser cette abstraction par la connexion de Galois formée par la fonction α qui déduit l'état des deux composants à partir de

leurs états indépendants, et sa fonction réciproque γ :

$$\begin{array}{ccccc} \{(a_1,b_1),(a_2,b_2)\} & \underset{\alpha}{\rightarrow} & (\{a_1,a_2\}, & \{b_1,b_2\}) & \underset{\gamma}{\rightarrow} \{(a_1,b_1),(a_1,b_2), \quad (a_2,b_1),(a_2,b_2)\} \\ d & \underset{\alpha}{\rightarrow} & d^{\#} & & \underset{\gamma}{\rightarrow} \quad \gamma(d^{\#}) \succeq d \end{array} \tag{2.8}$$

Comme pour toute abstraction, nous devons faire un compromis entre précision (surestimation) et performance (complexité) de l'analyse. Mais des sémantiques qui sont peu corrélées par nature (par exemple, l'analyse de cache et l'analyse de la prédiction de branchement) n'ont presque pas de relation et la perte de précision est acceptable. Face à la complexité des programmes et de la micro-architecture, la capacité de calcul dont nous disposons aujourd'hui ne nous laisse pas beaucoup de choix.

2.4 Vérification de modèles

Parmi les techniques utilisées dans le domaine de l'analyse statique, la vérification de modèles (CLARKE, 1997 ; BURCH et al., 1992) est certainement une des techniques les plus connues et utilisées. Elle est également applicable au problème du calcul de WCET. Le principe de la vérification de modèles est de représenter l'exécution du programme comme un automate fini et de parcourir tout l'espace du domaine d'états et ainsi prouver la validité de certaines propriétés. Dans son application au problème du calcul de WCET, on peut le voir comme une simulation du modèle temporel du processeur. Comme on s'intéresse spécialement aux aspects temporels du programme, une abstraction est faite à la construction du modèle temporel – certaines informations non-relatives au comportement temporel sont écartées. Cependant, à cause du modèle du calcul de la vérification de modèles, la représentation du comportement temporel par un automate fini doit être complète et il est par conséquent difficile de diviser le modèle temporel en sous-modèles. Au contraire, dans la structure d'analyse de WCET présentée dans la section précédente, le modèle de la micro-architecture est divisé en modèles du cache, du prédicteur de branchement et du pipeline. Ce type d'abstraction est moins direct pour les approches par vérification de modèles et, à cause de la taille du domaine d'états, l'approche peut se heurter à des problèmes de performance. Cependant, avec moins d'abstraction, les approches par vérification de modèles ont un avantage imbattable – la précision. Comme elles ne font presque pas d'abstraction, elles ne souffrent pas des *anomalies temporelles* (EISINGER et al., 2006) – cela revient pratiquement à découvrir explicitement tous les états possibles en prenant en compte les divergences d'état.

L'outil UPPAAL (LARSEN et al., 1997) est un des outils les plus connus pour la vérification de modèles. Dans le framework présenté dans CASSEZ et al., 2013 ; CASSEZ, 2011, le problème du calcul de WCET est décomposé en (a) encoder le modèle temporel du processeur en un automate temporel, et (b) exécuter le programme (avec chaque entrée possible) dans l'automate et trouver le temps d'exécution le plus long. Il est clair que le problème de performance est dû au nombre d'entrées possibles et à la complexité de la micro-architecture. Pour atténuer ce problème, la technique de *program slicing* CASSEZ et al., 2013 est appliquée pour éliminer les instructions qui n'ont pas d'effet sur le comportement temporel. Mais pour la plupart des processeurs, presque toutes les instructions ont un effet sur l'état de la micro-architecture et ne peuvent donc pas être ignorées dans le calcul du WCET. Les critères de sélection des instructions décrits dans l'article (seulement les instructions conditionnelles

et les instructions d'accès à la mémoire – lecture ou écriture, sont gardées) nous paraissent non adaptés à des micro-architectures plus compliquées que celle présentée dans l'article.

Les résultats expérimentaux dans CASSEZ et al., 2013 ; CASSEZ, 2011 ne portent que sur des programmes simples de la collection de benchmarks *Mälardalen*. Et même pour ce sous-ensemble simple, l'entrée des programmes est supposée unique [3]. Par conséquent, il existe un seul chemin d'exécution possible, ce qui réduit énormément la combinatoire des états produits.

Pour conclure, la vérification de modèles applique, par nature, peu d'abstractions, ce qui limite sa performance pour des programmes et des micro-architectures réellement complexes. Par contre, son avantage sur la précision et la possibilité de trouver le chemin exact du WCET est remarquable dans les cas où elle est effectivement applicable, c'est-à-dire sur de petits programmes simples ou bien si l'entrée du programme est statiquement connue. Ceci étant, la recherche dans ce domaine est encore très active.

2.5 AbsInt

L'outil *aiT* développé par *AbsInt* (FERDINAND et al., 2004) est un des outils les plus utilisés dans l'industrie pour l'analyse statique de WCET. Comme son nom l'indique, l'outil est fortement basé sur la théorie de l'interprétation abstraite. Les analyses globales dans aiT sont parmi les plus avancées du domaine, notamment les analyses de cache (TOUZEAU et al., 2019 ; STOCK et al., 2019). Mais au niveau de la structure de l'analyse de WCET, d'après la structure d'aiT publiée dans FERDINAND et al., 2001, la différence la plus remarquable en comparaison avec OTAWA (voir section 2.3) est que les variations temporelles sont prises en compte différemment dans l'analyse de pipeline. Autrement dit nos modèles de calcul de temps dans le pipeline sont différents. Néanmoins, dans les deux outils, l'analyse du pipeline produit la même sortie : le (ou les) WCET de chaque BB, qui est ensuite fourni à la méthode IPET pour calculer le WCET final.

Note Le fonctionnement interne d'aiT n'est, à notre connaissance, pas complètement publié. Cette situation est normale pour un outil industriel. Par conséquent, la description donnée ici est issue des publications existantes et de ce que nous avons pu en déduire et elle n'est donc probablement pas complètement à jour par rapport à la version actuelle de l'outil.

2.5.1 Le Modèle de pipeline dans *aiT*

Le modèle de pipeline de *aiT* (SCHNEIDER et al., 1999 ; THESING, 2004) s'apparente à une simulation cycle par cycle du comportement temporel du processeur. Contrairement aux approches par vérification de modèle, cette "simulation" est uniquement appliquée dans l'analyse de pipeline ; le comportement des autres mécanismes d'accélération est traité par des analyses globales indépendantes. Concrètement, l'état de pipeline décrit : (a) les instructions dans chaque composant (étage de

3. Tel quel, le benchmark fournit un état initial unique qui est souvent ignoré pour fournir un cadre plus réaliste dans l'évaluation du calcul de WCET.

pipeline), et (b) le temps restant de l'exécution de chaque instruction dans le composant où elle se situe. Donc, le domaine concret du pipeline est :

$$Pipe := \mathcal{I} \rightarrow \mathsf{S} \times \mathbb{N} \tag{2.9}$$

où $\mathcal{I}$ est l'ensemble des instructions, S l'ensemble des ressources du pipeline (typiquement des étages) et $\mathbb{N}$ un compteur dénotant le temps que l'instruction doit encore rester dans son composant courant.

La sémantique *cycle* du pipeline décrit comment les états évoluent après un cycle de processeur :

$$cycle : Pipe \rightarrow Pipe \tag{2.10}$$

Normalement, après un cycle, le compteur de chaque composant est diminué d'un. À cela, il faut ajouter les contraintes d'exécution dans le pipeline : les *aléas de données* – si les opérandes sont disponibles, les *aléas structurels* – si les unités fonctionnelles nécessaires sont disponibles, etc. Par conséquent, une instruction peut être bloquée à un étage même si son compteur atteint 0 (compteur bloqué à 0) si les ressources nécessaires pour son exécution dans l'étage suivant ne sont pas encore disponibles. Ces contraintes d'exécution sont décrites en logique et sont donc assez complexes. Un exemple complet peut être trouvé dans HAHN et al., 2020 qui décrit complètement le comportement temporel d'un petit pipeline.

En profitant du cadre de l'interprétation abstraite, la sémantique abstraite de pipeline est définie comme la *sémantique collectrice* de $Pipe$:

$$Pipe^{\#} := \wp(Pipe) \tag{2.11}$$

Le domaine abstrait $Pipe^{\#}$ est l'ensemble des parties de $Pipe$. La fonction de *Join* est simplement l'union. La fonction d'*Update* est héritée de *cycle*, mais considère les variations temporelles : la transformation d'un état concret dans le domaine abstrait peut diverger en plusieurs états abstraits selon l'occurrence des variations temporelles (évènements).

En suivant le calcul classique en interprétation abstraite, l'analyse doit atteindre un point fixe et les états trouvés forment donc un sur-ensemble des états concrets possibles de l'exécution réelle. À partir des états obtenus pour chaque BB, un pire temps est déduit et est ensuite exploité selon la méthode IPET, qui calcule finalement le WCET du programme.

2.5.2 Performances d'*aiT*

Comme aiT intègre des analyses globales séparées de l'analyse de pipeline, il est soumis au même genre de problème qu'OTAWA : les variations temporelles produites par les analyses globales et gérées dans l'analyse de pipeline produisent une explosion combinatoire du nombre d'états. Le modèle de pipeline d'*aiT* étant basé sur la *sémantiques collectrice* des états concrets, il peut faire face à une explosion combinatoire en relation avec nombre d'évènements. En plus, le flot de contrôle cause aussi de la divergence d'états ce qui crée encore plus d'états. C'est pourquoi, plusieurs techniques pour atténuer le problème de la performance ont été publiées.

Compactage de la représentation d'état et de transition

Dans WILHELM, 2007, la fonction de transition (*cycle*) et les états ($Pipe^{\#}$) sont représentés par un automate à états finis et des Binary Decision Diagrams (BDD)

sont utilisés pour représenter les états et les transitions de manière plus efficace. Mais jusqu'à présent, pour ce que nous avons pu trouver dans la littérature, il n'existe ni évaluation concrète de cette technique, ni preuve que cette technique toute seule peut résoudre le problème de performance dans des applications réelles.

Suppression des états

Comme on s'intéresse uniquement au pire cas d'exécution, il n'est pas obligatoire de garder tous les états possibles mais seulement le pire état global (ce qui peut introduire de la surestimation car le pire cas n'arrive pas toujours). Mais en présence des anomalies temporelles, il est difficile de savoir, lors de la divergence d'état causée par des variations temporelles, quel est le pire cas effectif. REINEKE et al., 2009 propose une technique pour pré-analyser la fonction *cycle* et construire un ordre partiel entre les états ($\in Pipe$) de manière à ce qu'un état supérieur (selon l'ordre partiel) à un autre soit prouvé d'être globalement pire qu'un autre. Par conséquent, les états n'étant pas les *pires* peuvent être écartés pour réduire la taille de l'état abstrait ($\in Pipe^{\#}$). En présence d'anomalies temporelles, comme toute paire d'états n'est pas forcément comparable (sinon cela prouverait l'absence des *anomalies*), l'ordre obtenu n'est que partiel ce qui permet seulement de réduire la taille de l'ensemble d'états concrets sans toujours pouvoir se ramener à un seul pire état.

Cette technique présente deux problèmes majeurs : (a) l'espace du domaine $Pipe$ et de la fonction de transition *cycle* est déjà assez grand, et trouver tous les ordres partiels paraît complexe ; (b) le ratio des paires comparables par rapport au nombre total de paires peut être petit. Par conséquent, les cas effectifs où les états peuvent être écartés sont relativement rares ce qui nuit à l'efficacité de cette technique. Cependant, à notre connaissance, aucune expérimentation permettant d'évaluer ces deux problèmes n'a été publiée.

Pipeline d'exécution strictement dans l'ordre

L'article HAHN et al., 2015a introduit la notion de pipeline à exécution strictement dans l'ordre basé sur la *monotonie*. La monotonie qualifie la fonction de transition *cycle*, c'est-à-dire que, après une application de *cycle*, l'état résultant est forcément "pire" que l'état précédent ; du point de vue temporel, l'état a progressé. Avec cette propriété, lors de la divergence d'un état s en s_1 et s_2 (à cause d'une variation temporelle), il est toujours possible de trouver un état s' qui est pire que s_1 et s_2. Cet état s' peut être s_1 ou s_2 s'ils sont comparables directement. Mais il peut aussi être un état différent, donc dans ce sens, les états ne sont pas "écartés" mais "fusionnés" en un état abstrait qui est supérieur. En tout cas, cette technique permet de réduire un ensemble d'états concrets en un seul état, ce qui réduit énormément la complexité de l'analyse. Bien-sûr, comme l'abstraction perd de la précision, il devient possible pour l'analyse de choisir de moins fusionner les états en fonction de leur nombre, ce qui permet à l'analyse de dynamiquement ajuster le compromis entre performance et précision.

La *monotonie* du pipeline est prouvée si le pipeline a une exécution strictement dans l'ordre, c'est-à-dire si toutes les ressources du pipeline sont allouées dans l'ordre du programme. Avec cette condition, l'absence des *anomalies temporelles* est automatiquement prouvée par la monotonie du pipeline. Cette preuve a été également donnée dans WENZEL et al., 2005 sans la notion de monotonie. Ses auteurs concluent "la présence des ressources allouées dans le désordre est une condition nécessaire aux anomalies temporelles". HAHN et al., 2020 propose un guide de conception et

de preuve d'un cœur d'exécution strictement dans l'ordre. Un tel cœur est réellement implémenté dans GRUIN et al., 2021 sur la base d'Ariane, un cœur RISC-V open-source (ZARUBA et al., 2019).

Les processeurs à exécution strictement dans l'ordre améliorent effectivement la prédictibilité en réduisant la complexité de l'analyse de pipeline, mais avec le coût de limiter le parallélisme dans le pipeline. Autrement dit, cette approche sacrifie la performance du processeur aux dépens de la performance de l'analyse.

Discussion

Pour conclure, les méthodes utilisées dans l'outil aiT sont assez satisfaisantes si le processeur cible a une exécution strictement dans l'ordre mais risquent de souffrir du problème de performance dans le cas général. La *monotonie* du pipeline contribue beaucoup à réduire la complexité de l'analyse. Au-delà de l'analyse de pipeline, la *monotonie* garantit la composabilité, une propriété très importante pour le calcul du WCET sur des processeurs multi-cœurs (HAHN et al., 2015b ; HAHN et al., 2016 ; HAHN, 2018). Cependant, renforcer la monotonie au niveau pipeline se paye par une certaine perte de performance du processeur. Ici aussi, des travaux en cours permettront peut-être de compenser cet inconvénient.

2.6 Graphe d'exécution

Le graphe d'exécution (eXecution Graph) est une autre approche pour l'analyse de pipeline et est utilisée dans l'outil Chronos (LI et al., 2007). Le principe du graphe d'exécution est de modéliser le comportement temporel du pipeline par les dépendances entre l'exécution des instructions dans ses étages. Ce principe a été premièrement proposé par ENGBLOM, 2002 et maturé dans LI et al., 2006 ; ROCHANGE et al., 2009. Le modèle de ROCHANGE et al., 2009 est une variante des graphes d'exécution qui est implémentée dans OTAWA. Cette implantation est l'ancienne approche de l'analyse de pipeline qui sera utilisée pour comparer avec les approches proposées dans cette thèse. Bien qu'il existe plusieurs variantes des graphes d'exécution, dans cette section, nous allons commencer par présenter la version proposée dans LI et al., 2006 que nous nommons *graphe d'exécution original*.

La figure 2.3 montre un exemple de graphe d'exécution original construit à partir d'une séquence d'instructions situées à gauche du graphe. Un graphe d'exécution est un graphe dont les nœuds représentent l'exécution des instructions dans les étages du pipeline. À chaque nœud est associée une latence qui correspond à son temps d'exécution. Les arcs solides représentent les dépendances entre les nœuds. Dans l'exemple de la figure 2.3, quatre types de dépendances sont pris en compte :

— Les instructions doivent traverser le pipeline dans l'ordre des étages, par exemple, pour une instruction I, on a un arc IF(I) $\rightarrow$ ID(I).
— L'exécution dans l'ordre des instructions dans chaque étage du pipeline. Par exemple, pour une instruction I et la suivante (I+1), IF(I) $\rightarrow$ IF(I+1).
— Les dépendances de données : si une instruction I produit une valeur qui est utilisée par une instruction suivante J, un arc WB(I) $\rightarrow$ EX(J) est créé. Par exemple, WB(1) $\rightarrow$ EX(3).
— L'effet de capacité des mémoires tampon entre les étages est également modélisé. Supposons que les files sont toutes FIFO (First-In-First-Out) et ont toutes deux entrées, pour une instruction I, la deuxième instruction suivante, I+2 avec IF(I+2), doit attendre que ID(I) libère une entrée de la file pour commencer son exécution.

Avec ces *dépendances*, la date de démarrage de chaque nœud peut être déduite comme la date au plus tôt à laquelle toutes les dépendances sont satisfaites, c'est-à-dire le maximum des dates de fin des prédécesseurs.

Les arcs pointillés bidirectionnels modélisent la contention entre les nœuds lors de l'accès à une ressource allouée dans le désordre. Dans cet exemple, la contention est engendrée par les unités fonctionnelles à l'étage EX, allouées selon la politique FCFS (First-Come-First-Served), qui sont partagées par toutes les instructions exécutées à cet étage. L'arc EX(1) $\leftrightarrow$ EX(4) modélise l'utilisation de l'unité de multiplication par l'instruction 1 et l'instruction 4. Si l'unité est libre et que l'instruction 1 et 4 sont toutes les deux prêtes au même cycle, l'instruction 1 a la priorité et va occuper l'unité. Une fois que l'instruction 1 a fini d'utiliser l'unité, l'instruction 4 peut l'occuper. Mais si l'instruction 4 est disponible avant l'instruction 1, elle peut occuper l'unité de multiplication avant l'instruction 1.

```
1: mult r6  r10  4
2: mult r1  r10  r1
3: sub  r6  r6   r2
4: mult r4  r8   r4
5: add  r1  r1   r4
```

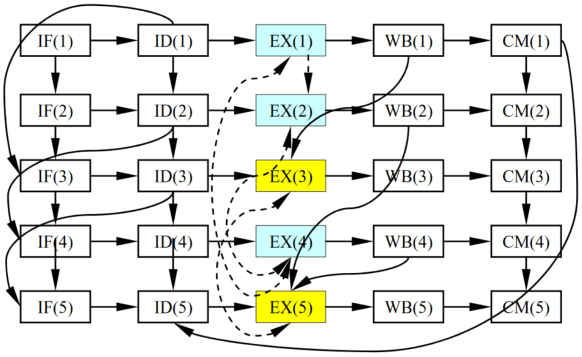


FIGURE 2.3 – Exemple de graphe d'exécution original.

Dans le calcul du graphe d'exécution original, la partie *dépendance* (les arcs solides) et la partie *contention* (les arcs pointillés) sont considérées séparément. La partie *dépendance* modélise le comportement *statique* du pipeline parce que l'ordre de résolution de ces dépendances ne dépend pas de l'état dynamique du pipeline mais uniquement de l'ordre des instructions dans le programme – c'est la partie du pipeline dans laquelle chaque ressource est allouée dans l'ordre du programme. C'est pour cette raison qu'il n'existe pas d'arc solide qui pointe en arrière. En fait, le graphe contenant uniquement ces arcs solides est un DAG (Directed Acyclic Graph) pour lequel on peut produire au moins un ordre topologique par lequel les dépendances sont résolues dans un ordre correct. Au contraire, les arcs pointillés modélisent la contention lors de l'allocation dynamique des ressources allouées dans le désordre. Dans ce cas, avec la politique FCFS, l'allocation dépend de la date de démarrage des nœuds qui est calculée à partir de la partie correspondant à l'exécution dans l'ordre.

Pour prendre en compte les évènements ainsi que la latence variable à cause de la contention, le graphe d'exécution original utilise les intervalles pour représenter les temps d'exécution possibles des nœuds. À partir de ce principe, la résolution du temps d'exécution est présenté dans l'algorithme 1 qui est une version plus compacte de l'algorithme proposé dans LI et al., 2006.

Ligne 1, la variable *noeuds_separes* est initialisée à l'ensemble vide. Cette variable enregistre l'ensemble des paires de nœuds qui sont en contention, c'est-à-dire liés par un arc pointillé et dont les intervalles de temps d'exécution se superposent. Lignes 2-4, les dates minimales et maximales du démarrage de chaque nœud (ρ_{min} et ρ_{max}) ainsi que les dates minimales et maximales de la fin de chaque nœud (ρ^*_{min} et ρ^*_{max}) sont initialisées à 0 et ∞. Cela signifie que chaque nœud démarre dans l'intervalle de date $[0, \infty)$ et finit dans l'intervalle $[0, \infty)$. Donc, initialement, tous les nœuds

Algorithm 1 Calcul du graphe d'exécution original

1: $noeuds\_separes \leftarrow \emptyset$
2: **for** $node \in G$ **do**
3: $\quad \rho_{min}[node] \leftarrow 0\,; \rho_{max}[node] \leftarrow \infty\,; \rho^*_{min}[node] \leftarrow 0\,; \rho^*_{max}[node] \leftarrow \infty\,;$
4: **end for**
5: **do**
6: $\quad \rho_{max}[\omega] \leftarrow 0$
7: $\quad$ **for** $node \in G$ dans l'ordre topologique **do**
8: $\quad\quad$ MAJ($\rho_{min}[node], \rho_{max}[node], \rho^*_{min}[node], \rho^*_{max}[node]$)
9: $\quad\quad$ MAJ-CONT($\rho_{min}[node], \rho_{max}[node], \rho^*_{min}[node], \rho^*_{max}[node]$)
10: $\quad$ **end for**
11: $\quad$ VERIFIER($noeuds\_separes$)
12: **while** $noeuds\_separes$ est inchangé

peuvent être en contention.

Ligne 6, la date de démarrage maximale du nœud d'entrée du graphe ω est mis à 0 (parce que la séquence commence son exécution à la date 0). Ensuite, lignes 7-9, pour chaque nœud du graphe dans l'ordre topologique, les dates de démarrage et de fin du nœud sont mises à jour en fonction des *dépendances*. MAJ-CONT met à jour toutes ces dates en considérant la contention avec d'autres nœuds (qui sont liés par un arc pointillé et dont la paire est présente dans *noeuds_separes*). Ligne 11, l'ensemble des paires en contention est vérifié : les paires qui ne sont effectivement plus en contention (les intervalles ne se superposant plus) sont éliminées de l'ensemble. Cette opération réduit l'ensemble des paires en contention et donc à la prochaine itération de la ligne 9, l'intervalle des dates se réduit. Cette procédure (lignes 5-12) est répétée à chaque itération et comme le nombre de paires de nœuds en contention (*noeuds_separes*) se réduit à chaque itération, l'algorithme va finir par atteindre un point fixe. À ce moment-là, l'intervalle de la date de fin du dernier nœud est l'intervalle de temps d'exécution du graphe et la borne supérieure donne le WCET du bloc. Comme l'algorithme est itératif et comme l'intervalle initialement associé à chaque nœud est $[0, \infty)$, l'algorithme peut s'arrêter à tout moment et les intervalles de temps obtenus sont toujours valides mais pas forcément précis.

2.6.1 Discussion

En comparant avec le modèle d'analyse du pipeline d'*aiT*, nous voyons intuitivement que l'approche originale des graphes d'exécution est plus abstraite – moins précise car les temps d'exécution sont abstraits par la représentation par intervalle. Les contentions sont caractérisées par la superposition des intervalles de temps d'exécution ce qui crée des contentions qui sont parfois impossibles (qui ne se produiraient pas si les dates exactes étaient disponibles). De plus, cette méthode de calcul peut être négativement impactée quand le nombre d'évènements augmente : plus il y a d'évènements, plus les intervalles sont grands et imprécis. Cette situation fait qu'il y a beaucoup de fausses contentions prises en compte par l'analyse, ce qui à nouveau amplifie l'imprécision. Les résultats expérimentaux présentés dans Li et al., 2006 sont impressionnants car seuls les évènements du cache d'instructions sont considérés. Cependant, pour les processeurs qui en sont équipés, le cache de données est sans conteste la source principale des variations temporelles parce que l'utilisation des données suit des motifs plus complexes et son analyse dépend fortement de l'analyse d'adresse dont la précision est aussi très variable. Nous n'avons

pas trouvé d'expérimentation ultérieure pour évaluer l'approche originale dans ce genre de conditions, plus réalistes, mais nous pensons que cette approche risque de souffrir des problèmes de précision lors que le nombre d'évènements augmente.

Malgré ce défaut de précision, il nous semble que modéliser le comportement du pipeline à la granularité de l'étage et de l'instruction est un bon choix. Au niveau de la complexité de l'analyse, nous pouvons intuitivement justifier le choix du graphe d'exécution par deux avantages en comparaison avec le modèle à la granularité du cycle (typiquement, le modèle de pipeline dans *aiT*) : (a) la granularité "instructions dans les étages" est plus grosse (potentiellement plus rapide) que la granularité du cycle – les dates de démarrage entre les nœuds peuvent différer de plusieurs cycles ce qui permet d'avancer de plusieurs cycles dans un calcul de temps ; (b) le modèle d'analyse d'*aiT* doit garder l'état du pipeline à chaque cycle du processeur ce qui est plus lourd que les temps calculés à travers un graphe d'exécution. De plus, si le processeur permet une exécution dans l'ordre, les *dépendances* sont suffisantes pour représenter le comportement **exact** du pipeline ; l'imprécision dans le graphe original est causée par l'utilisation des intervalles pour représenter le temps d'exécution des nœuds. On peut imaginer que, si les temps d'exécution étaient explicitement conservés, on serait capable de calculer le temps d'exécution **exact** de chaque nœud (bien-sûr, avec un algorithme différent de l'original). La raison qui nous empêche de faire un tel calcul est bien sûr le nombre combinatoire de temps à traiter, lui-même amplifié par le nombre d'évènements. C'est pour cette raison que le graphe d'exécution original fait une abstraction (par intervalle) sur le temps d'exécution.

Le nouveau modèle d'analyse du pipeline que nous présentons dans la suite de ce document (chapitre 4) se base aussi sur le principe des graphes d'exécution : modéliser les *dépendances* dans le pipeline. Par contre, nous allons conserver les dates précises des nœuds à l'aide d'une nouvelle structure de données, les XDD. Plus de détails sur les XDD sont donnés dans le chapitre 3 mais nous pouvons les voir comme une représentation exacte et efficace de la relation entre les temps d'exécution et les évènements, ce qui améliore la précision par rapport à la représentation du temps par des intervalles. Notre objectif, avec les XDD, a été d'obtenir un calcul temporel exact du pipeline avec les graphes d'exécution.

Il existe également une deuxième source d'imprécision avec les graphes d'exécution originaux : la surestimation du contexte. Le contexte temporel d'exécution d'un BB est l'effet temporel de l'exécution antérieure (des BB précédents) sur l'exécution du BB courant. Quand on traite chaque BB indépendamment, une façon classique de prendre en compte l'effet du contexte est de supposer un pire contexte juste avant chaque BB prédécesseur. Cependant, cela peut introduire de la surestimation pour deux raisons : (a) si on considère le pire contexte mais que le pire contexte arrive rarement, le temps d'exécution sera surestimé ; (b) comme le pire contexte n'est pas le contexte précis, lors du calcul de contention, le pire contexte amplifie l'imprécision dans le calcul de contention et ainsi augmente la surestimation. Autrement dit, il y aura des contentions qui ne sont pas possibles dans la réalité mais prises en compte dans le calcul du temps.

Pour calculer précisément le contexte d'exécution des BB, il est préférable de faire l'analyse de pipeline au niveau du CFG. Les algorithmes implantant l'interprétation abstraite fournissent une bonne base pour faire cela mais exigent de représenter les *états* complets du pipeline. Le modèle par graphe d'exécution est, comme son nom l'indique, sous forme de graphe et un graphe doit être construit pour chaque BB. Il est difficile de les convertir sous forme compatible avec l'analyse au niveau CFG ou sous forme supportée par l'interprétation abstraite.

2.7 Conclusion

Le problème du WCET étant un problème proposé depuis plus de 30 ans, beaucoup d'approches existent déjà. Mais le fossé entre le WCET réel et le WCET calculé est toujours grand. Les approches existantes ont toutes plus ou moins des problèmes de précision ou de performance, et se situent souvent aux deux extrêmes : précis mais peu performant, ou performant mais pas suffisamment précis.

Cette situation est engendrée principalement par quatre facteurs : le nombre très important de chemins d'exécution, les évènements, la structure complexe du pipeline et la présence d'anomalies temporelles qui résultent des facteurs précédents. L'ensemble de ces quatre facteurs impose que l'analyse de pipeline a seulement deux choix extrêmes lors de l'occurrence d'évènements : garder explicitement les états divergents – ce qui est de complexité combinatoire, ou couvrir l'effet des évènements avec une surestimation – ce qui cause généralement une perte de précision considérable, particulièrement en présence de ressources allouées dans le désordre.

Après avoir exploré l'état de l'art, nous avons décidé de concevoir l'analyse du pipeline selon deux idées essentielles. D'abord, nous allons nous baser sur les graphes d'exécution, c'est-à-dire modéliser le comportement temporel du pipeline à la granularité d'une instruction par étage (les nœuds du graphe). Comme discuté dans la section 2.6, nous pensons que cette représentation fournit une bonne abstraction de la sémantique temporelle.

Ensuite, nous imposons que notre analyse de pipeline garde des temps d'exécution précis car, dans le cas contraire, la latence causée par les contentions risque d'être largement surestimée en cas de présence de ressources allouées dans le désordre. L'approche originale par graphe d'exécution nous le démontre : l'imprécision sur le temps d'exécution oblige l'analyse de pipeline à considérer toutes les contentions **potentielles** ce qui aboutit à un temps d'exécution beaucoup plus grand que celui de l'exécution réelle. Par conséquent, nous choisissons de représenter les temps d'exécution explicitement mais nous allons utiliser les XDD pour limiter le problème de la complexité.

Chapitre 3

Diagramme de Décision d'Exécution – XDD

L'essentiel de la conception d'une analyse de pipeline performante est de trouver une bonne abstraction de l'ensemble des états concrets du pipeline – l'ensemble des informations nécessaires pour calculer le temps d'exécution. En présence des mécanismes de parallélisme au niveau des instructions machine (*instruction level parallelism*), l'état du pipeline à chaque instant lors de l'exécution d'une séquence d'instructions n'est pas une fonction temporelle linéaire, mais une fonction complexe de l'état précédent et des évènements de la séquence (qui peuvent être actifs ou inactifs selon le chemin d'exécution). Le problème majeur de l'analyse de pipeline est alors de représenter la relation entre l'état de pipeline et les évènements de manière efficace et précise.

Le coût de la précision est une augmentation de la complexité combinatoire induite par les évènements : pour une séquence d'instructions contenant $|\mathcal{E}|$ évènements, il y a $2^{|\mathcal{E}|}$ combinaisons possibles à considérer. Néanmoins, notre recherche préliminaire utilisant les graphes d'exécution pour calculer explicitement toutes les combinaisons d'évènements (qui, par conséquent, rend l'analyse peu performante) a montré que le nombre de temps d'exécution possibles est généralement beaucoup plus petit que $2^{|\mathcal{E}|}$.

Ce phénomène est à rapprocher des *fonctions booléennes* qui ont une propriété similaire : pour n variables booléennes en entrée (i.e. 2^n affectations possibles), seules deux valeurs de sorties sont possibles : TRUE ou FALSE. Cette propriété a donné lieu à des optimisations qui ont permis obtenir une représentation efficace des fonctions booléennes. L'une des plus connues est le *Diagramme de décision binaire* (BDD – Binary Descision Diagram) (Akers, 1978 ; Bryant, 1992).

En s'inspirant du principe des BDD, nous avons créé les XDD (eXecution Decision Diagram ou *diagrammes de décision d'exécution*) (Bai et al., 2020) pour représenter la fonction entre le temps d'exécution et les évènements. Un XDD peut être vu comme une généralisation du BDD telle que *le domaine de* sortie de la fonction booléenne n'est pas seulement $\{TRUE, FALSE\}$ mais un temps d'exécution (dans $\mathbb{Z}$).

Un XDD se présente comme une structure de donnée générique dont les propriétés principales sont exposées et discutées dans ce chapitre. Son utilisation dans l'analyse de pipeline est exposée dans les chapitres suivants.

Dans ce chapitre, la section 3.1 présente d'abord les BDD puis, dans la section 3.2, nous montrons qu'un XDD est une représentation canonique et sans perte de la relation entre les évènements et le temps d'exécution correspondant. Les opérateurs binaires entre les XDD sont présentés dans la section 3.3. Les optimisations dans l'implantation des XDD et ses opérateurs sont discutées dans la section 3.4. Finalement, la section 3.5 montre que le domaine des XDD peut former un demi-anneau commutatif. Par conséquent, il est possible de profiter des propriétés algébriques afférentes

pour définir des matrices et la multiplication matricielle dans le domaine des XDD. Comme cette dernière est associative (car le demi-anneau XDD est associatif), il est possible de résumer une séquence de code par une seule matrice qui, multipliée par un état entrant, permet d'obtenir l'état après toute une séquence d'instructions machine. Cette propriété peut être utilisée pour accélérer l'analyse du pipeline.

3.1 Diagramme de Décision Binaire

Les XDD sont inspirés des BDD (AKERS, 1978 ; BRYANT, 1992) et de ses variantes Algebraic Decision Diagram (ADD) (BAHAR et al., 1997) et Multi-Terminal BDD (MTBDD) (FUJITA et al., 1997). Un BDD représente une fonction booléenne ($\mathbb{B}^n \rightarrow \mathbb{B}$).

Une fonction booléenne peut avoir un nombre important d'entrées, mais une seule sortie qui a deux valeurs possibles. Par conséquent, une propriété importante des fonctions booléennes est qu'il existe des conditions suffisantes permettant à la valeur de certaines variables d'entrée de déterminer directement la sortie. Par exemple, pour la fonction $f \wedge a$, même si f est une fonction complexe, si $a = FALSE$, la valeur de toute la fonction est FALSE. De manière similaire pour $f \vee a$, si $a = TRUE$, le résultat est toujours $TRUE$.

Les BDD profitent de cette propriété pour représenter les fonctions booléennes de manière compacte. Tout d'abord, un BDD représente la fonction par un arbre de décision binaire dans lequel les nœuds représentent des variables de la fonction et les deux sous-arbres de chaque nœud correspondent, respectivement, aux cas où la variable est TRUE ou FALSE. Les feuilles sont les valeurs de sortie, donc dans $\mathbb{B}$. Ensuite, des règles de réduction sont appliquées pour réduire la taille de l'arbre : si les deux sous-arbres d'un nœud sont identiques, ce nœud est supprimé parce que la valeur de la variable correspondante n'a pas d'effet sur la sortie. De plus, un ordre sur les variables est ajouté pour garantir la canonicité de l'arbre. Un tel BDD réduit et ordonné est appelé un ROBDD (Reduced and Ordered BDD). Comme le ROBDD est la forme la plus utilisée, dans la suite de cette thèse, nous dirons simplement BDD pour le désigner.

La figure 3.1 montre deux BDD qui représentent $a \vee b \vee c$ (3.1A) et $a \wedge b \wedge c$ (3.1B). Dans le cas 3.1A, si une des variables vaut 1, toute la fonction vaut 1 indépendamment des autres variables. De manière similaire, dans le cas 3.1B, si une des variables vaut 0, toute la fonction vaut 0. Les deux BDD montrent bien qu'en profitant de la propriété des fonctions booléennes, la taille des BDD est fortement réduite : comparé à l'arbre binaire complet de $2^3 - 1 = 7$ nœuds, les BDD ont seulement 2 feuilles et 3 nœuds. La représentation compacte du BDD aide également à exprimer les opérateurs entre les fonctions booléennes de manière efficace parce que le parcours d'arbre est rapide.

(A) $a \vee b \vee c$ (B) $a \wedge b \wedge c$

FIGURE 3.1 – Deux exemples de BDD.

Le succès des BDD a permis de nombreuses applications et ses extensions sont utilisées dans des domaines variés. Les ADD (*Algebraic Descision Diagram*) et les MTBDD (*Multi-Terminal BDD*) sont des variantes similaires qui étendent le domaine des valeurs de sortie à un domaine plus large que $\mathbb{B}$. Ces deux variantes ont comme but de représenter des matrices et des opérations sur les matrices. Les XDD, présentés dans la suite de ce chapitre, suivent la même démarche mais sont appliquées au calcul du temps.

3.2 Représenter le temps avec des XDD

Un XDD a pour but de représenter la relation entre des évènements et les temps d'exécution correspondants. Formellement, un XDD est récursivement défini par :

Définition 3.2.1.

$$\begin{aligned} &\forall f^{\#} \in \mathcal{XDD}, \\ &f^{\#} = \text{LEAF}(k) \mid \text{NODE}(v, \overline{f^{\#}}, f^{\#}) \\ &\text{où } k \in \mathbb{Z}^{\infty} = \mathbb{Z} \cup \{+\infty, -\infty\}, \\ &v \in \mathbb{B} \text{ une variable booléenne,} \\ &\text{et } (f^{\#}, \overline{f^{\#}}) \in \mathcal{XDD}^2 \end{aligned} \tag{3.1}$$

Un XDD est un arbre qui peut être soit une feuille de valeur k – $\text{LEAF}(k)$, soit un nœud $\text{NODE}(v, \overline{f^{\#}}, f^{\#})$ avec une variable booléenne v, le sous-arbre où v est fausse – $\overline{f^{\#}}$ et le sous-arbre où v est vraie – $f^{\#}$. Dans cette section, les variables sont définies dans le domaine $\mathbb{B}$ sachant que les évènements ont exactement la même nature que des variables booléennes : les valeurs $VRAI$ et $FAUX$ de la variable booléenne correspondent à l'activation et l'inactivation de l'évènement. Dans la suite de ce chapitre, nous allons définir les variables sur le domaine $\mathbb{B}$ puisque les XDD forment une structure algébrique indépendante de leur domaine d'application. Nous pouvons également remarquer que le domaine des valeurs des feuilles peut facilement être étendu à $\mathbb{Z}^{\infty} = \mathbb{Z} \cup \{-\infty, +\infty\}$. Cela apporte des propriétés utiles au $\mathcal{XDD}$ qui seront exploitées dans la section 3.5.

Un XDD $f^{\#} \in \mathcal{XDD}$ peut être vu comme une fonction $f \in \mathbb{D} = \mathbb{B}^n \to \mathbb{Z}^{\infty}$ qui représente de manière explicite l'association entre les entrées et les sorties de $f^{\#}$ – sous forme de table d'association par exemple. f peut être vue comme la valeur *concrète* de $f^{\#}$ et va servir de pivot pour montrer que les optimisations de $f^{\#}$ préservent la fonction calculée. :

Définition 3.2.2. Un XDD peut être évalué avec une combinaison des valeurs booléennes en entrée, appelée une *configuration* γ :

$$\forall f^{\#} \in \mathcal{XDD}, \gamma \in \mathbb{B}^n, f^{\#}[\gamma] = \begin{cases} k & if\ f^{\#} = \text{LEAF}(k) \\ \overline{g^{\#}}[\gamma] & if\ f^{\#} = \text{NODE}(v, \overline{g^{\#}}, g^{\#}) \wedge \neg\, \gamma(v) \\ g^{\#}[\gamma] & if\ f^{\#} = \text{NODE}(v, \overline{g^{\#}}, g^{\#}) \wedge \gamma(v) \end{cases}$$

avec $\gamma(v)$ signifiant que la variable v est vraie dans la *configuration* γ.

Si un XDD $f^{\#}$ est bien construit par rapport à une fonction $f \in \mathbb{D}$, i.e. :

$$\forall \gamma \in \mathbb{B}^n, f^{\#}[\gamma] = f(\gamma) \tag{3.2}$$

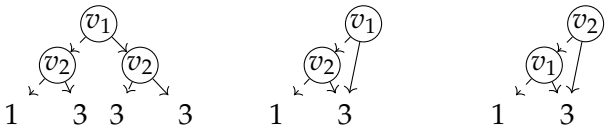
alors $f^\#$ et f sont équivalentes du point de vue de l'application. Par conséquent, nous pouvons utiliser des XDD pour représenter le temps $f^\#[\gamma]$ correspondant à chaque configuration γ.

3.2.1 Canonicité

Pour une fonction dans $\mathbb{D}$, il est toujours possible de trouver un XDD qui la représente correctement, c'est-à-dire qui est équivalent à cette fonction du point de vue de l'évaluation (équation 3.2). Néanmoins, un tel XDD n'est pas unique : autrement dit la bijection entre $\mathcal{XDD}$ et $\mathbb{D}$ n'est pas garantie. La figure 3.2A montre un exemple de fonction dans $\mathbb{D}$ et les figures 3.2B, 3.2C et 3.2D sont des XDD différents mais qui sont tous équivalents.

$\mathbb{B}^n$ valeur	$\mathbb{Z}^\infty$ image
v_1, v_2	3
$v_1, \overline{v_2}$	3
$\overline{v_1}, v_2$	3
$\overline{v_1}, \overline{v_2}$	1

(A)



(B) (C) (D)

FIGURE 3.2 – Des XDD non-canoniques

Ceci étant, pour minimiser la taille de l'arbre, il est préférable d'avoir les formes 3.2C ou 3.2D plutôt que 3.2B. Par conséquent, nous introduisons la première contrainte sur les XDD qui est fortement inspirée des règles de réduction utilisées pour réduire la taille des BDD.

Compacité Pour restreindre les XDD à une forme la plus compacte possible, la propriété de compacité (noté *Comp*) est définie par :

Définition 3.2.3. $\forall f^\# \in \mathcal{XDD}$, $Comp(f^\#) =$

$$\begin{cases} \top & if\ f^\# = \text{LEAF}(k) \\ (\overline{g^\#} \neq g^\#) \wedge Comp(\overline{g^\#}) \wedge Comp(g^\#) & if\ f^\# = \text{NODE}(v, \overline{g^\#}, g^\#) \end{cases}$$

Cette propriété interdit l'existence de nœuds dont les sous-arbres sont identiques car cela veut simplement dire que ce nœud n'a pas d'effet sur le choix des sous-arbres. Par exemple, le XDD de la Figure 3.2B n'est pas compact.

Le fait qu'un XDD soit compact ne suffit pas pour qu'il constitue une représentation unique de la fonction originale dans $\mathbb{D}$. Dans l'exemple de la Figure 3.2, le XDD 3.2C et le XDD 3.2D sont tous deux compacts et équivalents. Pour obtenir la canonicité, la même stratégie utilisée par les ROBDD est appliquée en contraignant l'ordre des variables dans l'arbre.

Ordre Étant donné un ordre total $\preceq$ sur l'ensemble des variables booléennes, la propriété d'ordre, notée $Ordered_\preceq$, respectant l'ordre $\preceq$, est définie par :

Définition 3.2.4. $\forall f^{\#} \in \mathcal{XDD}$, $Ordered_{\preceq}(f^{\#}) =$

$$\begin{cases} \top & if\ f^{\#} = \text{LEAF}(k) \\ (var(\overline{g^{\#}}) \preceq v) \wedge (var(g^{\#}) \preceq v) & \\ \wedge\ Ordered_{\preceq}(\overline{g^{\#}}) \wedge Ordered_{\preceq}(g^{\#}) & if\ f^{\#} = \text{NODE}(v, \overline{g^{\#}}, g^{\#}) \end{cases}$$

avec *var* qui extrait la variable d'un nœud, i.e. $var(\text{NODE}(v, \_, \_)) = v$ et $var(LEAF(\_))$ renvoie $\bot_{\preceq}$ qui est l'élément minimal dans l'ordre $\preceq$.

Cette propriété garantit que l'ordre est respecté dans les XDD : une variable d'ordre plus grand est plus proche de la racine de l'arbre. Par $\preceq$, les feuilles (avec la pseudo-variable $\bot_{\preceq}$) sont toujours inférieures à tous les nœuds parce qu'elles sont au plus loin de la racine.

Canonicité En combinant l'ordre et la compacité, et en considérant que les feuilles et nœuds identiques partagent la même instance[1], le XDD devient une représentation canonique de $\mathbb{D}$. Pour une fonction $f \in \mathbb{D}$ donnée, une seule représentation canonique $f^{\#} \in \mathcal{XDD}$ existe : un isomorphisme entre $\mathcal{XDD}$ et $\mathbb{D}$ peut être défini.

Définition 3.2.5.

$$\mathcal{XDD} \underset{\beta}{\overset{\alpha}{\leftrightarrows}} (\mathbb{B}^n \to \mathbb{Z}^{\infty})$$
$$\forall f^{\#} \in \mathcal{XDD}, \beta(f^{\#}) = \lambda\gamma.f^{\#}[\gamma]$$
$$\forall f \in \mathbb{D}, \alpha(f) = f^{\#} \mid \beta(f^{\#}) = f \wedge Ordered(f^{\#}) \wedge Comp(f^{\#}))$$

La fonction de concrétisation β convertissant un XDD en la représentation explicite consiste simplement à se servir du XDD comme une fonction avec la définition 3.2.2. La fonction d'abstraction α **définit** le XDD correspondant comme le XDD ordonné et compact qui s'évalue comme la représentation explicite, **sans préciser comment le $f^{\#}$ est calculé à partir de f**. Ce n'est pas problématique parce que ni la fonction d'abstraction, ni la fonction de concrétisation ne sont utilisées dans les calculs de l'analyse de pipeline. Leur présence ne sert qu'à prouver l'équivalence théorique entre $\mathbb{D}$ et $\mathcal{XDD}$. La construction des arbres de XDD est réalisée par les opérateurs binaires dans le domaine des $\mathcal{XDD}$ qui seront présentés dans la section 3.3.

Théorème 3.2.1. *α est une bijection entre $\mathbb{D}$ et $\mathcal{XDD}$:*

$$\forall f_1, f_2 \in \mathbb{D}^2, \alpha(f_1) = \alpha(f_2) \Leftrightarrow f_1 = f_2$$

Démonstration. Nous allons prouver l'équivalence dans les deux directions séparément, d'abord :

$$(a) \forall f_1, f_2 \in \mathbb{D}^2, \alpha(f_1) = \alpha(f_2) \Rightarrow f_1 = f_2$$

Comme α est simplement définie par le XDD $f^{\#}$ qui s'évalue de la même manière que f (mais ordonné et compact), cela revient à prouver que, **pour deux XDD qui sont identiques, l'évaluation est unique** :

$$\forall f_1^{\#}, f_2^{\#} \in \mathcal{XDD}^2, f_1^{\#} = f_2^{\#} \Rightarrow \forall \gamma \in \Gamma, f_1^{\#}[\gamma] = f_2^{\#}[\gamma]$$

1. Cette contrainte est surtout importante au niveau de l'implémentation. Mathématiquement, deux nœuds/feuilles identiques sont le même nœud.

ce qui peut être prouvé trivialement par la définition de l'évaluation de XDD (définition 3.2.2).

Ensuite, dans l'autre direction :

$$(b)\forall f_1, f_2 \in \mathbb{D}^2, f_1 = f_2 \Rightarrow \alpha(f_1) = \alpha(f_2)$$

D'après la définition de α, cela revient à prouver que **deux XDD qui s'évaluent de manière identique sont le même XDD :**

$$\forall f_1^\#, f_2^\# \in \mathcal{XDD}^2, (\forall \gamma \in \mathbb{B}^n, f_1^\#[\gamma] = f_2^\#[\gamma]) \Rightarrow f_1^\# = f_2^\#$$

Pour deux XDD, $f_1^\#$ et $f_2^\#$, nous allons analyser tous les cas possibles :

(b.i) Les deux XDD sont des feuilles : $f_1^\# = \text{LEAF}(k) \wedge f_2^\# = \text{LEAF}(l)$. Dans ce cas, il est trivial que $f_1^\#[\gamma] = f_2^\#[\gamma] \Rightarrow k = l$.

(b.ii) Les deux XDD sont des nœuds :

$$f_1^\# = \text{NODE}(v, \overline{g_1}, g_1) \wedge f_2^\# = \text{NODE}(w, \overline{g_2}, g_2)$$

(b.ii.1) Les deux sous-arbres des deux XDD sont deux feuilles :

$$f_1^\# = \text{NODE}(v, \text{LEAF}(k), \text{LEAF}(k')) \wedge f_2^\# = \text{NODE}(w, \text{LEAF}(l), \text{LEAF}(l'))$$

Dans ce cas, d'après la Définition 3.2.2, pour que $\forall \gamma \in \mathbb{B}^n, f_1^\#[\gamma] = f_2^\#[\gamma]$, on aura forcément $v = w, k = l, k' = l'$ et donc $f_1^\# = f_2^\#$.

(b.ii.2) Le sous-arbre droit est une feuille, le sous-arbre gauche est un nœud.

$$f_1^\# = \text{NODE}(v, \overline{g_1}, \text{LEAF}(k')) \wedge f_2^\# = \text{NODE}(w, \overline{g_2}, \text{LEAF}(l')) \wedge \overline{g_1} \neq \text{LEAF} \wedge \overline{g_2} \neq \text{LEAF}$$

Dans ce cas, nous supposons que $\overline{g_1}$ et $\overline{g_2}$ sont canoniques (hypothèse H_n de la preuve par récurrence). Par la propriété de l'ordre, toute variable dans $\overline{g_1}$ est d'ordre inférieur à celui de v et toute variable dans $\overline{g_2}$ est d'ordre inférieur à celui de w. Donc, pour que $\forall \gamma \in \mathbb{B}^n, f_1^\#[\gamma] = f_2^\#[\gamma]$, on aura forcément $\overline{g_1} = \overline{g_2}, k' = l'$ et donc $f_1^\# = f_2^\#$. Or, **(b.ii.1)** a montré que si $\overline{g_1}$ et $\overline{g_2}$ sont des feuilles, la propriété est vraie (H_0). Et comme un XDD est défini récursivement, par récurrence, **(b.ii.2)** est vraie.

(b.ii.3) Le sous-arbre gauche est une feuille, le sous-arbre droit est un nœud. La preuve est symétrique à **(b.ii.2)**.

(b.ii.4) Les deux sous-arbres sont deux nœuds :

$$f_1^\# = \text{NODE}(v, \overline{g_1}, g_1), f_2^\# = \text{NODE}(w, \overline{g_2}, g_2), \overline{g_1}, g_1, \overline{g_2}, g_2 \neq \text{LEAF}$$

D'après la propriété de compacité, $\overline{g_1} \neq g_1 \wedge \overline{g_2} \neq g_2$, en supposant que $\overline{g_1}, g_1, \overline{g_2}, g_2$ sont tous canoniques (hypothèse $H_{(n,n)}$), pour que $\forall \gamma \in \mathbb{B}^n, f_1^\#[\gamma] = f_2^\#[\gamma]$, on aura forcément $v = w$,. Donc, en combinant **(b.ii.2)** ($H_{(n+1,n)}$), **(b.ii.3)** ($H_{(n,n+1)}$) et **(b.ii.1)** ($H_{(0,0)}$), par double récursion sur les deux sous arbres, **(b.ii.4)** est vrai.

(b.ii.5) Les deux XDD n'ont pas la même structure, i.e. pas les mêmes nœuds ou feuilles. D'après la propriété de compacité, il ne peut pas y avoir de nœud dont les sous-arbres sont identiques, cela veut dire que le XDD NODE est forcément différent de LEAF. Donc ce cas est en effet impossible si $\forall \gamma \in \mathbb{B}^n, f_1^\#[\gamma] = f_2^\#[\gamma]$. □

Théorème 3.2.2. *α et β forment un isomorphisme entre $\mathbb{D}$ et $\mathcal{XDD}$.*

Démonstration. Nous avons déjà prouvé que α est une bijection de $\mathbb{D}$ à $\mathcal{XDD}$. Or, la fonction β est **par définition** la fonction réciproque de α. □

L'isomorphisme entre $\mathcal{XDD}$ est $\mathbb{D}$ assure que XDD est une représentation canonique de $\mathbb{D}$. La canonicité est obtenue par la compacité et l'ordre total des variables, notée :

$$Can \Leftrightarrow Ordered \wedge Comp$$

3.3 Opérateurs sur le domaine $\mathcal{XDD}$

Lors de l'analyse de pipeline, on veut appliquer des opérateurs binaires (typiquement, *max* et $+$) entre les temps. En considérant les évènements, ces opérateurs doivent être appliqués entre les temps **pour chaque configuration d'évènements**. Dans le domaine des XDD, il faut appliquer ces opérateurs entre les feuilles correspondant à chaque configuration.

Pour illustrer le fonctionnement de ces opérations, nous allons d'abord montrer un exemple dans le domaine $\mathbb{D}$. La figure 3.3 montre l'application de l'addition (notée $\boxtimes$ pour l'opérateur correspondant transféré dans le domaine $\mathbb{D}$)[2] entre deux fonctions dans le domaine $\mathbb{D}$.

a	b	1
a	$\overline{b}$	2
$\overline{a}$	b	3
$\overline{a}$	$\overline{b}$	4

(A) f_1

a	b	2
a	$\overline{b}$	3
$\overline{a}$	b	4
$\overline{a}$	$\overline{b}$	1

(B) f_2

a	b	$1+2=3$
a	$\overline{b}$	$2+3=5$
$\overline{a}$	b	$3+4=7$
$\overline{a}$	$\overline{b}$	$4+1=5$

(C) $f_1 \boxtimes f_2$

FIGURE 3.3 – Un exemple de l'addition dans le domaine $\mathbb{D}$.

Les deux opérandes f_1 et f_2 sont présentés à gauche de la figure. Le résultat de l'opération consiste simplement en l'addition de la valeur des opérandes pour chaque configuration.

Suivant ce principe, non seulement l'addition, mais n'importe quel opérateur binaire peut être appliquée dans le domaine $\mathbb{D}$ et également dans le domaine $\mathcal{XDD}$.

Définition 3.3.1. Soit $\bullet$ un opérateur binaire quelconque dans le domaine Z^∞ :

$$\bullet : \mathbb{Z}^\infty \times \mathbb{Z}^\infty \to \mathbb{Z}^\infty$$

Nous pouvons transférer cet opérateur dans le domaine $\mathbb{D}$, noté $\boxdot$ en appliquant $\bullet$ pour chaque configuration :

$$\boxdot : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$$
$$\forall f_1, f_2 \in \mathbb{D}^2, f_1 \boxdot f_2 = \lambda\gamma.f_1(\gamma) \bullet f_2(\gamma)$$

2. L'usage de l'opération $\boxtimes$ peut surprendre mais c'est une notation classique dans les demi-anneaux.

Cet opérateur peut également être transféré dans le domaine $\mathcal{XDD}$. En profitant de la structure arborescente du XDD et inspiré de l'implantation des opérateurs logiques des BDD, la définition 3.3.2 permet de produire l'algorithme ci-dessous pour l'opérateur $\bullet$ dans les XDD.

Définition 3.3.2. Soit l'opérateur $\odot$ qui est l'opérateur $\bullet$ transféré dans le domaine de $\mathcal{XDD}$,

$$\forall f_1^\#, f_2^\# \in \mathcal{XDD}, f_1^\# \odot f_2^\# =$$

$$\begin{cases} \text{LEAF}(k_1 \bullet k_2) & \text{if } f_1^\# = \text{LEAF}(k_1) \wedge f_2^\# = \text{LEAF}(k_2) \textbf{(a)} \\ g_1^\# \odot g_2^\# & \text{if } f_1^\# = \text{NODE}(v, \overline{g_1^\#}, g_1^\#) \\ & \wedge f_2^\# = \text{NODE}(v, \overline{g_2^\#}, g_2^\#) \\ & \wedge \overline{g_1^\#} \odot \overline{g_2^\#} = g_1^\# \odot g_2^\# \quad \textbf{(b)} \\ f_1^\# \odot \overline{g_2^\#} & \text{if } f_2^\# = \text{NODE}(v_2, \overline{g_2^\#}, g_2^\#) \\ & \wedge (var(f_1^\#) \preceq v_2) \\ & \wedge ((f_1^\# \odot \overline{g_2^\#}) = (f_1^\# \odot g_2^\#)) \quad \textbf{(c)} \\ \overline{g_1^\#} \odot f_2^\# & \text{if } f_1^\# = \text{NODE}(v_1, \overline{g_1^\#}, g_1^\#) \\ & \wedge (var(f_2^\#) \preceq v_1) \\ & \wedge ((\overline{g_1^\#} \odot f_2^\#) = (g_1^\# \odot f_2^\#)) \quad \textbf{(d)} \\ \text{NODE}(v, \overline{g_1^\#} \odot \overline{g_2^\#}, g_1^\# \odot g_2^\#) & \text{if } f_1^\# = \text{NODE}(v, \overline{g_1^\#}, g_1^\#) \\ & \wedge f_2^\# = \text{NODE}(v, \overline{g_2^\#}, g_2^\#) \quad \textbf{(e)} \\ \text{NODE}(v_2, f_1^\# \odot \overline{g_2^\#}, f_1^\# \odot g_2^\#) & \text{if } f_2^\# = \text{NODE}(v_2, \overline{g_2^\#}, g_2^\#) \\ & \wedge var(f_1^\#) \prec v_2 \quad \textbf{(f)} \\ \text{NODE}(v_1, f_2^\# \odot \overline{g_1^\#}, f_2^\# \odot g_1^\#) & \text{if } f_1^\# = \text{NODE}(v_1, \overline{g_1^\#}, g_1^\#) \\ & \wedge var(f_2^\#) \prec v_1 \quad \textbf{(g)} \end{cases}$$

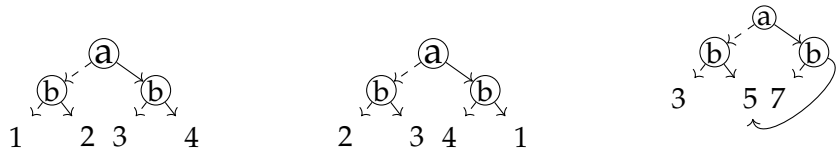
L'algorithme consiste à parcourir les deux XDD opérandes en parallèle et à appliquer l'opérateur aux sous-arbres qui correspondent à la même affectation de la variable de chaque nœud. Concrètement, si les deux XDD sont des feuilles (a), l'opérateur $\bullet$ est appliqué directement ; si les deux XDD sont des nœuds qui ont la même variable v, le calcul se propage de manière identique récursivement dans les deux sous-arbres des deux opérandes (b) et (e) (ce qui revient à affecter la même valeur à une variable commune). Si les résultats de l'opérateur des deux sous-arbres sont identiques, le nœud n'est pas construit (b) pour garantir la *compacité* du XDD résultant.

Si les nœuds courants ont des variables différentes, l'opérateur ne peut pas être appliqué et le calcul se propage en fonction de l'ordre des variables : l'opérande ayant la variable d'ordre le plus grand est maintenue et l'opérateur est appliqué aux sous-arbres de l'opérande ayant la variable d'ordre le plus petite pour garantir l'*ordre* des variables (définition 3.2.4) dans l'arbre résultat – cas (c), (d), (f) et (g).

L'exemple de la figure 3.3 est repris dans la figure 3.4 avec des XDD et nous permet de voir cet algorithme à l'œuvre.

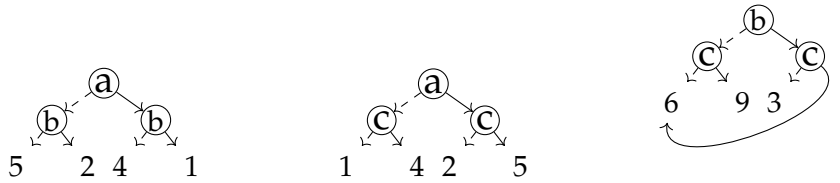
Il faut noter que certains nœuds peuvent paraître manquants mais, en fait, ils ont été supprimés par la propriété de compacité (définition 3.2.3). Par conséquent, certaines variables peuvent être présentes dans un opérande mais manquantes dans l'autre. La figure 3.5 montre un tel exemple.

Nous pouvons constater que la variable c est manquante dans le XDD $f_1^\#$ de la figure 3.5A. De manière similaire, la variable b est manquante dans la figure 3.5B.



(A) $f_1^{\#}$ (B) $f_2^{\#}$ (C) $f_1^{\#} \otimes f_2^{\#}$

FIGURE 3.4 – Un exemple d'application de l'opérateur $\otimes$ dans $\mathcal{XDD}$.



(A) $f_1^{\#}$ (B) $f_2^{\#}$ (C) $f_1^{\#} \otimes f_2^{\#}$

FIGURE 3.5 – Un exemple d'application de l'opérateur $\otimes$ dans $\mathcal{XDD}$.

Mais quand ces deux XDD sont additionnés, les variables manquantes sont considérées comme des nœuds supprimés grâce à la compacité. L'opération correspondante dans le domaine $\mathbb{D}$ est illustrée par la figure 3.6 où les variables manquantes sont entre parenthèses.

a	b	(c)	1
a	b	$(\overline{c})$	1
a	$\overline{b}$	(c)	4
a	$\overline{b}$	$(\overline{c})$	4
$\overline{a}$	b	(c)	2
$\overline{a}$	b	$(\overline{c})$	2
$\overline{a}$	$\overline{b}$	(c)	5
$\overline{a}$	$\overline{b}$	$(\overline{c})$	5

(A) f_1

a	(b)	c	5
a	$(\overline{b})$	c	5
a	(b)	$\overline{c}$	2
a	$(\overline{b})$	$\overline{c}$	2
$\overline{a}$	(b)	c	4
$\overline{a}$	$(\overline{b})$	c	4
$\overline{a}$	(b)	$\overline{c}$	1
$\overline{a}$	$(\overline{b})$	$\overline{c}$	1

(B) f_2

(a)	b	c	$1+5=6$
(a)	$\overline{b}$	c	$4+5=9$
(a)	b	$\overline{c}$	$1+2=3$
(a)	$\overline{b}$	$\overline{c}$	$4+2=6$
$(\overline{a})$	b	c	$2+4=6$
$(\overline{a})$	$\overline{b}$	c	$5+4=9$
$(\overline{a})$	b	$\overline{c}$	$2+1=3$
$(\overline{a})$	$\overline{b}$	$\overline{c}$	$5+1=6$

(C) $f_1 \boxtimes f_2$

FIGURE 3.6 – Un exemple de l'addition dans le domaine $\mathbb{D}$

En comparant la taille de la représentation explicite dans le domaine $\mathbb{D}$ (figure 3.6) et la représentation par XDD de la figure 3.5, nous pouvons constater que la représentation XDD est plus compacte parce que les XDD profitent de la compacité pour éliminer des nœuds qui n'ont pas d'impact sur leur valeur. Cet effet de compactage peut aussi apparaître dans l'application des opérateurs. Par exemple, dans le XDD résultat de la figure 3.6C, la variable commune des deux opérandes, a, a disparu. Mais nous pouvons confirmer qu'elle n'a effectivement pas d'impact dans le résultat à la figure 3.6C puisque les 4 premières lignes où a est active ont la même valeur que les 4 dernières lignes où a est inactive, c'est-à-dire que a n'a pas d'effet sur la valeur de sortie. Cependant, cette simplification est plus difficile à identifier dans la représentation explicite alors que le XDD est capable de la prendre en compte automatiquement et donc de réduire sa taille.

Canonicité du XDD produit

L'implémentation des opérateurs binaires entre les XDD garantie la canonicité du XDD produit car nous avons bien géré la compacité et l'ordre des variables à la construction du XDD produit. Formellement, cela signifie que, si les deux XDD opérandes sont sous forme canonique, le résultat de l'application de l'opérateur avec la définition 3.3.2 est également canonique :

Théorème 3.3.1. *un opérateur binaire $\odot$ quelconque défini selon 3.3.2 est compatible avec la canonicité :*

$$\forall f_1^\#, f_2^\# \in \mathcal{XDD}^2, Can(f_1^\#) \wedge Can(f_2^\#) \Rightarrow Can(f_1^\# \odot f_2^\#) \tag{3.3}$$

Démonstration. $Can(f_1^\# \odot f_2^\#) \Leftrightarrow Comp(f_1^\# \odot f_2^\#) \wedge Ordered(f_1^\# \odot f_2^\#)$.

(i) Compacité Dans le cas (b) de l'algorithme de la définition 3.3.2, les sous-arbres produits sont testés pour éviter de créer des arbres identiques quand les nœuds sont créés dans l'appel récursif dans (e). Suivant le même principe, (c) assure que le (f) ne crée pas de sous-arbres identiques. On fait de même pour (d) et (g). Donc, les trois règles qui créent de nouveaux nœuds ne peuvent pas produire de nœud avec des sous-arbres identiques.

(ii) Ordre L'ordre des variables des opérandes est toujours testé. Et la construction des nœuds est cohérente par rapport à l'ordre des variables d'après les conditions de (e), (f) et (g).

Comme $f_1^\# \odot f_2^\#$ est compact et ordonné, il est canonique. □

Validité des opérateurs

Pour utiliser les opérateurs dans le domaine $\mathcal{XDD}$, il est aussi impératif de prouver qu'il y a équivalence avec l'application de la fonction binaire dans le domaine $\mathbb{D}$:

Théorème 3.3.2. *Soit un opérateur binaire $\bullet : \mathbb{Z}^\infty \times \mathbb{Z}^\infty \to \mathbb{Z}^\infty$, l'opérateur correspondant dans $\mathbb{D}$, $\boxdot : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ et l'opérateur correspondant dans $\mathcal{XDD}$, $\odot : \mathcal{XDD} \times \mathcal{XDD} \to \mathcal{XDD}$. L'application de l'opérateur $\odot$ donne un résultat pour les XDD équivalent à $\boxdot$ pour $\mathbb{D}$. Formellement :*

$$\forall f_1, f_2 \in \mathbb{D}^2, \forall \gamma \in \mathbb{B}^n, (\alpha(f_1) \odot \alpha(f_2))[\gamma] = (f_1 \boxdot f_2)(\gamma) \tag{3.4}$$

Pour simplifier la preuve, nous pouvons remplacer $\alpha(f)$ par $f^\#$, ce qui donne :

$$\begin{aligned} \forall f_1, f_2 \in \mathbb{D}^2, f_1^\#, f_2^\# \in \mathcal{XDD}^2, f_1^\# = \alpha(f_1), f_2^\# = \alpha(f_2), \\ \forall \gamma \in \mathbb{B}^n, (f_1^\# \odot f_2^\#)[\gamma] = (f_1 \boxdot f_2)(\gamma) \end{aligned} \tag{3.5}$$

Par la définition de $\boxdot$, cela consiste simplement à appliquer l'opérateur $\bullet$ pour chaque configuration (définition 3.3.1). Nous pouvons donc réécrire la formule à prouver en :

$$\begin{aligned} \forall f_1, f_2 \in \mathbb{D}^2, f_1^\#, f_2^\# \in \mathcal{XDD}^2, f_1^\# = \alpha(f_1), f_2^\# = \alpha(f_2), \\ \forall \gamma \in \mathbb{B}^n, (f_1^\# \odot f_2^\#)[\gamma] = f_1(\gamma) \bullet f_2(\gamma) \end{aligned} \tag{3.6}$$

Par la définition de α (définition 3.2.5), f_1, f_2 s'évaluent exactement comme $f_1^\#$ et $f_2^\#$, nous pouvons donc remplacer $f_1(\gamma)$ par $f_1^\#[\gamma]$ et $f_2(\gamma)$ par $f_2^\#[\gamma]$:

$$\forall f_1^\#, f_2^\# \in \mathcal{XDD}, \forall \gamma \in \mathbb{B}^n, (f_1^\# \odot f_2^\#)[\gamma] = f_1^\#[\gamma] \bullet f_2^\#[\gamma] \tag{3.7}$$

Intuitivement, cette dernière formule à prouver signifie : appliquer une opération binaire entre deux XDD est équivalent à appliquer indépendamment l'opérateur correspondant $\bullet$ aux feuilles correspondant à chaque configuration. Comme les XDD s'évaluent exactement comme les $f \in \mathbb{D}$ (isomorphisme entre $\mathcal{XDD}$ et $\mathbb{D}$), le XDD résultant représentent exactement le résultat de l'opération $\boxdot$ dans $\mathbb{D}$ (la formule initiale).

Finalement, ce que nous allons prouver dans la suite est : pour deux XDD opérandes canoniques, ***l'algorithme d'évaluation de*** $\odot$ ***réalise effectivement*** $\bullet$ *entre les feuilles correspondantes des deux opérandes pour chaque configuration.*

Démonstration. La preuve de ce théorème est obtenue par induction sur la structure du XDD, des feuilles jusqu'à la racine. Comme un XDD est implémenté comme un graphe orienté acyclique, un nœud peut avoir plusieurs prédécesseurs et l'induction consiste à trouver en arrière le prédécesseur correspondant au chemin induit par chaque configuration γ des valeurs des variables booléennes.

Soit $\pi^{\gamma}_{g^{\#}}(f^{\#})$ la fonction qui renvoie le prédécesseur du sous-arbre $g^{\#}$ dans un XDD $f^{\#}$ sur le chemin induit par la configuration γ, ou $\perp$ si $g^{\#}$ n'est pas sur le chemin qui satisfait γ dans $f^{\#}$. Formellement, il est défini par :

Définition 3.3.3. $\forall f^{\#}, g^{\#} \in \mathcal{XDD}, \forall \gamma \in \mathbb{B}^n$

$$\pi^{\gamma}_{g^{\#}}(f^{\#}) = \begin{cases} f^{\#} & \text{if } (f^{\#} = \text{NODE}(v, g^{\#}, \_) \wedge \neg\gamma(v)) \\ & \vee\ (f^{\#} = \text{NODE}(v, \_, g^{\#}) \wedge \gamma(v)) \\ & \vee\ f^{\#} = g^{\#} \\ \pi^{\gamma}_{g^{\#}}(\overline{h}) & \text{if } f^{\#} = \text{NODE}(v, \overline{h}, h) \wedge \neg\gamma(v) \\ \pi^{\gamma}_{g^{\#}}(h) & \text{if } f^{\#} = \text{NODE}(v, \overline{h}, h) \wedge \gamma(v) \\ \perp & \text{else} \end{cases}$$

Nous allons profiter de l'opérateur π pour conduire la preuve du théorème 3.3.2 par récurrence :

D'abord, le *cas initial (H_0)* – les deux XDD opérandes sont des feuilles : $f_1^{\#} = \text{LEAF}(k_1)$ et $f_2^{\#} = \text{LEAF}(k_2)$:

$$\begin{aligned} \forall \gamma \in \mathbb{B}^n, (f_1^{\#} \odot f_2^{\#})[\gamma] &= k_1 \bullet k_2 \\ &= f_1^{\#}[\gamma] \bullet f_2^{\#}[\gamma] \end{aligned}$$

Ensuite, le *cas d'induction (H_n) :* Soit $g_1^{\#}$ et $g_2^{\#}$ respectivement les XDD sous-arbres de $f_1^{\#}$ et $f_2^{\#}$ sur le chemin induit par γ. Supposons donc que $(g_1^{\#} \odot g_2^{\#})[\gamma] = g_1^{\#}[\gamma] \bullet g_2^{\#}[\gamma]$. La preuve est terminée si $g_1^{\#} = f_1^{\#}$ et $g_2^{\#} = f_2^{\#}$. Sinon, nous procédons par induction. À l'exception du cas initial, $\pi^{\gamma}_{g^{\#}}(f^{\#})$ renvoie finalement toujours un nœud $\text{NODE}(v, g^{\#}, \_)$ si $\gamma(v)$, $\text{NODE}(v, \_, g^{\#})$ sinon. Dans la suite, nous allons discuter de 6 cas possibles : cas 1 et 2 où la variable du nœud prédécesseur de $g_1^{\#}$ et $g_2^{\#}$ est le même et cas 1 si cette variable est inactive dans la configuration γ et cas 2 si elle est active dans γ ; cas 3 et 4 où la variable du nœud prédécesseur de $g_1^{\#}$ est inférieur à celle de $g_2^{\#}$, le cas 3 où la variable de $g_2^{\#}$ est inactive et le cas 3 où elle est inactive. Les cas 5 et 6 sont similaires aux cas 3 et 4 mais avec la variable du nœud prédécesseur de $g_1^{\#}$ supérieure à celle du $g_2^{\#}$.

1. Si $var(\pi^{\gamma}_{g_1^{\#}}(f_1^{\#})) = var(\pi^{\gamma}_{g_2^{\#}}(f_2^{\#})) = v \wedge \neg\gamma(v)$ alors

$$\begin{aligned} (\pi^{\gamma}_{g_1^{\#}}(f_1^{\#}) \odot \pi^{\gamma}_{g_2^{\#}}(f_2^{\#}))[\gamma] &= (\text{NODE}(v, g_1^{\#}, \_) \odot \text{NODE}(v, g_2^{\#}, \_))[\gamma] \\ &= \text{NODE}(v, g_1^{\#} \odot g_2^{\#}, \_)[\gamma] \\ &= (g_1^{\#} \odot g_2^{\#})[\gamma] \\ &= g_1^{\#}[\gamma] \bullet g_2^{\#}[\gamma] \\ &= \pi^{\gamma}_{g_1^{\#}}(f_1^{\#})[\gamma] \bullet \pi^{\gamma}_{g_2^{\#}}(f_2^{\#})[\gamma] \end{aligned}$$

2. Si $var(\pi^{\gamma}_{g_1^\#}(f_1^\#)) = var(\pi^{\gamma}_{g_2^\#}(f_2^\#)) = v \wedge \gamma(v)$: similaire au cas (1) mais en utilisant le sous-arbre droit à la place de $g_1^\#$ et $g_2^\#$
3. Si $var(\pi^{\gamma}_{g_1^\#}(f_1^\#)) \preceq var(\pi^{\gamma}_{g_2^\#}(f_2^\#)) \wedge \neg\gamma(var(\pi^{\gamma}_{g_2^\#}(f_2^\#)))$, alors

$$\pi^{\gamma}_{g_2^\#}(f_2^\#) = \text{NODE}(e_2, g_{2,}^\#, \_)$$

donc,

$$\begin{aligned}(\pi^{\gamma}_{g_1^\#}(f_1^\#) \odot \pi^{\gamma}_{g_2^\#}(f_2^\#))[\gamma] &= (\pi^{\gamma}_{g_1^\#}(f_1^\#) \odot \text{NODE}(e_2, g_2^\#, h))[\gamma] \\ &= \text{NODE}(v, \pi^{\gamma}_{g_1^\#}(f_1^\#) \odot g_2, \pi^{\gamma}_{g_1^\#}(f_1^\#) \odot h)[\gamma] \\ &= (\pi^{\gamma}_{g_1^\#}(f_1^\#) \odot g_2^\#)[\gamma] \\ &= \pi^{\gamma}_{g_1^\#}(f_1^\#)[\gamma] \bullet g_2^\#[\gamma] \\ &= \pi^{\gamma}_{g_1^\#}(f_1^\#)[\gamma] \bullet \pi^{\gamma}_{g_2^\#}(f_2^\#)[\gamma]\end{aligned}$$

4. Si $var(\pi^{\gamma}_{g_1^\#}(f_1^\#)) \preceq var(\pi^{\gamma}_{g_2^\#}(f_2^\#)) \wedge \gamma(var(\pi^{\gamma}_{g_2^\#}(f_2^\#)))$: similaire au cas (3) mais en utilisant le sous-arbre droit.
5. (et 6.) Sinon $var(\pi^{\gamma}_{g_1^\#}(f_1^\#)) \succeq var(\pi^{\gamma}_{g_2^\#}(f_2^\#))$: similaire à (3) et (4) mais en échangeant $g_1^\#$ et $g_2^\#$ (trivial si $\odot$ et $\bullet$ sont commutatifs).

Grâce aux six cas d'induction ci-dessus, nous prouvons que, pour une configuration γ quelconque, si un sous-arbre XDD est valide, alors son arbre parent (i.e. l'arbre de niveau supérieur, dont la racine est son prédécesseur) est aussi valide. Donc par récurrence, des feuilles jusqu'à la racine, l'équivalence est valide pour le XDD complet. □

Lors de l'utilisation des XDD dans l'analyse du pipeline, les XDD sont construits à partir des feuilles et des évènements sous forme de $\text{NODE}(e, \text{LEAF}(t_1), \text{LEAF}(t_2))$ où e est l'évènement, t_1 est la latence en cas d'inactivation et t_2 la latence en cas d'activation. Comme prouvé par les théorèmes 3.3.1 et 3.3.2, les XDD construits à partir des opérateurs binaires sont canoniques. Donc, tout élément canonique du domaine $\mathcal{XDD}$ peut être généré par des opérations binaires entre des feuilles et des évènements. Comme nous utilisons uniquement les XDD canoniques, dans la suite, la notation $f \in \mathcal{XDD}$ sous-entend que f est canonique.

Dans cette section, nous avons démontré qu'un XDD est équivalent à la représentation explicite, i.e. domaine $\mathbb{D} : \mathbb{B}^n \to \mathbb{Z}^\infty$, et que n'importe quel opérateur binaire peut être appliqué dans le domaine $\mathcal{XDD}$ comme dans le domaine $\mathbb{D}$. Il faut seulement voir un XDD comme une compression sans perte et sous forme graphique de $\mathbb{D}$, ce qui permet d'utiliser les XDD pour représenter des temps d'exécution dans l'analyse de pipeline. Par conséquent, on peut facilement transférer les opérateurs les plus utilisés dans l'analyse de pipeline par graphe d'exécution, $max, min, +$ et $-$, dans le domaine des XDD (et dans le domaine $\mathbb{D}$). Leur correspondance est listée dans la table 3.1.

$\mathbb{Z}^\infty$	$\mathbb{D}$	$\mathcal{XDD}$
max	$\boxplus$	$\oplus$
min	$\boxminus$	$\ominus$
$+$	$\boxtimes$	$\otimes$
$-$	$\boxslash$	$\oslash$

TABLE 3.1 – Correspondance des opérateurs dans les différents domaines.

3.4 Optimisations des XDD

3.4.1 Mémoïsation

La mémoïsation est une technique d'optimisation dans laquelle on stocke des valeurs déjà calculées pour pouvoir les réutiliser si les mêmes calculs sont demandés. La mémoïsation est la clé pour obtenir une bonne performance avec les XDD. Deux types de mémoïsation sont utilisés :

— Une *table d'unicité* qui stocke les instances des sous-arbres (nœuds ou feuilles) des XDD pour garantir l'unicité des sous-arbres. Cette table est aussi utilisée dans la plupart des implantations de BDD (BRACE et al., 1991 ; STORNETTA et al., 1996).

— Une *table d'opération* par opérateur qui stocke les résultats de l'opérateur appliqué entre les XDD (y compris entre les sous-arbres XDD lors des appels récursifs). Cette méthode est inspirée de la technique de *hash consing* utilisée dans les langages fonctionnels comme *Lisp* (GOTO, 1974).

La *table d'unicité* est implantée avec une table de hachage qui stocke les nœuds et feuilles créées. Dans la table, chaque nœud est identifié (i.e. la clé des éléments dans la table) avec le triplet (v, l, r) – la variable v, la référence (pointeur) des sous-arbres gauches l et droits r. Les feuilles sont identifiées directement par leur valeur. La valeur renvoyée par la table pour une clé donnée est la référence vers le nœud de XDD correspondant. Autrement dit, la table garde un tableau associatif de l'instance des nœuds/feuilles vers leur référence. La relation réciproque est la référence elle-même, i.e. déréférencer une référence donne son instance.

Lors de la création d'un sous-arbre XDD (nœud ou feuille), il est donné à la table d'unicité et la table renvoie la référence du XDD s'il est déjà enregistré dans la table. Sinon, le XDD est ajouté à la table et la référence est renvoyée.

La *table d'opération* est aussi implantée avec une table de hachage qui enregistre la relation entre le tuple des opérandes et le XDD résultant. Si l'opérateur est commutatif (ce qui est le cas de $\oplus$ et $\otimes$), l'ordre entre les opérandes est ignoré. Comme l'unicité des nœuds et des feuilles est garantie par la *table d'unicité*, le tuple contient simplement les références des opérandes. Comme décrit dans la définition 3.3.2, les opérateurs dans le domaine $\mathcal{XDD}$ sont appliqués récursivement sur les sous-arbres des opérandes. Durant ce calcul, les XDD résultant des sous-arbres sont tous mémorisés dans la *table d'opération* et un test est effectué à chaque appel récursif pour profiter des calculs déjà faits. D'après nos observations dans l'analyse du pipeline, les opérateurs effectués profitent systématiquement de la réutilisation des résultats existants et par conséquent, cela permet de considérablement accélérer l'analyse.

3.4.2 Ordre des variables

Pour supporter la canonicité dans $\mathcal{XDD}$, un ordre total doit être défini sur l'ensemble des variables. Néanmoins, un tel ordre n'est pas unique. Les BDD ont le

même problème et il est déjà prouvé que trouver un tel ordre optimal pour lequel la taille des BDD serait minimale est un problème très complexe (MEINEL et al., 1994). Heureusement, nous avons trouvé une heuristique permettant d'ordonner facilement les variables dans l'analyse du pipeline. Comme les variables sont en fait des évènements, l'ordre des évènements/variables est défini de manière croissante selon l'ordre d'avancement de l'analyse. Les évènements dans les premiers BB sont plus proches des feuilles des XDD (c'est-à-dire d'ordre plus petit) que les évènements dans les BB suivants. Les évènements au début de chaque BB ont une grande chance d'être plus proches des feuilles des XDD (c'est-à-dire d'ordre plus petit) que les évènements des instructions suivantes dans le même BB. Nous avons pu observer que cet ordre permet de bonnes performances dans le calcul réalisé par les opérateurs. Par exemple, une opération très fréquente dans l'analyse de pipeline est $f \otimes \text{NODE}(v, \text{LEAF}(0), \text{LEAF}(k_e))$: cela permet d'ajouter un évènement dans le temps d'exécution avec f un grand XDD. Si v est d'ordre supérieur à toutes les variables de f, le résultat est $\text{NODE}(v, f, f \otimes \text{LEAF}(k_e))$: f est réutilisé directement comme le sous-arbre gauche du XDD résultant. Dans le cas contraire (sans appliquer cette heuristique pour ordonner les évènements), v est d'ordre inférieur à toutes les variables du f, le sous-arbre gauche du résultat doit être calculé en insérant v tout au fond (vers la racine) de f, ce qui provoque la reconstruction complète de f. Donc, dans ce genre de cas, assigner à v un ordre supérieur (proche de la racine) économise presque la moitié des calculs.

3.4.3 Optimisation des opérateurs sur $\mathcal{XDD}$

Un autre avantage des XDD est leur flexibilité pour concevoir des optimisations pour les opérateurs binaires grâce à leur structure arborescente. Par exemple, pour le $\oplus$ (l'opérateur maximum), nous pouvons profiter de la nature du *max* pour optimiser l'opérateur sur $\mathcal{XDD}$:

Théorème 3.4.1.

$$\forall f_1^\#, f_2^\# \in \mathcal{XDD}, f_1^\# \oplus f_2^\# = f_1^\# \Leftrightarrow \forall \gamma \in \mathbb{B}^n, f_1^\#[\gamma] \geq f_2^\#[\gamma]$$

Si la valeur de $f_1^\#$ est plus grande que celle de $f_2^\#$ dans chaque configuration, le maximum entre $f_1^\#$ et $f_2^\#$ est simplement $f_1^\#$. Cette condition est toujours compliquée à évaluer car le calcul doit vérifier toutes les feuilles en correspondance. Une condition plus faible mais également plus simple à évaluer est :

Théorème 3.4.2.

$$\forall f_1^\#, f_2^\# \in \mathcal{XDD}, f_1^\# \oplus f_2^\# = f_1^\# \iff min^*(f_1^\#) > max^*(f_2^\#).$$

où min^* et max^* donne respectivement la valeur minimale, maximale, de toutes les feuilles d'un XDD.

Les fonctions min^* et max^* sont définies dans le domaine de $\mathcal{XDD}$ par :

Définition 3.4.1.

$$min^* : \mathcal{XDD} \to \mathbb{Z}^\infty, min^*(f^\#) = \begin{cases} min(min^*(\overline{g^\#}), min^*(g^\#)) \text{ si } f^\# = \text{NODE}(\_, \overline{g^\#}, g^\#) \\ k \text{ if } f^\# = \text{LEAF}(k) \end{cases}$$

La fonction max^* peut être déduite de la même manière en changeant tous les min par max et min^* par max^*. Néanmoins, la définition récursive de ces fonctions exige de parcourir tout l'arbre d'un XDD. Pour éviter la complexité d'un tel parcours, les informations de valeurs minimales et maximales peuvent être directement embarquées dans les nœuds lors de leur construction. Formellement, cela correspond à étendre la définition de $\mathcal{XDD}$ par :

Définition 3.4.2.

$$\forall f^\# \in \mathcal{XDD}^\oplus, f^\# = \text{LEAF}(k) | \text{NODE}(v, \overline{f^\#}, f^\#, min, max)$$

Avec la nouvelle définition, nous pouvons définir les nouvelles fonctions $min^\#$ et $max^\#$ qui font la même chose que min^* et max^*.

$$min^\# : \mathcal{XDD}^\oplus \to \mathbb{Z}^\infty, min^\#(f^\#) = \begin{cases} min \text{ si } f^\# = \text{NODE}(\_, \_, \_, min, max) \\ k \text{ si } f^\# = \text{LEAF}(k) \end{cases}$$

$$max^\# : \mathcal{XDD}^\oplus \to \mathbb{Z}^\infty, max^\#(f^\#) = \begin{cases} max \text{ si } f^\# = \text{NODE}(\_, \_, \_, min, max) \\ k \text{ si } f^\# = \text{LEAF}(k) \end{cases}$$

Avec ces informations supplémentaires, le calcul de l'opérateur $\oplus$ peut s'arrêter dès que la valeur minimale des feuilles d'un opérande est plus grande que la valeur maximale des feuilles de l'autre opérande. De plus, comme ces informations sont embarquées dans les nœuds, la condition d'arrêt peut être vérifiée tout le long du calcul du $\oplus$. Une version optimisée du $\oplus$ est proposée dans la définition 3.4.3.

Pour faciliter la comparaison avec l'algorithme original, le nom des règles est le même que dans la définition 3.3.2. La règle (a), qui permet d'arrêter le calcul si la condition du théorème 3.4.2 est satisfaite, est remplacée par (a0) et (a1). La construction des feuilles est automatiquement couverte par (a0) et (a1) car le min/max des feuilles est la valeur des feuilles elles-mêmes. Les règles (b), (c) et (d) sont les mêmes. Les règles (e), (f) et (g) sont étendues en calculant la valeur min/max des nouveaux sous-arbres.

Cette optimisation améliore la performance de l'opérateur $\oplus$ mais elle a aussi un sur-coût en termes d'espace mémoire pour stocker les valeurs min/max dans les nœuds. Mais le problème du temps d'analyse étant plus important que le problème de la place mémoire, cette optimisation est en général utile dans l'analyse de pipeline.

Définition 3.4.3.

$$\forall f_1^\#, f_2^\# \in \mathcal{XDD}, f_1^\# \oplus f_2^\# =$$

$$\begin{cases}
f_1^\# & \text{if } min^\#(f_1^\#) \geq max\#(f_2^\#) \textbf{ (a0)} \\
f_2^\# & \text{if } max^\#(f_1^\#) \leq min\#(f_2^\#) \textbf{ (a1)} \\
g_1^\# \oplus g_2^\# & \text{if } f_1^\# = \text{NODE}(v, \overline{g_1^\#}, g_1^\#) \\
 & \wedge f_2^\# = \text{NODE}(v, \overline{g_2^\#}, g_2^\#) \\
 & \wedge \overline{g_1^\#} \oplus \overline{g_2^\#} = g_1^\# \oplus g_2^\# \textbf{ (b)} \\
f_1^\# \oplus \overline{g_2^\#} & \text{if } f_2^\# = \text{NODE}(v_2, \overline{g_2^\#}, g_2^\#) \\
 & \wedge (var(f_1^\#) \preceq v_2) \\
 & \wedge ((f_1^\# \oplus \overline{g_2^\#}) = (f_1^\# \oplus g_2^\#)) \textbf{ (c)} \\
\overline{g_1^\#} \oplus f_2^\# & \text{if } f_1^\# = \text{NODE}(v_1, \overline{g_1^\#}, g_1^\#) \\
 & \wedge (var(f_2^\#) \preceq v_1) \\
 & \wedge ((\overline{g_1^\#} \oplus f_2^\#) = (g_1^\# \oplus f_2^\#)) \textbf{ (d)} \\
\text{NODE}(v, \overline{g_1^\# \oplus \overline{g_2^\#}}, g_1^\# \oplus g_2^\#, & \text{if } f_1^\# = \text{NODE}(v, \overline{g_1^\#}, g_1^\#) \\
min(min^\#(\overline{g_1^\# \oplus \overline{g_2^\#}}), min^\#(g_1^\# \oplus g_2^\#)), & \wedge f_2^\# = \text{NODE}(v, \overline{g_2^\#}, g_2^\#) \textbf{ (v)} \\
max(max^\#(\overline{g_1^\# \oplus \overline{g_2^\#}}), max^\#(g_1^\# \oplus g_2^\#)) & \\
\text{NODE}(v_2, f_1^\# \oplus \overline{g_2^\#}, f_1^\# \oplus g_2^\#, & \text{if } f_2^\# = \text{NODE}(v_2, \overline{g_2^\#}, g_2^\#) \\
min(min^\#(f_1^\# \oplus \overline{g_2^\#}), min^\#(f_1^\# \oplus g_2^\#)), & \wedge var(f_1^\#) \prec v_2 \textbf{ (f)} \\
max(max^\#(f_1^\# \oplus \overline{g_2^\#}), max^\#(f_1^\# \oplus g_2^\#))) & \\
\text{NODE}(v_1, f_2^\# \oplus \overline{g_1^\#}, f_2^\# \oplus g_1^\#, & \text{if } f_1^\# = \text{NODE}(v_1, \overline{g_1^\#}, g_1^\#) \\
min(min^\#(f_2^\# \oplus \overline{g_1^\#}), min^\#(f_2^\# \oplus g_1^\#)), & \wedge var(f_2^\#) \prec v_1 \textbf{ (g)} \\
max(max^\#(f_2^\# \oplus \overline{g_1^\#}), max^\#(f_2^\# \oplus g_1^\#))) &
\end{cases}$$

3.5 Algèbre linéaire sur le demi-anneau $\mathcal{XDD}$

Dans cette section, nous allons discuter des propriétés algébriques du domaine $\mathcal{XDD}$ en se focalisant spécialement sur les opérateurs utilisés dans l'analyse de pipeline : $\oplus$, $\ominus$, $\otimes$ et $\oslash$. La correspondance de ces opérateurs entre les domaines $\mathbb{Z}^\infty$, $\mathbb{D}$ et $\mathcal{XDD}$ est donnée dans la Table 3.1. La définition de ces opérateurs n'a pas besoin d'être présentée explicitement car ce sont juste des cas particuliers de la définition générale des opérateurs binaires (3.3.1 et 3.3.2).

En fait, les propriétés algébriques du domaine $\mathcal{XDD}$ héritent essentiellement des propriétés du domaine $\mathbb{Z}^\infty$.

Lemme 3.5.1. *La structure $\langle \mathbb{Z}^\infty, max, +, -\infty, 0\rangle$ est un demi-anneau commutatif.*

Démonstration. (a) $\langle \mathbb{Z}^\infty, max, -\infty\rangle$ est un monoïde commutatif avec l'élément neutre ∞ ;

(b) $\langle \mathbb{Z}^\infty, +, 0\rangle$ est un monoïde commutatif avec l'élément neutre 0.

En fait, $\langle \mathbb{Z}^\infty, max, +, -\infty, 0\rangle$ est un *demi-anneau tropical* (COHEN et al., 1999 ; HEIDERGOTT et al., 2006) où max est l'addition et $+$ la multiplication. □

Ensuite, nous pouvons voir que ce demi-anneau existe également sur le domaine de la représentation explicite de la relation $\mathbb{D} : \mathbb{B}^n \to \mathbb{Z}^\infty$:

Lemme 3.5.2. *$\langle \mathbb{D}, \boxplus, \boxtimes, \mathbb{0}_\boxplus, \mathbb{1}_\boxtimes\rangle$ est un demi-anneau commutatif où $\boxplus$ est le max transféré dans $\mathbb{D}$ et $\boxtimes$ l'addition transférée dans $\mathbb{D}$.*

Démonstration. Pour $\langle \mathbb{D}, \boxplus, \boxtimes, \mathbb{0}_\boxplus, \mathbb{1}_\boxtimes\rangle$:

$\langle \mathbb{D}, \boxplus, \mathbb{0}_\boxplus\rangle$ est un monoïde commutatif :

$\boxplus$ est associatif :

$$\begin{aligned}
&\forall d_1, d_2, d_3 \in \mathbb{D}^3, (d_1 \boxplus d_2) \boxplus d_3 \\
&= \lambda\gamma.max(max(d_1, d_2), d_3) \\
&= \lambda\gamma.max(d_1, max(d_2, d_3)) \\
&\stackrel{\text{def}}{=} d_1 \boxplus (d_2 \boxplus d_3)
\end{aligned}$$

et est équipé de l'élément neutre $\mathbb{0}_\boxplus = \lambda\gamma. -\infty$.

En fait, les opérateurs dans le domaine $\mathbb{D}$ héritent des propriétés des opérateurs sur $\mathbb{Z}^\infty$ car, par définition (3.3.1), les opérateurs binaires sur $\mathbb{D}$ appliquent simplement les opérateurs correspondant dans $\mathbb{Z}^\infty$ pour chaque configuration. Or, $\langle \mathbb{Z}^\infty, +, \mathbb{1}_+\rangle$ est un monoïde commutatif, donc $\langle \mathbb{D}, \boxtimes, \mathbb{1}_\boxtimes\rangle$ est également un monoïde commutatif avec $\mathbb{1}_\boxtimes = \lambda\gamma.0$.

Finalement, $\langle \mathbb{D}, \boxplus, \boxtimes, \mathbb{0}_\boxplus, \mathbb{1}_\boxtimes\rangle$ est également un *demi-anneau tropical*.

□

Théorème 3.5.3. *La structure $\langle \mathcal{XDD}, \oplus, \otimes, \mathbb{0}_\oplus, \mathbb{1}_\otimes\rangle$ est un demi-anneau commutatif.*

Démonstration. Les éléments neutres dans $\mathcal{XDD}$ sont :

$$\begin{aligned}
\mathbb{0}_\oplus &= \text{LEAF}(-\infty) \\
\mathbb{1}_\otimes &= \text{LEAF}(0)
\end{aligned}$$

En fait, un XDD est seulement une représentation graphique de valeurs définies sur $\mathbb{D}$. Les opérateurs dans le domaine $\mathbb{D}$ sont équivalents à ceux transférés dans le domaine $\mathcal{XDD}$: nous l'avons déjà démontré dans le théorème 3.3.2. Par conséquent,

toute propriété des opérateurs dans $\mathbb{D}$ est automatiquement héritée par les opérateurs correspondants dans $\mathcal{XDD}$. Donc, $\langle \mathcal{XDD}, \oplus, \otimes, \mathbb{0}_{\oplus}, \mathbb{1}_{\otimes} \rangle$ est aussi un demi-anneau commutatif. □

Corollaire 3.5.3.1. *La fonction de concrétisation β (la réciproque de α) définie par 3.2.5 ainsi que le transfert des opérateurs binaires (défini dans la définition 3.3.2) forment un isomorphisme entre les demi-anneaux :*

$$\langle \mathcal{XDD}, \oplus, \otimes, \mathbb{0}_{\oplus}, \mathbb{1}_{\otimes} \rangle \underset{\beta}{\overset{\alpha}{\rightleftharpoons}} \langle \mathbb{D}, \boxplus, \boxtimes, \mathbb{0}_{\boxplus}, \mathbb{1}_{\boxtimes} \rangle$$

i.e. :
- $\beta(f_1^\# \oplus f_2^\#) = \beta(f_1^\#) \boxplus \beta(f_2^\#)$
- $\beta(f_1^\# \otimes f_2^\#) = \beta(f_1^\#) \boxtimes \beta(f_2^\#)$
- *β est bijective*

Démonstration. Le théorème 3.2.1 (page 33) a permis de montrer que β est bijective.

Grâce au théorème 3.3.2, nous savons que l'application des opérateurs binaires dans le domaine des XDD est équivalente à leur application dans le domaine $\mathbb{D}$, i.e. pour un opérateur binaire quelconque :

$$\forall f_1, f_2 \in \mathbb{D}^2, \forall \gamma \in \mathbb{B}^n, (\alpha(f_1) \odot \alpha(f_2))[\gamma] = (f_1 \boxdot f_2)(\gamma)$$

On peut remplacer $\alpha(f_1), \alpha(f_2)$ par $f_1^\#$ et $f_2^\#$; f_1, f_2 par $\beta(f_1^\#)$ et $\beta(f_2^\#)$ car α et β forment in isomorphisme (théorème 3.2.1) :

$$\forall f_1^\#, f_2^\# \in \mathcal{XDD}^2, \forall \gamma \in \mathbb{B}^n, (f_1^\# \odot f_2^\#)[\gamma] = (\beta(f_1^\#) \boxdot \beta(f_2^\#))(\gamma)$$

Ensuite, par la définition de β (définition 3.2.5), comme $f_1^\# \odot f_2^\#$ s'évalue exactement comme $\beta(f_1^\#) \boxdot \beta(f_2^\#)$, on obtient :

$$\forall f_1^\#, f_2^\# \in \mathcal{XDD}^2, \beta(f_1^\# \odot f_2^\#) = \beta(f_1^\#) \boxdot \beta(f_2^\#)$$

ce qui est vrai pour tout opérateur binaire $\odot$ sur $\mathcal{XDD}$ et $\boxdot$ sur $\mathbb{D}$. Il ne reste qu'à instancier ces deux opérateurs pour $\oplus$ ($\boxplus$) et $\otimes$ ($\boxtimes$).

□

Un XDD est uniquement une représentation graphique de $\mathbb{D}$, les opérations correspondantes (déduite de la définition 3.3.2) sont aussi équivalentes à celles dans le domaine $\mathbb{D}$.

À partir du demi-anneau $\mathcal{XDD}$, les matrices de XDD et la multiplication matricielle sur le domaine $\mathcal{XDD}$ peuvent être définies.

Définition 3.5.1. Soit $\mathcal{XDD}^{N \times M}$, le domaine de matrices des XDD, la multiplication matricielle est définie par :

$$\cdot : \mathcal{XDD}^{N \times M} \times \mathcal{XDD}^{M \times L} \to \mathcal{XDD}^{N \times L},$$

$$B \cdot C = \begin{bmatrix} A_{i,j} \end{bmatrix} \mid A_{i,j} = \bigoplus_{1 \le k \le M} (B_{i,k} \otimes C_{k,j})$$

qui est très similaire à la multiplication entre des matrices de nombres réels, mais en remplaçant les nombres réels par des XDD et, l'addition et la multiplication par les opérateurs correspondants dans $\mathcal{XDD}$, *max* et $+$.

Théorème 3.5.4. *Comme $\mathcal{XDD}$ est un demi-anneau ($\otimes$ est associatif), la multiplication matricielle est associative :*

$$\forall \mathcal{M}^A, \mathcal{M}^B, \mathcal{M}^C \in \mathcal{XDD}^{N\times M} \times \mathcal{XDD}^{M\times L} \times \mathcal{XDD}^{L\times K},$$
$$(\mathcal{M}^A \cdot \mathcal{M}^B) \cdot \mathcal{M}^C = \mathcal{M}^A \cdot (\mathcal{M}^B \cdot \mathcal{M}^C)$$

Démonstration. Soit $\mathcal{M}^{AB} = \mathcal{M}^A \cdot \mathcal{M}^B, \mathcal{M}^{BC} = \mathcal{M}^B \cdot \mathcal{M}^C$.

$$\begin{aligned}
\mathcal{M}^{AB}_{i,j} &\stackrel{\text{def}}{=} \bigoplus_{1\le k\le M} (\mathcal{M}^A_{i,k} \otimes \mathcal{M}^B_{k,j}) \\
(\mathcal{M}^{AB} \cdot \mathcal{M}^C)_{i,j} &\stackrel{\text{def}}{=} \bigoplus_{1\le l\le L} (\mathcal{M}^{AB}_{i,l} \otimes \mathcal{M}^C_{l,j}) \\
&= \bigoplus_{1\le l\le L} \Big(\bigoplus_{1\le k\le M} (\mathcal{M}^A_{i,k} \otimes \mathcal{M}^B_{k,l}) \otimes \mathcal{M}^C_{l,j} \Big) \\
&\quad \text{Comme } \otimes \text{ est distributif par rapport à } \oplus, \\
&= \bigoplus_{1\le l\le L} \Big(\bigoplus_{1\le k\le M} (\mathcal{M}^A_{i,k} \otimes \mathcal{M}^B_{k,l} \otimes \mathcal{M}^C_{l,j}) \Big) \\
&= \bigoplus_{1\le k\le M} \Big(\bigoplus_{1\le l\le L} (\mathcal{M}^A_{i,k} \otimes \mathcal{M}^B_{k,l} \otimes \mathcal{M}^C_{l,j}) \Big) \\
&= \bigoplus_{1\le k\le M} \Big(\mathcal{M}^A_{i,k} \otimes \bigoplus_{1\le l\le L} (\mathcal{M}^B_{k,l} \otimes \mathcal{M}^C_{l,j}) \Big) \\
&\stackrel{\text{def}}{=} \bigoplus_{1\le k\le M} (\mathcal{M}^A_{i,k} \otimes \mathcal{M}^{BC}_{k,j}) \\
&\stackrel{\text{def}}{=} (\mathcal{M}^A \cdot \mathcal{M}^{BC})_{i,j}
\end{aligned}$$

□

Dans l'analyse du pipeline définie plus loin dans ce document, nous allons représenter l'état du pipeline sous forme de vecteur (i.e. matrice d'une seule ligne) de XDD. Nous pouvons représenter une fonction de transition d'état de pipeline comme une fonction affine (dans le domaine $\mathcal{XDD}$) des éléments du vecteur de l'état entrant, c'est-à-dire comme une multiplication matricielle.

L'associativité de la multiplication matricielle nous permet donc de pré-calculer une séquence d'applications de fonctions de transition en une seule matrice. C'est-à-dire, pour l'état de pipeline entrant d'un BB représenté par un vecteur $\overrightarrow{v}$, les transitions à appliquer sur $\overrightarrow{v}$ sont représentées par une séquence de matrices $\mathcal{M}_{1..n}$ que nous pouvons pré-calculer et résumer en une seule matrice $\mathcal{M}_{1-n}$:

$$\overrightarrow{v} \cdot \mathcal{M}_1 \cdot \mathcal{M}_2 \cdot ... \cdot \mathcal{M}_n = v \cdot (\mathcal{M}_1 \cdot \mathcal{M}_2 \cdot ... \cdot \mathcal{M}_n) = \overrightarrow{v} \cdot \mathcal{M}_{1-n}$$

3.6 Conclusion

Dans ce chapitre, nous avons présenté la structure de données des XDD qui est utilisée pour représenter efficacement la relation entre des variables binaires (les évènements) et des valeurs entières (les temps d'exécution). Nous avons montré que la représentation par XDD est sans perte par rapport à la représentation explicite, i.e.

elle conserve explicitement la relation entre chaque configuration de variable et la valeur correspondante. De plus, toutes les opérations binaires qui s'appliquent sur la représentation explicite à chaque configuration peuvent également être transférées dans le domaine des XDD, plus compact, qui réalise les mêmes opérations mais de manière plus efficace. Cela signifie que nous pouvons réaliser l'analyse de pipeline pour toutes les configurations en une seule passe à l'aide des XDD et des opérateurs transférés dans ce domaine. Les propriétés algébriques du demi-anneau $\mathcal{XDD}$ peuvent aussi aider à accélérer l'analyse de pipeline si on arrive à représenter les calculs sous forme des multiplication matricielle.

Chapitre 4

Analyse de pipeline avec les XDD

Pour calculer le temps d'exécution d'un programme à la précision du nombre de cycles de processeur, il est indispensable de considérer la micro-architecture du processeur car le jeu d'instructions spécifie uniquement le comportement fonctionnel du programme et manque d'informations pour modéliser précisément le comportement temporel. Par exemple, considérons la séquence d'instructions (en ARM) ci-dessous :

```
I0: add r3, r0, #4
I1: add r1, r0, r1, lsl #2
I2: ldr r2, [r3]
I3: cmp r2, ip
I4: ldrgt ip, [r3]
I5: add r3, r3, #4
```

Le temps d'exécution de l'instruction **ldr**, qui charge des données depuis la mémoire, dépend de la latence de la mémoire. En outre, si le processeur est équipé d'un cache de données, son temps d'exécution devient variable tout en dépendant du comportement de l'accès au cache, *Miss* ou *Hit*.

En plus de la latence variable d'exécution, l'analyse temporelle doit aussi considérer la structure du pipeline. Les instructions sont en fait exécutées en parallèle dans les étages du pipeline avec des contraintes entre leurs exécutions respectives pour gérer les *aléas d'exécution*. Par exemple, la comparaison dans I3 ne peut pas être effectuée avant que l'opérande r2 soit calculé et écrit par I2 (*aléa de données*). Nous allons appeler ces contraintes des *dépendances*.

Les *dépendances* spécifient les contraintes d'exécution entre des **composants du pipeline** (souvent les étages). Par exemple, la dépendance de données entre I3 et I2 est en fait entre **l'étage d'exécution** où I2 produit le registre r2 et **l'étage d'accès mémoire** où I3 consomme r2.

Du point de vue de l'analyse temporelle, la présence du pipeline et des *dépendances* signifie que nous ne pouvons pas simplement calculer le temps d'exécution de la séquence par la somme du temps d'exécution de chaque instruction : cela reviendrait à ignorer le parallélisme d'exécution des instructions dans les étages et le temps d'exécution serait largement surestimé. Pour modéliser l'effet du parallélisme dans le pipeline et des *dépendances* – qui lui imposent des restrictions, un modèle de pipeline doit être construit au sein de l'analyse de pipeline. Par exemple, le graphe d'exécution original et la sémantique *cycle* que nous avons vus au chapitre 2 sont des modèles de pipeline.

Les latences variables à l'exécution – modélisées par les évènements, la structure du pipeline et les *dépendances* sont les trois facteurs principaux à considérer pour construire le modèle d'exécution.

Comme présenté dans le chapitre 2, la présence des évènements crée de la divergence pour modéliser l'exécution des instructions dans le pipeline. L'analyse doit faire le choix entre garder explicitement tous les états de pipeline possibles – avec une complexité combinatoire, ou couvrir la latence supplémentaire par une surestimation du temps d'exécution. Le choix que nous avons fait est de calculer l'exécution explicite dans chaque configuration d'évènements en profitant des XDD pour atténuer le problème de complexité.

Dans ce chapitre, nous allons présenter une nouvelle variante du graphe d'exécution qui hérite du graphe d'exécution proposé par ROCHANGE et al., 2009. La différence la plus remarquable est que nous allons utiliser des XDD pour représenter tous les temps d'exécution possibles. Dans la suite, sauf spécification explicite, la terminologie *graphe d'exécution* est réservé à notre variante de graphe d'exécution.

4.1 Le Graphe d'exécution

Définition 4.1.1. Un *graphe d'exécution* exprime (ou modélise) l'exécution temporelle d'une séquence d'instructions sur une micro-architecture donnée.

Considérons un processeur de N_S étages, notés $S = [s_1, s_2, ..., s_{N_S}]$ et la séquence d'instructions notée $[I_1, I_2, ..., I_{N_I}] \in \mathcal{I}^{N_I}$.

Un graphe d'exécution est un DAG (V_{XG}, E_{XG}) dont les nœuds $V_{XG} \subseteq \mathcal{I} \times S$ sont des paires $[I_i/s_j]$ qui représentent l'exécution de l'instruction I_i à l'étage s_j.

Chaque nœud $v = [I_i/s_j]$ est associé à une latence $\lambda_v \in \mathbb{Z}$ [1] qui représente le temps d'exécution de I_i passé dans l'étage s_j. Dans cette section, nous ne considérons pas les évènements, le temps d'exécution de chaque nœud est donc statique.

Les arcs $E_{XG} \subset V_{XG} \times V_{XG}$ représentent les dépendances entre l'exécution des nœuds. Un arc $v \to w$ peut être un arc plein ou pointillé, ce qui est indiqué par la fonction $\delta : \delta(v \to w) = 0$ si l'arc est en pointillé, 1 si l'arc est plein. Ces deux types d'arc représentent deux types de dépendances : l'arc plein signifie que "w peut démarrer seulement après la fin de v " ; l'arc pointillé signifie que "w peut démarrer en même temps que v mais pas plus tôt". Les arcs pointillés sont utilisés pour exprimer l'utilisation concurrente de plusieurs instructions sur une même ressource : des instructions peuvent être exécutées dans le même cycle mais pas exécutées dans le désordre. Un exemple de graphe d'exécution est présenté dans la figure 4.1 (page 54).

Dans notre variante de graphe d'exécution, les contentions liées des ressources allouées dans le désordre ne sont pas modélisables parce que l'ordre d'allocation de ces ressources ne peut pas être déterminé statiquement et donc les dépendances (les arcs) ne peuvent pas être construites statiquement. C'est la raison pour laquelle un graphe d'exécution est toujours un DAG. En comparaison, avec le graphe d'exécution original présenté à la section 2.6 (page 23), les contentions sont marquées par des arcs non-orientés et calculées par un algorithme de point fixe.

Notre nouveau graphe d'exécution est capable de modéliser différentes micro-architectures : la différence se joue au niveau de l'utilisation des ressources du pipeline, représentée par des arcs (dépendances). Nous utiliserons, dans la suite, comme exemple, un ensemble de règles de construction d'arc qui modélisent un pipeline. Bien évidemment, cet ensemble de règles doit être ajusté selon la micro-architecture réelle en question mais les règles ci-dessous seront probablement communes à l'ensemble du spectre des processeurs d'exécution dans l'ordre.

1. La latence des nœuds est toujours positive, nous l'avons définie dans $\mathbb{Z}$ uniquement pour être cohérents avec le domaine des valeurs de sortie dans les XDD.

Ordre induit par pipeline (PipeOrd) Toute instruction doit traverser le pipeline dans l'ordre des étages :

$$\forall i \in [1, N_I - 1], \forall j \in [1, N_S - 1], \begin{cases} [I_i/s_j] \rightarrow [I_i/s_{j+1}] \in E_{XG} \\ \delta([I_i/s_j] \rightarrow [I_i/s_{j+1}]) = 1 \end{cases} \tag{4.1}$$

Ordre de fetch (FeOrd) À cause de l'organisation de la mémoire et du pipeline, les instructions sont chargées depuis la mémoire dans l'ordre du programme :

$$\forall i \in [1, N_I - 1], [I_i/s_1] \rightarrow [I_{i+1}/s_1] \in E_{XG} \tag{4.2}$$

avec s_1 le premier étage du pipeline que nous supposons être l'étage de *fetch*.

S'il n'y a pas de cache d'instructions, seule une instruction peut être chargée par cycle :

$$\forall i \in [1, N_I - 1], \delta([I_i/s_1] \rightarrow [I_{i+1}/s_1]) = 1 \tag{4.3}$$

S'il y a un cache d'instructions, seules des instructions qui sont dans le même bloc de mémoire du cache peuvent être chargées dans le même cycle. La fonction $\mathcal{B} : \mathcal{I} \rightarrow \mathbb{N}$ donne, pour chaque instruction, un identifiant de son bloc de cache (en concaténant le *tag* et le numéro d'ensemble de bloc [2]).

$$\forall i \in [1, N_I - 1], \delta([I_i/s_1] \rightarrow [I_{i+1}/s_1]) = \begin{cases} 0 & \text{if } \mathcal{B}(I_i) = \mathcal{B}(I_{i+1}) \\ 1 & \text{else} \end{cases} \tag{4.4}$$

Ordre du programme (ProgOrd) Un étage s_K est partagé par les instructions exécutées en parallèle dans le pipeline. Si l'étage est utilisé par les instructions dans l'ordre du programme, des arcs pointillés sont créés pour assurer cet ordre :

Soit $[I_1, ..., I_{N_k}] \in \mathcal{I}^{N_k}$ les instructions utilisant l'étage S_k, les arcs à construire sont :

$$\forall i \in [1, N_k - 1], \forall j \in [1, N_S - 1], \begin{cases} [I_i/s_j] \rightarrow [I_{i+1}/s_j] \in E_{XG} \\ \delta([I_i/s_j] \rightarrow [I_{i+1}/s_j]) = 0 \end{cases} \tag{4.5}$$

Au contraire, si l'étage peut être alloué dans le désordre, aucune dépendance n'est créée parce que notre modèle de pipeline présenté dans ce chapitre est insuffisant pour gérer les ressources allouées dans le désordre. Nous allons essayer de les gérer indépendamment plus tard dans le chapitre 7.

Ordre de Capacité (CapOrd) Un étage a généralement une capacité limitée. Un étage s_k de taille $|s_k|$ signifie qu'au plus $|s_k|$ instructions peuvent être exécutées en même temps dans l'étage (souvent, cela signifie que $|s_k|$ *unités fonctionnelles* sont disponibles à cet étage).

Pour les instructions qui sont exécutées dans un étage s_k selon une politique d'émission FIFO (*First-In-First-Out*), un arc plein est ajouté pour les instructions séparées d'une distance de $|s_k|$ des autres instructions. Soit $[I_1, ..., I_{N_k}] \in \mathcal{I}^k$ les instructions utilisant l'étage S_k, les arcs à construire sont :

$$\forall i \in [1, (N_k - |s_k| - 1)], \forall k \in [1, N_S - 1], \begin{cases} [I_i/s_k] \rightarrow [I_{i+|s_k|}/s_k] \in E_{XG} \\ \delta([I_i/s_k] \rightarrow [I_{i+|s_k|}/s_k]) = 1 \end{cases} \tag{4.6}$$

2. Si le cache est *associatif par ensemble*, ce qui est le cas le plus commun.

Ordre induit par les données (DataDep) Les *aléas de données* peuvent se présenter lors de l'exécution : une instruction lisant des registres à un certain étage doit attendre que l'écriture de ces registres soit terminée (s'il y en a) avant de réaliser son calcul.

Les fonctions ci-dessous sont définies pour simplifier la présentation :

— $rr : \mathcal{I} \to 2^{\mathcal{R}eg}$ – pour une instruction donnée, renvoie l'ensemble des registres lus par l'instruction. $\mathcal{R}eg$ représente l'ensemble des registres du processeur.

— $wr : \mathcal{I} \to 2^{\mathcal{R}eg}$ – pour une instruction donnée, renvoie l'ensemble des registres écrits par l'instruction.

— $rs : \mathcal{I} \times \mathcal{R}eg \to \mathsf{S}$ – pour une instruction qui lit un registre, renvoie l'étage où la lecture est effectuée.

— $ws : \mathcal{I} \times \mathcal{R}eg \to \mathsf{S}$ – pour une instruction qui écrit un registre, renvoie l'étage où l'écriture est effectuée.

La dépendance de données est exprimée par des arcs pleins entre l'instruction qui produit le contenu d'un registre et l'instruction qui le lit :

$$\begin{array}{l} \forall i,j \in [1, N_I - 1]^2, (k,l) \in [1, N_{\mathsf{S}} - 1]^2, \\ \left.\begin{array}{l} [I_i/\mathsf{s}_k] \to [I_j/\mathsf{s}_l] \in E_{\mathrm{XG}} \\ \delta([I_i/\mathsf{s}_k] \to [I_j/\mathsf{s}_l]) = 1 \end{array}\right\} \iff \begin{array}{l} i < j \\ \wedge\ wr(I_i) \cap rd(I_j) \neq \emptyset \\ \wedge\ \mathsf{s}_k \in \{ws(I_i, reg) \mid reg \in wr(I_i)\} \\ \wedge\ \mathsf{s}_l \in \{rs(I_j, reg) \mid reg \in rd(I_j)\} \end{array} \end{array} \tag{4.7}$$

Ordre de mémoire (MemOrd) Pour assurer la consistance de la mémoire centrale, les accès mémoire sont ordonnés. Les contraintes suivantes doivent être prises en compte :

— L'ordre entre une lecture et une écriture doit être préservé ;

— L'ordre entre les écritures doit être préservé ;

— Les lectures peuvent être re-ordonnées si les deux contraintes ci-dessus sont respectées.

La construction des arcs représentant ces contraintes nécessite des informations supplémentaires fournies par les fonctions suivantes :

— Si une instruction fait une lecture depuis la mémoire, le prédicat $il : \mathcal{I} \to \mathbb{B}$ renvoie vrai.

— Si une instruction fait de l'écriture en mémoire, le prédicat $is : \mathcal{I} \to \mathbb{B}$ renvoie vrai,

— $ss : \mathcal{I} \to \mathsf{S}$ renvoie l'étage où l'accès mémoire commence,

— $ts : \mathcal{I} \to \mathsf{S}$ renvoie l'étage où l'accès mémoire termine.

Souvent, l'étage auquel commence l'accès mémoire et auquel il termine est le même, mais nous avons déjà vu des exceptions : *ss* et *ts* seront utiles uniquement pour ces cas exceptionnels.

Les arcs sont créés selon la règle :

$$\begin{array}{l} \forall i,j \in [1, N_I - 1]^2, \mathsf{s}_k, \mathsf{s}_l \in \mathsf{S}^2, \\ \left.\begin{array}{l} [I_i/\mathsf{s}_k] \to [I_j/\mathsf{s}_l] \in E_{\mathrm{XG}} \\ \delta([I_i/\mathsf{s}_k] \to [I_j/\mathsf{s}_l]) = 1 \end{array}\right\} \iff \begin{array}{l} i < j \\ \wedge\ (il(I_i) \vee is(I_i)) \\ \wedge\ is(I_j) \\ \wedge\ \mathsf{s}_k = ts(I_i) \\ \wedge\ \mathsf{s}_l = ss(I_j) \end{array} \end{array} \tag{4.8}$$

Ordre de branchement (BrOrd) Quand un branchement est pris, le chargement de l'instruction cible peut seulement commencer son exécution après avoir déterminé l'adresse cible (pour pouvoir charger l'instruction cible). Cependant, considérant que le branchement peut être conditionnel (e.g. $BEQ\ 0x1000$ dans ARM) ou inconditionnel (e.g. $B\ 0x1000$) ; direct (e.g. $B\ 0x1000$) ou indirect (e.g. $MOV\ PC, LR$), les fonctions définies ci-dessous nous seront utiles :

— $ib : \mathcal{I} \to \mathbb{B}$ indique si l'instruction est de type branchement ;
— $db : \mathcal{I} :\to \mathbb{B}$ si l'instruction fait un branchement et peut poursuivre en séquence (dans le cas de branchement conditionnel) ;
— $bs : \mathcal{I} \to \mathsf{S}$ donne l'étage dans lequel l'adresse de branchement est calculée.

Dans le cas où il n'y a pas de prédicteur de branchement :

si l'instruction à exécuter après le branchement est consécutive dans la mémoire (branchement non pris, $\neg db(I_i)$), aucun arc n'est généré parce que ces instructions sont pré-chargées et donc ne sont pas soumises aux latences supplémentaires ;

si l'instruction à exécuter après le branchement est l'instruction cible (branchement pris, $db(I_i)$), l'instruction cible peut commencer son exécution uniquement après le calcul d'adresse :

$$\left.\begin{array}{r} \forall i \in [1, N_I - 1], \forall k \in [1, N_{\mathsf{S}} - 1], [I_i/\mathsf{s}_k] \to [I_{i+1}/\mathsf{s}_1] \in E_{\mathrm{XG}} \\ \delta([I_i/\mathsf{s}_k] \to [I_{i+1}/\mathsf{s}_1]) = 1 \end{array}\right\} \iff \begin{array}{l} ib(I_i) \\ \wedge\, db(I_i) \\ \wedge\, bs(I_i) = \mathsf{s}_k \end{array} \tag{4.9}$$

avec s_1 le premier étage du pipeline que nous supposons être l'étage de fetch.

Dans la suite de ce document, les expériences sont faites pour le cas sans prédicteur de branchement. Mais notre modèle est capable de le supporter avec une petite extension.

Dans le cas où le prédicteur est présent, le comportement du pipeline est le suivant : les instructions sont pré-fetchées en fonction de la prédiction, l'exécution continue normalement jusqu'au moment où l'adresse cible est effectivement calculée (à $bs(I_i)$). Si la prédiction est correcte, aucune latence supplémentaire n'est générée ; sinon, le pipeline est vidé et l'instruction correcte commence son exécution.

Donc, l'arc entre l'étage calculant l'adresse cible et l'étage fetch de l'instruction suivante est créé que le branchement soit pris ou non. Mais cette dépendance devient conditionnelle en fonction de la correction de la prédiction, annotée par un évènement e_{br}, actif si mal prédit, inactif si bien prédit. Pour représenter cette dépendance conditionnelle, une latence $0 + (1 - e_{br}) \times -\infty$ est associée à l'arc : si l'évènement est actif (mauvaise prédiction), la latence est 0 et l'arc fonctionne comme un arc plein normal ; si l'évènement est inactif (bonne prédiction), la latence $-\infty$ rend l'arc inutile.

Ordre induit par la capacité des files (QueueOrd) Dans un processeur pipeliné, des mémoires tampons existent généralement entre les étages pour sauvegarder les instructions et leur état. Ces tampons permettent de (a) stocker des instructions quand elles sont exécutées dans l'étage, et (b) libérer l'étage précédent tant que l'étage suivant n'est pas disponible (pour atténuer des blocages à cause des *aléas d'exécution*). Ces mémoires tampons sont le plus souvent des files (FIFO). Une file q de l'ensemble de files $\mathcal{Q}$ du pipeline ($q \in \mathcal{Q}$) est définie par :

— un étage producteur s_q^p qui ajoute des instructions dans la file ;
— un étage consommateur s_q^c qui prélève des instructions de la file ;

— une capacité $|q|$ qui donne le nombre maximal d'instructions qui peuvent être stockées dans la file simultanément.

Quand une file est positionnée entre deux étages, sa capacité est toujours supérieure ou égale à la capacité de l'étage producteur (l'étage précédent) car la file est au moins utilisée comme tampon d'instructions par l'étage producteur, ce qui correspond au rôle (a) discuté ci-dessus. Un tampon de taille plus petite que la taille de l'étage producteur signifie qu'une partie de l'étage producteur (certaines unités fonctionnelles) ne peut jamais être utilisée et, par conséquent, une telle configuration n'existe pas dans des processeurs réels. Si la capacité de la file est égale à la capacité de l'étage producteur, la file joue uniquement le rôle (a) – c'est une mémoire tampon pour les instructions exécutées dans l'étage producteur. Le rôle (b) – atténuer des *aléas d'exécution*, est effectif seulement si la capacité de la file est supérieure à celle de l'étage producteur. Cela est typiquement le cas après l'étage de fetch : cela permet d'amortir les aléas d'accès à la mémoire des instructions.

Nous nous intéressons uniquement aux files de politique FIFO, type utilisé la plupart du temps dans les processeurs à exécution dans l'ordre. Même pour des processeurs à exécution dans le désordre, les étages fetch, decode et certains d'autres réalisent une exécution dans l'ordre et par conséquent, les tampons sont implémentés par des files.

Avec ce type de file, les contraintes temporelles imposées par les tampons sont :

— L'exécution d'une instruction dans un étage producteur peut commencer seulement s'il y a au moins une entrée disponible dans la file d'attente.

— Une instruction libère une entrée de la file d'attente quand elle commence son exécution dans l'étage consommateur (elle peut attendre dans la file après la fin d'exécution dans l'étage producteur et jusqu'au démarrage dans l'étage consommateur).

Les arcs ci-dessous sont créés dans le graphe d'exécution.

$$\forall q \in \mathcal{Q}, \forall 1 \leq i < N_I - |q| \Rightarrow \begin{cases} [I_i/\mathsf{s}_q^c] \rightarrow [I_{i+|q|}/\mathsf{s}_q^p] \in E_{\mathrm{XG}} \\ \delta([I_i/\mathsf{s}_q^c] \rightarrow [I_{i+|q|}/\mathsf{s}_q^p]) = 0 \end{cases} \tag{4.10}$$

Suivant ces règles, nous pouvons construire le graphe d'exécution présenté à la figure 4.1, pour la séquence d'instructions qui se situe dans la partie gauche de la figure. Dans cet exemple, nous considérons un processeur à 5 étages (FE-DE-EX-ME-WB), avec une exécution dans l'ordre et soumis aux contraintes décrites par toutes les règles ci-dessus. La capacité de chaque étage et de chaque file est de 2.

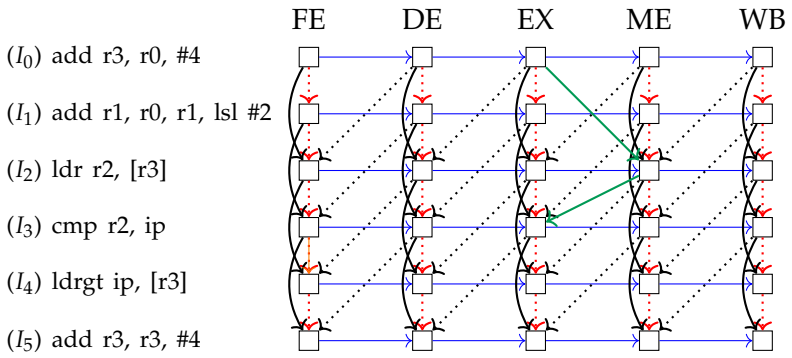


FIGURE 4.1 – Un exemple de graphe d'exécution.

Les arcs bleus représentent l'ordre du pipeline alors que les arcs rouges pointillés verticaux représentent l'ordre de programme dans chaque étage. Pour l'étage

FE, l'ordre du programme coïncide avec l'ordre du fetch quand les instructions se trouvent dans le même bloc de cache. L'arc orange plein vertical (entre $[I_3/FE]$ et $[I_4/FE]$) est l'ordre du fetch pour des instructions dans différents blocs de cache. Les arcs noirs verticaux courbés représentent l'ordre de capacité de chaque étage. Les arcs noirs pointillés diagonaux prennent en compte l'ordre des files. Les deux arcs pleins verts matérialisent les dépendances de données : I_0 doit écrire r_3 avant la lecture par I_2 ; I_2 doit écrire r_2 avant la lecture par I_3. La dépendance de données entre I_0 et I_4 n'est pas considérée car l'étage ME est alloué dans l'ordre du programme et donc la dépendance $[I_0/EX] \rightarrow [I_2/ME]$ (*CapOrd*) suffit pour modéliser la dépendance de données entre I_0 et I_4. De manière similaire, la dépendance de données $[I_0/EX] \rightarrow [I_5/EX]$ est également n'est pas représentée puisque l'étage EX a une exécution dans l'ordre et est de taille 2.

4.2 Le calcul de temps avec le graphe d'exécution

Une fois le graphe d'exécution construit, nous pouvons nous en servir pour calculer le temps d'exécution. Comme les arcs du graphe représentent toutes les contraintes à respecter durant l'exécution, la date effective du démarrage de chaque nœud est la date au plus tôt où toutes les dépendances sont satisfaites. Cette date est calculée en prenant le maximum des dates de début (noté ρ_v) ou de fin (noté ρ^*_v) des prédécesseurs en fonction du type d'arc (temps de fin si l'arc est plein, temps de démarrage si l'arc est pointillé). La date de fin d'un nœud est simplement le temps de démarrage augmenté de sa latence. Formellement, le calcul peut se résumer par l'équation 4.11 ci-dessous :

$$\forall w \in V_{\mathrm{XG}}, \rho_w = \max_{v \rightarrow w \in E_{\mathrm{XG}}} (\rho_v + \delta_{v \rightarrow w} \times \lambda_v) \tag{4.11}$$

Pour simplifier la notation, nous définissons aussi la date de fin de chaque nœud :

$$\forall v \in V_{\mathrm{XG}}, \rho^*_v = \rho_v + \lambda_v \tag{4.12}$$

Pour le graphe de la figure 4.1, si on suppose que le premier nœud $[I_0/FE]$ commence à la date 0, nous pouvons déterminer le temps de fin de $[I_0/FE]$ qui est son temps de démarrage plus la latence du nœud : $\rho^*_{[I_0/FE]} = 0 + \lambda_{[I_0/FE]}$. Ensuite, nous pouvons déterminer le temps de démarrage de $[I_0/DE]$ car sa seule dépendance est un arc plein $[I_0/FE] \rightarrow [I_0/DE]$: comme $\lambda_{[I_0/FE]} = 1$, nous aurons $\rho_{[I_0/DE]} = 1$. De la même manière, les dates de $[I_0/EX]$, $[I_0/ME]$, $[I_0/WB]$, puis $[I_1/FE]$, ... sont calculées jusqu'au dernier nœud du graphe.

Comme un graphe d'exécution est un DAG, il existe au moins un ordre topologique entre les nœuds qui permet de calculer le temps de chaque nœud en respectant les dépendances. Dans un processeur à exécution dans l'ordre, comme les dépendances *ProgOrd*, *PipeOrd* et *QueueOrd* sont toujours présentes, nous pouvons ainsi trouver un ordre topologique qui est toujours valide : l'évaluation par instruction et par l'ordre des étages comme montré par la flèche grise en fond de la figure 4.2. En suivant cet ordre topologique, les dépendances sont résolues correctement et le temps de début et fin de chaque nœud est déterminé.

Pour mieux modéliser le contexte d'exécution d'un BB, un graphe d'exécution est construit pour le BB et chacun de ses prédécesseurs. Ces graphes sont associés à leurs arcs respectifs $a \rightarrow b$ et qualifiés d'exécution de b dans le contexte de a. La date 0 que nous supposons au premier nœud du graphe est en fait la date du début de la première instruction du BB prédécesseur a. Le temps d'exécution du BB

b est calculé par la date de début du dernier nœud moins la date du début de la dernière instruction du BB a au dernier étage. Comme les dernières instructions du BB prédécesseur sont exécutées en parallèle avec les premières instructions du BB calculé, cette opération évite de compter plusieurs fois le temps de ces instructions : leur exécution correspond au temps d'exécution commun entre a et b.

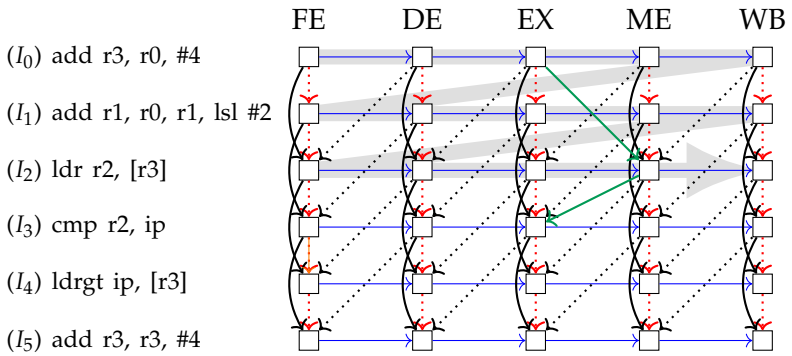


FIGURE 4.2 – Un ordre d'évaluation valide de graphe d'exécution pour tout processeur d'exécution dans l'ordre.

4.3 Les évènements dans les graphes d'exécution

Jusqu'à présent, nous considérions que la latence des nœuds était une constante : $\lambda \in \mathbb{Z}$. Mais pour de nombreux processeurs, certains étages ont une latence variable en fonction de l'état de la micro-architecture durant l'exécution. Ces variations temporelles sont gérées par les analyses globales et représentées par des évènements au sein de l'analyse de pipeline.

Dans cette section, nous allons considérer un exemple typique qui introduit des évènements : la mémoire cache. Si une instruction fait un accès mémoire (de données ou d'instruction) à un certain étage, la latence est courte si le cache contient déjà le bloc recherché (*Hit*). Dans le cas contraire, le bloc est chargé depuis la mémoire centrale (ou un cache du niveau plus bas) et par conséquent l'accès prend un temps beaucoup plus long (*Miss*).

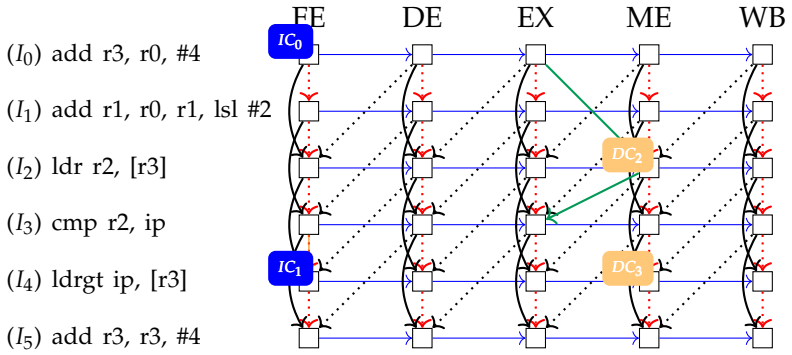


FIGURE 4.3 – Le graphe d'exécution avec des évènements

La figure 4.3 montre le graphe d'exécution de l'exemple précédent décoré avec des évènements. Quatre évènements sont présents dans cette séquence d'instructions : IC_0 et IC_1 sont des évènements de cache d'instruction identifiés par l'analyse de cache d'instructions. DC_2 et DC_3 sont des évènements de cache de données fournis par l'analyse de cache de données. Comme expliqué dans le chapitre précédent,

les évènements sont exactement des variables booléennes. Pour des évènements de cache (d'instruction ou de données), l'activation d'un évènement signifie qu'un *Miss* a lieu et la valeur de l'évènement vaut 1 (ou $VRAI$), et l'inactivation d'un évènement signifie un *Hit* et la valeur de l'évènement est 0 (ou $FAUX$).

Une manière triviale de calculer le temps d'exécution sur le graphe avec des évènements pourrait être de résoudre les dates du graphe avec le calcul sans évènements (présenté dans la section précédente) puis de répéter ce calcul pour chaque configuration d'évènements. C'est-à-dire, dans notre graphe qui contient 4 évènements, il faut faire $2^4 = 16$ parcours complets, ce qui est coûteux.

Pour éviter un nombre de parcours exponentiel en fonction du nombre d'évènements, une autre idée peut être de calculer symboliquement tous les temps possibles pour chaque nœud du graphe comme une fonction des évènements. En reprenant la méthode de calcul du graphe d'exécution sans évènements, cela signifie que la latence λ des nœuds devient une fonction de l'activation des évènements. Par exemple, le temps d'exécution de $[I_0/FE]$ devient $\lambda_{[I_0/FE]} = max(1, 10.IC_0)$, i.e. 10 cycles si l'évènement de cache est actif (*Miss*), 1 cycle sinon (*Hit*). De la même manière, les temps de $[I_2/ME]$, $[I_4/FE]$ et $[I_4/ME]$ deviennent également variables.

Nous pouvons appliquer à nouveau la résolution du temps d'exécution du graphe, mais cette fois avec les nouvelles latences λ, les dates des nœuds deviennent :

— $\rho_{[I_0/FE]}(\gamma) = 0$
— $\lambda_{[I_0/FE]}(\gamma) = \max(1, 10.\gamma(IC_0))$
— $\rho_{[I_0/DE]}(\gamma) = \rho_{[I_0/FE]}(\gamma) + \lambda_{[I_0/FE]}(\gamma) = \max(1, 10.\gamma(IC_0))$
— ...
— $\rho_{[I_2/EX]}(\gamma) = 2 + \max(1, 10.\gamma(IC_0))$
— $\rho^*_{[I_2/ME]}(\gamma) = 3 + \max(1, 10.\gamma(IC_0)) + \max(1, 10.\gamma(DC))$
— ...
— $\rho_{[I_3/DE]}(\gamma) = 2 + \max(1, 10.\gamma(IC_0))$
— ...

Néanmoins, comme on peut le constater, le temps d'exécution des nœuds est une fonction qui contient des max et $+$ ce qui rend la représentation et le calcul peu efficace car elle est difficile à canoniser et à simplifier. En outre, la résolution du graphe d'exécution (équation 4.11) nécessite de réaliser des calculs faisant intervenir des max et $+$ sur des variables (représentant les évènements), ce qui complique encore le calcul et rend difficile l'obtention d'une représentation efficace. Or, c'est exactement dans ce genre de situation que les XDD deviennent utiles parce qu'ils peuvent représenter ces fonctions de manière efficace et compacte.

Les XDD pour représenter le temps Les fonctions de temps d'exécution en fonction des évènements peuvent être définies dans le domaine $\mathbb{D} : \mathbb{B}^n \to \mathbb{Z}^\infty$, où $\mathbb{B}^n = \mathcal{E}^n$ représente toutes les combinaisons (configurations) possibles des évènements (c'est-à-dire des variables binaires) et $\mathbb{Z}^\infty$ le temps d'exécution. Noter que travailler sur $\mathbb{Z}^\infty$ permet par la suite des simplifications dans nos formules et est également adéquat pour représenter des temps d'exécution positifs comme $\mathbb{N} \subset \mathbb{Z}^\infty$. Par conséquent, nous pouvons profiter des XDD pour représenter efficacement la fonction (ou la relation) entre le temps d'exécution des nœuds du graphe et les évènements. Les opérateurs utilisés dans le calcul sont max, $+$ et $-$ [3] qui peuvent être implantés dans

3. $-$ est utile uniquement pour soustraire le temps d'exécution du BB prédécesseur à la fin du BB courant.

le domaine des XDD avec la définition des opérateurs binaires présentés dans la définition 3.3.2 (page 36).

En remplaçant tous les opérateurs $max, +$ et $-$ par l'opérateur correspondant dans le domaine $\mathcal{XDD}$ (voir table 3.1), nous pouvons transférer les calculs du graphe d'exécution dans le domaine $\mathcal{XDD}$ en réécrivant l'équation de calcul 4.11 en :

$$\rho_w = \bigoplus_{v \to w \in E_{XG}} \rho_v \otimes (\delta_{v \to w} \times \lambda_v)$$
$$\rho_v^* = \rho_v \otimes \lambda_v$$

La soustraction du temps des derniers nœuds de chaque BB est remplacée par l'opérateur $\oslash$, la soustraction transférée dans le domaine $\mathcal{XDD}$. La multiplication dans $\delta_{v \to w} \times \lambda_v$ ne nécessite pas d'être transférée dans le domaine des XDD car $\delta_{v \to w}$: le type d'arc, est statiquement connu, donc on calcule soit $\rho_v \otimes \lambda_v$ si $\delta_{v \to w} = 1$, soit ρ_v si $\delta_{v \to w} = 0$.

Maintenant, le temps (ρ ou ρ^*) de chaque nœud du graphe est exprimé sous forme de XDD, qui n'est plus un temps d'exécution unique, mais exactement tous les temps possibles en fonction des évènements. Par conséquent, un seul parcours du graphe devient nécessaire (au lieu de $2^{|\mathcal{E}|}$ sans utiliser les XDD) pour résoudre tous les temps d'exécution dans toutes les configurations. Comme présenté dans le chapitre 3, les XDD peuvent représenter de manière exacte et efficace la relation entre le temps d'exécution et les évènements, ainsi que les opérations entre ces fonctions. Ainsi, ils permettent d'accélérer l'analyse du graphe d'exécution en considérant explicitement les évènements.

La performance effective du calcul du graphe d'exécution dépend fortement du taux de compression des XDD utilisés durant le calcul, i.e. de la réduction de la taille des XDD grâce à la *compacité*. Cette performance est quantifiée sur le benchmark TACLe dans la section 4.4, mais nous pouvons déjà avancer quelques explications théoriques sur les bonnes performances obtenues. En fait, le pipeline porte naturellement en lui des propriétés qui favorisent la compression des XDD. Nous les nommons *effets d'amortissement* du pipeline. Parmi tous ces effets, deux d'entre eux jouent le rôle majeur : *l'amortissement de parallélisme* et *la régularité des latences*.

Amortissement de parallélisme Pour avoir une meilleure performance, un des principes de conception du pipeline est de favoriser le parallélisme et d'éviter le plus possible les blocages induits par les *aléas*. Cela fait que les latences supplémentaires (par exemple, les latences causées par l'activation des évènements) ont souvent moins d'effet de blocage.

Intuitivement, dans le calcul graphe d'exécution, ce phénomène est illustré dans la figure 4.4.

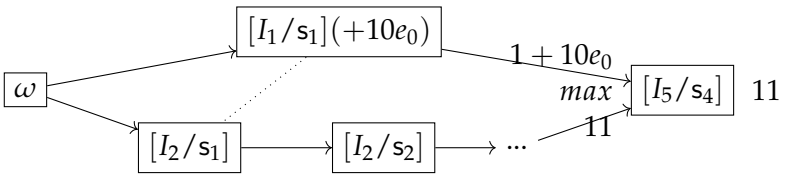


FIGURE 4.4 – Effet d'amortissement du pipeline.

À partir d'un nœud ω du graphe, deux chemins d'exécution dans le pipeline (c'est-à-dire deux chemins de dépendances dans le graphe) correspondent à des exécutions en parallèle. Le premier subit un évènement à l'étage s_1 qui coûte, par exemple, 10 cycles. Mais le pipeline n'est pas totalement bloqué en cas d'activation

de cet évènement, d'autres instructions, par exemple $[I_2/\mathsf{s}_1]$, qui ne dépend pas de $[I_1/\mathsf{s}_1]$, peuvent avancer parallèlement dans le pipeline. Et l'exécution de ces deux instructions reste parallèle jusqu'à une "synchronisation" plus tard (souvent à l'étage *commit/write back*). Au moment de la synchronisation, la latence supplémentaire de l'évènement e_0 est *amortie* grâce l'exécution parallèle des autres instructions. Cet amortissement est pris en compte dans le calcul du graphe d'exécution par l'application du *max* entre les temps d'exécution des deux chemins lors de la synchronisation. L'exemple ci-dessus ne porte que sur une instruction mais, en réalité, c'est plutôt plusieurs instructions en parallèle qui peuvent en profiter. Dans l'exemple de la figure 4.3, nous pouvons également observer des occurrences de l'effet d'amortissement pour le calcul de la date du début de $[I_3/EX]$:

$$\begin{aligned}\rho_{[I_3/EX]}(\gamma) &= max(4 + max(1, 10\gamma(IC_0)),\ 3 + max(1, 10\gamma(IC_0)),\\ &\qquad 4 + max(1, 10\gamma(IC_0)) + max(1, 10\gamma(DC_2)))\\ &= 4 + max(1, 10\gamma(IC_0)) + max(1, 10\gamma(DC_2))\\ &= 6 + 9\gamma(IC_0) + 9\gamma(DC_2)\end{aligned}$$

Grâce au *max*, la fonction est largement simplifiée et l'évènement du cache amorti.

Régularité des latences La latence des étages du pipeline est souvent régulière. La latence supplémentaire liée aux évènements est également très souvent la même. Par exemple, l'évènement de cache prend t_{Miss} cycles de pénalité en cas de *Miss*. Cela fait que la contribution des évènements est régulière dans le temps d'exécution final. Par exemple, si le temps d'exécution est de la forme de $t = 1 + max(1, 10e_1) + max(1, 10e_2) + max(1 + 10e_3)$, il y a seulement quatre temps d'exécution possibles (4, 14, 24 et 34), ce qui est inférieur à $2^3 = 8$, le nombre de configurations possibles d'évènements.

En réalité, ces deux effets ont des implications l'un sur l'autre, mais nous pensons que l'effet d'amortissement de parallélisme est plus important dans un processeur superscalaire parce que le parallélisme est plus fréquent, alors que l'effet de la régularité des latences est plus important pour les pipelines monoscalaires.

4.4 Évaluation

Les *effets d'amortissement* du pipeline sont les effets qui ont le plus d'impact sur la taille des XDD et sont donc également des facteurs majeurs pour les performances de l'analyse. Malgré les explications données dans la section précédente, l'effet de ces phénomènes reste dépendant du programme analysé et du pipeline du processeur. Pour mieux les quantifier, nous avons expérimenté notre modèle de calcul sur des benchmarks et des processeurs réalistes afin de mieux appréhender les performances de l'analyse.

4.4.1 Mise en place de l'expérience

L'expérimentation des XDD est réalisée sur l'outil OTAWA. Comme décrit dans la structure de l'analyse de WCET (section 2.3, page 14), les analyses globales ainsi que l'infrastructure du calcul de WCET sont fournies par OTAWA. Dans l'expérience, nous avons seulement implanté l'analyse de pipeline. La performance de la nouvelle méthode est comparée avec *etime*, l'ancienne implantation qui construit le

graphe d'exécution de la même manière mais parcourt explicitement les $2^{|\mathcal{E}|}$ configurations des évènements.

L'expérience est réalisée sur le benchmark TACLe (FALK et al., 2016). À cause de certains problèmes au sein d'OTAWA et par manque de certaines informations de flot, seulement 81% des benchmarks de l'ensemble sont considérés.

Deux micro-architectures sont testées : une représentative des processeurs simples dans le domaine des systèmes embarqués, l'autre plutôt complexe, similaire au *Tricore Aurix*, un processeur très utilisé dans les applications temps-réel automobiles. Nous pensons que ces deux pipelines sont représentatifs des processeurs à exécution dans l'ordre typiquement utilisés dans les systèmes embarqués et nous espérons donc que l'expérience reflètera correctement les performances de notre méthode pour de vraies applications. La spécification de ces deux architectures est détaillée dans la table 4.1.

Simple	Complexe
5 étages : FE, DE, EX, MEM, WB	4 étages : FE, DE, EX, WB
pas de file de fetch	file de fetch de capacité = 3
1 instruction/cycle	3 instructions/cycle (superscalaire)
1 étage d'exécution	1 ALU, 1 FPU, 1 MU (calcul d'adresse compris)
Multiplication = 6 cycles, Division = 12 cycles	Multiplication = 2 cycles, Division = 7 cycles
Addition virgule flottante = 10 cycle	Addition virgule flottante = 1 cycles
Multiplication virgule flottante = 12 cycles	Multiplication virgule flottante = 2 cycles
Division virgule flottante = 37 cycles	Division virgule flottante = 7 cycles
cache d'instruction L1 2-voies 16KB LRU, Hit = 1 cycle, Miss = 8 cycles	
cache de données L1 2-voies 8KB LRU à écriture différée, Hit = 1 cycle, Miss = 8 cycles	

TABLE 4.1 – Détails des micro-architectures expérimentées.

L'expérience est réalisée sur un serveur composé de 8 cœurs Intel Xeon E25 (@2.4GHZ) qui partagent 32GB de RAM. Notre implantation est mono-thread mais plusieurs analyses sont lancées en parallèle sur le serveur (et partagent donc l'ensemble des 32GB de RAM).

Une archive qui contient tous les codes source et des données expérimentales est disponible sur Zenodo [4].

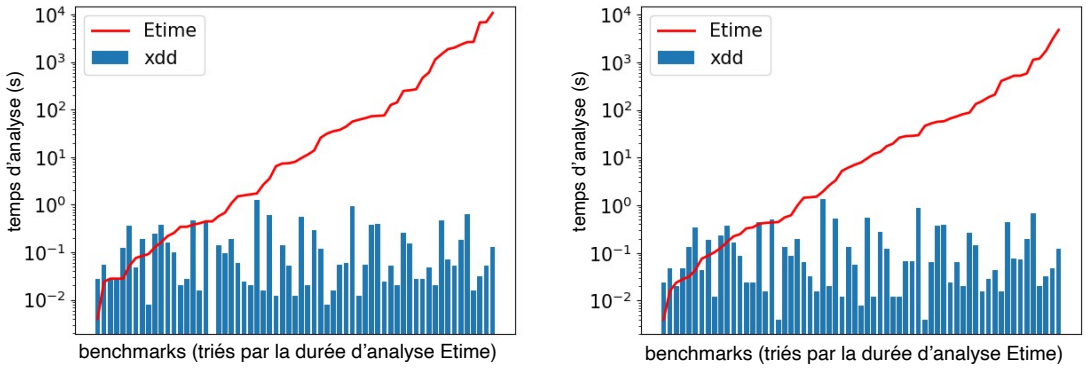
Seuil de coupure L'algorithme exhaustif *etime* a une performance très limitée à cause de la complexité combinatoire du traitement de toutes les configurations. Dans les expériences, nous avons observé qu'il est généralement capable d'analyser, en un temps raisonnable, des BB supportant moins de 15 évènements. Pour comparer avec la nouvelle méthode dans les mêmes conditions, une politique de seuil de coupure est appliquée dans notre article BAI et al., 2020) qui coupe les BB en morceaux de moins de 15 évènements. Dans les expériences où *etime* est comparé avec la nouvelle méthode, ce seuil est appliqué pour les deux approches. Cette technique introduit de la surestimation parce que le contexte entre les morceaux coupés devient moins précis. Au contraire, en utilisant les XDD, la performance est généralement suffisante sans coupure : nous avons d'ailleurs observé une capacité efficace jusqu'à 136 évènements dans un rare graphe d'exécution. Un tel seuil permet de couvrir 99% de cas du benchmark TACLe. Seuls les programmes **rijndael_enc** et **gsm_dec**, qui contiennent certains BB de plus de 300 évènements, n'ont pu être analysés dans un temps raisonnable, même avec la nouvelle méthode, sans appliquer la technique de coupure. Par conséquent, dans les expériences où la nouvelle méthode est évaluée sans comparaison avec *etime*, le seuil choisi est 120 sur l'architecture simple et

4. https://zenodo.org/record/3756621/files/LCTES.tar?download=1

100 sur l'architecture complexe. Dans cette expérimentation, nous avons fixé le seuil statiquement, mais nous envisageons de rajouter la possibilité d'ajuster le seuil dynamiquement en fonction de la taille des XDD durant l'analyse.

4.4.2 Le temps d'analyse

L'expérimentation la plus importante est la comparaison du temps d'analyse entre la nouvelle méthode et l'ancien algorithme exhaustif *etime*. Le résultat est visible sur les figures 4.5A et 4.5B, pour le pipeline simple et complexe, respectivement.



(A) Architecture simple (B) Architecture complexe

FIGURE 4.5 – Le temps d'analyse

L'axe x représente les benchmarks ordonnés par le temps d'analyse selon l'approche *etime*. Il fournit une estimation grossière de la complexité de base (étant donné que *etime* est un algorithme exhaustif). L'axe y représente le temps d'analyse (sans compter les analyses globales) avec une échelle logarithmique. Pour les deux analyses, le seuil de coupure est fixé à 15 . La ligne rouge montre la croissance du temps d'analyse en utilisant *etime* sur l'ensemble de benchmarks et les barres bleus montrent le temps d'analyse en utilisant les XDD sur les mêmes benchmarks.

Nous pouvons constater que l'analyse par *etime* suit une tendance exponentielle sur l'ensemble de benchmarks en prenant 7 minutes pour le pire benchmark. D'un autre côté, le temps d'analyse avec la nouvelle méthode est très faible (moins d'une seconde) sur l'ensemble des benchmarks et ne suit pas une tendance exponentielle. Comme *etime* est exhaustif, son temps d'analyse est exponentiel en fonction du nombre d'évènements dans le BB mais le seuil de coupure, étant positionné à 15, limite l'explosion combinatoire. En fait, le temps d'analyse d'un benchmark dépend majoritairement du nombre d'évènements total du benchmark et du nombre de BB qui ont plus de 15 évènements. Les benchmarks qui prennent le plus de temps, **rijndael_enc** et **statemate**, sont aussi ceux qui ont le plus d'évènements au total et qui ont les plus grands BB. Des données détaillées de l'expérience peuvent être trouvées dans l'archive Zenodo (dont le lien est fourni dans la section précédente).

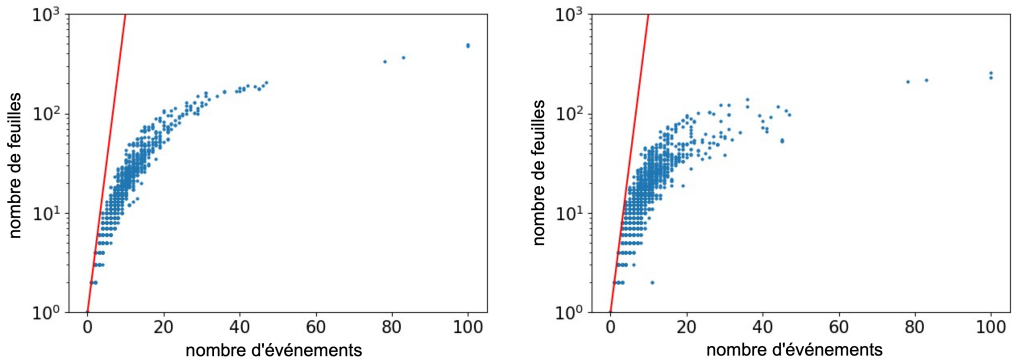
Dans la figure 4.5A, il y a une barre manquante : en fait, l'analyse a pris très peu de temps (moins de 1 ms), temps qui est plus petit que la granularité de l'horloge du système d'exploitation (Linux Ubuntu).

Un petit ensemble de benchmarks (9 sur la figure 4.5A et 10 sur la figure 4.5B) nécessitent un temps d'analyse supérieur en utilisant les XDD. Cette hausse est cependant trop petite en temps absolu pour être représentative, surtout en considérant la précision de l'horloge du système et l'instabilité de la plate-forme de test – des machines virtuelles fonctionnant sur des serveurs.

4.4.3 La compacité des XDD

Les performances en termes de temps d'analyse dépendent de la taille des XDD produits. Celle-ci dépend à son tour du nombre d'occurrences des *effets d'amortissement*. L'évaluation sur le temps d'analyse a déjà donné une estimation qualitative de ces effets : comme l'analyse est assez performante, les *effets d'amortissement* doivent se produire systématiquement. Dans cette section, nous essayons d'évaluer ces effets d'un point de vue quantitatif, par la taille des XDD durant l'analyse.

Lors de cette expérimentation, le nombre de nœuds et de feuilles des XDD sont mesurés dans les XDD résultant de l'analyse par BB. Pour avoir un résultat plus réaliste, le seuil de coupure est mis à 120 pour l'architecture simple et à 100 pour l'architecture complexe [5]. La figure 4.6 montre le nombre de feuilles (donc le nombre de temps d'exécution possibles) dans les XDD obtenus à la fin de l'analyse de chaque BB en fonction du nombre d'évènements dans le BB (4.6A pour l'architecture simple, 4.6B pour l'architecture complexe).



(A) Architecture Simple (B) Architecture Complexe

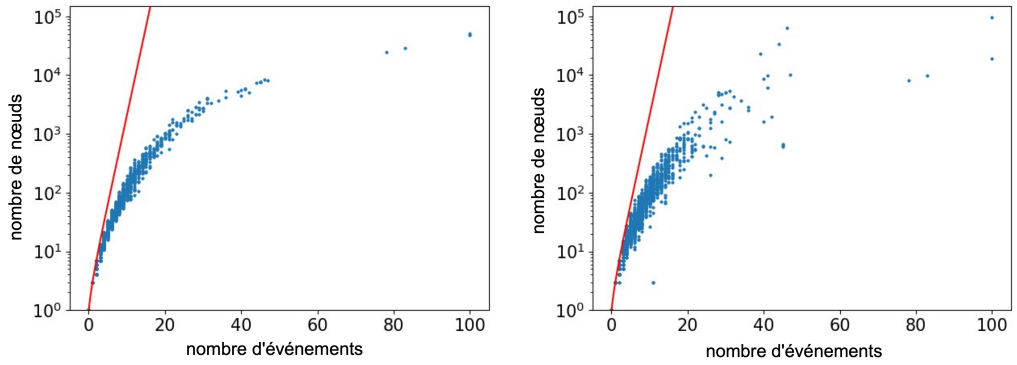
FIGURE 4.6 – Le nombre de feuilles des XDD en fonction du nombre d'évènements.

L'axe x est le nombre d'évènements et l'axe y est le nombre de feuilles selon une échelle logarithmique. Chaque point de la figure représente un XDD résultant de l'analyse de pipeline d'un BB (et de son prédécesseur pour le contexte). La ligne rouge représente la taille des XDD si les effets d'amortissement n'avaient pas lieu ($2^{|\mathcal{E}|}$). Ce résultat montre que, quand le nombre d'évènements augmente, l'écart entre la borne supérieure théorique et le nombre effectif des feuilles s'élargit. Le nombre effectif des feuilles ne suit pas la tendance exponentielle que l'on aurait si on n'utilisait pas de XDD. Cela valide notre supposition que "le nombre de temps d'exécutions possibles est beaucoup plus petit que le nombre de combinaisons possibles des évènements" qui est un phénomène prouvant l'occurrence des *effets d'amortissement*.

Nous avons aussi mesuré le nombre de nœuds et de feuilles total des XDD obtenus à la fin de l'analyse de pipeline sur les BB. Le résultat est présenté à la figure 4.7. L'axe x est le nombre d'évènements du BB, l'axe y est le nombre de nœuds et de feuilles total des XDD avec une échelle logarithmique. La ligne rouge est la borne supérieure théorique si aucun effet d'amortissement ne se produit ($2^{|\mathcal{E}|+1} - 1$). Cette mesure confirme que la taille de XDD est beaucoup plus petite que si l'*effet d'amortissement* ne se produisait pas, ce qui valide encore l'occurrence systématique de ces effets durant l'analyse.

5. Comme il n'y a plus de comparaison avec *etime*, le seuil n'est plus limité à 15.

Les deux expériences montrent des résultats similaires pour les deux architectures, simple et complexe. Nous pouvons cependant voir que les nuages de point sont plus groupés pour l'architecture simple. Cela signifie que la taille des XDD est moins variable. Nous pensons que c'est dû au fait que le pipeline complexe est équipé des ressources superscalaires qui favorisent le parallélisme d'exécution au niveau des instructions et par conséquent crée des modèles d'exécution avec plus de variabilité.



(A) Architecture simple (B) Architecture complexe

FIGURE 4.7 – Le nombre de nœuds des XDD en fonction du nombre d'évènements..

4.5 Conclusion

Dans ce chapitre, nous avons présenté notre version des graphes d'exécution qui analyse le temps d'exécution d'une séquence d'instructions dans le pipeline en considérant les dépendances entre l'exécution des instructions dans les étages. Ces dépendances sont construites en fonction de la structure du pipeline en question. Nous avons montré les règles de construction de certaines dépendances qui sont valides pour la plupart des processeurs d'exécution dans l'ordre.

Ensuite, nous avons exploité les XDD pour représenter les temps d'exécution possibles des nœuds du graphe en fonction des évènements. Comme les XDD fournissent une représentation exacte de la relation entre le temps d'exécution et les configurations des évènements, cette méthode n'introduit pas de surestimation. Ceci étant, l'utilisation des XDD permet de réduire la complexité empirique de l'analyse qui est initialement exponentielle en fonction du nombre d'évènements. Le gain de performance en utilisant les XDD est fortement lié à l'*effet d'amortissement* qui est empiriquement validé par des expériences sur deux micro-architectures typiques et sur les benchmarks TACLe.

Chapitre 5

Amélioration de la méthode IPET

Avec l'analyse de pipeline présentée dans la section précédente, nous sommes capables de calculer tous les temps d'exécution possibles pour chaque BB. Le calcul du WCET du programme entier nécessite maintenant d'introduire ces temps dans le système ILP produit par la méthode IPET. Cependant, l'approche originale considère un seul temps par BB. Il bien sûr possible de prendre le maximum des temps fournis par les XDD mais, dans ce cas-là, l'effort d'analyse et le potentiel gain de précision apportés par le calcul par XDD serait purement et simplement abandonné. Il est donc préférable de modifier la formulation classique d'IPET pour prendre en compte les temps d'exécution fournis par les XDD.

La version classique d'IPET a déjà été présentée dans la section 2.3.3 mais le petit rappel ci-dessous permettra d'expliquer plus facilement notre démarche. Avec IPET, le problème du WCET est représenté par un système ILP dont la fonction objectif est la somme des temps d'exécution de chaque BB multiplié par le nombre d'exécutions du BB sur le pire chemin :

$$WCET = \sum_{a \in V_{\text{CFG}}} (t_a \times x_a) \tag{5.1}$$

Les compteurs d'exécution x_a sont soumis aux contraintes de flot et de bornes de boucle. Le WCET du programme est obtenu en maximisant cette fonction.

5.1 La méthode IPET avec plusieurs temps par BB

La nouvelle analyse de pipeline basée sur les XDD produit tous les temps possibles en fonctions des *configurations* des évènements. Si on les représente trivialement dans le système ILP, la fonction objectif devient :

$$WCET = max \sum_{a \to b \in E_{\text{CFG}}} \left(\sum_{\gamma \in \Gamma_{a \to b}} t^{\gamma}_{a \to b} x^{\gamma}_{a \to b} \right) \tag{5.2}$$

où

— E_{CFG} est l'ensemble des arcs du CFG ;

— $\Gamma_{a \to b}$ est l'ensemble des configurations pour lesquelles le temps d'exécution de b est calculé le long de l'arc $a \to b$. Comme présenté dans la section 4.1, notre modèle de graphe d'exécution prend en compte non seulement chaque BB mais aussi chacun de ses prédécesseurs (pour modéliser le contexte), d'où l'usage de $a \to b \in E_{\text{CFG}}$;

— $t^{\gamma}_{a \to b}$ est le temps d'exécution du BB b précédé par BB a dans la configuration γ.

— $x^{\gamma}_{a \to b}$ est le nombre d'exécutions de l'arc $a \to b$ avec la configuration γ.

Pour maintenir la consistance des contraintes de flots, nous ajoutons une contrainte spécifiant que le nombre total d'exécutions le long d'un arc, $x_{a\to b}$ est égal à la somme de son nombre d'exécutions dans chaque configuration :

$$\forall a \to b \in E_{\text{CFG}}, x_{a\to b} = \sum_{\gamma \in \Gamma_{a\to b}} x^{\gamma}_{a\to b} \tag{5.3}$$

Avec ces modifications, la taille du système ILP devient problématique. D'après notre expérience dans le calcul de WCET, un programme de taille moyenne est composé d'environ 50 000 BBs et 75 000 arcs. Chaque arc, pour un micro-processeur moyen qui a un cache d'instructions et un cache de données, contient environ 8 évènements en moyenne, ce qui correspond à 256 configurations possibles. En conséquence, le système ILP aurait $75,000 \times 256$, c'est-à-dire plus de 19 millions de variables pour couvrir toutes les configurations. À notre connaissance, cette taille n'est gérable par aucun des solveurs d'ILP existants sachant que le problème d'optimisation ILP est de base un problème NP-complet (SCHRIJVER, 1998a). En conséquence, il est indispensable de réduire le nombre de variables tout en essayant de perdre le moins de précision possible.

5.2 Partitionnement des configurations

Pour réduire le nombre de variables, pour chaque arc, l'ensemble des configurations $\Gamma_{a\to b}$ dans le XDD résultat de l'analyse de pipeline est partitionné en $P = \{\Gamma_1, \Gamma_2, ..., \Gamma_n\}$. P doit être un partitionnement, c'est-à-dire qu'il doit satisfaire les conditions suivantes :

— $\Gamma_{a\to b} = \bigcup_{1\leq i\leq n} \Gamma_i$
— $\forall 1 \leq i \neq j \leq n, \Gamma_i \cap \Gamma_j = \emptyset$
— $\forall 1 \leq i \leq n, \Gamma_i \neq \emptyset$

Au lieu d'avoir une variable (compteur) par configuration, on aura, après le partitionnement, seulement un compteur par partie. Le compteur de chaque partie Γ_i représente la somme du nombre d'exécutions de toutes les configurations dans la partie :

$$\forall 1 \leq i \leq n, x_i = \sum_{\gamma \in \Gamma_i} x^{\gamma}_{a\to b} \tag{5.4}$$

Les contraintes de flots dans l'équation 5.3 deviennent donc :

$$x_{a\to b} = \sum_{\gamma \in \Gamma_{a\to b}} x^{\gamma}_{a\to b} = \sum_{1\leq i\leq n} x_i \tag{5.5}$$

De la même manière, les variables x_i doivent aussi remplacer les $x^{\gamma}_{a\to b}$ dans la fonction objectif :

$$WCET = max \sum_{a\to b \in E_{\text{CFG}}} \left(\sum_{1\leq i\leq n_{a\to b}} t_i^{a\to b}\, x_i^{a\to b} \right) \tag{5.6}$$

Comme le partitionnement est fait par BB, nous allons nous focaliser sur un arc particulier $a \to b$ et simplifier la notation : n désignera $n_{a\to b}$, t_i pour $t_i^{a\to b}$ et x_i pour $x_i^{a\to b}$. Il est entendu que les n, t_i et x_i sont différents d'un BB à l'autre.

Pour garantir la validité du WCET obtenu après la ré-écriture du système (en remplaçant les compteurs des configurations par les compteurs des parties), le coefficient/temps $t_i \in \mathbb{N} \cup \{0\}$ [1] de chaque partie doit être supérieur au t^γ de chaque configuration γ de la partie :

$$\forall i \in \mathbb{N}, 1 \leq i \leq n, t_i = \max_{\gamma \in \Gamma_i} t^\gamma_{a\to b} \tag{5.7}$$

La validité du WCET signifie simplement que le WCET ne peut qu'être surestimé (supérieur) après la ré-écriture :

Théorème 5.2.1.

$$\sum_{\gamma \in \Gamma_{a\to b}} t^\gamma_{a\to b} x^\gamma_{a\to b} \leq \sum_{1\leq i\leq n} t_i\, x_i \tag{5.8}$$

Démonstration.

$$\begin{aligned}
\sum_{\gamma \in \Gamma_{a\to b}} t^\gamma_{a\to b} x^\gamma_{a\to b} &= \sum_{1\leq i\leq n} \sum_{\gamma \in \Gamma_i} t^\gamma_{a\to b} x^\gamma_{a\to b} \\
&\leq \sum_{1\leq i\leq n} \sum_{\gamma \in \Gamma_i} t_i\, x^\gamma_{a\to b} && \text{(avec l'équation 5.7)} \\
&= \sum_{1\leq i\leq n} t_i \left(\sum_{\gamma \in \Gamma_i} x^\gamma_{a\to b} \right) \\
&= \sum_{1\leq i\leq n} t_i\, x_i && \text{(avec l'équation 5.4)}
\end{aligned}$$

□

La démonstration est assez triviale. La seule étape nécessitant une explication est l'injection du $\leq$ à la deuxième ligne, qui est vrai parce que t_i est supérieur à tous les $t^\gamma_{a\to b}$ avec $\gamma \in \Gamma_i$ (Équation 5.7).

5.3 Effet de maximisation

Que ce soit avec ou sans partitionnement, le système ILP généré présente une propriété intéressante que nous nommons *effet de maximisation*.

Exemple d'effet de maximisation Étant donné un système ILP avec une fonction objectif,

$$max\ T = ax + by + cz \tag{5.9}$$

où $a < b < c$ sont des coefficients constants et **positifs**, x, y, z sont des variables **positives** et $C \in \mathbb{N}$ constant (cf. la fonction objectif dans l'équation 5.1).

Et la contrainte sur la somme des variables :

$$x + y + z = C \tag{5.10}$$

qui indique que la somme est constante (cf. équation 5.5).

1. Nous avons utilisé $\mathbb{Z}^\infty$ comme domaine de valeurs de sortie des XDD, mais le temps d'exécution de chaque BB finalement obtenu par l'analyse de pipeline est forcément positif ou zéro. Ce n'est pas gênant puisque $\mathbb{N} \subset \mathbb{Z}^\infty$.

Si nous ne posons pas de contraintes supplémentaires sur x, y et z, la solution du système est simplement $T = c\ C$, i.e. en affectant C à z, qui a le coefficient maximal.

Si on rajoute des contraintes, par exemple :

$$\begin{aligned} x &\leq C_x \\ z &\leq C_z \end{aligned} \tag{5.11}$$

en supposant que $C_x \leq C$ et $C_z \leq C$, $C_x, C_z \in \mathbb{N}$, la solution du système devient $c\ C_z + b\ (C - C_z)$. Nous pouvons résumer cet effet par le comportement suivant : comme C, la somme de x, y, z, est connue, nous pouvons commencer par affecter la valeur la plus grande possible à z, qui a le coefficient maximal. Sans contrainte sur z, toute la valeur C est affectée à z. Si z est borné par C_z, C_z est affecté à z et est enlevé de C. La procédure se répète avec le reste des variables dont la somme est C, i.e. $C - C_z$ et avec les variables non affectées. Donc la variable qui a le deuxième plus grand coefficient (y) sera affectée et ainsi de suite jusqu'à l'épuisement du C. Mais dans l'exemple, comme la variable y n'est pas bornée, elle prend tout le reste de la somme $C - C_z$ et la contrainte supplémentaire sur x est finalement inutile.

Cette procédure est appliquée par l'algorithme de *simplex* qui est souvent la base des solveurs ILP (KLEE et al., 1972 ; SCHRIJVER, 1998b). Cependant, nous nous focalisons ici sur un BB particulier : l'algorithme de *simplex*, utilisé pour résoudre le problème de programmation linéaire non-entière, est de complexité polynomiale alors qu'un système ILP quelconque a une complexité exponentielle (car le problème ILP est un problème NP-complet). Cela ne veut pas dire que la résolution complète du système est de complexité polynomiale. Même si les compteurs des configurations de **chaque BB** sont soumis à des contraintes très simples, les variables de flots $x_{a\to b}$ subissent les effets des autres contraintes du système ILP : elles sont liées entre elles par les contraintes de flots du CFG et donc ne profitent pas de l'effet de maximisation. Autrement dit, la résolution du système ILP généré dans l'approche IPET est toujours de complexité exponentielle et *l'effet de maximisation* est valide uniquement pour le sous-système ILP d'un BB donné limité aux compteurs de configuration.

Si on revient à la fonction objectif d'un système ILP pour un BB donné, on considère maintenant un partitionnement strictement ordonné par les temps d'exécution $t_i : P = [\Gamma_1, ...\Gamma_n]$ et $\forall 1 \leq i < n,\ t_i \leq t_{i+1}$. Si la seule contrainte sur les compteurs x_i est $\sum x_i = x_{a\to b}$, par l'effet de maximisation, le compteur x_n (la partie de plus grand coefficient) va prendre toute la quantité affectée à $x_{a\to b}$ lors de la résolution du système ILP et tous les x_i restant sont mis à 0. Le WCET final va être équivalent à avoir une seule partie $[\Gamma^* = \cup_{1\leq i\leq n}\Gamma_i = \Gamma_{a\to b}]$ ce qui revient à l'approche IPET classique qui prend le temps maximum comme temps de BB.

Heureusement, certaines analyses globales peuvent fournir des contraintes sur le nombre d'occurrences de certains évènements :

$$X_e \geq x_e \tag{5.12}$$

où x_e est le compteur d'exécutions des configurations où e est actif et X_e la borne supérieure sur le nombre d'occurrences (activation) de l'évènement e. La borne X est généralement fournie par les analyses globales.

Les évènements pour lesquels une telle borne peut être trouvée sont appelés des évènements bornés. Dans la suite, on considère que les évènements non-bornés ont $X_e = \infty$ comme borne.

Par exemple, pour les accès de cache qui sont classifiés comme *persistants* (qui ont déjà été discutés dans la section 2.3.3, page 16), dans une boucle lp, les *Miss*

se produisent uniquement à la première itération de la boucle lp. Donc, le nombre d'occurrences de l'évènement est égal au nombre de fois où la boucle est démarrée :

$$X_{e(PERS)} = \sum_{edge \in ENTRY(lp)} x_{edge} \tag{5.13}$$

où $ENTRY$ donne l'ensemble des arcs entrants de la boucle lp.

Les x_e sont liés au système de contraintes par les configurations où un évènement est actif. Donc, pour un arc $a \rightarrow b$ donné, x_e est égal au nombre d'occurrences de l'ensemble des configurations où e est actif ($\gamma(e) = TRUE$) :

$$x_e = \sum_{\gamma \in \Gamma_{a \rightarrow b} \wedge \gamma(e)} x^{\gamma}_{a \rightarrow b} \tag{5.14}$$

Ces bornes peuvent être utilisées pour contraindre le nombre d'exécutions d'une partie. Mais il faut tout d'abord explorer l'effet des évènements non-bornés.

Une configuration qui contient soit des évènements non-bornés, soit des évènements inactifs est dite non-bornée car il n'y a pas de contrainte supplémentaire sur son compteur :

$$unbounded(\gamma) \Leftrightarrow \forall e \in \mathcal{E}, \neg\gamma(e) \vee X_e = \infty \tag{5.15}$$

Par conséquent, une partie qui contient au moins une configuration non-bornée est aussi non-bornée :

$$unbounded(\Gamma^*) \Leftrightarrow \exists \gamma^* \in \Gamma^*, unbounded(\gamma^*) \tag{5.16}$$

Puisque le compteur de la partie Γ^*, x^*, est supérieur à x^{γ^*} (le compteur de la configuration non-bornée γ^*), et que x^{γ^*} est non-borné, donc x^* est non-borné. Dans ce cas, la seule contrainte sur x^* est la somme des compteurs de toutes les parties égales à l'exécution totale : $\sum x_i = x_{a \rightarrow b}$ où x^* est un des x_i.

Soit $P = [\Gamma_1, ..., \Gamma_n]$ un partitionnement ordonné par t_i avec la partie Γ_j non-bornée et $\Gamma_{j<i\leq n}$ des partitions bornées, *l'effet de maximisation* va affecter aux $x_{j<i\leq n}$ la plus grande quantité possible de $x_{a \rightarrow b}$ en fonction de leurs contraintes d'évènement et x_j, n'étant pas borné, recevra le reste de $x_{a \rightarrow b}$, i.e. $x_j = x_{a \rightarrow b} - \sum_{j<i\leq n} x_i$. Les variables $x_{i<j}$ sont donc mises à zéro par *l'effet de maximisation*. Comme elles sont mises à zéro, la solution du système ILP sans elles reste la même ce qui est équivalent au partitionnement :

$$\left[\left(\Gamma^* = \bigcup_{1 \leq i \leq j} \Gamma_i \right), \Gamma_{j+1}, ..., \Gamma_n \right]$$

où Γ^* est appelé *partie nullifiée*. Nous définissons la *partie de nullifiée* comme la partie qui contient toutes les configurations dont le compteur est mis à zéro à cause de l'effet de maximisation sauf une partie de configuration qui a le coefficient maximal parmi toutes les configurations de la partie et dont le compteur prend la valeur du reste de $x_{a \rightarrow b}$ après la soustraction des compteurs des parties bornées.

La *partie nullifiée* peut être qualifiée par la configuration non-bornée qui a le temps maximal t^* parmi toutes les configurations non-bornées :

$$\Gamma^* = \{\gamma \in \Gamma_{a \rightarrow b} \mid t^{\gamma}_{a \rightarrow b} \leqslant t^*\} \text{ avec } t^* = \max_{\gamma \in \Gamma_{a \rightarrow b} \wedge unbounded(\gamma)} t^{\gamma}_{a \rightarrow b} \tag{5.17}$$

En effet, si γ^* est la configuration non-bornée avec le temps (coefficient) maximal,

même si une autre configuration ayant un coefficient plus petit est bornée, il est quand même nullifié à cause de l'*effet de maximisation*.

Avec la *partie de nullifiée*, toutes les configurations nullifiées n'ont pas besoin de variables dans le système ILP (car elles vaudront toutes 0) et seule la variable x^{γ^*} est créée, ce qui permet de réduire un peu le nombre de variables du système ILP.

5.4 Borner les compteurs de parties

Dans la section précédente, nous avons montré comment supprimer des variables du système ILP en regroupant celles qui étaient nullifiées. Dans cette section, nous allons voir comment créer le système ILP avec des contraintes sur les parties/-configurations bornées qui ne sont donc pas contenues dans la partie nullifiée.

Tout d'abord, les configurations bornées et les parties bornées sont définies comme l'opposé des configurations et des parties non-bornées. Pour qu'une configuration soit bornée, il doit exister au moins un évènement actif et borné. Pour qu'une partie soit bornée, elle doit contenir uniquement des configurations bornées. Mais cela ne signifie pas qu'une partie bornée est bornée par l'union des évènements bornant chaque configuration de la partie car les configurations peuvent être bornées par des évènements différents. En fonction des évènements qui bornent une partie, elle peut être classifiée en trois catégories :

Entièrement bornée Une partie est entièrement bornée par un évènement $e \in \mathcal{E}$ si e est borné et si e est actif dans toutes les configurations de la partie. $\mathcal{E}_{\Gamma_t}^{Entire}$ donne l'ensemble des évènements qui borne entièrement la partie Γ_t et est défini par :

$$\mathcal{E}_{\Gamma_t}^{Entire} = \{e \in \mathcal{E}_{a\to b} \mid X_e \neq \infty \wedge (\forall \gamma \in \Gamma_t, \gamma(e))\} \tag{5.18}$$

Dans la suite, nous disons qu'une partie est entièrement bornée (sans préciser l'évènement) si elle est entièrement bornée par au moins un évènement, notée $entirely\_bounded(\Gamma_t)$:

$$entirely\_bounded(\Gamma_t) \Leftrightarrow \mathcal{E}_{\Gamma_t}^{Entire} \neq \emptyset \tag{5.19}$$

Partiellement bornée Une partie est partiellement bornée par un évènement $e \in \mathcal{E}$ si e est borné et s'il est actif dans au moins une configuration. $\mathcal{E}_{\Gamma_t}^{Partial}$ donne l'ensemble d'évènement qui borne partiellement Γ_t et est défini par :

$$\mathcal{E}_{\Gamma_t}^{Partial} = \{e \in \mathcal{E}_{a\to b} \mid X_e \neq \infty \wedge (\exists \gamma \in \Gamma_t, \gamma(e))\} \tag{5.20}$$

On peut voir que $\mathcal{E}_{\Gamma_t}^{Entire} \subseteq \mathcal{E}_{\Gamma_t}^{Partial}$.

Une partie Γ_t est partiellement bornée (sans préciser l'évènement) si elle est bornée par au moins un évènement et si la partie ne contient pas de configuration non-bornée :

$$partially\_bounded(\Gamma_t) \leftrightarrow \mathcal{E}_{\Gamma_t}^{Partial} \neq \emptyset \wedge \neg unbounded(\Gamma_t) \tag{5.21}$$

Non-bornée Une partie est non-bornée si elle contient au moins une configuration non-bornée comme discuté dans la section précédente. Cette catégorie n'est pas soumise aux contraintes du système ILP parce que toutes les configurations non-bornées sont regroupées dans la partie nullifiée qui est **la seule partie non-bornée dans la partition**.

5.4.1 Contraintes sur les parties entièrement bornées

Les contraintes sur les compteurs de chaque partie bornée peuvent maintenant être créées en fonction des bornes fournies par les évènements. Nous rappelons que le nombre d'exécutions de l'évènement e (x_e) est égal la somme des compteurs des configurations où e est actif (équation 5.14). Par conséquent, si une partie est entièrement bornée par e, elle est un sous-ensemble de l'ensemble des configurations qui ont e actif et leur nombre d'exécution est inférieur à x_e. Nous pouvons ainsi en déduire que :

$$\begin{aligned} & x_e = \sum_{\gamma \in \Gamma_{a\to b} \ \wedge \ \gamma(e)} x^{\gamma}_{a\to b} \\ \iff & x_e = \left(\sum_{\gamma \in \Gamma_i \ \wedge \ \gamma(e)} x^{\gamma}_{a\to b} \right) + \left(\sum_{\gamma \in (\Gamma_{a\to b} \setminus \Gamma_i) \ \wedge \ \gamma(e)} x^{\gamma}_{a\to b} \right) \\ \text{si } e \in \mathcal{E}^{Entire}_{\Gamma_i} \iff & x_e = x_i + \left(\sum_{\gamma \in (\Gamma_{a\to b} \setminus \Gamma_i) \ \wedge \ \gamma(e)} x^{\gamma}_{a\to b} \right) \end{aligned} \tag{5.22}$$

Cette procédure peut être répétée pour chaque partie qui est entièrement bornée par e ce qui donne la contrainte suivante :

$$x_e = \left(\sum_{j<i\leq n \ \wedge \ e \in \mathcal{E}^{Entire}_{\Gamma_i}} x_i \right) + \left(\sum_{\gamma \in (\Gamma_{a\to b} \setminus B_i) \ \wedge \ \gamma(e)} x^{\gamma}_{a\to b} \right) \tag{5.23}$$

où B_i est l'ensemble des configurations inclues dans les parties qui sont entièrement bornées par e, i.e.

$$B_i = \bigcup_{j<i\leq n \ \wedge \ e \in \mathcal{E}^{Entire}_{\Gamma_i}} \Gamma_i \tag{5.24}$$

avec $\gamma \in (\Gamma_{a\to b} \setminus B_i)$ les configurations qui ne sont pas regroupées dans les parties entièrement bornées.

5.4.2 Contraintes sur les parties partiellement bornées

Si la partie Γ_i est partiellement bornée mais pas entièrement bornée ($\mathcal{E}^{Partial}_{\Gamma_t} \neq \emptyset \wedge \mathcal{E}^{Entire}_{\Gamma_t} = \emptyset \wedge \neg unbounded(\Gamma_t)$), x_i ne peut pas correspondre à un sous-ensemble de configurations qui sont toutes bornées par certains évènements et donc les contraintes ne peuvent pas être générées directement comme dans le cas entièrement borné. Ces configurations restent sous la forme $x^{\gamma}_{a\to b}$ dans l'équation 5.23.

Puisque nous sommes dans un système ILP dont tous les coefficients et toutes les variables sont positifs, plus on a de contraintes, plus précis sera le WCET. Pour profiter quand même les bornes des évènements qui bornent partiellement des parties, deux techniques peuvent être appliquées :

Supprimer la variable d'une contrainte. Dans le système ILP généré pour notre problème de WCET, supprimer un terme à droite d'une contrainte de l'équation 5.14 (page 69) et passer de l'égalité à l'inégalité est une modification *valide* :

$$x_e = \sum_{\gamma \in \Gamma_{a\to b} \wedge \gamma(e)} x^{\gamma}_{a\to b} \tag{5.25}$$

$$\Rightarrow x_e \geq \sum_{\gamma \in (\Gamma_{a\to b} \setminus \gamma_j) \wedge \gamma(e)} x^{\gamma}_{a\to b} \tag{5.26}$$

où γ_j est la configuration dont le compteur est supprimé de la contrainte.

Cette opération est valide parce que tous les $x^{\gamma}_{a\to b}$ sont positifs et donc en supprimer certains seulement relaxe les contraintes associées. Du point de vue d'un polyèdre représenté par les contraintes, cette modification ne fait que l'agrandir et donc sa solution ne peut être que plus grande ou égale. Néanmoins, il est important de noter que cette opération est valide uniquement pour le système ILP au sein de la méthode IPET dans notre situation : cette propriété ne peut pas être généralisée pour le problème ILP en général car on n'est pas sûr de toujours agrandir le polyèdre.

Nous pouvons utiliser cette technique pour supprimer des variables dans l'équation 5.23 : les variables de la partie droite de l'équation ($x^{\gamma}_{a\to b}$) ne sont pas contenues dans des parties entièrement bornées. Comme nous n'avons pas pu leur trouver des contraintes supplémentaires [2], elles sont donc nullifiées [3]. On obtient alors la contrainte :

$$x_e \geq \sum_{j<i\leq n \ \wedge \ \forall \gamma \in \Gamma_i, \gamma(e)} x_i \tag{5.27}$$

où les variables nullifiées ($x^{\gamma}_{a\to b}$) sont supprimées de la contrainte. Comme certains termes nuls ou positifs sont enlevés à droite de l'équation (seule la variable nullifiée de plus grand coefficient n'est pas nulle), l'égalité devient une inégalité mais la solution est toujours valide (éventuellement surestimée). Cette propriété est aussi utilisée dans la technique pour *Diviser une partie* présentée ci-dessous.

Diviser une partie Si une partie Γ_i est partiellement mais pas entièrement bornée, nous pouvons la diviser en sous-parties $\Gamma_{i.e_1}$, ..., $\Gamma_{i.e_m}$ dont chaque sous-partie est entièrement bornée par un évènement $e_1, e_2, ..., e_m$. Cela consiste à modifier la partition initiale des configurations en :

$$[\Gamma^*, \Gamma_{j+1}, ..., \Gamma_i, \Gamma_{i+1}, ..., \Gamma_n] \rightsquigarrow [\Gamma^*, \Gamma_{j+1}, ..., \Gamma_{i.e_1}, ..., \Gamma_{i.e_m}, \Gamma_{i+1}, ..., \Gamma_n]$$

où la partie Γ_i est divisée en fonction des évènements $e_1, ..., e_m$.

Les compteurs des sous-parties $x_{i.e_k}$ sont insérés dans les contraintes d'évènements (équation 5.27) parce que, maintenant, les parties obtenues sont entièrement bornées. Cependant, nous devons préciser que cette technique introduit un nombre considérable de nouvelles variables (plus de variables que si on ne divisait pas, mais moins que sans partitionnement) et qu'elle peut amener de la surestimation. Pour mieux comprendre pourquoi il y a cette surestimation, considérons l'exemple ci-dessous.

La partie Γ_i est composée de trois configurations : $\gamma_1 = \overline{e_1}e_2, \gamma_2 = e_1\overline{e_2}, \gamma_3 = e_1e_2$ (la barre indique si l'évènement est inactif ou non dans la configuration). Γ_i n'est

2. Dans la suite, nous allons utiliser des évènements bornant partiellement ces parties pour créer des contraintes.

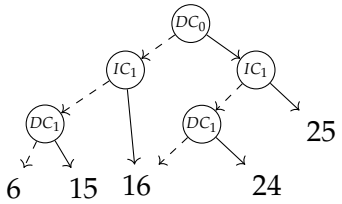
3. Cette technique peut également être utilisée pour réduire le nombre de variables, aux dépens de la précision bien sûr.

donc pas entièrement bornée car ni e_1, ni e_2 ne sont actifs pour toutes les configurations. Par conséquent, on peut la décomposer en $\Gamma_{i.e_1}$ qui est entièrement bornée par e_1 et $\Gamma_{i.e_2}$ qui est entièrement bornée par e_2. Cependant, l'effet de cette division est que γ_3 est bornée à la fois par e_1 et e_2. Comme la division de la partie doit toujours être un partitionnement, le γ_3 doit être mis dans l'une des deux sous-parties. Si nous la mettons dans $\Gamma_{i.e_1}$, elle sera manquante dans $\Gamma_{i.e_2}$. Ainsi, dans la contrainte sur e_2, i.e. l'équation 5.22 pour e_2, γ_3 doit être supprimée car γ_3 fait partie de $\gamma \in \Gamma_{a\to b} \ \wedge \ \gamma(e)$. La règle de *supprimer la variable d'une contrainte* est appliquée, l'égalité deviendra $\geq$ et le résultat pourra être surestimé. Symétriquement, si nous mettons γ_3 dans $\Gamma_{i.e_2}$, la surestimation est présente de la même manière mais dans la contrainte du e_1. Finalement, nous devons noter que, malgré l'éventuelle surestimation introduite durant cette procédure pour borner les parties partiellement bornées, la précision est quand même améliorée par rapport au cas où aucune contrainte n'est générée pour cette partie. Il faut noter que, si cette division n'est pas réalisée, la partie serait simplement nullifiée et son nombre d'exécutions serait simplement compté comme une partie non-bornée avec un coefficient (temps d'exécution) maximal.

5.5 Génération du système ILP avec des XDD

Après avoir discuté des propriétés des systèmes ILP avec partitionnement, nous présentons ici la génération du système – ou comment orienter notre partitionnement avec les XDD.

L'analyse du pipeline produit donc des XDD, qui contiennent tous les temps d'exécution possibles des BB en fonction des configurations. Dans un XDD, une feuille $\text{LEAF}(k)$ et tous les chemins de sa racine jusqu'à cette feuille représentent l'ensemble des configurations pour lesquelles le temps d'exécution est k. Cela amène naturellement à un partitionnement en fonction du temps. La figure 5.1 donne l'exemple d'un XDD avec le partitionnement des configurations par le temps. Les parties sont notées avec, pour indice, le temps d'exécution. Les configurations dans chaque partie sont notées sous forme de chemin dans le XDD et comme le XDD ne représente pas les nœuds qui n'ont pas d'effet (propriété de *compacité*), les évènements qui n'ont pas d'effet sont naturellement omis. Par exemple $\{\overline{DC_0}.IC_1\}$ représente les configurations $\{\overline{DC_0}.IC_1.DC_1, \overline{DC_0}.IC_1.\overline{DC_1}\}$.



DC_1 is unbounded.

$$
\begin{aligned}
P &= \{\Gamma_6, \Gamma_{15}, \Gamma_{16}, \Gamma_{24}, \Gamma_{25}\} \\
\Gamma_6 &= \{\overline{DC_0}.\overline{IC_1}.\overline{DC_1}\} \\
\Gamma_{15} &= \{\overline{DC_0}.\overline{IC_1}.DC_1\} \\
\Gamma_{16} &= \{\overline{DC_0}.IC_1, DC_0.\overline{IC_1}.\overline{DC_1}\} \\
\Gamma_{24} &= \{DC_0.\overline{IC_1}.DC_1\} \\
\Gamma_{25} &= \{DC_0.IC_1\}
\end{aligned}
$$

FIGURE 5.1 – Exemple de représentation des temps par un $\mathcal{XDD}$.

Nous avons déjà démontré dans la section 5.3 que la *partie nullifiée* doit regrouper les configurations dont le compteur est mis à zéro dans la résolution du système ILP ce qui, par conséquent, réduit le nombre de variables. Partitionner par le temps facilite la procédure pour trouver la *partie nullifiée* parce qu'elle est directement déterminée par la configuration non-bornée qui a le temps maximal. Justement, la structure du XDD fournit un très bon support pour réaliser cette recherche : un XDD étant

un DAG, nous pouvons parcourir tous ses chemins. Lors du parcours, on sait qu'un évènement est inactif sur un chemin si on passe par la branche gauche d'un nœud contenant cet évènement et qu'il est actif si on passe par la branche droite. Si un chemin de la racine à une feuille $\text{LEAF}(k)$ contient seulement des évènements inactifs ou non-bornés, ce chemin représente **au moins une** configuration non-bornée (parce que les nœuds sans effet sont éliminés dans le XDD, un chemin peut représenter plusieurs configurations). Un tel chemin est appelé un *chemin non-borné*. Maintenant, le problème de trouver la configuration non-bornée qui a le temps maximal (γ^*) devient un problème de graphe qui est de trouver le k maximal des feuilles $\text{LEAF}(k)$ qui est le nœud final d'un *chemin non-borné*. Dans l'exemple de la figure 5.1, supposons que seul DC_1 soit non-borné, la *partie nullifée* est $\Gamma_6 \cup \Gamma_{15}$: les autres parties sont bornées par DC_0 ou IC_1.

Après avoir regroupé les parties par le temps et avoir déterminé la *partie nullifiée*, nous avons à déterminer les bornes des parties restantes (c'est-à-dire $\mathcal{E}^{Entire}$ et $\mathcal{E}^{Partial}$ des parties). Comme le parcours de tous les chemins d'un XDD peut quand même consommer du temps (de complexité linéaire en fonction du nombre de nœuds et de feuilles, et de complexité exponentielle en fonction du nombre d'évènements), nous proposons une représentation des chemins des XDD par l'ensemble des évènements bornés et actifs sur le chemin, c'est-à-dire un chemin π est dans le domaine $\wp(\mathcal{E})$. Pour simplifier la présentation, nous définissons la fonction Ω_t qui collecte l'ensemble de chemins de la racine jusqu'à une feuille d'un XDD $\text{LEAF}(t)$:

Définition 5.5.1. $\forall f^\# \in \mathcal{XDD}, t \in \mathbb{N}, \Omega_t : \wp(\mathcal{E}) \times \mathcal{XDD} \to \wp(\wp(\mathcal{E}))$

$$\Omega_t(\pi, f) = \begin{cases} \{\pi\} & if\ f^\# = \text{LEAF}(t) \\ \{\emptyset\} & if\ f^\# = \text{LEAF}(k) \wedge k \neq t \\ \Omega_t(\pi \cup \{e\}, g) \cup \Omega_t(\pi, \overline{g}) & if\ f^\# = \text{NODE}(e, \overline{g^\#}, g^\#) \wedge X_e \neq \infty \\ \Omega_t(\pi, g) \cup \Omega_t(\pi, \overline{g}) & if\ f^\# = \text{NODE}(e, \overline{g^\#}, g^\#) \wedge X_e = \infty \end{cases}$$

La fonction Ω_t traverse un XDD depuis la racine vers les feuilles en enregistrant les évènements actifs et bornés sur chaque chemin, jusqu'à atteindre la feuille $\text{LEAF}(t)$. Ici la définition de cette fonction est récursive mais elle est réellement implémentée comme un algorithme de parcours BFS (Breadth-First Search) de l'arbre de XDD en propageant l'ensemble des évènements représentant les chemins de la racine jusqu'aux feuilles. Ω_t a donc une complexité logarithmique en fonction du nombre de nœuds.

Finalement, pour une feuille $\text{LEAF}(t)$, et la partie correspondante Γ_t, les informations de bornes sont :

$$entirely\_bounded(\Gamma_t) \Leftrightarrow \mathcal{E}^{Entire}_{\Gamma_t} \neq \emptyset \Leftrightarrow \left(\bigcap_{\pi \in \Omega_t(\emptyset, root)} \pi \right) \neq \emptyset \tag{5.28}$$

$$unbounded(\Gamma_t) \Leftrightarrow \emptyset \in \Omega_t(\emptyset, root) \tag{5.29}$$

$$partially\_bounded(\Gamma_t) \Leftrightarrow \neg unbounded(\Gamma_t) \wedge \neg entirely\_bounded(\Gamma_t) \tag{5.30}$$

La *partie nullifiée*, initialement inconnue, est également déterminée durant le parcours par Ω_t dans l'ordre décroissant de la valeur des feuilles. Une fois que la première configuration non-bornée $unbounded(\Gamma_t)$ est trouvée, l'analyse s'arrête parce

que Γ_t ainsi que tous les $\Gamma_i | i < t$ forment l'ensemble de la *partie nullifiée*. Les évènements qui bornent ces parties peuvent également être calculés à partir de Ω_t :

$$\mathcal{E}^{Entire}_{\Gamma_t} = \bigcap_{\pi \in \Omega_t(\emptyset, root)} \pi \tag{5.31}$$

$$\mathcal{E}^{Partial}_{\Gamma_t} = \bigcup_{\pi \in \Omega_t(\emptyset, root)} \pi \tag{5.32}$$

Maintenant que toutes les informations nécessaires sont prêtes, nous pouvons générer le système ILP. Tout d'abord, rappelons qu'avant le partitionnement, les contraintes initiales sont données dans l'équation 5.14 :

$$x_e = \sum_{\gamma \in \Gamma_{a \to b} \wedge \gamma(e)} x^{\gamma}_{a \to b}$$

Pour chaque partie Γ_t partitionnée en fonction du temps, nous appliquons le traitement suivant :

Cas 1 – Γ_t entièrement bornée. Γ_t est un sous-ensemble des configurations de $\Gamma_{a \to b}$ où $e \in \mathcal{E}^{Entire}_{\Gamma_t}$ est actif :

$$\forall e \in \mathcal{E}^{Entire}_{\Gamma_t}, \Gamma_t \subset \{\gamma \in \Gamma_{a \to b} | \gamma(e)\} \tag{5.33}$$

Cependant, pour un évènement e, il peut exister plusieurs parties qui sont entièrement bornées par e. En considérant les compteurs liés à e (équation 5.22), toutes ces parties partageant sont regroupées dans une seule contrainte :

$$\forall e \in \mathcal{E}, \sum_{\Gamma_t \in P^{Entire}_e} x_t \leq x_e \tag{5.34}$$

où P^{Entire}_e est l'ensemble de parties entièrement bornées par e et qui partagent l'évènement e :

$$\forall e \in \mathcal{E}, P^{Entire}_e = \{\Gamma_t \in P \mid e \in \mathcal{E}^{Entire}_{\Gamma_t}\} \tag{5.35}$$

Cette façon de générer les contraintes est valide car certains termes positifs sont enlevés de la contrainte dans l'équation 5.23 : les compteurs des parties qui ne sont pas entièrement bornées par e sont supprimés de la partie droite de l'équation par la technique *supprimer une variable d'une contrainte*. Mais cela peut introduire une surestimation parce qu'il peut exister des compteurs non-nullifiés parmi les compteurs enlevés. En relaxant la contrainte, la solution du système ILP peut ainsi être supérieure. Par exemple, dans la figure 5.1, Γ_{24} et Γ_{25} sont entièrement bornées par DC_0 mais la contrainte correspondante générée dans le système est $x_{24} + x_{25} \leq x_{DC_0}$. Γ_{16} contient aussi une configuration bornée par DC_0 mais qui n'est pas prise en compte dans la contrainte. Par conséquent, nous allons essayer de borner Γ_{16} comme une partie partiellement bornée par DC_0.

Cas 2 – Γ_t est partiellement bornée. Ce cas est plus compliqué parce que le compteur x_t de la partie Γ_t partiellement bornée regroupe des configurations qui sont bornées par des évènements différents, c'est-à-dire que nous ne pouvons pas remplacer tous les compteurs par x_t. Pour éviter ce problème, la technique de *Diviser une partie* est appliquée, ce qui produit sur Γ_t des sous-ensembles $\{\Gamma_{t.e} \mid e \in \mathcal{E}^{Partial}_{\Gamma_t}\}$ qui sont

entièrement bornés. Comme chaque $\Gamma_{t.e}$ est bornée par e, les contraintes peuvent être générées de la même manière qu'avec des parties entièrement bornées.

Dans l'exemple de la figure 5.1, Γ_{16} est partiellement bornée par DC_0 et IC_1, donc Γ_{16} est divisée en $\Gamma_{16.DC_0}$ et $\Gamma_{16.IC_1}$. Comme Γ_{24} et Γ_{25} sont entièrement bornées par DC_0 et Γ_{25} par IC_1. Les contraintes générées sont :

$$\begin{aligned} x_{16.DC_0} + x_{24} + x_{25} &\leq x_{DC_0} \\ x_{16.IC_1} + x_{25} &\leq x_{IC_1} \end{aligned} \tag{5.36}$$

Cas 3 – Γ_t est non-bornée. Comme décrit dans la section 5.5, les informations de bornes sont déterminées dans l'ordre décroissant de la valeur des feuilles et à la première rencontre d'un chemin non-borné, la partie de nullifiée Γ^* est identifiée. Son compteur x^* est seulement borné par le nombre total d'exécutions de $a \rightarrow b$. Donc, la contrainte de flot est générée sous la forme :

$$x_{a\rightarrow b} = x^* + x_{16.DC_0} + x_{16.IC_1} + x_{24} + x_{25} \tag{5.37}$$

Finalement, le système ILP complet généré pour pour l'arc $a \rightarrow b$:

$$\begin{aligned} Obj : max \;\; X_{a\rightarrow b} &= 15x^* + 16x_{16.DC_0} + 16x_{16.IC_1} + 24x_{24} + 25x_{25} \\ x_{a\rightarrow b} &= x^* + x_{16.DC_0} + x_{16.IC_1} + x_{24} + x_{25} \\ x_{16.DC_0} + x_{24} + x_{25} &\leq x_{DC_0} \\ x_{16.IC_1} + x_{25} &\leq x_{IC_1} \end{aligned} \tag{5.38}$$

5.6 Évaluation

L'algorithme de génération de système ILP décrit précédemment a pour but de réduire le nombre de variables et en même temps de préserver la précision du WCET au maximum. Cette section évalue cette approche en la comparant avec l'algorithme existant dans OTAWA nommé *etime* et avec deux algorithmes extrêmes, *groupement exhaustif* et *groupement maximum* :

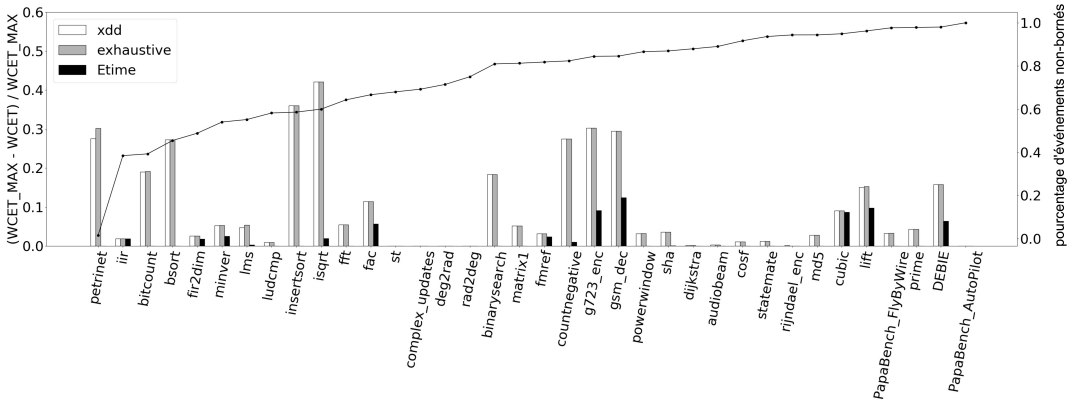
- *etime* – L'algorithme original d'OTAWA partitionne les configurations en deux parties, *High Time Set* et *Low Time Set*. Nous pouvons intuitivement les voir respectivement comme une partie non-nullifiée et une partie nullifiée. Il détermine ce partitionnement en triant les configurations par le temps et de manière à ce qu'une estimation heuristique de la solution du WCET final soit minimale. Il est seulement capable de produire des parties complètement bornées par des évènements (pas de division de partie).
- *Groupement exhaustif* – Cet algorithme ne réalise pas de partitionnement. En réalité, il est impossible de créer $2^{|\mathcal{E}|}$ variables pour **toutes les configurations** car le solveur ILP est incapable d'en supporter autant. Par conséquent, nous créons une variable par **chemin** du XDD mais sans appliquer la technique de partitionnement présentée dans ce chapitre. Donc, comparé à l'algorithme avec partitionnement, cet algorithme ne fait pas de surestimation mais a un temps de calcul qui devrait être plus élevé.
- *Groupement maximum* – Cela correspond à la version originale de la méthode IPET : un seul temps est utilisé par BB. En fait, du point de vue de notre nouvel algorithme, c'est comme s'il y avait qu'une seule partie, la *partie nullifiée*.

Bien évidemment, cet algorithme minimise la taille du système ILP mais produit le WCET le plus surestimé parmi tous les algorithmes.

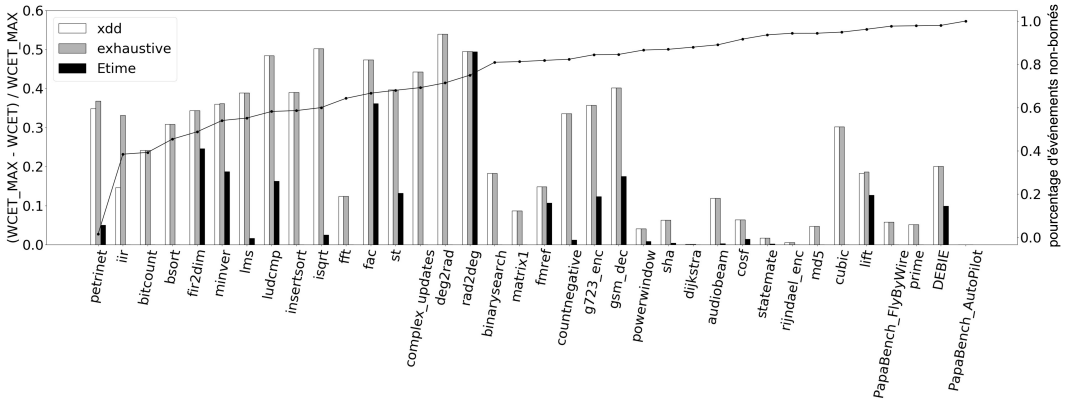
Ces quatre algorithmes sont expérimentés sur les mêmes benchmarks et les mêmes micro-architectures considérées dans l'expérimentation du graphe d'exécution avec des XDD décrite dans la table 4.1 (page 60). Le *seuil de coupure* (présenté également à la la section 4.4.1, page 60) est mis à 15 à cause de la limitation induite par les performances d'*etime*.

5.6.1 Précision

Tout d'abord, la précision est comparée. L'amélioration de précision est calculée avec comme base le WCET du *groupement maximum* : $(w_{max} - w_g)/w_{max}$ où w_{max} est le WCET du *groupement maximum* et w_g est le WCET des autres algorithmes (sauf le *groupement maximum* bien sûr). Les résultats sont présentés sur la figure 5.2 dont l'axe x est le nom des benchmarks et l'axe y est le gain de précision (plus le résultat est grand, meilleure est la précision).



(A) Architecture simple



(B) Architecture complexe

FIGURE 5.2 – Amélioration du WCET.

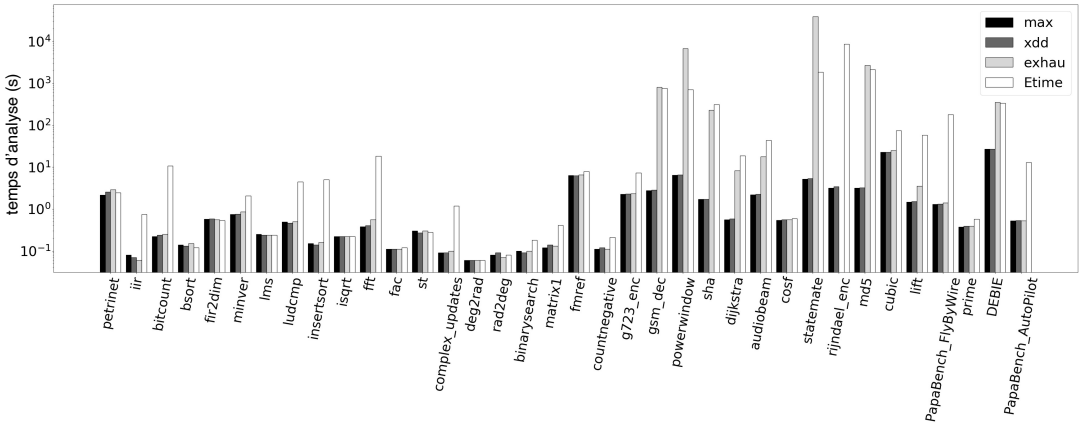
Cette expérience montre que le *groupement par XDD* (l'approche présentée dans ce chapitre) et le *groupement exhaustif* fournissent une précision similaire sur approximativement 80% des benchmarks. Ces deux algorithmes produisent de meilleurs WCET que le *groupement maximum* et *etime*. Comparé au *groupement maximum*, le gain de précision maximal est de 42.1% sur le benchmark *isqrt* pour l'architecture simple et 53.9% sur le benchmark *deg2rad* pour l'architecture complexe. L'algorithme original d'OTAWA, *etime* a une précision proche du *groupement maximum* sur la plupart des benchmarks pour l'architecture simple. Pour l'architecture complexe, *etime* est bien meilleur que le *groupement maximum* sur une dizaine de benchmarks mais a une précision similaire pour le reste des benchmarks. Par rapport à *etime*, le *groupement par XDD* apporte un gain significatif sur une dizaine de benchmarks pour l'architecture simple et sur la plupart de benchmarks pour l'architecture complexe. Nous pouvons ainsi conclure que le *groupement par XDD* est bien meilleur en termes de précision que *etime* et le *groupement maximum*, et d'une précision très proche du *groupement exhaustif*.

Cette évaluation de la précision basée sur le WCET final dépend fortement du résultat des analyses globales et si ces analyses produisent des bornes pour la plupart des évènements. Ces bornes sont nécessaires à un fonctionnement efficace et un WCET précis pour le *groupement exhaustif* et le *groupement par XDD*. Au contraire, si peu d'informations pour les bornes des évènements sont produites, les différentes stratégies de partitionnement ne peuvent pas compenser le manque d'information.

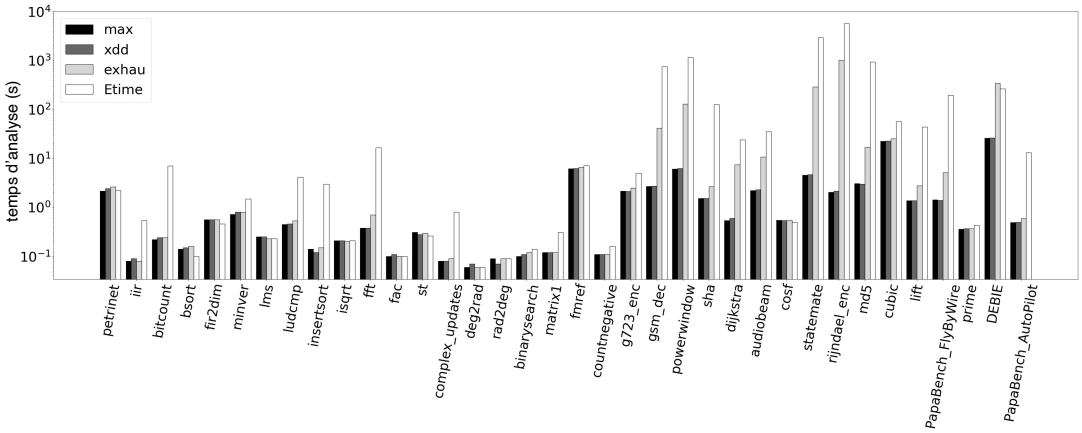
Nous pouvons visualiser cet effet par la ligne présente dans les sous-figures de 5.2 qui donnent le pourcentage d'évènements non-bornés (le taux est affiché sur l'axe y à droite). Nous pouvons constater que plus il y a d'évènements non-bornés, plus la précision du *groupement exhaustif* et du *groupement par XDD* se rapproche de la précision des deux autres méthodes qui ne prennent pas, ou mal, en compte les bornes des évènements. Les benchmarks qui ont le plus d'évènements non-bornés sont à droite de la figure et on peut voir que, dans ces cas-là, la précision est fortement dégradée. Mais cette corrélation n'est pas absolue, il y a aussi des benchmarks qui ont peu d'évènements bornés mais dont le WCET est plus précis que ceux qui ont plus d'évènements bornés. C'est parce que le WCET dépend non seulement de la proportion d'évènements bornés mais également de beaucoup d'autres facteurs. Par exemple, il se peut que les évènements bornés se trouvent dans un morceau de code qui est exécuté un grand nombre fois et leur contribution au WCET final est donc plus importante que des parties moins exécutées avec des évènements non-bornés. Néanmoins, à cause du manque d'estimation précise entre le WCET calculé et le WCET réel, l'impact effectif de ces facteurs est difficile à qualifier.

5.6.2 Temps d'analyse

Une autre métrique importante des algorithmes de partitionnement est leur performance en temps d'analyse. Comme les analyses globales sont les mêmes pour tous les algorithmes de partitionnement, leur temps d'analyse total est comparé directement. Ce temps comprend la chaîne complète du calcul du WCET : le décodage du fichier binaire, les analyses globales, l'analyse de pipeline, la construction et la résolution du système ILP jusqu'à la résolution du WCET. Dans cette expérience, le *seuil de coupure* est encore de 15, à cause de la limitation nécessaire au fonctionnement d'*etime*. Cependant, même avec ce seuil de coupure assez limité, l'analyse du programme *rijndael_enc* avec le *groupement exhaustif* ne finit pas au bout de 12 heures et est considéré comme échoué (pour *etime* seulement). Ce résultat se distingue par une barre manquante dans la figure 5.3A.



(A) Architecture simple



(B) Architecture complexe

FIGURE 5.3 – Temps d'analyse total.

La figure 5.3 montre le temps d'analyse (en secondes) par benchmark (le résultat le plus petit est le meilleur). L'axe y a une échelle logarithmique. Sur les deux architectures, sauf pour le benchmark *petrinet*, le *groupement par XDD* prend des temps d'analyse similaires à celui du *groupement maximum* mais a un temps très inférieur à ceux du *groupement exhaustif* et d'*etime*. Pour des petits benchmarks dont le temps d'analyse est inférieur à une seconde, la différence n'est pas significative en considérant les imprécisions possiblement introduites par l'horloge du système d'exploitation (Linux Ubuntu dans une machine virtuelle exécutée sur un serveur).

5.6.3 Conclusion

En combinant les deux expériences, nous pouvons conclure que le *groupement par XDD* apporte une amélioration significative par rapport à la version originale de la méthode IPET parce qu'il prend presque le même de temps d'analyse comparé au *groupement maximum* classique. De plus, il est presque aussi précis que le *groupement exhaustif* qui est, lui, très coûteux en temps de calcul.

Nous avons observé dans ces expériences que certaines analyses globales produisent beaucoup d'évènements non-bornés ce qui peut fortement impacter la précision des méthodes de calcul de WCET basées sur IPET. Néanmoins, cet effet est difficile à évaluer : nous ne pouvons pas simplement activer/désactiver des évènements car ils sont profondément intriqués au sein l'analyse de pipeline.

Chapitre 6

Analyse de pipeline basée sur les ressources

Dans le chapitre 4, nous avons présenté notre analyse de pipeline basée sur les graphes d'exécution mais en utilisant des XDD pour représenter des temps d'exécution soumis aux évènements. Le contexte d'exécution des BB est alors pris en compte en réalisant les calculs en présence des BB prédécesseurs. Cette approche est valide pour des processeurs à exécution dans l'ordre (ROCHANGE et al., 2009) mais, pour des processeurs qui contiennent des ressources allouées dans le désordre, à cause de la présence d'*anomalies temporelles*, il est difficile de trouver un **pire contexte local** (uniquement en fonction des BB prédécesseurs), ce qui pose des problèmes de validité du WCET calculé. Même si un tel pire contexte est calculable, il introduit généralement une surestimation considérable parce que la latence supplémentaire, causée par l'attente des ressources allouées dans le désordre, est en général beaucoup moins importante en situation réelle que dans les estimations pire cas.

D'autre part, pour supprimer cette surestimation, l'analyse de pipeline n'a pas d'autre choix que de considérer précisément le contexte d'exécution de chaque BB, i.e. les états précis du pipeline à l'entrée de chaque BB. Cela exige un modèle de pipeline capable d'exprimer **l'état** du pipeline et, pour l'analyse de pipeline, de fonctionner **au niveau du CFG**. Malheureusement, le modèle d'analyse basé sur les graphes d'exécution, présenté dans le chapitre 4, représente l'exécution de chaque BB sous forme de graphe et non sous forme d'**état**.

Par conséquent, l'objectif de ce chapitre est de convertir le modèle par graphe d'exécution présenté dans le chapitre 4 en un modèle qui peut exprimer l'**état** du pipeline ce qui permettra de créer une analyse fonctionnant au niveau du CFG et il sera ainsi possible de réutiliser les techniques classiques d'analyse statique du CFG. Ce modèle de calcul a pour principe de décomposer le pipeline en *ressources* et est donc qualifié de "basé sur les ressources" (BAI et al., 2023).

6.1 État temporel du pipeline

Rappelons que dans l'analyse basée sur le graphe d'exécution présenté dans la section 4.1, les contraintes d'exécution dans le pipeline sont représentées par les *dépendances* : les arcs du graphe. En fait, ces contraintes existent parce que les *ressources* du pipeline sont allouées et libérées en fonction de certaines règles. Autrement dit, une dépendance spécifie la condition d'allocation et de libération des ressources. Par exemple, l'*Ordre du Pipeline* établit que l'allocation d'un étage doit être faite après la libération de l'étage précédent. De ce point de vue, nous pouvons voir la résolution du graphe d'exécution comme la résolution des dates d'allocation et de libération des ressources.

Par exemple, dans le processeur que nous avons utilisé dans le chapitre 4, les règles de construction des arcs du graphe d'exécution sont décrites dans la section 4.1. D'après ces règles, pour déterminer la date de démarrage d'une instruction $I_i \in \mathcal{I}$ à l'étage DE, du point de vue de la ressource, il faut connaître :

- La date du début de l'instruction précédente à l'étage DE : $\rho_{[I_{i-1}/DE]}$, qui correspond à la dépendance d'*Ordre du Programme* – l'utilisation de l'étage DE doit suivre l'ordre du programme.
- La date de la fin de la deuxième instruction précédente (l'étage est de largeur 2) à l'étage DE : $\rho^*_{[I_{i-2}/DE]}$ qui correspond à la dépendance de l'*Ordre de Capacité* – l'utilisation de l'étage DE avec la limite de capacité de l'étage.
- La date de la fin de la même instruction à l'étage FE : $\rho^*_{[I_i/FE]}$ qui correspond à l'*Ordre du Pipeline* – l'utilisation des étages par une instruction doit respecter l'ordre des étages dans le pipeline.
- La date du début de la deuxième instruction précédente à l'étage EX : $\rho_{[I_{i-2}/EX]}$ qui correspond à l'*Ordre induit par la capacité des files* à cause de la file entre les étages DE et EX de taille 2.

où l'instruction I_{i-j} dénote la $j^{ème}$ instruction précédant I_i.

Dans l'analyse par graphe d'exécution, ces dépendances sont illustrées comme des arcs entrants du nœud $[I_i/DE]$ mais, finalement, si les dates de libération des ressources correspondantes (les dates mentionnées ci-dessus) sont connues, les dépendances peuvent être résolues avec la même règle que pour les graphes d'exécution – la date au plus tôt où toutes les dépendances sont satisfaites :

$$\begin{aligned} \rho_{[I_i/DE]} &= \rho_{[I_{i-1}/DE]} \oplus \rho^*_{[I_{i-2}/DE]} \\ &\quad \oplus \rho^*_{[I_i/FE]} \oplus \rho_{[I_{i-2}/EX]} \\ \text{avec} & \\ \rho^*_{[I_{i-2}/DE]} &= \rho_{[I_{i-2}/DE]} \otimes \lambda_{[I_{i-2}/DE]} \\ \rho^*_{[I_i/FE]} &= \rho_{[I_i/FE]} \otimes \lambda_{[I_i/FE]} \end{aligned} \tag{6.1}$$

La table 6.1 montre les informations nécessaires pour déterminer la date de début d'une instruction quelconque à l'étage DE.

Nous notons que, pour résoudre la dépendance de l'*Ordre de Capacité* à l'instruction I_i, seul le temps $\rho^*_{[I_{i-2}/DE]}$ est nécessaire, mais comme l'analyse avance instruction par instruction, ce $\rho^*_{[I_{i-2}/DE]}$ est en fait calculé depuis $\rho^*_{[I_{i-1}/DE]}$ à l'instruction précédente. Autrement dit, nous pouvons le voir comme une fenêtre glissante qui contient toujours le temps des deux dernières instructions. Et c'est pour la même raison que l'*Effet de File* nécessite aussi deux temps.

Ordre de Programme	Ordre de Capacité		Ordre du Pipeline	Ordre des files	
$\rho_{[I_{i-1}/DE]}$	$\rho^*_{[I_{i-2}/DE]}$	$\rho^*_{[I_{i-1}/DE]}$	$\rho^*_{[I_i/FE]}$	$\rho_{[I_{i-2}/EX]}$	$\rho_{[I_{i-1}/EX]}$

TABLE 6.1 – Informations nécessaires pour déterminer le temps de démarrage d'une instruction I_i quelconque exécutée à l'étage DE.

Le vecteur présenté dans la table 6.1 peut être construit pour chaque étage du pipeline ce qui, finalement, aboutit au vecteur synthétique présenté dans la table 6.2. Nous pouvons considérer cette table, en l'aplatissant, comme un vecteur dont chaque

élément est un état partiel du pipeline. Le symbole $-\infty$ dénote l'absence d'une dépendance à un étage [1]. Dans cette table, I_{fetch}, I_{load}, I_{store} et I_{R_i} sont respectivement la dernière instruction qui charge une instruction depuis la mémoire, charge une donnée depuis la mémoire, écrit de données dans la mémoire et écrit dans le registre R_i dans l'étage s_{R_i}.

Ce vecteur contient toutes les informations nécessaires pour déterminer la date de démarrage de chaque instruction à chaque étage et, par conséquent, nous le nommons *état temporel*. Ceci étant, les informations qu'il contient doivent être mises à jour tout au long de l'exécution pour que les dates calculées soient correctes. Avant de montrer comment l'utiliser pour réaliser l'analyse de pipeline, nous allons définir et clarifier les termes utilisés dans ce chapitre.

	Ordre prog.	Ordre cap.		Ordre pipe.	Ordre file	
FE	$\rho_{[I_{i-1}/FE]}$	$\rho^*_{[I_{i-1}/FE]}$	$\rho^*_{[I_{i-2}/FE]}$	$-\infty$	$\rho_{[I_{i-1}/DE]}$	$\rho_{[I_{i-2}/DE]}$
DE	$\rho_{[I_{i-1}/DE]}$	$\rho^*_{[I_{i-1}/DE]}$	$\rho^*_{[I_{i-2}/DE]}$	$\rho^*_{[I_i/FE]}$	$\rho_{[I_{i-1}/EX]}$	$\rho_{[I_{i-2}/EX]}$
EX	$\rho_{[I_{i-1}/EX]}$	$\rho^*_{[I_{i-1}/EX]}$	$\rho^*_{[I_{i-2}/EX]}$	$\rho^*_{[I_i/DE]}$	$\rho_{[I_{i-1}/ME]}$	$\rho_{[I_{i-2}/ME]}$
ME	$\rho_{[I_{i-1}/CM]}$	$\rho^*_{[I_{i-1}/ME]}$	$\rho^*_{[I_{i-2}/ME]}$	$\rho^*_{[I_i/EX]}$	$\rho_{[I_{i-1}/CM]}$	$\rho_{[I_{i-2}/CM]}$
CM	$\rho_{[I_{i-1}/ME]}$	$\rho^*_{[I_{i-1}/CM]}$	$\rho^*_{[I_{i-2}/CM]}$	$\rho^*_{[I_i/ME]}$	$-\infty$	$-\infty$

	Ordre fetch	Ordre mémoire		Ordre données		
FE	$\rho^*_{[I_{fetch}/FE]}$	$-\infty$		$-\infty$		
DE	$-\infty$	$-\infty$		$-\infty$		
EX	$-\infty$	$-\infty$		$\rho^*_{[I_{R0}/\mathsf{s}_{R0}]}$	$\rho^*_{[I_{R0}/\mathsf{s}_{R1}]}$	...
ME	$-\infty$	$\rho^*_{[I_{load}/ME]}$	$\rho^*_{[I_{store}/ME]}$	$\rho^*_{[I_{R0}/\mathsf{s}_{R0}]}$	$\rho^*_{[I_{R1}/\mathsf{s}_{R1}]}$	...
CM	$-\infty$	$-\infty$		$-\infty$		

TABLE 6.2 – L'*état temporel*

Définition 6.1.1. Une *ressource temporelle* de l'ensemble des *ressources temporelles* $r \in \mathcal{R}$ est une ressource abstraite et unitaire de l'état temporel du pipeline. Elle est unitaire car son état est décrit par **un seul** XDD.

Définition 6.1.2. Un *vecteur temporel* $\overrightarrow{S} \in \mathcal{S}$ est l'*état temporel* du pipeline. Ses éléments sont des XDD : le domaine de l'*état temporel* est $\mathcal{S} = \mathcal{XDD}^{|\mathcal{R}|}$ où $|\mathcal{R}|$ est le nombre de *ressources temporelles* nécessaires pour modéliser un pipeline.

Une *ressource temporelle* $r \in \mathcal{R}$ correspond à un élément du vecteur de la table 6.2, identifiée par un indice unique i_r, dont l'état est accessible par $\overrightarrow{S}[i_r]$.

Réciproquement, l'état temporel du pipeline est le vecteur d'état pour chaque *ressource temporelle*.

Définition 6.1.3. Une *dépendance temporelle* de l'ensemble des *dépendances temporelles* $d \in \mathcal{D}$ est définie par une *ressource temporelle* r_d qui permet de déterminer la date à laquelle la *dépendance temporelle* est satisfaite.

Soit une instruction I_i exécutée à l'étage s (noté $[I_i/\mathsf{s}]$ comme dans le graphe d'exécution), un *état temporel* $\overrightarrow{S} \in \mathcal{S}$ avant son exécution et un ensemble de *dépendances temporelles* $\mathcal{D}_{[I_i/s]}$ qui sont nécessaires pour le démarrage de $[I_i/\mathsf{s}]$. Son temps de démarrage est calculé par :

$$\rho_{[I_i/s]} = \bigoplus_{d \in \mathcal{D}_{[I_i/\mathsf{s}]}} \overrightarrow{S}[i_{r_d}] \tag{6.2}$$

1. $-\infty$ est neutre par rapport à l'opération *max*.

Il s'agit simplement de trouver le maximum des dates de toutes les *ressources temporelles* liées aux *dépendances temporelles* nécessaires. On peut déjà voir que cette formule est très similaire au calcul de dépendance dans le graphe d'exécution (l'équation 4.11, page 55). La seule différence est que nous nous basons sur les XDD du *vecteur temporel* alors que, dans le graphe d'exécution, on utilisait les dates de démarrage et de fin des nœuds.

6.2 Analyse de pipeline avec les états temporels

L'*état temporel* présenté dans la section précédente nous donne l'ensemble des informations nécessaires pour déterminer le temps de démarrage de chaque instruction à chaque étage. Ce vecteur doit être mis à jour tout au long de l'avancement de l'analyse, ce qui est différent du calcul avec les graphes d'exécution. Ce dernier est construit pour une séquence d'instructions donnée et la date de début/fin de chaque nœud est conservée tout au long de la résolution du graphe. Néanmoins, certaines dates ne sont plus nécessaires en fonction de l'avancement de l'analyse, précisément, une fois que les arcs **sortants** de ces nœuds ont tous été traités. Ces informations, n'étant plus utiles, ne sont pas stockées dans le modèle basé sur les ressources. En fait, en suivant l'ordre d'évaluation du graphe d'exécution (figure 4.2, page 56), il existe un nombre statiquement fixé de nœuds dont la date de démarrage (ou de fin) est suffisante pour continuer la résolution. Les *ressources temporelles* sont exactement ces dates du point de vue des *ressources*. Par conséquent, nous pouvons voir *l'état temporel* comme une représentation de l'**état de la résolution** du graphe d'exécution, ce qui rend le calcul basé sur les ressources équivalent la résolution du par graphe d'exécution.

Le fait de représenter seulement "une partie du graphe d'exécution" nous permet d'améliorer les performances d'analyse (car moins d'informations inutiles nécessitent d'être stockées). D'autre part, l'analyse se doit de mettre à jour l'état à mesure qu'elle avance tout au long du programme (dans l'ordre d'évaluation des nœuds) parce que l'état des *ressources temporelles* peut être relatif à une instruction. Par exemple, nous devons garder la date du début de la **dernière instruction** traitée pour résoudre les dépendances d'*ordre du programme*. C'est pour cette raison que dans la table 6.2, l'état enregistre toujours les dates des instructions I_{i-1} ou I_{i-2} – des instructions relatives et précédant I_i.

Nous allons maintenant montrer comment décrire les *mises à jour* de l'état temporel. D'un point de vue *interprétation abstraite*, l'*état temporel* correspond à l'état abstrait du pipeline et la mise à jour de l'état correspond à la fonction $Update$ (voir section 2.2). Pour ce faire, 4 **transitions** d'état basiques sont définies : $\tau_{reset}, \tau_{wait}, \tau_{move}$ et $\tau_{consume}$.

Pour simplifier le calcul, une *ressource temporelle* **virtuelle** ϱ est ajoutée au *vecteur temporel* à l'indice i_ϱ. Cette ressource représente la "date courante" durant l'analyse, désigné sous le terme *pointeur de temps*. Les transitions basiques sont définies ci-dessous :

$$\begin{aligned} &\tau_{reset} : \mathcal{S} \to \mathcal{S}, \\ &\tau_{reset}(\overrightarrow{S}) = \overrightarrow{S}' \mid \begin{cases} \overrightarrow{S}'[i] = \overrightarrow{S}[i] \otimes \mathbb{0} \text{ if } i = i_\varrho \\ \overrightarrow{S}'[i] = \overrightarrow{S}[i] \text{ otherwise} \end{cases} \end{aligned} \tag{6.3}$$

La transition *reset* initialise simplement le *pointeur de temps* à $\mathbb{0} = -\infty$.

$$\tau_{wait} : \mathbb{N} \times \mathcal{S} \to \mathcal{S},$$
$$\tau_{wait}(x, \overrightarrow{S}) = \overrightarrow{S}' \mid \begin{cases} \overrightarrow{S}'[i] = \overrightarrow{S}[i] \oplus \overrightarrow{S}[x] \text{ if } i = i_{\varrho} \\ \overrightarrow{S}'[i] = \overrightarrow{S}[i] \text{ otherwise} \end{cases} \tag{6.4}$$

La transition *wait* représente l'attente de la libération d'une ressource, ce qui applique max ($\oplus$) entre le *pointeur de temps* et l'état de la *ressource temporelle* à l'indice x.

$$\tau_{move} : \mathbb{N} \times \mathbb{N} \times \mathcal{S} \to \mathcal{S},$$
$$\tau_{move}(i_{dest}, i_{src}, \overrightarrow{S}) = \overrightarrow{S}' \mid \begin{cases} \overrightarrow{S}'[i] = \overrightarrow{S}[i_{src}] \text{ if } i = i_{dest} \\ \overrightarrow{S}'[i] = \overrightarrow{S}[i] \text{ otherwise} \end{cases} \tag{6.5}$$

La transition *move* copie un élément du vecteur d'indice i_{src} dans un autre élément d'indice i_{dest} et écrase l'ancienne valeur dans i_{dest}.

$$\tau_{consume} : \mathbb{N} \times \mathcal{S} \to \mathcal{S},$$
$$\tau_{consume}(\lambda, \overrightarrow{S}) = \overrightarrow{S}' \mid \begin{cases} \overrightarrow{S}'[i] = \overrightarrow{S}[i] \otimes \lambda \text{ if } i = i_{\varrho} \\ \overrightarrow{S}'[i] = \overrightarrow{S}[i] \text{ otherwise} \end{cases} \tag{6.6}$$

La transition *consume* rajoute ($\otimes$) une latence dans le *pointeur de temps*.

Définition 6.2.1. Les transitions sont appliquées en quatre étapes pour exprimer l'exécution (par la méthode des graphes d'exécution) du nœud $[I_i, \mathsf{s}]$:

- Étape 1. Suivant le même principe que les graphes d'exécution, un nœud (une instruction exécutée à un étage) peut démarrer si toutes les dépendances sont satisfaites. Avec le modèle basé sur les ressources, ce comportement est exprimé par : (étape 1.a) le *pointeur de temps* est remis à $\mathbb{0}$ à l'aide de τ_{reset} et (étape 1.b) pour chaque dépendance nécessaire pour le démarrage de ce nœud ($\mathcal{D}_{[I/S]}$), on attend jusqu'à la libération de chacune des ressources correspondantes : τ_{wait} sur chacune de ces ressources. Après l'étape 1, la valeur dans le *pointeur de temps* est la date de démarrage du nœud. En fait, l'étape 1 réalise l'équation 6.2 mais avec les transitions de l'*état temporel*.
- Étape 2. Certaines *ressources temporelles* doivent être mises à jour si elles doivent mémoriser la date de démarrage du nœud courant $[I_i/\mathsf{s}]$ – qui est présente dans le *pointeur de temps*. On peut les retrouver dans la table 6.2 : il s'agit de l'*Ordre du Programme* et de l'*Ordre induit par la capacité des files*.
  Pour l'*Ordre du Programme*, $move(j, i_T P)$ est suffisant pour mettre à jour l'état en copiant la valeur du *pointeur de temps* à l'indice j.
  Cependant, pour l'*Ordre induit par la capacité des files*, plusieurs *ressources temporelles* doivent être mises à jour. La mise à jour est faite tout en respectant la sémantique de la file FIFO : le temps de libération de la file par I_i devient celui de I_{i-1}, et celui de I_{i-1} devient celui de I_{i-2} et ainsi de suite jusqu'à $I_{i-|q|}$ où $|q|$ est la capacité de la file. Le temps de I_i est alors l'état du *pointeur de temps*.
  Pour simplifier la notation, une convention de placement de ces *ressources temporelles* est fixée par l'ordre du programme. Par exemple pour l'*Ordre induit par les files* de l'étage FE, si le $\rho_{[I_i/FE]}$ est placé à l'indice j, alors $\rho_{[I_{i-1}/FE]}$ sera placé à $j+1$ et $\rho_{[I_{i-2}/FE]}$ aura l'indice $j+2$ et ainsi de suite. Donc cette mise à jour consiste en :

$$\begin{aligned} &\forall x \in [j, j+|q|-2], \tau_{move}(x, x+1, \overrightarrow{S}); \\ &\tau_{move}(j, i_\varrho, \overrightarrow{S}); \end{aligned} \tag{6.7}$$

où j est l'index de la ressource temporelle qui enregistre la date de libération de la file par l'instruction courante.

- Étape 3. L'instruction prend un certain temps ($\lambda_{[I_i/\mathsf{s}]}$) pour s'exécuter dans l'étage courant. Ce temps est ajouté ($\otimes$ dans le domaine $\mathcal{XDD}$) dans le *pointeur de temps*, i.e. $\tau_{consume}(\lambda_{[I_i/\mathsf{s}]}, \overrightarrow{S})$.
- Étape 4. L'instruction finit son exécution dans l'étage courant, la valeur du *pointeur de temps* est sa date de fin. Donc, la mise à jour des ressources temporelles qui dépendent de la date de fin est faite en suivant la même logique que l'étape 3. Pour les *dépendances temporelles* de plusieurs *ressources* – l'*Ordre de Capacité*, la mise à jour est effectuée de la même manière que celle de l'*Ordre induit par la capacité des files* car on retrouve le même comportement FIFO. Pour les *dépendances* qui concernent le type d'instruction – l'*Ordre de Fetch*, l'*Ordre de Mémoire* et la *Dépendance de données*, la mise à jour est effectuée uniquement si l'instruction est du type correspondant.

Les étapes ci-dessus décrivent les transitions à appliquer pour une instruction sur un étage. Pour une séquence d'instructions, l'ordre d'évaluation reste le même que la résolution des graphes d'exécution, i.e. instruction par instruction dans l'ordre du programme, puis étage par étage dans l'ordre des étages du pipeline (voir la figure 4.2, page 56). Finalement, on se rend compte que les calculs réalisés par le modèle basé sur les ressources sont les mêmes que pour la résolution d'un graphe d'exécution. Cela nous garantit que les états des *ressources temporelles* sont correctement mis à jour car les **dépendances** sont résolues dans le même ordre.

6.3 Calcul avec des matrices

Le modèle basé sur les ressources est équivalent au modèle utilisant des graphes d'exécution mais il plus performant. Tout d'abord parce qu'il garde seulement "une partie" des dates calculées sur le graphe d'exécution tel que présenté dans la section 6.2 (page 84). Ensuite, tel que présenté dans cette section, il permet de profiter des propriétés algébriques des XDD pour amplifier l'amélioration des performances et fournir des états concrets avec l'analyse au niveau du CFG.

Nous avons montré dans la section 3.5 que le domaine des XDD, $\langle \mathcal{XDD}, \oplus, \otimes, \mathbb{0}, \mathbb{1} \rangle$ avec $\mathbb{0} = LEAF(-\infty)$ et $\mathbb{1} = LEAF(0)$, forme un demi-anneau. Les fonctions de transitions τ sont en fait affines sur ce demi-anneau. Par conséquent, leur application peut être exprimée sous forme de produit matriciel. De plus, $\mathcal{XDD}$ étant un demi-anneau, la multiplication est associative et donc le produit matriciel l'est aussi. Cette propriété permet de pré-calculer et de résumer l'analyse de pipeline de chaque BB en une seule matrice de XDD car les transitions à appliquer sont statiquement connues.

Le produit matriciel sur le demi-anneau $\mathcal{XDD}$ est similaire à l'algèbre linéaire sur $\mathbb{R}$ mais en remplaçant $+$ par $\oplus$ et $\times$ par $\otimes$ [2] :

2. Attention cependant, sémantiquement, $\otimes$ est l'addition dans $\mathbb{R}$ et $\oplus$ est le maximum (voir section 3.3).

Définition 6.3.1. Nous pouvons définir la matrice d'identité Id :

$$Id = \begin{bmatrix} A_{i,j} \end{bmatrix} \mid A_{i,j} = \begin{cases} \mathbb{1} \text{ if } i = j \\ \mathbb{0} \text{ otherwise} \end{cases}$$

qui satisfait $\overrightarrow{S} \cdot Id = \overrightarrow{S}$ car chaque élément de la $i^{\text{ème}}$ ligne est $\mathbb{0}$ sauf à la $i^{\text{ème}}$ colonne qui est $\mathbb{1}$. La matrice Id a un effet identique à l'identité dans $\mathbb{R}$. Maintenant, nous pouvons montrer comment représenter toutes les transitions τ sous forme matricielle. Pour simplifier l'explication, nous allons prendre la matrice Id comme base et nous modifierons seulement les éléments qui ont un effet sur le vecteur.

Un $\mathbb{0}$ sur la diagonale (qui remplace $\mathbb{1}$ dans Id) à la même colonne que l'indice du pointeur de temps (i_ϱ) le remet à zéro (car $\overrightarrow{S}[i_\varrho] \otimes \mathbb{0} = \mathbb{0}$) :

$$\begin{aligned} \tau_{reset}(\overrightarrow{S}) &= \overrightarrow{S} \cdot \mathcal{M}_{reset} \\ &= \overrightarrow{S} \cdot \begin{bmatrix} A_{i,j} \end{bmatrix} | A_{i,j} = \begin{cases} \mathbb{0} \text{ if } i = j = i_\varrho \\ Id_{i,j} \text{ otherwise} \end{cases} \end{aligned} \tag{6.8}$$

Pour un indice x dans $\overrightarrow{S}$, $\tau_{wait}(x, \overrightarrow{S})$ est représentée par une matrice $\mathcal{M}_{wait(x)}$ avec $\mathbb{1}$ à la $x^{\text{ème}}$ ligne et $i_\varrho^{\text{ème}}$ colonne (noté (x, i_ϱ)) ce qui remplace l'état du *pointeur de temps* ϱ par $\varrho \oplus (\mathbb{1} \otimes \overrightarrow{S}[x]) = \varrho \oplus \overrightarrow{S}[x]$:

$$\begin{aligned} \tau_{wait}(x, \overrightarrow{S}) &= \overrightarrow{S} \cdot \mathcal{M}_{wait(x)} \\ &= \overrightarrow{S} \cdot \begin{bmatrix} A_{i,j} \end{bmatrix} | A_{i,j} = \begin{cases} \mathbb{1} \text{ if } i = i_\varrho \wedge j = x \\ Id_{i,j} \text{ otherwise} \end{cases} \end{aligned} \tag{6.9}$$

$\tau_{move}(i_{src}, i_{dest}, \overrightarrow{S})$ est représentée par une matrice $\mathcal{M}_{move(i_{src}, i_{dest})}$ dont l'élément à la position (i_{dest}, i_{dest}) est mis à $\mathbb{0}$ et l'élément à (i_{dest}, i_{src}) est mis à $\mathbb{1}$. Comme cela, l'élément d'indice i_{dest} dans le vecteur sortant devient $(\mathbb{0} \otimes \overrightarrow{S}[i_{dest}]) \oplus (\mathbb{1} \otimes \overrightarrow{S}[i_{src}]) = \overrightarrow{S}[i_{src}]$:

$$\begin{aligned} \tau_{move}(i_{src}, i_{dest}, \overrightarrow{S}) &= \overrightarrow{S} \cdot \mathcal{M}_{move(i_{src}, i_{dest})} \\ &= \overrightarrow{S} \cdot \begin{bmatrix} A_{i,j} \end{bmatrix} | A_{i,j} = \begin{cases} \mathbb{0} \text{ if } i = j = i_{dest} \\ \mathbb{1} \text{ if } i = i_{dest} \wedge j = i_{src} \\ Id_{i,j} \text{ otherwise} \end{cases} \end{aligned} \tag{6.10}$$

$\tau_{consume}(\lambda, \overrightarrow{S})$ est utilisé pour ajouter une latence $\lambda \in \mathcal{XDD}$ dans le *pointeur de temps*. La matrice correspondante est $\mathcal{M}_{consume(\lambda)}$, met λ à la position (i_ϱ, i_ϱ). Par la multiplication de matrice (opérateur $\otimes$), λ sera ajouté à $\overrightarrow{S}[i_\varrho]$:

$$\begin{aligned} \tau_{consume}(\lambda, \overrightarrow{S}) &= \overrightarrow{S} \cdot \mathcal{M}_{consume(\lambda)} \\ &= \overrightarrow{S} \cdot \begin{bmatrix} A_{i,j} \end{bmatrix} | A_{i,j} = \begin{cases} \lambda \text{ if } i = j = i_\varrho \\ Id_{i,j} \text{ otherwise} \end{cases} \end{aligned} \tag{6.11}$$

Maintenant, comme nous avons représenté toutes les transitions τ sous forme de matrice, nous pouvons représenter le calcul de la date de démarrage d'un nœud (Étape 1) par une matrice, par exemple :

$$\mathcal{M}_{Etape_1[I_i/\mathsf{s}]} = \mathcal{M}_{reset} \cdot \prod_{d \in \mathcal{D}_{[I_i/\mathsf{s}]}} \mathcal{M}_{wait(i_{r_d})} \tag{6.12}$$

avec $d \in \mathcal{D}_{[I_i/\mathsf{s}]}$ les dépendances de $[I_i/\mathsf{s}]$ et r_d les *ressources temporelles* nécessaires pour l'exécution de I_i à l'étage s.

De la même manière, nous pouvons écrire la matrice pour les étapes 2, 3 et 4 ($\mathcal{M}_{Etape_2[I_i/\mathsf{s}]}, \mathcal{M}_{Etape_3[I_i/\mathsf{s}]}$ et $\mathcal{M}_{Etape_4[I_i/\mathsf{s}]}$) en invoquant les τ des transitions correspondantes. Comme chaque *étape* est une fonction affine sur le demi-anneau $\mathcal{XDD}$, l'analyse de pipeline pour une instruction I_i à l'étage s est aussi affine, et la matrice correspondante peut être calculée :

$$\mathcal{M}_{[I_i/\mathsf{s}]} = \mathcal{M}_{Step_1[I_i/\mathsf{s}]} \cdot \mathcal{M}_{Step_2[I_i/\mathsf{s}]} \cdot \mathcal{M}_{Step_3[I_i/\mathsf{s}]} \cdot \mathcal{M}_{Step_4[I_i/\mathsf{s}]} \tag{6.13}$$

Finalement, l'analyse de pipeline sur un BB $a \in V_{\mathrm{CFG}}$ est composée de l'analyse de chaque instruction et chaque étage, suivant l'ordre d'évaluation dans le bloc et le graphe d'exécution :

$$\mathcal{M}_a = \prod_{i=[0,\ldots,|a|-1]} \prod_{k=[0,\ldots,|\mathsf{S}|-1]} \mathcal{M}_{[I_i/\mathsf{s}_k]} \tag{6.14}$$

Au total, les $\mathcal{M}_a$ sont pré-calculées et représentent complètement l'analyse de pipeline du BB a. Avec un état de pipeline (*vecteur temporel*) entrant $\overrightarrow{S}$, l'état à la fin du BB $\overrightarrow{S}'$ est calculé simplement par :

$$\overrightarrow{S}' = \overrightarrow{S} \cdot \mathcal{M}_a \tag{6.15}$$

6.4 Analyse de pipeline au niveau du CFG

Dans les sections précédentes, nous avons présenté comment le modèle basé sur les ressources peut être appliqué à une séquence d'instructions (typiquement un BB). Cependant, l'objectif final de ce modèle est de supporter l'analyse de pipeline au niveau du CFG. Une telle analyse ne doit pas calculer les temps d'exécution de chaque BB indépendamment mais également considérer son **contexte** d'exécution, c'est-à-dire **l'effet de l'historique d'exécution sur le temps d'exécution du BB**.

Pour un processeur qui contient uniquement des ressources allouées dans l'ordre du programme, il est possible de trouver un pire contexte pour tous les BB. De plus, l'article ROCHANGE et al., 2009 a déjà montré qu'il existe des cas où prendre un ou plusieurs BB prédécesseurs est suffisant pour modéliser le contexte avec une précision raisonnable.

Cependant, si l'architecture contient des ressources accédées dans le désordre, la présence d'*anomalies temporelles* rend difficile de trouver un **pire** contexte valide (c'est-à-dire pour lequel le temps d'exécution calculé est forcément supérieur au vrai WCET du BB). De plus, avec le pire contexte, l'analyse tend à largement surestimer le temps d'exécution en présence des contentions liées aux ressources allouées dans le désordre parce que ces contentions peuvent être rares et causent, en général, moins d'attente que dans le pire cas. Pour des raisons de précision, il est donc préférable

que l'analyse de pipeline puisse calculer le contexte précis de chaque BB au niveau du CFG.

Heureusement, nous avons déjà toutes les informations nécessaires pour construire ce contexte : *l'état temporel* calculé par le modèle basé sur les ressources à la fin de chaque BB contient les dates de libération de toutes les ressources du pipeline. Cependant, nous ne pouvons pas l'utiliser directement comme contexte des BB successeurs parce qu'il contient des dates relatives au début du BB prédécesseur. Autrement dit, nous pouvons certes calculer tous les temps d'exécution d'un seul chemin d'exécution mais, à cause des boucles, il est impossible de les calculer pour tous les chemins d'exécution du CFG. En fait, pour convertir *l'état temporel* à la fin d'un BB en un contexte utilisable par les BB successeurs, il suffit de *changer* la date de référence (initialement relative au début du BB **prédécesseur**) pour la rendre relative au début du BB **successeur**. L'objectif de cette section est de clarifier cette opération que nous nommons *rebasage*.

6.4.1 Début et fin de BB

Comme les instructions sont exécutées en parallèle dans le pipeline, nous devons d'abord bien définir ce que sont le début et la fin de BB. De manière conventionnelle (ou intuitive), on aimerait dire que le début d'un BB est le début de la première instruction au premier étage et la fin du BB est la fin de la dernière instruction au dernier étage. Comme les BB sont exécutés en parallèle dans le pipeline, cette façon de compter introduit une importante surestimation du WCET. Considérons par exemple l'exécution de deux BB consécutifs tel que représenté sur la figure 6.1. Les dates de début sont notées d_{pred} et d_{succ} et les dates de fin d^*_{pred} et d^*_{succ}. Les temps t_{succ} et t_{pred} sont respectivement le temps d'exécution du BB successeur et du BB prédécesseur calculé avec cette définition pour le début et la la fin du BB. Sur la figure, il est clair que la partie superposée est comptée deux fois (plus précisément, plusieurs fois en fonction du nombre d'occurrences des BB sur le pire chemin) quand on calcule le temps d'exécution des deux BB, c'est-à-dire $t_{succ} + t_{pred}$.

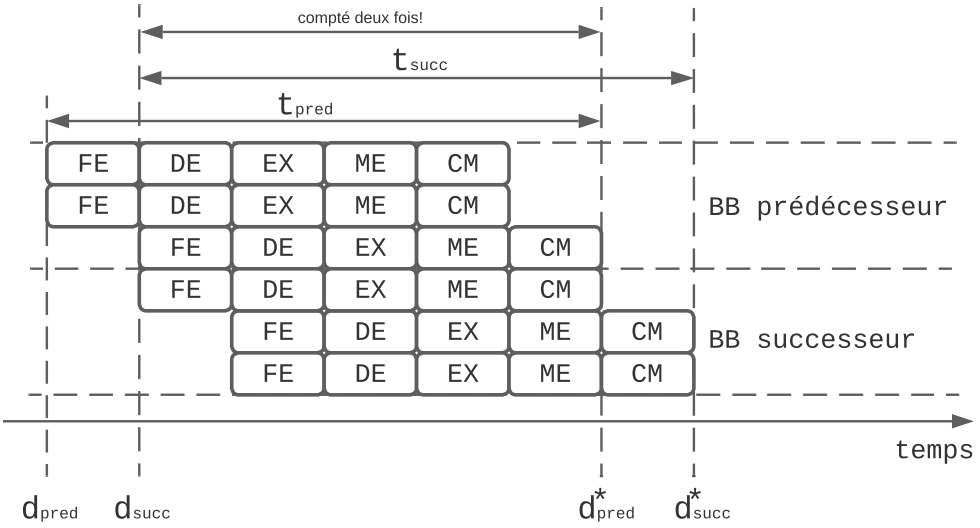


FIGURE 6.1 – Temps d'exécution des BB – définition conventionnelle.

Pour supprimer cette surestimation, nous définissons la fin d'un BB comme le début de sa dernière instruction au premier étage (l'étage de fetch). Cette date a l'avantage d'être très probablement le début de la première instruction au premier

étage du **prochain BB** comme illustré sur la figure 6.2. La dernière instruction du BB prédécesseur se situe dans le même bloc mémoire que la première instruction du BB successeur. Donc, si on considère un processeur superscalaire de degré 2, ces deux instructions sont exécutées en parallèle dans le pipeline. Ainsi, la date de fin du BB prédécesseur (d^*_{pred}) est égale à celle du début du BB successeur (d_{succ}). Le temps d'exécution des deux BB est calculé comme la fin de chaque bloc moins sa date de début, respectivement t_{pred} et t_{succ} . Nous pouvons effectivement constater que, en comptant de cette manière, il n'y a plus de latence comptée plusieurs fois.

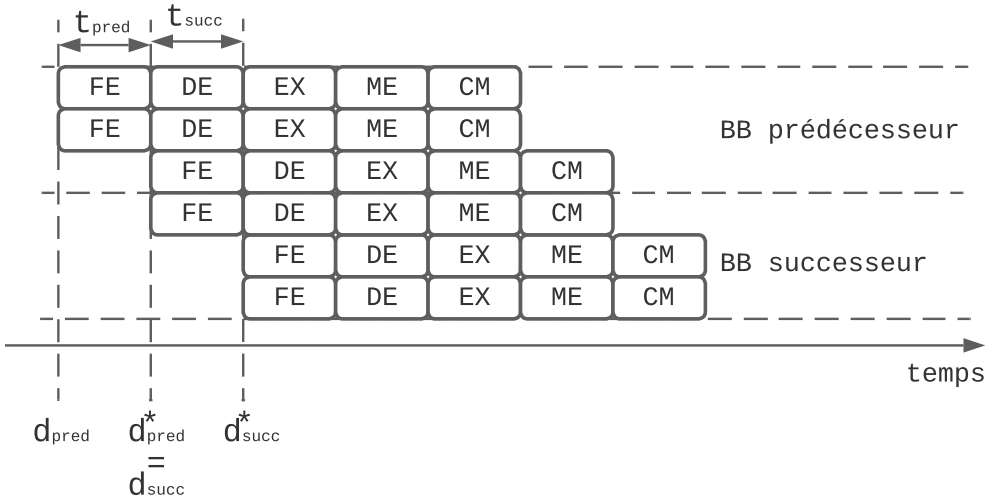


FIGURE 6.2 – Temps d'exécution des BB – notre définition (cas 1)

Cependant, la première instruction du BB successeur n'est pas forcément exécutée en parallèle avec la dernière instruction du BB prédécesseur, par exemple parce qu'elles ne situent pas dans le même bloc de mémoire. La figure 6.3 montre un tel exemple : le début de la première instruction du BB successeur arrive plus tard que la date de fin du BB prédécesseur. C'est pour cette raison que nous avons défini de manière **artificielle** que la date du début d'un BB est exactement la date de fin du BB prédécesseur ($d^*_{pred} = d_{succ}$ est toujours vrai dans notre définition). Ainsi, le temps est compté de manière continue, et il n'y a pas de latence non comptée. Comme nous avons défini d^* et forcé $d_{succ} = d^*_{pred}$, cela fonctionne comme si la date du début du BB successeur n'était pas 0 mais retardée par le BB prédécesseur, dans cet exemple, 1 cycle.

En fait, il existe plusieurs manières de définir une date de fin mais celle que nous avons choisie ici simplifie l'implantation. La date de fin est avant tout une convention qui doit respecter comme condition, afin d'être valide, (a) de ne pas sous-estimer les temps d'exécution, (b) de rester précise afin de ne pas les surestimer, et (c) de faire coïncider la date de fin d'un bloc avec la date de début du suivant.

Le contexte

De manière générale, le contexte d'exécution d'un BB est l'état du processeur (ou du programme) à la fin de l'exécution des BB prédécesseurs, qui détermine l'exécution du BB courant. Du point de vue temporel :

Définition 6.4.1. Le contexte temporel d'un BB est l'état temporel du processeur, à la fin des BB prédécesseurs, qui détermine le temps d'exécution du BB courant.

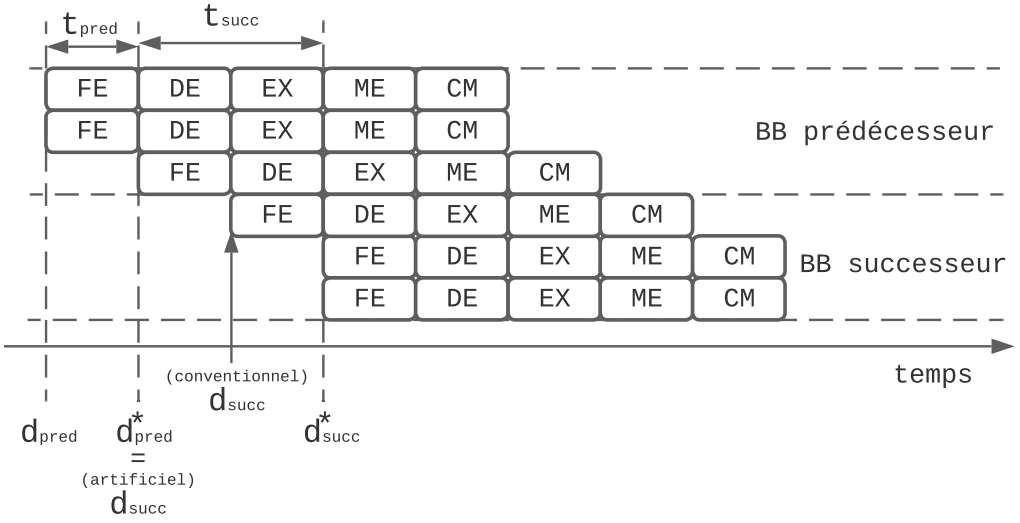


FIGURE 6.3 – Temps d'exécution des BB – notre définition (cas 2)

Le contexte (dans la suite, le qualificatif "temporel" est omis) peut influencer l'exécution des BB successeurs par l'utilisation des **ressources** dans les BB prédécesseurs. L'*état temporel* obtenu à la fin des BB prédécesseurs donne déjà cette information. Néanmoins, les temps dans cet état sont encore relatifs au début des BB prédécesseurs. En suivant notre définition du début et de la fin des BB, ces temps doivent être *rebasés* par rapport au début du BB successeur pour former son contexte. Pour ce faire, nous définissons l'opération de *rebasage* qui permet de calculer ces temps relativement à un temps de base arbitrairement choisi. Par exemple, nous avons un temps t sous forme de XDD (en fait plusieurs temps) qui est relatif à une date 0 (l'origine du temps). Nous pouvons *rebaser* t relativement à un autre temps $t_{ref} \in \mathcal{XDD}$ en faisant $t \oslash t_{ref}$ où $\oslash$ est la soustraction transférée dans le domaine des $\mathcal{XDD}$. Comme cette opération est faite avec des XDD, la relation entre les temps et les évènements est préservée.

Comme nous pouvons rebaser un temps, un *état temporel* peut l'être également en *rebasant* l'état de chaque *ressource temporelle* du vecteur :

$$\forall \overrightarrow{S} \in \mathcal{XDD}^{|\mathcal{R}|}, \forall r \in \mathcal{R}, rebase(\overrightarrow{S}, t_{ref})[r] = \overrightarrow{S}[r] \oslash t_{ref} \tag{6.16}$$

Rebaser un état temporel est une opération sans perte parce que nous pouvons toujours restaurer l'état initial en appliquant $\otimes$:

$$\forall \overrightarrow{S} \in \mathcal{XDD}^{|\mathcal{R}|}, \forall r \in \mathcal{R}, rebase(\overrightarrow{S}, t_{ref})[r] \otimes t_{ref} = \overrightarrow{S}[r] \tag{6.17}$$

car $\otimes$ est la fonction réciproque de $\oslash$:

$$t \oslash t_{ref} \otimes t_{ref} = t \tag{6.18}$$

Pour convertir l'*état temporel* $\overrightarrow{S'}$ calculé à la fin d'un BB en un contexte pour les BB successeurs, il faut simplement rebaser $\overrightarrow{S'}$ par rapport à la date de la fin du BB (qui est également la date du début du BB successeur) – la date du début de la dernière instruction au premier étage.

Le rebasage permet également de contrer les effets de boucle qui donneraient des dates relatives au début du programme qui sont toujours croissantes : les dates sont

désormais relatives au début des blocs quelle que soit l'itération. Bien sûr d'une itération à une autre, le contexte peut être changeant. Il faut noter que le rebasage permet aussi de limiter la quantité de XDD créés et donc d'améliorer les performances de l'analyse.

Analyse de pipeline sur le CFG

Finalement, l'analyse de pipeline au niveau du CFG est la *sémantique collectrice* des états temporels définis par le modèle basé sur les ressources. L'état entrant et sortant de chaque BB est l'ensemble d'*états temporels* avant et après le BB. Comme un BB peut avoir plusieurs prédécesseurs, l'état entrant est l'union des états sortants des prédécesseurs :

$$\begin{aligned} \forall b \in V_{\text{CFG}}, IN[b], OUT[b] \in \wp(\mathcal{XDD}^{|\mathcal{R}|})^2 \\ \forall b \in V_{\text{CFG}}, IN[b] = \bigcup_{a \in PRED(b)} OUT[a] \end{aligned}$$

où $PRED(b)$ donne l'ensemble des prédécesseurs de b.

La sémantique **concrète** de l'exécution d'un BB (*Update* pour un BB) est calculée par la procédure suivante :

$$\begin{aligned} &\forall b \in V_{\text{CFG}}, \\ &\quad OUT[b] = \{\overrightarrow{S_{IN}} \cdot \mathcal{M}_b \oslash t^{ref} \mid \overrightarrow{S_{IN}} \in IN[b] \wedge t^{ref} = (\overrightarrow{S_{IN}} \cdot \mathcal{M}_b^{ref})[\varrho]\} \end{aligned} \tag{6.19}$$

Pour chaque état concret, c'est-à-dire chaque *état temporel* $\overrightarrow{S_{IN}}$ en entrée, l'analyse de pipeline est appliquée avec le modèle de calcul basé sur les ressources, jusqu'au début de la dernière instruction dans le premier étage. La sémantique de cette étape est résumée en une matrice $\mathcal{M}_b^{ref}$. La date de la fin de ce BB est obtenue par la valeur du *pointeur de temps* de l'état temporel de ce nœud. Cette date est utilisée comme le temps d'exécution du BB (lors de la construction du système ILP par la méthode IPET) et comme date de référence pour le rebasage, donc notée t^{ref}. Ensuite, l'état temporel à la fin du BB, $\overrightarrow{S_{OUT}}$ (c'est-à-dire à la fin de la dernière instruction au dernier étage), est calculé en multipliant l'état entrant par la matrice $\mathcal{M}_b$ pré-calculée pour toute la séquence d'instructions. Cet état est ensuite rebasé par rapport à t^{ref} pour déterminer l'état concret sortant du BB. Finalement, l'état (abstrait) sortant du BB est l'ensemble des états concrets sortants du BB.

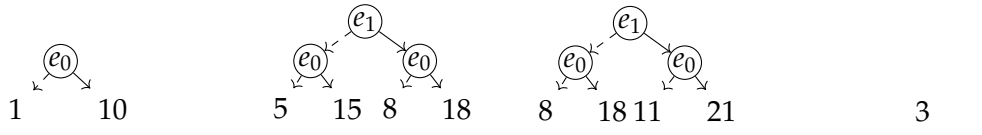
Effet du rebasage sur le contexte

Le rebasage est tout spécialement utile en présence des évènements. Les évènements qui n'ont plus d'effet sur l'exécution des instructions suivantes sont supprimés lors du calcul du contexte (autrement dit, ils ont un effet linéaire sur le WCET en fonction de nombre d'exécutions du BB). Considérons l'exemple de la figure 6.4. L'analyse de pipeline s'applique sur un BB avec comme état entrant le XDD de la figure 6.4A [3] qui contient un évènement e_0 provenant du BB précédent. L'état obtenu à la fin du BB (t_{out} est un élément de $\overrightarrow{S_{OUT}}$) est le XDD de la figure 6.4C avec

3. En réalité, il s'agit d'un vecteur XDD mais, pour simplifier l'exemple, nous considérons un élément du vecteur en gardant à l'esprit que tous les autres éléments vont subir les mêmes opérations.

comme temps de référence le XDD de la figure 6.4B. Ces deux XDD contiennent un évènement e_1 qui est introduit dans le BB courant. Le rebasage est appliqué par $t_{out} \oslash t_{ref}$ ce qui donne simplement un temps $t_{context}$ égal à 3 comme contexte. Ce résultat est induit par l'effet des évènements qui se trouve être, dans ce cas, une pénalité constante : dans t_{ref} et t_{out}, les différences entre les branches des sous-arbres $\text{NODE}(e_0, 5, 15)$, $\text{NODE}(e_0, 8, 18)$, $\text{NODE}(e_0, 11, 21)$ sont de 10 cycles. L'évènement e_0 a donc un coût constant sur l'exécution du BB **courant** : une latence supplémentaire de 10 cycles. Pour la même raison, les sous-arbres gauches et droits de l'évènement e_1 diffèrent de 3 dans t_{ref} et t_{out} et donc cela signifie que e_1 a un effet **constant** de 3 cycles pour l'exécution **ultérieure**, e_1 a ainsi disparu du contexte.

Ces évènements contribuent donc avec un temps linéaire en fonction de leur nombre d'exécutions au WCET final, n'influent donc pas sur l'exécution des instructions suivantes et n'ont donc pas besoin d'être pris en compte par l'analyse de pipeline des BB suivants. Par contre, comme ils sont présents dans t_{ref}, compté comme le temps d'exécution du BB dans le système ILP par la méthode IPET, cette contribution temporelle est bien prise en compte dans le calcul du temps d'exécution. L'expérimentation montre que ce genre de situation arrive assez souvent : le rebasage permet donc également d'améliorer les performances de notre analyse.



(A) État entrant (B) t_{ref} (C) t_{out} (D) $t_{context}$

FIGURE 6.4 – Effet du rebasage.

6.5 Génération des évènements dans les boucles

Les évènements sont générés par les analyses globales (voir figure 2.2, page 15) et chaque évènement est associé à **une instruction**. Par exemple, un accès de mémoire cache qui n'est classifié ni *Always Hit*, ni *Always Miss* est un évènement associé à l'instruction qui réalise l'accès. L'activation ou l'inactivation de cet évènement est par conséquent directement associée à l'instruction pour une exécution donnée du BB qui la contient. Cependant, au niveau de l'analyse du CFG, en présence de boucles, il peut arriver que certains évènements d'une même instruction dans un même BB mais provenant d'itérations différentes coexistent dans les XDD de l'état de pipeline. Toutefois rien ne nous assure que leur activation ou inactivation soit corrélée : d'une itération à l'autre, un même évènement peut être activé puis désactivé. Le fait de les considérer comme un seul évènement peut causer un calcul erroné.

Pour résoudre ce problème, un *numéro de génération/itération* est associé à chaque évènement pour le distinguer des évènements d'itérations différentes. Ce *numéro de génération* n'est pas le numéro d'itération effectif dans la vraie exécution du programme mais un nombre symbolique pour distinguer son exécution dans les contextes différents. Concrètement, ce nombre est initialisé à 0 et est incrémenté chaque fois que l'analyse passe par le BB produisant cet évènement avec un état de pipeline différent. Comme on calcule la *sémantique collective*, ce numéro croît jusqu'à ce que l'analyse atteigne un point fixe.

Théoriquement, le *numéro de génération* n'a pas de borne supérieure parce qu'aucune contrainte ne peut limiter la durée de l'effet d'un évènement, autrement dit, un évènement peut avoir un effet temporel sur un nombre infini d'itérations. Dans la

pratique, cela n'arrive presque jamais pour trois raisons : (a) l'effet d'un évènement est généralement de court terme, grâce à l'*effet d'amortissement* du pipeline (voir section 4.3), (b) les évènements d'effet constant sont supprimés grâce au *rebasage* et (c) le *numéro de génération* est borné par le nombre d'itérations maximal des boucles (limite de boucle). La limite de boucle est forcément disponible : dans le cas contraire, nous ne pourrions pas calculer le WCET de toute façon lors de la phase IPET.

6.6 Analyse du pipeline sur le CFG

Finalement, l'analyse du pipeline est réalisée par la méthode classique d'*analyse de flot de données* en utilisant une liste de travail (SHARIR et al., 1978 ; ALLEN et al., 1976). L'algorithme détaillé est présenté dans 2.

Algorithm 2 Analyse de pipeline sur CFG

1: **for** $b \in V_{\text{CFG}}$ **do**
2: $OUT[b] \leftarrow \emptyset$
3: $T[b] \leftarrow \emptyset$
4: **end for**
5: $OUT[CFG\_entry] \leftarrow \{\overrightarrow{0}\}$
6: $WorkList : LIFO \leftarrow [CFG\_entry]$
7: **while** $WorkList \neq \emptyset$ **do**
8: $b \leftarrow WorkList.pop()$
9: $IN \leftarrow \emptyset$
10: **for** $pred \in PRED(b)$ **do**
11: $IN \leftarrow IN \cup OUT[pred]$
12: **end for**
13: **for** $\overrightarrow{S_{IN}} \in IN$ **do**
14: $\overrightarrow{S_{IN}} \leftarrow MAJ\_GENERATION(\overrightarrow{S_{IN}}, b)$
15: $t_{ref} \leftarrow (\overrightarrow{S_{IN}} \cdot \mathcal{M}_b^{ref})[\varrho]$
16: $\overrightarrow{S_{OUT}} = \overrightarrow{S_{IN}} \cdot \mathcal{M}_b \oslash t^{ref}$
17: **if** $\overrightarrow{S_{OUT}} \notin OUT[b]$ **then**
18: $OUT[b] \leftarrow OUT[b] \cup \{\overrightarrow{S_{OUT}}\}$
19: $WorkList.push(SUCC(b))$
20: $T[b] \leftarrow T[b] \cup \{t_{ref}\}$
21: **end if**
22: **end for**
23: **end while**

Dans les lignes 1-4, l'état (i.e. l'ensemble des états temporels) sortant de chaque BB OUT, ainsi que les temps d'exécution des BB T sont initialisés à l'ensemble vide. À la ligne 5, l'état sortant du point d'entrée du CFG est initialisé à un ensemble contenant un seul *vecteur temporel* dont l'état de chaque *ressource temporelle* est initialisé à $\text{LEAF}(0)$ car le programme est considéré comme commençant son exécution à la date 0. À la ligne 6, la liste de travail est initialisée avec le point d'entrée du CFG.

La boucle principale des lignes 7-23 réalise l'analyse du pipeline en partant du premier BB de la liste de travail jusqu'à épuisement de cette dernière, c'est-à-dire jusqu'au point fixe des états entrants/sortants de chaque BB. Les lignes 8-12 calculent l'ensemble des états concrets entrants du BB par l'union des états sortants des BB prédécesseurs.

La boucle des lignes 13-22 interprète l'exécution du bloc b pour chaque état concret entrant $\overrightarrow{S_{IN}}$ (un *état temporel* étant un vecteur de XDD) : d'abord, le numéro de génération des évènements dans $\overrightarrow{S_{IN}}$ est incrémenté, ce qui consiste simplement à vérifier les évènements dans chaque XDD dans $\overrightarrow{S_{IN}}$. Si un évènement est associé à une instruction de b, son numéro de génération est incrémenté (ligne 14) ; ensuite, le temps d'exécution du BB (t_{ref}) et l'état sortant $\overrightarrow{S_{OUT}}$ sont calculés à l'aide des matrices pré-calculées (lignes 15-16, en suivant l'équation 6.19). Comme un BB peut être visité plusieurs fois pour trouver tous les états entrants/sortants possibles, le fait de précalculer ces matrices permet effectivement d'accélérer l'analyse.

Dans les lignes 17-21, si l'état n'a jamais été enregistré, l'état concret sortant et le temps d'exécution sont tous deux enregistrés dans OUT et T et les BB successeurs sont rajoutés dans la liste de travail. La boucle principale est répétée, de nouveaux états entrants/sortants sont ajoutés dans l'ensemble jusqu'à ce qu'aucun nouvel état n'apparaisse : l'analyse finit ainsi par atteindre un point fixe.

6.7 Conclusion

Dans ce chapitre, nous avons vu le modèle d'analyse du pipeline basé sur les ressources. Dans ce modèle de calcul, on est capable d'exprimer **l'état du pipeline** sous forme de vecteur de XDD. Chaque élément du vecteur enregistre une partie de l'état temporel du pipeline. Du point de vue du graphe d'exécution, le vecteur contient toute l'information nécessaire pour calculer le temps de chaque nœud en suivant un ordre d'évaluation basé sur l'ordre des instructions et sur l'ordre des étages. Autrement dit, le contenu du vecteur est la date du début ou de fin des nœuds qui sont nécessaires pour résoudre les dépendances des nœuds ultérieurs. Par conséquent, le modèle basé sur les ressources est équivalent à la méthode initiale par graphe d'exécution.

De plus, le fait de pouvoir représenter **l'état du pipeline** ouvre la possibilité de réaliser l'analyse de pipeline au niveau du CFG. À l'aide du calcul de contexte avec le *rebasage*, nous avons simplement construit la *sémantique collectrice* des états du pipeline grâce à laquelle nous calculons l'ensemble des *états temporels du pipeline* à la sortie de chaque BB. En outre, l'*état temporel*, étant un vecteur de XDD, profite des propriétés algébriques des XDD – les transitions d'état sont des fonctions affines sur le demi-anneau XDD. Cela signifie que les transitions sont représentables par des matrices et une séquence de transitions se résume (se pré-calcule) en une seule matrice qui peut s'appliquer sur les XDD du vecteur d'état temporel.

Le but de l'analyse du pipeline basée sur les ressources est d'obtenir un contexte précis qui permet de modéliser les ressources allouées dans le désordre de manière exacte. Par conséquent, dans le prochain chapitre, nous allons expérimentalement évaluer ce modèle de calcul après avoir présenté comment nous pouvons supporter les ressources allouées dans le désordre.

Chapitre 7

Modélisation des ressources allouées dans le désordre

Les modèles d'analyse de pipeline que nous avons présentés dans cette thèse, le premier basé sur le graphe d'exécution (chapitre 4) et le second basé sur les ressources (chapitre 6) fonctionnent uniquement pour les processeurs à exécution dans l'ordre. D'autres variantes du graphe d'exécution ont leur propre méthode pour modéliser les ressources allouées dans le désordre mais elless souffrent soit d'un problème de précision, soit d'un problème de performance : le graphe d'exécution original (voir section 2.6, page 23) utilise le domaine des intervalles pour représenter les temps d'exécution, ce qui peut nuire à la précision ; le graphe d'exécution proposé par ROCHANGE et al., 2009 considère les évènements de manière combinatoire, ce qui cause une complexité exponentielle de l'analyse. Au contraire, notre modèle d'analyse du pipeline calcule explicitement les états temporels du pipeline en prenant en compte les évènements : cela fournit une bonne précision et permet de mitiger le problème de la performance du calcul à l'aide des XDD. Dans ce chapitre, nous proposons d'étendre notre modèle d'analyse du pipeline au support des ressources allouées dans le désordre.

Nous commençons par analyser pourquoi notre modèle basé sur ressources ne peut pas modéliser les ressources allouées dans le désordre. Les *dépendances*, représentant l'utilisation de ressources par le pipeline, sont résolues dans l'ordre topologique du graphe, typiquement l'ordre d'évaluation présenté dans la figure 4.2 (page 56) qui est valide pour tous les processeurs d'exécution dans l'ordre. Cependant, les dépendances ne sont pas suffisantes pour modéliser les ressources allouées dans le désordre parce que l'ordre de résolution du temps d'exécution n'est plus statique mais doit respecter l'ordre d'allocation effective de la ressource par les instructions concernées. Par exemple, pour une ressource allouée selon la politique FCFS, l'ordre de résolution dépend de la date à laquelle les instructions sont prêtes à utiliser la ressource : celle qui arrive (prête) en premier obtient la ressource en premier.

En réalité, les ressources dans le pipeline ne sont pas toutes allouées dans le désordre. La date à laquelle une instruction est prête est déterminée par la partie "d'exécution dans l'ordre". Par conséquent, le calcul du temps d'exécution relatif à une ressource allouée dans le désordre est une procédure en deux étapes : la première étape consiste à déterminer l'ordre d'allocation de la ressource par les instructions concernées (avec leur temps d'exécution sans compter l'usage de cette ressource) et la deuxième étape consiste à résoudre le temps d'exécution effectif passé à utiliser la ressource dans l'ordre d'allocation en considérant les contentions lors des accès concurrents à la ressource partagée.

Heureusement, avec notre modèle de graphe d'exécution, nous avons des dates exactes, sous forme de XDD, auxquelles les instructions sont prêtes à allouer la ressource partagée. Malheureusement, en présence d'évènements, il n'y a pas une

seule date à laquelle les instructions sont prêtes mais plusieurs dates en fonction des évènements : l'ordre d'allocation n'est pas unique mais dépend de l'état des évènements.

L'objectif principal de ce chapitre est de résoudre ce problème : comment calculer l'ordonnancement des accès aux ressources alloués dans le désordre pour **toute configuration** d'évènements et ainsi, par la suite, calculer le temps d'exécution des instructions concernées par ces ressources.

Comme les politiques d'ordonnancement des ressources allouées dans le désordre sont assez variables, cette section s'intéresse à une ressource commune à de nombreux processeurs : le bus partagé par le cache d'instructions et le cache de données pour communiquer avec la mémoire centrale ou avec un cache L2 unifié. Le bus est alloué selon une politique FCFS avec priorité aux accès à la mémoire de donnée, donc le bus peut être alloué dans le désordre. Par exemple, un accès au cache d'instructions qui fait un *Miss* peut prendre le bus avant une instruction d'accès mémoire **antérieure** [1] qui accède au bus à cause d'un *Miss* dans le cache de données.

7.1 Ordre d'allocation du bus partagé FCFS

Soit le pipeline classique de 5 étages (FE-DE-EX-ME-CM) considéré dans le chapitre 4. Il a basiquement une exécution dans l'ordre mais, si on y ajoute un bus FCFS partagé par le cache d'instruction (accédé à l'étage FE) et le cache de données (accédé à l'étage ME), on obtient un pipeline qui n'a plus une exécution purement dans l'ordre. Une contention peut arriver lors de l'accès à ce bus partagé : lors d'un cache *Miss* aux étages FE ou ME. Donc, les *anomalies temporelles* deviennent possibles car l'allocation du bus est dynamique. Un exemple concret de l'occurence des *anomalies temporelles* dans cette configuration du pipeline est présenté dans HAHN et al., 2016. Mais comme le pipeline a, de base, une exécution dans l'ordre, la contention n'est pas totalement libre. Par exemple, une instruction accédant au bus à l'étage ME ne peut être en contention avec aucune instruction suivante au même étage parce que l'étage ME a une exécution dans l'ordre. De la même manière, les accès au bus partagé à l'étage FE doivent respecter l'ordre du programme et donc ne sont pas en contention entre eux. De plus, l'*ordre du pipeline* assure qu'un accès au bus à l'étage FE est forcément antérieur à l'accès à l'étage ME de la même instruction.

Toutes ces contraintes signifient que l'allocation du bus partagé se fait souvent dans l'ordre du programme sauf dans un cas : un accès *Miss* au cache de données à l'étage ME (noté ME_0) peut être retardé à cause de la contention avec une instruction postérieure (noté $FE_{i|i>0}$). Pour simplifier la notation, ME_0 et FE_i dénote à la fois l'instruction concernée par l'accès et le nœud concerné du graphe d'exécution. Par exemple, ME_0 dénote aussi $[ME_0/ME]$, FE_i dénote $[FE_i/FE]$. Or, FE_i peut retarder ME_0 seulement si FE_i est prêt à rentrer dans l'étage FE avant que ME_0 soit prêt à rentrer dans ME. Cette situation peut seulement arriver quand FE_i ne dépend pas de ME_0, i.e. dans le graphe d'exécution correspondant, il n'existe pas de *chemin de dépendance* – une chaîne d'arcs dans le graphe, de ME_0 à FE_i. Dans le pipeline considéré, l'occurrence de cette situation est limitée par la taille des files entre les étages car nous pouvons trouver **au pire** le *chemin de dépendance* ci-dessous :

$$[I_i/ME] \dashrightarrow [I_{i+2}/EX] \dashrightarrow [I_{i+4}/DE] \dashrightarrow [i_{i+6}/FE] \rightsquigarrow [i_{i+6+k}/FE] \tag{7.1}$$

1. L'étage d'accès mémoire à une donnée se trouvant forcément après l'étage de fetch, ce genre de contention ne peut se produire qu'avec une instruction antérieure.

dont les trois premières dépendances sont causées par l'*ordre induit par la capacité des files* avec des files de taille 2. Ces dépendances assurent que I_{i+6} à l'étage FE ne peut pas démarrer avant I_i à l'étage ME. De plus, avec l'*ordre de fetch*, les instructions dans différents blocs de mémoire sont forcément accédées dans l'ordre. Si on considère un cas où un bloc de cache fait 16 octets (pour un processeur RISC 32 bits, c'est 4 instructions), au bout d'au maximum 4 instructions après I_{i+6}, nous aurons une dépendance d'arc solide qui garantit que $[I_{i+6+k}/FE]$ avec $k \leq 4$ est exécuté après $[I_{i+6}/FE]$ et par conséquent après $[I_i/ME]$: les instructions au-delà de $i+6+3$ ne peuvent pas être en contention avec l'instruction I_i.

Quand on s'intéresse à la contention entre FE_i et ME_0, les autres instructions se trouvant entre elles ne sont pas considérées car elles n'utilisent pas le bus. Mais elles sont prises en compte par le modèle d'analyse basé sur les ressources qui est capable de gérer l'exécution dans l'ordre. En effet, nous avons ici une approche similaire à celle du graphe d'exécution original (cf. section 2.6, page 23) qui est de gérer la partie d'exécution dans l'ordre séparément de la partie d'exécution dans le désordre.

En prenant en compte les contraintes sur l'allocation du bus, la table 7.1 montre toutes les situations possibles pour un accès à ME_0 en contention avec trois nœuds postérieurs : FE_1, FE_2 et FE_3 **pour une configuration donnée des évènements**. La première colonne montre l'ordre d'allocation du bus par les 4 nœuds. Considérant la politique FCFS, l'ordonnancement de ME_0 est déterminé en comparant la date à laquelle ME_0 est prêt à prendre le bus (ρ_{ME_0}) avec celle de FE_1, FE_2 et FE_3, respectivement ρ_{FE_1}, ρ_{FE_2} et ρ_{FE_3}. Les ρ sont en fait les dates de démarrage de ces nœuds sans considérer le bus, calculables par le modèle basé sur ressources (la valeur du *pointeur de temps* de l'*état temporel* au moment correspondant). La deuxième colonne de la table est la condition qui détermine l'ordonnancement des accès qui reflète simplement le politique FCFS. En fonction de l'ordre d'accès au bus, le temps d'attente de ME_0 est différent. La date à laquelle ME_0 accède effectivement au bus est matérialisé dans la troisième colonne : si ME_0 est le premier à être prêt, il prend le bus sans attente ; si seul FE_1 est prêt avant ME_0, FE_1 prend le bus à la date où il est prêt (ρ_{FE_1}) et libère le bus après une latence d'utilisation du bus (λ_{BUS}), donc si ME_0 est prêt avant la libération du bus par FE_1, ME_0 doit attendre jusqu'à la libération $(\rho_{FE_1} + \lambda_{BUS})$ alors que si ME_0 est prêt après la libération par FE_1, ME_0 prend le bus quand il est prêt (ρ_{ME_0}). Finalement, cela donne $max(\rho_{FE_1} + \lambda_{BUS}, \rho_{ME_0})$. La situation est similaire si FE_2 ou FE_3 prennent le bus avant ME_0 sachant que si FE_2 prend le bus avant ME_0, c'est que FE_1 prend forcément le bus avant FE_2 et ME_0 et le ρ_{FE_2} doit être mis à jour en considérant l'utilisation du bus à cause de FE_1, avant de comparer avec ρ_{ME_0}. Donc, la procédure de calcul de contention doit comparer les dates des FE_i avec celle de ME_0 dans l'ordre : d'abord FE_1 avec ME_0, si FE_1 prend le bus avant, considérons l'accès du bus par FE_1 et calculer ρ_{FE_2}, puis comparer ρ_{FE_2} avec ρ_{ME_0} et ainsi de suite jusqu'au FE_i qui ne passe plus avant ME_0.

Ordonnencement	Condition	Date d'ordonnencement de ME_0
ME_0, FE_1, FE_2, FE_3	$\rho_{ME_0} \leq \rho_{FE_1}$	ρ_{ME_0}
FE_1, ME_0, FE_2, FE_3	$\rho_{FE_1} < \rho_{ME_0} \leq \rho_{FE_2}$	$max(\rho_{FE_1} + \lambda_{BUS}, \rho_{ME_0})$
FE_1, FE_2, ME_0, FE_3	$\rho_{FE_2} < \rho_{ME_0} \leq \rho_{FE_3}$	$max(\rho_{FE_2} + \lambda_{BUS}, \rho_{ME_0})$
FE_1, FE_2, FE_3, ME_0	$\rho_{FE_3} < \rho_{ME_0}$	$max(\rho_{FE_3} + \lambda_{BUS}, \rho_{ME_0})$

TABLE 7.1 – Ordonnancement de ME_0 avec les FE_is suivants.

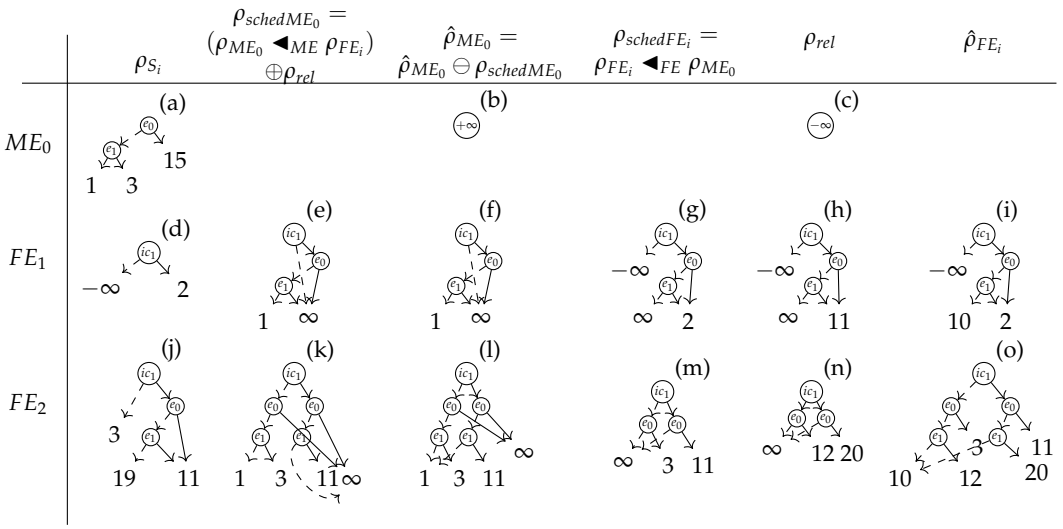


FIGURE 7.1 – Ordonnancement du bus avec des XDD.

7.2 Ordonnancement du bus avec les XDD

La table 7.1 montre les différents ordonnancements de ME_0 pour **toutes les configurations possibles en fonction des évènements**. Ainsi, l'ordonnancement peut être différent en fonction de la configuration et cette section montre comment calculer l'ordonnancement pour toutes les configurations au sein d'un XDD.

La figure 7.1 donne un exemple du calcul réalisé pour un cas simple : ME_0 accède au bus et peut être retardé par FE_1 et FE_2. Le *fetch* de FE_1 peut faire un *Miss* ou un *Hit*, représenté par l'évènement ic_1. Nous rappelons que le bus est seulement accédé dans le cas *Miss* ; dans le cas *Hit*, FE_1 n'est pas en contention avec ME_0. Le *fetch* de FE_2 est classifié comme *Always Miss* donc le bus est toujours accédé. Nous supposons que la latence d'utilisation (λ_{BUS}) du bus est de 9 cycles. Les autres évènements (e_0, e_1) sont des évènements qui déterminent le temps d'exécution des nœuds (donc qui peuvent déterminer l'ordre d'allocation du bus) mais qui ne sont pas liés à l'allocation du bus par les nœuds concernés.

Dans cet exemple, l'état initial du pipeline avant ME_0, nommé $\overrightarrow{S_0}$, est choisi de manière arbitraire mais il n'est pas montré dans la figure (seul le *pointeur de temps* en est extrait – ρ_{ME_0}). L'exécution entre les nœuds accédant le bus n'est pas concernée par la contention. Ainsi, l'effet de leur exécution est pré-calculée par des matrices : la matrice $\mathcal{M}_{ME_0-FE_1}$ résume l'exécution du pipeline entre ME_0 et FE_1. Les dates auxquelles les nœuds sont prêts à accéder le bus, ρ_{S_i}, peuvent ainsi être calculées à partir de l'état $\overrightarrow{S_0}$. Grâce à cet exemple, nous allons montrer comment les contentions peuvent être déterminées à partir des ρ_{ME_0} et ρ_{FE_i}.

Le XDD (a) de la figure est la date à laquelle ME_0 est prêt, obtenu à partir de l'état $\overrightarrow{S_0}$. Le XDD (b) est la valeur initiale de $\hat{\rho}_{ME_0}$ qui enregistre la date d'ordonnancement définitif de ME_0. La valeur initiale $+\infty$ signifie que cet accès n'est ordonnancé pour aucune des configurations. Le XDD (c) est la valeur initiale de ρ_{rel} qui enregistre la date de libération du bus (par les FE_i) : la valeur $-\infty$ signifie que le bus n'est accédé par aucun des FE_i pour l'instant.

La date de démarrage de FE_1 (d) est calculée à partir de l'état de pipeline avant ME_0 ($\overrightarrow{S_0}$) et l'exécution (dans l'ordre) entre ME_0 et FE_1, représentée par la matrice $\mathcal{M}_{ME_0-FE_1}$. Autrement dit, c'est la date à laquelle FE_1 est prêt à allouer le bus. L'évènement ic_1 est annoté avec $-\infty$ quand il est inactif car le bus n'est pas accédé par

FE_1.

D'abord, ρ_{ME_0} (a) est comparé avec ρ_{FE_1} (d) avec l'opérateur $\blacktriangleleft_{ME}$ qui calcule les configurations et les dates auxquelles ME_0 prend le bus $\rho_{schedME_0}$ (e). Comme le bus est alloué selon la politique FCFS, $\blacktriangleleft_{ME}$ donne simplement le bus à ME_0 si la date de démarrage de ME_0 est inférieure ou égale à celle de FE_1. La définition formelle de $\blacktriangleleft_{ME}$ est présentée dans l'équation 7.3. Les configurations où ME_0 ne prend pas le bus avant FE_1 sont marquées par $+\infty$ dans $\rho_{schedME_0}$ (e). Notons que le $-\infty$ dans ρ_{FE_1} ne permet pas à ME_0 d'être ordonnancé car c'est le cas où FE_1 n'accède pas au bus mais les FE suivants (FE_2 dans l'exemple) peuvent prendre le bus avant ME_0 et nous n'avons pas encore comparé la date de FE_2 avec celle de ME_0. Ce comportement est bien représenté par le $\blacktriangleleft_{ME}$ car $-\infty$ est inférieur à toutes les dates de ρ_{ME_0} ce qui produit un $+\infty$ dans $\rho_{schedME_0}$, c'est-à-dire que ME_0 n'est pas ordonnancé pour ces configurations. Ensuite, $\hat{\rho}_{ME_0}$ qui était $+\infty$ (b) (pas encore ordonnancé) est mis à jour en considérant $\rho_{schedME_0}$. La mise à jour est faite avec l'opérateur $\ominus$ (le minimum transféré dans le domaine $\mathcal{XDD}$) ce qui donne une nouvelle date $\hat{\rho}_{ME_0}$ (f). Dans (f), nous pouvons voir que la seule configuration où ME_0 est définitivement ordonnancé : $ic_1\overline{e_0e_1}$ est la seule configuration où ME_0 est prêt à la date 1, inférieure à celle de FE_1. Dans les autres configurations, soit ME_0 a une date supérieure que FE_1 donc ME_0 accède au bus après FE_1, soit FE_1 n'accède pas au bus et donc l'ordonnancement de ces configurations reste à déterminer avec les comparaisons des dates de démarrage des FE_i suivants. Dans les deux cas, l'ordonnancement de ME_0 sera déterminé plus tard en comparant avec FE_2.

Dans $\rho_{schedFE_1}$ (g), les dates auxquelles FE_1 prend le bus sont calculées en comparant ρ_{FE_1} et ρ_{ME_0} avec l'opérateur $\blacktriangleleft_{FE}$. La seule différence entre $\blacktriangleleft_{FE}$ et $\blacktriangleleft_{ME}$ est que $\blacktriangleleft_{ME}$ compare avec $\leq$ au lieu de $<$ car l'étage ME a la priorité sur FE quand ils sont prêts à prendre le bus au même cycle (cf. équation 7.3). En rajoutant ($\otimes$) la latence d'utilisation du bus (λ_{BUS}) à $\rho_{schedFE}$, nous pouvons mettre à jour (avec $\oplus$) la date de libération du bus (par les FE) après FE_i : ρ_{rel} (h). Finalement, nous pouvons mettre à jour l'ordonnancement définitif du FE_1, $\hat{\rho}_{FE_1}$ (i), qui est $\rho_{schedFE_1}$ pour les configurations où FE_1 est ordonnancé avant ME_0, sinon $(\hat{\rho}_{ME_0} \otimes \lambda_{BUS})$ si ME_0 est ordonnancé avant. Comme les FE_i sont forcément exécutés dans l'ordre, le $\hat{\rho}_{FE_1}$ est définitif pour les configurations où FE_1 accède le bus avant ME_0 (les configurations qui ne sont pas $-\infty$). La date définitive pour les configurations $-\infty$ est simplement la date correspondante dans ρ_{FE_1} car FE_1 n'est pas concerné par la contention dans ces configurations.

Une fois que l'ordonnancement définitif de FE_1 est connu, la latence due à la contention est ajoutée dans l'*état temporel*, c'est-à-dire dans le *pointeur de temps* de l'*état temporel* à la fin de FE_1 ce qui permet de calculer l'*état temporel* au début de FE_2 (et donc de calculer ρ_{FE_2}).

La deuxième itération suit presque la même procédure. Nous calculons l'ordonnancement de FE_2 par rapport à ME_0. Tout d'abord, l'état du pipeline avant FE_2 est calculé à partir de l'état temporel de FE_1 (après avoir compté la contention entre FE_1 et ME_0, parce que la date de démarrage de FE_2 dépend de l'utilisation du bus par FE_1), et la date de démarrage de FE_2 (ρ_{FE_2}) est extraite de cet *état temporel* (j). Ensuite, ρ_{ME_0} (a) est comparé avec ρ_{FE_2} (j) avec l'opérateur $\blacktriangleleft_{ME}$. L'ordonnancement de ME_0 $\rho_{schedME_0}$ (k) est calculé en considérant le maximum (avec $\oplus$) entre $\rho_{schedME_0}$ et la date de libération du bus ρ_{rel}, comme montré dans la troisième colonne de la table 7.1. Ensuite, $\hat{\rho}_{ME_0}$ est mis à jour (l). L'ordonnancement de FE_2 ($\rho_{schedFE_2}$ (m)) est calculé avec l'opérateur $\blacktriangleleft_{FE}$ appliqué à ρ_{FE_2} et à ρ_{ME_0} ce qui permet de mettre à jour la date de libération du bus ρ_{rel} (n). L'ordonnancement définitif de FE_2 (o) est calculé

de la même manière que pour FE_1 : à la libération du bus par ME_0 si ME_0 prend le bus avant, sinon à la date de démarrage de FE_2.

À la fin de la séquence des nœuds en contention, il n'y a plus d'instructions à prendre à compte[2]. Les $+\infty$ dans $\hat{\rho}_{ME_0}$ à la fin représentent des configurations où ME_0 ne prend pas le bus ni avant FE_1 ni avant FE_2. Il existe encore deux cas pour ces configurations $+\infty$:

Cas 1 : Si ME_0 est prêt **avant** la libération du bus après l'utilisation par FE_1 et FE_2, c'est-à-dire que ME_0 attend la libération du bus puisque FE_1 et FE_2 l'utilisent avant ME_0, ces $+\infty$ doivent être remplacés par la date de libération du bus.

Cas 2 : Si ME_0 est prêt **après** la libération du bus après l'utilisation par FE_1 et FE_2, cela signifie que ME_0 ne subit pas vraiment de contention avec FE_1 et FE_2 mais simplement est prêt plus tard – après que FE_1 et FE_2 aient déjà fini d'utiliser le bus. Dans ce cas, la date de démarrage définitive de ME_0 est simplement la date à laquelle il est prêt (ρ_{ME_0}).

En considérant ces deux cas, le remplacement des $+\infty$ est fait par :

$$\hat{\rho}_{ME_0}(\text{définitif}) = \hat{\rho}_{ME_0}(\text{avec} + \infty) \ominus (\rho_{ME_0} \oplus \rho_{rel}). \tag{7.2}$$

où $\rho_{ME_0} \oplus \rho_{rel}$ donne, pour chaque configuration, le maximum entre la date de démarrage de ME_0 et la date de libération du bus. Le $\ominus$ effectue le remplacement des $+\infty$ dans $\hat{\rho}_{ME_0}$ (les configurations non $+\infty$ ne sont pas changées parce que ces valeurs sont forcément inférieures à celles dans ρ_{rel}).

Les opérateurs $\blacktriangleleft_{FE}$ et $\blacktriangleleft_{ME}$ utilisés dans le calcul de contention sont définis ci-dessous :

$$\begin{aligned} &\forall f^{\#}_{ME}, f^{\#}_{FE} \in \mathcal{XDD}^2, \forall \gamma \in \Gamma, \\ &(f^{\#}_{ME} \blacktriangleleft_{ME} f^{\#}_{FE})[\gamma] = \begin{cases} f^{\#}_{ME}[\gamma] & \text{if } f^{\#}_{ME}[\gamma] \leq f^{\#}_{FE}[\gamma], \\ +\infty & \text{sinon} \end{cases} \\ &(f^{\#}_{FE} \blacktriangleleft_{FE} f^{\#}_{ME})[\gamma] = \begin{cases} f^{\#}_{FE}[\gamma] & \text{if } f^{\#}_{FE}[\gamma] < f^{\#}_{ME}[\gamma], \\ +\infty & \text{sinon} \end{cases} \end{aligned} \tag{7.3}$$

avec $+\infty$ dénotant les configurations ou les FE et ME ne prennent pas le bus.

Le calcul de contention pour supporter les XDD, présenté dans cette section, semble complexe mais les XDD représentent efficacement un grand nombre de configurations et permettent de résoudre l'ordonnancement de ces configurations en un nombre limité de passes, quel que soit le nombre de configurations, ce qui permet d'obtenir une performance importante dans notre analyse.

7.3 Algorithme de contention

L'exemple d'analyse de contention présenté dans la section précédente est décrit plus formellement ici. Tout d'abord, le modèle de pipeline présenté dans le chapitre 6 est étendu en divisant chaque BB selon les *points de contention* i.e. les nœuds pouvant potentiellement subir une contention à cause d'un accès au bus. Ils sont regroupés sous forme de séquences qui commencent par un ME_0 et suivi par plusieurs

2. Grâce au *chemin de dépendance* présenté par l'équation 7.1, la longueur de cette séquence est toujours bornée.

FE_i comme dans l'exemple de la section précédente (car c'est la seule forme dans laquelle la contention est possible, section 7.1). Les nœuds entre les *points de contention* sont exécutés dans l'ordre du programme et donc résumés par une matrice de XDD.

Dans la section précédente, nous avons également considéré une séquence simple $ME_0 - FE_{i\geq 1}$. Dans la pratique, il peut exister plusieurs nœuds ME_i qui utilisent le bus entre ME_0 et les FE_i mais comme l'étage ME a une exécution dans l'ordre, ces nœuds sont forcément exécutés après ME_0 : les ME_i ne sont pas en contention entre eux mais peuvent être en contention avec les FE_i qui suivent. Le calcul de contention de ces nœuds est effectué après celui de la séquence $ME_0 - FE_{i\geq 1}$. On procédera alors de la même manière qu'avec la séquence $ME_1 - FE_{i\geq 1}$ mais en prenant en compte les dates de démarrage déjà fixées pour les FE_i dans le calcul précédent.

L'algorithme 3 réalise le calcul de contention : il détermine la date définitive d'accès au bus pour une instruction à l'étage ME en considérant la contention avec les instructions suivantes dans l'étage FE. Il prend en entrée l'*état temporel* $\overrightarrow{S_0}$ avant ME_0 et la séquence en contention $(ME_0, FE_{1\leq i\leq n})$. Il renvoie en sortie la date de démarrage définitive de ME_0 : $\hat{\rho}_{ME_0}$, et la date de démarrage des FE_i en considérant la contention avec ME_0 : les $\hat{\rho}_{FE_i}$. Les dates $\hat{\rho}_{FE_i}$ ne sont pas globalement définitives (à l'exception des configurations où les FE_i acquièrent le bus avant ME_0) car, comme expliqué plus haut, il peut exister d'autres ME_i entre ME_0 et les FE_i qui donneront lieu à une nouvelle itération de l'algorithme qui complétera les $\hat{\rho}_{FE_i}$. Les $\hat{\rho}_{FE_i}$ deviennent définitives une fois que tous les ME les précédant sont traités.

Algorithm 3 Calcul de Contention.

Input $\overrightarrow{S}_0 \in \mathcal{S}, (ME_0, FE_{1\leq i\leq n})$
Output $(\hat{\rho}_{ME_0}, \hat{\rho}_{FE_{1\leq i\leq n}})$

1: $\hat{\rho}_{ME_0} = \text{LEAF}(+\infty)$
2: $\rho_{rel} := \text{LEAF}(-\infty)$
3: $\overrightarrow{S}_{FE_1} := \overrightarrow{S}_{ME_0} \cdot \mathcal{M}_{ME_0-FE_1}$
4: $i := 1$;
5: $\rho_{ME_0} := \overrightarrow{S}_0[i_\rho]$
6: **while** $i \leq n \wedge (\exists \gamma \in \Gamma \wedge \hat{\rho}_{ME_0}[\gamma] = +\infty)$ **do**
7: **if** $FE_i.mustUseBus()$ **then**
8: $\rho_{FE_i} := \overrightarrow{S}_{FE_i}[i_{FE}]$
9: **else**
10: $\rho_{FE_i} := \overrightarrow{S}_{FE_i}[i_{FE}] \otimes \text{NODE}(ic_i, -\infty, 0)$
11: **end if**
12: $\rho_{schedME_0} := (\rho_{ME_0} \blacktriangleleft_{ME} \rho_{FE_i}) \oplus \rho_{rel}$
13: $\hat{\rho}_{ME_0} := \hat{\rho}_{ME_0} \ominus \rho_{schedME_0}$
14: $\rho_{schedFE_i} := \rho_{FE_i} \blacktriangleleft_{FE} \rho_{ME_0}$
15: $\rho_{rel} := \rho_{rel} \oplus (\rho_{schedFE_i} \otimes \lambda_{BUS})$
16: $\hat{\rho}_{FE_i} := \rho_{schedFE_i} \ominus (\hat{\rho}_{ME_0} \otimes \lambda_{ME_0})$
17: $\overrightarrow{S}_{FE_{i+1}} := (\overrightarrow{S}_{FE_i} \oplus [\mathbb{0}, ..., \mathbb{0}, \hat{\rho}_{FE_i} \otimes \lambda_{BUS}]) \cdot \mathcal{M}_{FE_i-FE_{i+1}} \quad i = i + 1$
18: **end while**
19: $\hat{\rho}_{ME_0} := \hat{\rho}_{ME_0} \ominus (\rho_{rel} \oplus \rho_{ME_0})$

Initialement, ME_0 est considéré non-ordonnancé pour toute configuration et donc $\hat{\rho}_{ME0}$ est initialisé à $\text{LEAF}(+\infty)$ (ligne 1). Il sera mis à jour en analysant la contention avec chacun des FE_i suivants. Quand ME_0 ne contient plus $+\infty$ ou bien tous les

FE_i sont traités, l'ordonnancement de ME_0 devient complet (condition à la ligne 6). La deuxième ligne initialise ρ_{rel} (qui enregistre la date de libération du bus) à $-\infty$ car aucun des FE_i n'est traité et donc la date de libération du bus est inconnue.

À la ligne 3, l'*état temporel* pour FE_1 est calculé en appliquant la matrice $\mathcal{M}_{ME_0-FE_1}$ à l'état initial $\overrightarrow{S_0}$. Le compteur i est initialisé et aidera à itérer sur tous les *points de contention* de 1 à n. La date de démarrage de ME_0 est initialisée à la ligne 5 à partir de la valeur du *pointeur de temps* de l'*état temporel* initial. Les lignes 7-11 calculent la date de démarrage de FE_i si l'accès est classifié en *Always Miss* ou *Not Classified* (donné par la fonction $mustUseBus()$). Le cas *Not Classified* est supporté par un évènement, par exemple ic_1, et un XDD $\text{NODE}(ic_1, -\infty, 0)$ est additionné ($\otimes$) au ρ_{FE_i} pour représenter l'accès conditionnel au bus. La valeur $-\infty$ lorsque l'évènement n'est pas activé dénote que le bus n'est pas accédé dans cette configuration.

$\rho_{schedME_0}$, contenant les configurations où ME_0 prend le bus avant FE_i, est calculé avec $\blacktriangleleft_{ME}$ à la ligne 12. En accord avec la troisième colonne de la table 7.1, la date d'ordonnancement de ME_0 est le maximum entre la date de libération du bus (ρ_{rel}) et la date de démarrage de ME_0. L'ordonnancement de ME_0 à l'itération i (en considérant FE_i) est accumulé dans l'ordonnancement définitif de ME_0 ($\hat{\rho}_{ME_0}$) à la ligne 13. L'opération $\ominus$ à la même ligne permet d'enregistrer les configurations qui sont déjà ordonnancées dans $\hat{\rho}_{ME_0}$ mais les configurations non-ordonnancées ($+\infty$) sont remplacées par de nouvelles valeurs venant de $\rho_{schedME_0}$. À la ligne 14, l'ordonnancement de FE_i est calculé. Après ce calcul, il peut exister des $-\infty$ dans ρ_{FE_i} pour dénoter des cas où FE_i n'accède pas le bus, ces $-\infty$ sont préservées dans $\rho_{schedFE_i}$ grâce à $\blacktriangleleft_{FE}$. Ensuite, la date de libération du bus ρ_{rel} est mise à jour à la ligne 15. L'opérateur $\oplus$ utilisé pour la mise à jour assure que les $-\infty$ dans $\rho_{schedFE_i}$ n'écrasent pas les valeurs dans ρ_{rel} (car $-\infty \oplus x = x$). Nous notons aussi que les $+\infty$ dans ρ_{rel} ne peuvent pas écraser la date de libération de FE_i car, si les FE_i précédents $FE_j | j < i$ n'ont pas pris le bus en concurrence avec ME_0 (d'où le $+\infty$), FE_i ne peut pas prendre le bus non plus (les FE_i ont une exécution dans l'ordre).

À la ligne 16, l'ordonnancement des FE_i est calculé en remplaçant les $+\infty$ dans $\rho_{schedFE_i}$ (où FE_i perd la contention contre ME_0) par la date de libération du bus par ME_0 ($\hat{\rho}_{ME_0} \otimes \lambda_{ME_0}$).

Le remplacement peut être fait avec l'opérateur $\ominus$ pour deux raisons : (a) les configurations dont la valeur est $+\infty$ dans $\rho_{schedFE_i}$ ne sont pas $+\infty$ dans $\rho_{schedME_0}$ car un des deux, soit le FE_i soit ME_0 est gagnant lors de la contention ; (b) les configurations dans $\rho_{schedFE_i}$ qui ne sont pas $+\infty$ ont forcément une date antérieure à celle dans $\hat{\rho}_{ME_0}$ (car ce sont des configurations où FE_i gagne la contention). Pour ces deux raisons, utiliser $\ominus$ à la ligne 16 permet de remplacer uniquement les $+\infty$ dans $\rho_{schedFE_i}$ par la date correspondante dans $\hat{\rho}_{ME_0} \otimes \lambda_{ME_0}$.

À la ligne 17, l'*état temporel* est mis à jour en fonction de l'ordonnancement des FE_i en appliquant $\oplus$ (maximum) entre le *pointeur de temps* de l'état temporel et la date de libération du bus. L'état mis à jour est à nouveau multiplié par la matrice $\mathcal{M}_{FE_i-FE_{i+1}}$ qui représente l'exécution dans l'ordre entre FE_i et FE_{i+1} ce qui donne l'*état temporel* avant FE_{i+1} en prenant en compte l'ordonnancement de FE_i sur le bus.

À la ligne 19, les configurations qui ne sont pas encore ordonnancées (celles qui valent $+\infty$ dans $\hat{\rho}_{ME_0}$) sont remplacées par le maximum entre la date de libération du bus et la date démarrage de ME_0 (si ME_0 est prêt après la libération du bus par tous les FE) puisqu'il n'y a plus de FE qui peut être en contention avec ME_0 (équation 7.2).

En fait, le fait de définir la séquence des instructions en contention et d'arrêter le calcul au bout de la séquence est uniquement une optimisation car si nous calculions

le prochain FE qui accède le bus, sa date de démarrage serait forcément supérieure à celle de ME_0 pour toute configuration : elle est garantie par la chaîne de dépendance de l'équation 7.1. Comme la longueur de la chaîne de dépendance est connue statiquement, la taille de la séquence est connue statiquement. Nous pouvons donc éviter certains calculs en arrêtant le calcul quand la fin de la séquence est atteinte.

7.4 Évaluation

Dans la partie précédente, nous avons étendu le modèle d'analyse de pipeline basé sur les ressources (chapitre 6) en une analyse au niveau du CFG qui supporte le bus mémoire partagé. Cette section évalue maintenant les performances de la nouvelle analyse.

7.4.1 Mise en place

Comme nous nous intéressons aux comportements de l'exécution dans le désordre, nous avons choisi la micro-architecture complexe qui fournit plus de parallélisme. Le pipeline est de 4 étages (FE, DE, EX, CM). L'étage FE est capable de charger au maximum 4 instructions par cycle (si les instructions sont dans le même bloc de mémoire) avec une latence de 7 cycles en cas de *Miss* de cache (sans compter la contention), 1 cycle sinon. L'étage DE décode les instructions en 1 cycle. L'étage EX traite toutes les opérations arithmétiques, en virgule flottante ainsi que des opérations de mémoire avec plusieurs types d'unités fonctionnelles : 4 unites arithmétiques et logiques (ALU – Arithmetic and Logic Units) sont disponibles et sont capables d'exécuter 4 instructions arithmétiques simultanément. La latence des opérations arithmétiques est de 1 cycle pour l'addition et soustraction, 2 cycles pour la multiplication et 7 cycles pour la division. Une unité de calcul en nombre flottant (FPU – Floatting Point Unit) est disponible avec une latence de 3 cycles pour l'addition et la soustraction, 5 cycles pour la multiplication et 12 cycles pour la division. Une seule unité mémoire (MU – Memory Unit) est disponible pour exécuter les lectures et les écritures en mémoire. Dans le cas des instructions à lecture/écriture multiples (typiquement LDM, STM dans le jeu d'instruction *ARM*), les accès mémoire sont exécutés dans l'ordre et si un des accès multiples requiert le bus, ce dernier est occupé jusqu'à ce que tous les accès soient finis (donc aucun FE n'est autorisé à allouer le bus durant ce temps-là). La latence d'accès à la mémoire depuis le cache de données est la même que celle de la mémoire d'instruction (1 cycle en cas de *Hit*, 7 cycles en cas de *Miss*). À l'étage EX, un tampon d'émission (*Issue Buffer*) distribue les instructions aux unités fonctionnelles correspondantes. Les instructions utilisant la même unité fonctionnelle sont exécutées dans l'ordre. En revanche, les instructions utilisant des unités fonctionnelles différentes peuvent être exécutées dans le désordre à l'étage EX (si aucun aléa ne se produit). La définition de "l'exécution dans le désordre" n'est pas unifiée dans la littérature : ici, les instructions qui utilisent différentes unités fonctionnelles sont effectivement exécutées dans le désordre à l'étage EX mais cela ne pose pas de problème pour notre modèle basé sur les graphes d'exécution car toutes les dépendances sont statiquement connues : l'ordre de l'allocation de chaque type d'unité fonctionnelle est l'ordre du programme.

Le cache d'instructions a une taille de 16-Ko et est associatif par ensembles sur 2 voies avec une politique de remplacement LRU. Le cache de données est identique mais avec une taille de 8-Ko. Il existe un seul niveau de cache et les deux caches partagent le même bus FCFS pour accéder la mémoire centrale.

L'analyse est implantée dans l'outil OTAWA qui fournit aussi les analyses globales et d'autres éléments d'infrastructure nécessaires pour calculer le WCET. La seule modification est l'analyse du pipeline réalisée à partir de l'algorithme proposé dans le chapitre 6 et étendu dans ce chapitre : on a donc une analyse de pipeline basée sur ressources utilisant des XDD et étendue avec le calcul de contention. Le benchmark est TACLe compilé pour le jeu d'instructions *armv7* avec unité de calcul en nombre flottant matérielle. Sur les 79 benchmarks, 5 ne fonctionnent pas avec OTAWA à cause de constructions de flots de contrôle non supportés, principalement les branchements dynamiques [3]. Cinq autres benchmarks sont éliminés à cause du manque d'annotations sur les appels récursifs et les bornes de boucle [4].

7.4.2 Nombre d'états temporels

Comme l'analyse calcule la *sémantique collectrice* du modèle d'analyse du pipeline basé sur les ressources, un risque important est l'explosion du nombre d'états. La divergence d'*états temporels* est causée à la fois par les évènements et par le flot de contrôle (chemins d'exécution multiples). Ces deux facteurs ne sont pas orthogonaux : nous espérons que les évènements ont, dans la plupart des cas, un effet à court terme, ce qui fait que les *états temporels* suivant des chemins d'exécution différents ne varient pas trop.

Les premières mesures évaluent le nombre d'*états temporels* obtenus par arc du CFG (c'est-à-dire l'ensemble d'états sortant du BB prédécesseur et ceux entrant dans le BB successeur quand ils sont exécutés en suivant cet arc). Le résultat expérimental est présenté sur la figure 7.2. L'axe x montre le nombre d'*états temporels* du pipeline (ce qui peut représenter un nombre encore plus grand d'états **concrets** du pipeline sans utiliser des XDD). L'axe y, en échelle logarithmique, représente le nombre d'arcs du CFG ayant ce nombre d'états. Le nombre est cumulé pour tous les benchmarks.

Ces mesures nous montrent que la plupart des arcs ont moins de 20 états temporels possibles, signifiant que les variations temporelles ont empiriquement un effet court terme. Le fait qu'elles n'ont plus d'effet est représenté efficacement par les XDD et le modèle d'analyse du pipeline. Il existe également certains cas rares où le nombre d'états est beaucoup plus élevé mais comme ils sont rares, leur effet sur la performance générale de l'analyse est assez limité. La performance générale (le temps d'analyse) sera également évaluée dans la section 7.5. L'effet de compactage des XDD continue ici à jouer malgré ou grâce aux contentions. Même en présence de quelques cas rares avec un grand nombre d'états temporels, l'analyse reste efficace.

7.4.3 Durée de vie des évènements

La performance de l'analyse est fortement liée à l'effet des évènements à long ou à court terme. Plus courte est la durée de vie des évènements, plus petite est la taille des XDD et moins d'états temporels sont possibles pour chaque BB.

Par conséquent, le deuxième critère de performance mesure la durée de vie des évènements par la distance (en nombre d'instructions) entre l'apparition d'un évènement jusqu'à ce qu'il disparaisse du XDD par rebasage (voir section 6.4, page 88) ou par *effet d'amortissement* (voir section 4.3, page 57). Un évènement est considéré mort en un *point de contention* (les nœuds où la ressource d'exécution dans le désordre peut être accédée) si aucun XDD dans le vecteur d'état temporel ne contient cet évènement. À l'inverse, un évènement est vivant en un *point de contention* si l'évènement

3. *pm*, *mpeg*2, *gsm_enc*, *ammunition*, *rosace*
4. *recursion*, *huff_enc*, *quicksort*, *bitonic*, *anagram*

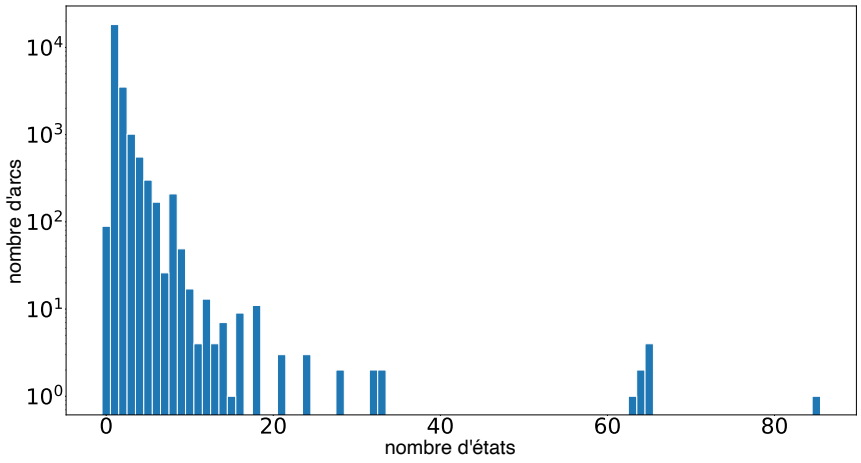


FIGURE 7.2 – Distribution du nombre d'états temporels.

existe dans au moins un XDD du vecteur d'état temporel. La durée de vie de l'évènement est ainsi mesurée par le nombre d'instructions depuis sa naissance jusqu'à sa mort, avec une granularité en termes de *points de contention*. Cette granularité est induite par notre algorithme de calcul de contention : l'analyse de pipeline s'intéresse uniquement aux *points de contention* et l'effet des instructions entre ces points est précalculé par des matrices qui permettent d'accélérer l'analyse. Il n'est donc possible de connaître le moment précis de la disparition des évènements entre les *points de contention* sans dégrader les performances, voire de la rendre infaisable. Par conséquent, la durée de vie des évènements est mesurée par le nombre d'**instructions** entre les points de contention conditionné par la granularité des *points de contention*. Un évènement mort en un *point de contention* est considéré mort pour l'instruction à laquelle appartient le *point de contention*.

Une autre remarque sur cette mesure est que, comme le rebasage est appliqué uniquement à la fin des BB, la durée de vie mesurée au milieu des BB ne prend pas en compte cet effet. En tout cas, la durée de vie mesurée dans cette expérience est une surestimation de la vraie durée de vie des évènements à un grain plus gros que l'instruction.

La figure 7.3 montre le résultat de cette mesure avec, sur l'axe x la durée de vie des évènements en nombre d'instructions et sur l'axe y, en échelle logarithmique, le nombre d'évènements ayant une telle durée de vie. Ces statistiques sont également cumulées pour l'ensemble des benchmarks de TACLe. Ce résultat montre que les évènements ont généralement une durée de vie courte – moins de 50 instructions. Nous avons observé une seule durée de vie de 602 instructions qui n'est pas représentée dans la figure pour préserver sa lisibilité. En fait, dans les situations où la durée de vie est supérieure à 50 instructions, les évènements sont mesurés dans des gros BB. Par exemple, le cas exceptionnel de 602 instructions est mesuré dans un BB de 617 instructions (dans le benchmark md5). Nous pensons que cette longue durée de vie est causée par le manque de rebasage au milieu des BB. À part ce cas exceptionnel, la plupart des évènements ont une durée de vie courte qui aide à garder la taille des XDD et le nombre d'*états temporels* dans un intervalle raisonnable et ainsi préserve la rapidité de l'analyse.

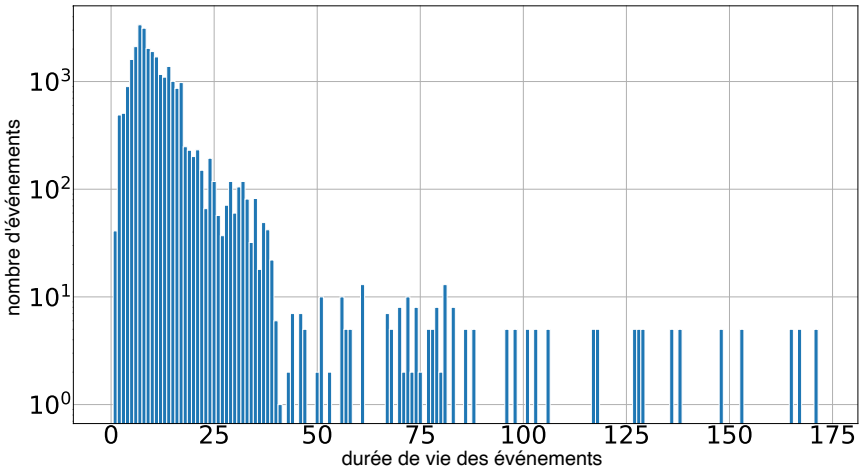


FIGURE 7.3 – Distribution de la durée de vie des évènements.

7.5 Temps d'analyse

Finalement, nous allons mesurer la performance de l'analyse en termes de temps. Le temps d'analyse inclut le temps pour pré-calculer les matrices pour effectuer l'analyse de pipeline (au niveau du CFG). Nous avons réalisé les mesures sur un serveur *cloud* avec une mémoire de 8GB RAM et 4 cœurs Intel Broadwell. Considérant l'utilisation de la mémoire, seuls deux benchmarks sont analysés simultanément (i.e. seuls 2 cœurs sont utilisés et toute la RAM est partagée par les deux benchmarks analysés).

Nous avons mesuré à la fois le temps d'analyse avec et sans appliquer l'optimisation de pré-calculer les matrices afin de montrer le gain de performance. Le temps d'analyse est limité à une heure pour l'expérience sans pré-calculer les matrices. Le résultat expérimental est présenté dans la figure 7.4 avec, sur l'axe x, le nom des benchmarks et sur l'axe y, le temps d'analyse en échelle logarithmique. Le temps d'analyse avec les matrices est noté avec des barres vertes. Dans le pire cas, l'analyse finit en 553 secondes (9 minutes 13 secondes). Dans la plupart des cas, l'analyse finit entre 1 à 20 secondes. L'analyse sans matrices pré-calculées rencontre à la fois des problèmes d'utilisation de mémoire (plus de 8GB utilisé, échec d'analyse) et des problèmes de performance (temps limité à une heure). Les cas d'explosion de la mémoire sont marqués par des barres rouges, les cas de dépassement de temps par des barres jaunes et les cas où l'analyse réussit sans matrices par des barres bleues. Dans les cas où elle réussit, l'optimisation en pré-calculant les matrices amène en moyenne à 217% d'accélération. Les rares cas où pré-calculer les matrices rend l'analyse plus longue sont de petits benchmarks pour lesquels le sur-coût de pré-calcul des matrices ne rattrape pas l'analyse directe. Ce résultat démontre que pré-calculer les matrices peut efficacement réduire les calculs intermédiaires redondants, ce qui améliore la performance de l'analyse en terme d'utilisation de mémoire et de temps d'analyse.

En général, l'analyse (avec matrices) est capable de traiter les benchmarks problématiques (qui produisent un grand nombre d'*états temporels* ou contiennent certains évènements avec une vie très longue) en un temps raisonnable. De plus, nous avons observé que les benchmarks les plus proches de vrais programmes dans des applications industrielles (e.g. Debbie et Papabench) sont analysés dans un temps raisonnable et court : en 10 et 15 minutes respectivement (la somme du temps d'analyse de chaque tâche). Par conséquent, nous pensons que ce modèle est adapté pour des

applications réalistes : par exemple, Airbus exige une procédure de moins de 48h entre la détection d'un bug et l'installation du correctif, en incluant la vérification temporelle.

7.6 Précision

La présence des ressources à allocation dans le désordre rend les *anomalies temporelles* possibles. L'analyse de pipeline devient incapable d'assumer le pire cas local lors de la divergence d'états à cause des évènements. Dans cette situation, un bon compromis entre la performance et la précision devient plus difficile à atteindre car l'analyse a seulement deux choix : soit garder explicitement tous les cas, soit couvrir tous les cas avec une latence supplémentaire qui est surestimée. Pour le bus partagé présenté dans ce chapitre, notre modèle de pipeline représente explicitement les dates d'exécution à l'aide des XDD. Mais nous pouvons imaginer que, si l'état de pipeline n'était pas gardé explicitement, il serait nécessaire de couvrir la latence de la contention du bus avec, par exemple, la pire latence en considérant des contentions avec toutes les instructions postérieures et antérieures qui utilisent potentiellement le bus. Pour estimer l'avantage de précision de notre modèle, nous le comparons avec *etime*, l'ancienne implantation de l'analyse de pipeline d'OTAWA. Concrètement, l'analyse *etime* implante le modèle de graphe d'exécution présenté dans la section 4.1 mais sans utiliser de XDD pour représenter les temps et il considère seulement le pipeline au niveau des BB (au lieu du CFG). Comme il ne prend pas en compte le contexte complet des BB, il peut choisir d'abandonner le contexte même au milieu d'un BB (ce qui est équivalent à couper le BB en deux ou plusieurs sous-blocs et à construire les graphes d'exécution indépendamment) pour réduire la complexité exponentielle de l'analyse. Cette technique de coupure n'est pas valide si le processeur contient des ressources allouées dans le désordre mais nous allons ignorer ce problème de validité dans cette comparaison. De plus, *etime* produit seulement deux temps, HTS et LTS par BB ce qui est moins précis que d'avoir tous les temps possibles. La conséquence de cette différence sur IPET et le WCET final a été discutée dans la section 5.6 (page 76) où nous avons introduit une version améliorée de la méthode IPET. Finalement, comme *etime* ne supporte pas initialement des ressources allouées dans le désordre, nous avons créé un nouveau type d'évènement avec la latence d'une seule utilisation du bus (λ_{BUS}) qui est associé à chaque accès mémoire *Always Hit* ou *Not Classified* pour prendre en compte la latence supplémentaire causée par une contention potentielle.

Les deux approches, *etime* et notre nouveau modèle sont implantés dans OTAWA. Par conséquent, nous pouvons assurer que le modèle de la micro-architecture et les analyses globales sont exactement les mêmes. À l'exception du problème de validité d'*etime* avec les *anomalies temporelles*, nous pensons que *etime* est représentative des approches qui ne considèrent pas explicitement les contentions mais les couvrent par des évènements avec des latences supplémentaires. Au contraire, notre nouvelle approche arrive à considérer la contention explicitement grâce aux XDD qui gardent l'état de pipeline explicitement (et bien-sûr également grâce au gain de performance avec le modèle d'analyse basé sur les ressources et au support des XDD dans le calcul de contention).

Le résultat expérimental est présenté sur la figure 7.5 avec l'axe x supportant le nom des benchmarks et l'axe y le gain de précision calculé par :

$$\frac{t_{etime} - t_{new}}{t_{etime}} \tag{7.4}$$

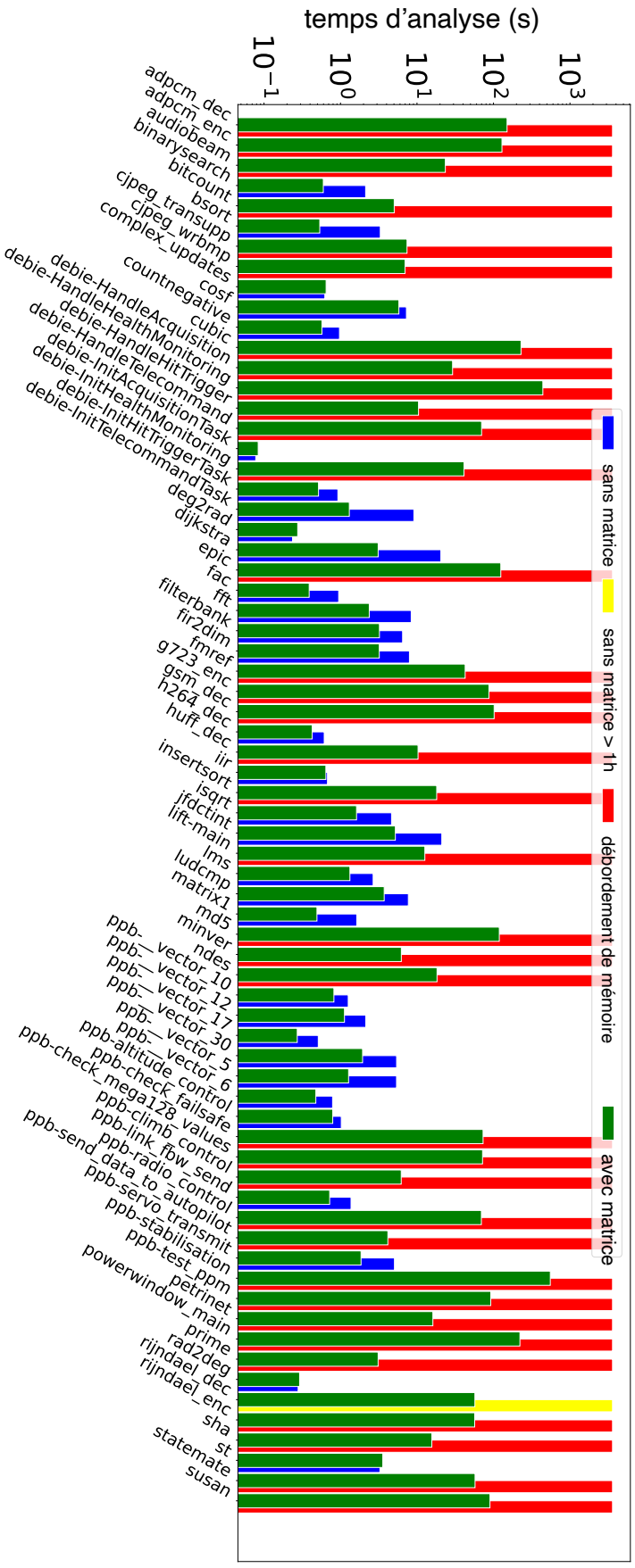


FIGURE 7.4 – Temps d'analyse.

où t_{etime} est le WCET obtenu avec *etime* et t_{new} le WCET obtenu avec la nouvelle méthode basée sur les XDD. À cause de la performance limitée d'*etime*, l'analyse du benchmark *epic* n'a pas pu finir au bout de 6 heures. Le benchmark *susan* a aussi échoué à cause d'un bug dans OTAWA. Les deux barres correspondantes sont ainsi absentes de la figure.

Le résultat montre que, pour la plupart des benchmarks, le WCET obtenu avec la nouvelle méthode est amélioré de 40% à 80%. Dans certains cas (*fft, papabench, rad2deg, rijndael_enc, rijndael_dec* and *statemate*), ce gain est moins important mais est toujours de 20% à 40%.

L'obtention d'un telle amélioration s'appuie fortement sur le fait que nous avons modélisé précisément l'état du pipeline, ce qui permet de déterminer précisément les contentions. Cela pourrait être reflété par le fait que les benchmarks ayant moins de gain de précision sont généralement moins intensifs en termes de taux d'instructions accédant à la mémoire de données, donc moins intensifs en termes de nombre de contentions lors des accès au bus. Intuitivement, on constate que plus un benchmark subit des contentions, plus la surestimation sur la latence par contention est importante. Cette surestimation existe uniquement si l'état de pipeline n'est pas précisément considéré : on est obligé de considérer le pire cas ce qui cause souvent une forte surestimation. Autrement dit, si nous considérons que les instructions ne perdent pas beaucoup de temps en raison de l'attente du bus en cas de contention, notre approche, qui prend en compte des temps d'attente exacts, a un gros avantage par rapport à celles qui modélisent cette attente par une pire latence constante.

De plus, les approches similaires à *etime*, qui couvrent les contentions par une latence surestimée et calculent un nombre limité de temps (WCET) par BB ont une imprécision amplifiée lors de la phase d'IPET (car le pire cas arrive peu souvent, voir section 5.6). Au contraire, notre nouveau modèle considère tous les temps d'exécution possibles et profite des bornes des évènements pour limiter le nombre d'occurrences des pires cas, ce qui permet ainsi d'améliorer la précision du WCET en exposant tous ces temps à la méthode IPET (chapitre 5).

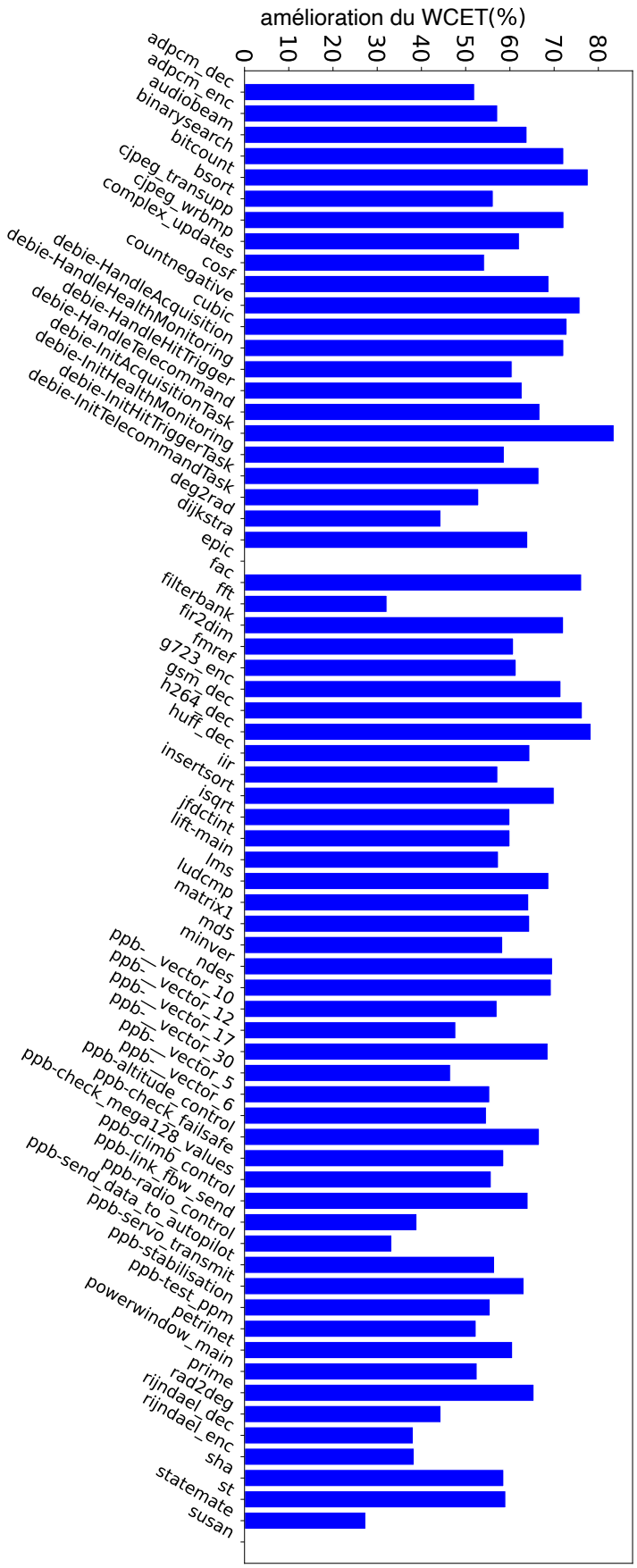


FIGURE 7.5 – Comparaison de précision du WCET.

Chapitre 8

Conclusion

Un des obstacles principaux du calcul de WCET est la modélisation du pipeline et des mécanismes d'accélération avancés dans la conception des micro-architectures modernes comme les caches, la prédiction de branchement, etc. Pour réduire la complexité et pour avoir une meilleure modularité parmi les composants de l'analyse du WCET, les mécanismes avancés sont généralement traités séparément par des analyses dédiées. Leur comportement temporel est représenté par des *évènements* qui sont insérés dans l'analyse de pipeline qui, finalement, calcule le WCET de chaque BB.

En présence des évènements, le temps d'exécution dans le pipeline n'est plus statiquement connu, mais dépend de l'occurrence des évènements. Selon qu'ils sont actifs ou inactifs, l'état temporel du pipeline peut être différent : les évènements créent des divergences d'état du pipeline. L'analyse de pipeline a deux choix face à la divergence : soit garder explicitement les états ayant divergé, soit couvrir la divergence avec un *pire état*. Il est clair que, dans le premier cas, on fait face une complexité combinatoire en fonction du nombre d'évènements alors que la seconde approche, par son principe, souffre d'un problème de précision (car seul un pire cas est considéré et, si le pire cas arrive peu souvent, le WCET sera surestimé) et de sûreté (à cause de la possible apparition d'*anomalies temporelles*).

8.1 Résumé des travaux réalisés

Le XDD (*eXecutable Decision Diagram*) est une structure de données inspirée des BDD qui permet de représenter efficacement la relation entre les variables binaires (les évènements) et des valeurs entières (les temps d'exécution). Avec des XDD, le problème de complexité dans l'analyse de pipeline est atténué, ce qui donne la possibilité de garder explicitement tous les états possibles durant l'analyse de pipeline. C'est pourquoi nous avons utilisé les XDD dans notre analyse de pipeline basée sur les graphes d'exécution et nous l'avons expérimenté sur deux micro-architectures typiques des systèmes embarqués et sur des benchmarks classiques. Les résultats expérimentaux ont montré que le temps d'analyse est tout à fait raisonnable. Les mesures sur le nombre de nœuds et de feuilles des XDD tout au long de l'analyse ont montré que les XDD sont capables de représenter efficacement la relation entre les temps d'exécution et les évènements. Ces effets sont issus des propriétés intrinsèques du pipeline. Dans une micro-architecture super-scalaire, l'*amortissement de parallélisme* joue un rôle important : les processeurs sont conçus pour diminuer les blocages (produits par les aléas dans le pipeline) durant l'exécution afin d'améliorer les performances, ce qui fait que les latences supplémentaires introduites par les évènements sont souvent "amorties" et leur effet sur le temps d'exécution disparaît à terme. Dans une micro-architecture où le parallélisme au niveau des instructions

est moins fort (typiquement mono-scalaire), la *régularité des latences* joue un rôle important : la latence supplémentaire causée par les évènements est régulière, ce qui réduit le nombre combinatoire de temps d'exécution possibles.

Dans l'approche classique, le WCET du programme entier est calculé en utilisant la méthode IPET, ce qui requiert un WCET unique pour chaque BB. Néanmoins, avec notre nouvelle analyse de pipeline, nous arrivons à calculer **tous** les temps possibles de chaque BB. Cependant, les prendre tous en compte nous expose à une explosion de la taille du système ILP utilisé par la méthode IPET, par excès de variables. De plus, sans contraintes supplémentaires sur les temps d'exécution différents, le système ILP considère simplement le pire WCET de chaque BB lors de la maximisation du système : les informations précises fournies par la nouvelle analyse de pipeline deviennent inutiles. Pour pallier ces problèmes, nous proposons une amélioration de la méthode IPET qui permet de trouver un bon compromis entre la taille du système ILP généré et la précision du WCET final. Cette amélioration consiste à partitionner les temps d'exécution en fonction des configurations d'événements qui produisent ces temps, et de profiter des bornes sur le nombre d'occurrences des évènements pour borner le nombre d'occurrences de ces temps. L'expérience a montré que cette amélioration permet d'obtenir des performances similaires à l'approche IPET classique mais avec une précision très supérieure.

À l'aide des XDD, notre nouvelle analyse de pipeline basée sur les graphes d'exécution est capable de calculer précisément tous les temps d'exécution possibles de chaque BB. Néanmoins, ce modèle supporte uniquement des processeurs dont les ressources de pipeline sont allouées dans l'ordre du programme. En présence de ressources allouées dans le désordre, l'allocation dépend de l'état (temporel) précis des ressources du pipeline. Or, si l'analyse de pipeline est appliquée à chaque BB séparément, il est difficile, en présence d'*anomalies temporelles*, de trouver un pire contexte valide pour chaque BB. En plus, le pire contexte supposé risque d'introduire de l'imprécision lors du calcul de l'allocation des ressources allouées dans le désordre, ce qui, à son tour, crée une surestimation considérable dans le WCET calculé. Par conséquent, notre première étape pour supporter les ressources allouées dans le désordre est d'étendre notre analyse de pipeline au niveau du CFG afin de modéliser précisément le contexte de chaque BB. Pour ce faire, nous avons introduit le modèle de pipeline *basé sur ressources*. La **sémantique** de ce nouveau modèle reste la même que celle du modèle des graphes d'exécution, mais il est capable de représenter l'**état temporel** du pipeline contrairement au modèle basé sur les graphes d'exécution. Ainsi, nous pouvons transporter ce nouveau modèle dans le cadre de l'interprétation abstraite. L'analyse de pipeline au niveau CFG est directement déduite de la *sémantique collectrice* de ce nouveau modèle de calcul.

Dans le modèle basé sur les ressources, l'état du pipeline est représenté sous forme d'un vecteur de XDD, ce qui permet de profiter des propriétés algébriques du XDD pour accélérer l'analyse. Comme le domaine des XDD avec les opérations max et $+$ forme un demi-anneau tropical, il est possible d'exprimer les mises à jour de l'état de pipeline sous forme de produit matriciel (dans le domaine des XDD). Ces matrices peuvent être pré-calculées indépendamment pour chaque BB et donc diminuer le coût des multiples calculs nécessaires pour chaque BB. Il est en effet nécessaire de recalculer le temps de chaque bloc de base pour chacun de ses contextes. À leur tour, ces nouveaux temps peuvent se propager comme nouveaux contextes pour les BB suivants. Au final, les résultats expérimentaux ont montré que cette technique accélère largement l'analyse.

Dans un second temps, nous avons étendu notre modèle pour supporter une ressource typique d'allocation dans le désordre : le bus mémoire partagé par le cache

d'instructions et le cache de données. Comme le bus est géré selon une politique FCFS, son allocation dépend de la date à laquelle les accès provenant des caches sont prêts. Si ces dates sont connues, et elles le sont [1], nous pouvons déterminer l'allocation exacte du bus. L'obstacle majeur pour cela est que ces dates ne sont pas uniques mais fonction de l'occurrence des évènements, ce qui peut résulter en des ordonnancements différents pour l'allocation. Par conséquent, nous avons conçu un algorithme en deux étapes pour résoudre l'allocation : la première étape détermine les dates auxquelles les accès sont prêts à utiliser le bus et la deuxième étape compare les dates calculées pour déterminer l'ordre d'allocation pour toutes les configurations. La première étape peut être réalisée simplement avec le modèle basé sur ressources car les dates auxquelles les accès sont prêts dépendent uniquement du comportement d'exécution dans l'ordre. Dans la deuxième étape, nous avons créé des comparateurs **entre XDD** pour déterminer l'ordonnancement des accès **pour toutes les configurations** : quelle instruction prend la main sur le bus et éventuellement retarde les autres. Finalement, il nous a été possible de calculer tous les temps d'exécution exacts de chaque BB, en présence du bus FCFS et au niveau du CFG. Du point de vue de la performance, les résultats expérimentaux ont montré que ce nouveau modèle est capable de traiter les benchmarks classiques en un temps raisonnable, mais bien-sûr, comme nous calculons les temps exacts, l'analyse est plus coûteuse que les approches classiques qui introduisent de la surestimation. Du point de vue de la précision, les résultats expérimentaux ont montré un gain impressionnant sur l'ensemble des benchmarks, en comparaison avec l'approche classique *etime* – l'implantation de l'analyse de pipeline d'OTAWA qui est représentative des analyses qui ne considèrent pas explicitement tous les temps d'exécution possibles.

8.2 Perspectives de recherche

Plusieurs directions sont envisageables dans la suite de ces travaux de thèse. Tout d'abord, nous voudrions nous intéresser à la modélisation des ressources allouées dans le désordre en général (par exemple, un étage d'exécution dans le désordre avec un *Re-Order Buffer*(TOMASULO, 1967)). Pour modéliser le bus de mémoire partagé par le cache de données et le cache d'instructions, nous avons profité des contraintes sur l'ordonnancement des accès qui sont bien plus fortes que celles rencontrées dans un étage d'exécution dans le désordre. Ainsi, l'algorithme que nous avons montré est uniquement valide pour le problème du bus. Néanmoins, pour une ressource allouée selon la politique FCFS en général, l'ordonnancement est calculable en comparant la date de disponibilité des accès. Les opérateurs de comparaisons ($\blacktriangleleft_{FE}, \blacktriangleleft_{ME}$) que nous avons introduits sont capables de comparer les dates et de sélectionner les configurations des XDD selon le résultat de la comparaison, ce qui supprime l'obstacle principal à utiliser des XDD dans le calcul de contention en général, au moins pour les ressources allouées selon la politique FCFS. La difficulté majeure que nous devons envisager dans cette extension est le problème de performance : comme la contention est *moins contrainte* (il n'y a plus des contraintes statiques sur les accès comme dans le problème du bus partagé), plus de calculs sont nécessaires pour déterminer l'ordonnancement des allocations. Les XDD doivent être soumis à nouveau à des expériences réalistes pour savoir s'ils sont suffisamment performants pour supporter de telles analyses afin de traiter des applications réalistes dans un temps raisonnable. Dans le cas où les XDD seraient insuffisants, il est encore envisageable de concevoir des algorithmes pour générer automatiquement des contraintes

1. Les dates calculées avec les techniques présentées précédemment sont exactes.

(statiques) sur les allocations (comme les contraintes exploitées dans notre problème du bus), ce qui peut éviter des calculs inutiles et ainsi accélérer l'analyse.

Une autre direction concerne l'amélioration du modèle de calcul sur les pipelines, qui peut être déclinée selon deux axes parallèles. Le premier est la simplification de la modélisation du pipeline de nouveaux processeurs : même si nous avons déjà proposé un modèle basé sur les ressources pour décrire un microprocesseur, il exige une compréhension profonde de notre modèle et des efforts importants pour atteindre un modèle correct. Dans notre modèle de pipeline actuel, le comportement *d'exécution dans l'ordre* du pipeline est totalement modélisé par les **fonctions de transitions** de vecteur de XDD. En fait, le problème de la modélisation du pipeline est un problème de compilation : étant donné une description de micro-architecture, on cherche à générer un *compilateur* du programme binaire en un programme de **fonctions de transitions basiques** comme les τ de la section 6.2. Le but ultime est de générer ce compilateur avec la description de l'ISA, par exemple avec le langage *SimNML* (HERBEGUE et al., 2014). Ces langages sont initialement conçus pour décrire le comportement fonctionnel des instructions par des fonctions de transition de **l'état fonctionnel** du processeur (par exemple en spécifiant les ressources utilisées par les instructions). Par conséquent, il n'y a théoriquement pas d'obstacle pour étendre ces langages à la modélisation du comportement temporel, par exemple en spécifiant les fonctions de transitions à invoquer lors de l'exécution de chaque instruction.

Le deuxième axe consiste à optimiser les calculs dans notre analyse par XDD. Par exemple, nous savons que l'ordre des évènements dans les XDD joue un rôle important sur la taille des XDD. Jusqu'à présent, nous avons utilisé une heuristique d'ordonnancement des évènements (par ordre d'apparition) qui est particulièrement performante pour optimiser les opérations entre XDD. Une amélioration pourrait être de chercher un meilleur ordre des évènements qui peut, à la fois, minimiser la taille des XDD et simplifier les opérations. Une autre direction pour améliorer la performance des calculs de XDD est d'optimiser les produits matriciels des XDD. Les matrices des XDD (pour représenter l'exécution des BB) sont souvent creuses parce que toutes les *ressources temporelles* n'ont pas de dépendances complètes entre elles : lors de la mise à jour d'une *ressource temporelle*, r, qui ne dépend pas d'une autre *ressource temporelle*, r', l'élément correspondant dans la matrice (l'élément $(i_r, i_{r'})$) est $\mathbb{0}$. Ce phénomène apparaît particulièrement pour la *dépendance de données* : l'*état temporel* contient au moins une *ressource temporelle* par registre, ces ressources sont généralement peu dépendantes entre elles et tous les registres ne sont pas utilisés. La zone correspondante dans la matrice est ainsi systématiquement creuse. Par conséquent, des optimisations de la représentation et de la multiplication des matrices creuses deviendront utiles pour réduire la complexité du calcul. Ces techniques existent déjà pour les matrices creuses de nombres réels (YUSTER et al., 2005) et devraient pouvoir être facilement transférées vers les matrices de XDD.

Bibliographie

ABELLA, Jaume, Damien HARDY, Isabelle PUAUT, Eduardo QUINONES et Francisco J CAZORLA (2014). « On the comparison of deterministic and probabilistic WCET estimation techniques ». In : *2014 26th Euromicro Conference on Real-Time Systems*. IEEE, p. 266-275.

AKERS, Sheldon B. (1978). « Binary decision diagrams ». In : *IEEE Transactions on computers* 27.06, p. 509-516.

ALLEN, Frances E. et John COCKE (1976). « A program data flow analysis procedure ». In : *Communications of the ACM* 19.3, p. 137.

AMALOU, Abderaouf N, Isabelle PUAUT et Gilles MULLER (2021). « WE-HML : hybrid WCET estimation using machine learning for architectures with caches ». In : *2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, p. 31-40. DOI : 10.1109/RTCSA52859.2021.00011.

BAHAR, R Iris, Erica A FROHM, Charles M GAONA, Gary D HACHTEL, Enrico MACII, Abelardo PARDO et Fabio SOMENZI (1997). « Algebric decision diagrams and their applications ». In : *Formal methods in system design* 10.2, p. 171-206.

BAI, Zhenyu, Hugues CASSÉ, Marianne DE MICHIEL, Thomas CARLE et Christine ROCHANGE (2020). « Improving the Performance of WCET Analysis in the Presence of Variable Latencies ». In : *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES '20, London, United Kingdom. New York, NY, USA : Association for Computing Machinery, 119–130. ISBN : 9781450370943. DOI : 10.1145/3372799.3394371. URL : https://doi.org/10.1145/3372799.3394371.

— (2022). « A Framework for Calculating WCET Based on Execution Decision Diagrams ». In : *ACM Trans. Embed. Comput. Syst.* 21.3. ISSN : 1539-9087. DOI : 10.1145/3476879. URL : https://doi.org/10.1145/3476879.

BAI, Zhenyu, Hugues CASSÉ, Thomas CARLE et Christine ROCHANGE (2023). « Computing Execution Times with eXecution Decision Diagrams in the Presence of Out-Of-Order Resources ». In : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, p. 1-1. DOI : 10.1109/TCAD.2023.3258752.

BAK, Stanley, Gang YAO, Rodolfo PELLIZZONI et Marco CACCAMO (2012). « Memory-aware scheduling of multicore task sets for real-time systems ». In : *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, p. 300-309.

BALLABRIGA, Clément, Hugues CASSÉ, Christine ROCHANGE et Pascal SAINRAT (2010). « OTAWA : An open toolbox for adaptive WCET analysis ». In : *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer, p. 35-46.

BENHAMAMOUCH, Bilel et Bruno MONSUEZ (2009). « Computing worst case execution time (wcet) by symbolically executing a time-accurate hardware model ». In : *International MultiConference of Engineers and Computer Scientists*. T. 2, p. 3-8.

BENHAMAMOUCH, Bilel, Bruno MONSUEZ et Franck VÉDRINE (2008). « Computing WCET using symbolic execution ». In : *Second International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS 2008)*, p. 1-12.

BERNAT, Guillem, Antoine COLIN et Stefan M PETTERS (2002). « WCET analysis of probabilistic hard real-time systems ». In : *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.* IEEE, p. 279-288.

BETTS, Adam, Nicholas MERRIAM et Guillem BERNAT (2010). « Hybrid measurement-based WCET analysis at the source level using object-level traces ». In : *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

BIERE, Armin, Jens KNOOP, Laura KOVÁCS et Jakob ZWIRCHMAYR (2013). « The auspicious couple : Symbolic execution and WCET analysis ». In : *13th International Workshop on Worst-Case Execution Time Analysis*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

BINDER, Benjamin, Mihail ASAVOAE, Florian BRANDNER, Belgacem Ben HEDIA et Mathieu JAN (2022). « The Role of Causality in a Formal Definition of Timing Anomalies ». In : *2022 IEEE 28th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, p. 91-102.

BINDER, Benjamin, Mihail ASAVOAE, Belgacem Ben HEDIA, Florian BRANDNER et Mathieu JAN (2021). « Is this still normal ? Putting definitions of timing anomalies to the test ». In : *2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, p. 139-148.

BONENFANT, Armelle, Hugues CASSÉ, Marianne DE MICHIEL, Jens KNOOP, Laura KOVÁCS et Jakob ZWIRCHMAYR (2012). « FFX : A portable WCET annotation language ». In : *Proceedings of the 20th International Conference on Real-Time and Network Systems*, p. 91-100.

BRACE, Karl S, Richard L RUDELL et Randal E BRYANT (1991). « Efficient implementation of a BDD package ». In : *Proceedings of the 27th ACM/IEEE design automation conference*, p. 40-45.

BRIL, Reinder J, Johan J LUKKIEN et Wim FJ VERHAEGH (2009). « Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption ». In : *Real-Time Systems* 42, p. 63-119.

BRYANT, Randal E (1992). « Symbolic boolean manipulation with ordered binary-decision diagrams ». In : *ACM Computing Surveys (CSUR)* 24.3, p. 293-318.

BURCH, Jerry R, Edmund M CLARKE, Kenneth L MCMILLAN, David L DILL et Lain-Jinn HWANG (1992). « Symbolic model checking : 1020 states and beyond ». In : *Information and computation* 98.2, p. 142-170.

CASSEZ, Franck (2011). « Timed games for computing WCET for pipelined processors with caches ». In : *2011 Eleventh International Conference on Application of Concurrency to System Design*. IEEE, p. 195-204.

CASSEZ, Franck et Jean-Luc BÉCHENNEC (2013). « Timing analysis of binary programs with UPPAAL ». In : *2013 13th International Conference on Application of Concurrency to System Design*. IEEE, p. 41-50.

CASSEZ, Franck, René Rydhof HANSEN et Mads Chr OLESEN (2012). « What is a timing anomaly ? » In : *12th International Workshop on Worst-Case Execution Time Analysis*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

CLARKE, Edmund M (1997). « Model checking ». In : *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, p. 54-56.

COHEN, Guy, Stéphane GAUBERT et Jean-Pierre QUADRAT (1999). « Max-plus algebra and system theory : where we are and where to go now ». In : *Annual reviews in control* 23, p. 207-219.

COUSOT, Patrick (1996). « Abstract interpretation ». In : *ACM Computing Surveys (CSUR)* 28.2, p. 324-328.

COUSOT, Patrick et Radhia COUSOT (1977). « Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints ». In : *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, p. 238-252.

— (1992). « Abstract interpretation frameworks ». In : *Journal of logic and computation* 2.4, p. 511-547.

DAVID, Laurent et Isabelle PUAUT (2004). « Static determination of probabilistic execution times ». In : *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.* IEEE, p. 223-230.

EISINGER, Jochen, Ilia POLIAN, Bernd BECKER, Stephan THESING, Reinhard WILHELM et Alexander METZNER (2006). « Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis ». In : *2006 IEEE Design and Diagnostics of Electronic Circuits and systems*. IEEE, p. 13-18.

ENGBLOM, Jakob (2002). « Processor pipelines and static worst-case execution time analysis ». Thèse de doct. Acta Universitatis Upsaliensis.

FALK, Heiko, Sebastian ALTMEYER, Peter HELLINCKX, Björn LISPER, Wolfgang PUFFITSCH, Christine ROCHANGE, Martin SCHOEBERL, Rasmus Bo SØRENSEN, Peter WÄGEMANN et Simon WEGENER (2016). « TACLeBench : A benchmark collection to support worst-case execution time research ». In : *16th International Workshop on Worst-Case Execution Time Analysis*.

FERDINAND, Christian et Reinhold HECKMANN (2004). « ait : Worst-case execution time prediction by static program analysis ». In : *Building the Information Society*. Springer, p. 377-383.

FERDINAND, Christian, Reinhold HECKMANN, Marc LANGENBACH, Florian MARTIN, Michael SCHMIDT, Henrik THEILING, Stephan THESING et Reinhard WILHELM (2001). « Reliable and precise WCET determination for a real-life processor ». In : *International Workshop on Embedded Software*. Springer, p. 469-485.

FUJITA, Masahiro, Patrick C. MCGEER et JC-Y YANG (1997). « Multi-terminal binary decision diagrams : An efficient data structure for matrix representation ». In : *Formal methods in system design* 10.2, p. 149-169.

GEBHARD, Gernot (2010). « Timing anomalies reloaded ». In : *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

GOTO, Eiichi (1974). *Monocopy and associative algorithms in an extended lisp*. Rapp. tech. Technical Report TR 74-03, University of Tokyo.

GRAHAM, Ronald L. (1969). « Bounds on multiprocessing timing anomalies ». In : *SIAM journal on Applied Mathematics* 17.2, p. 416-429.

GRAILLAT, Amaury, Claire MAIZA, Matthieu MOY, Pascal RAYMOND et Benoît Dupont de DINECHIN (2019). « Response Time Analysis of Dataflow Applications on a Many-Core Processor with Shared-Memory and Network-on-Chip ». In : *Proceedings of the 27th International Conference on Real-Time Networks and Systems*. RTNS '19. Toulouse, France : Association for Computing Machinery, 61–69. ISBN : 9781450372237. DOI : 10.1145/3356401.3356416. URL : https://doi.org/10.1145/3356401.3356416.

GRUIN, Alban, Thomas CARLE, Hugues CASSÉ et Christine ROCHANGE (2021). « Speculative Execution and Timing Predictability in an Open Source RISC-V Core ». In : *IEEE Real-Time Systems Symposium (RTSS 2021)*. IEEE, p. 393-404.

HAHN, Sebastian (2018). « On Static Execution-time Analysis : Compositionality, Pipeline Abstraction, and Predictable Hardware ». Thèse de doct. Universität des Saarlandes.

HAHN, Sebastian, Michael JACOBS, Nils HÖLSCHER, Kuan-Hsun CHEN, Jian-Jia CHEN et Jan REINEKE (2022). « LLVMTA : An LLVM-Based WCET Analysis Tool ». In : *20th International Workshop on Worst-Case Execution Time Analysis (WCET 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

HAHN, Sebastian, Michael JACOBS et Jan REINEKE (2016). « Enabling Compositionality for Multicore Timing Analysis ». In : *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. RTNS '16, Brest, France. New York, NY, USA : Association for Computing Machinery, 299–308. ISBN : 9781450347877. DOI : 10.1145/2997465.2997471. URL : https://doi.org/10.1145/2997465.2997471.

HAHN, Sebastian et Jan REINEKE (2020). « Design and analysis of SIC : a provably timing-predictable pipelined processor core ». In : *Real-Time Systems* 56.2, p. 207-245.

HAHN, Sebastian, Jan REINEKE et Reinhard WILHELM (2015a). « Toward compact abstractions for processor pipelines ». In : *Correct System Design*. Springer, p. 205-220.

— (2015b). « Towards Compositionality in Execution Time Analysis : Definition and Challenges ». In : *SIGBED Rev.* 12.1, 28–36. DOI : 10.1145/2752801.2752805. URL : https://doi.org/10.1145/2752801.2752805.

HEIDERGOTT, Bernd, Geert Jan OLSDER, Jacob VAN DER WOUDE et JW van der WOUDE (2006). *Max Plus at work : modeling and analysis of synchronized systems : a course on Max-Plus algebra and its applications*. T. 13. Princeton University Press.

HERBEGUE, Hajer, Mamoun FILALI et Hugues CASSÉ (2014). « Formal architecture specification for time analysis ». In : *International Conference on Architecture of Computing Systems*. Springer, p. 98-110.

HOLSTI, Niklas, Thomas LÅNGBACKA et Sami SAARINEN (2000). « Worst-case execution time analysis for digital signal processors ». In : *2000 10th European Signal Processing Conference*. IEEE, p. 1-4.

KEBBAL, Djemai et Pascal SAINRAT (2006). « Combining symbolic execution and path enumeration in worst-case execution time analysis ». In : *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

KIRNER, Raimund, Albrecht KADLEC et Peter PUSCHNER (2009). « Precise worst-case execution time analysis for processors with timing anomalies ». In : *2009 21st Euromicro Conference on Real-Time Systems*. IEEE, p. 119-128.

KLEE, Victor et George J MINTY (1972). « How good is the simplex algorithm ». In : *Inequalities* 3.3, p. 159-175.

LARSEN, Kim G, Paul PETTERSSON et Wang YI (1997). « UPPAAL in a nutshell ». In : *International journal on software tools for technology transfer* 1.1, p. 134-152.

LATTNER, Chris et Vikram ADVE (2004). « LLVM : A compilation framework for lifelong program analysis & transformation ». In : *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, p. 75-86.

LI, Xianfeng, Yun LIANG, Tulika MITRA et Abhik ROYCHOUDHURY (2007). « Chronos : A timing analyzer for embedded software ». In : *Science of Computer Programming* 69.1-3, p. 56-67.

LI, Xianfeng, Abhik ROYCHOUDHURY et Tulika MITRA (2006). « Modeling out-of-order processors for WCET analysis ». In : *Real-Time Systems* 34.3, p. 195-227.

LI, Yau-Tsun Steven et Sharad MALIK (1995). « Performance analysis of embedded software using implicit path enumeration ». In : *Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, p. 88-98.

MEINEL, Christoph et Anna SLOBODOVA (1994). « On the complexity of constructing optimal ordered binary decision diagrams ». In : *International Symposium on Mathematical Foundations of Computer Science*. Springer, p. 515-524.

MICHIEL, Marianne de, Armelle BONENFANT, Clément BALLABRIGA et Hugues CASSÉ (2010). « Partial Flow Analysis with oRange ». In : *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, p. 479-482.

PELLIZZONI, Rodolfo, Emiliano BETTI, Stanley BAK, Gang YAO, John CRISWELL, Marco CACCAMO et Russell KEGLEY (2011). « A predictable execution model for COTS-based embedded systems ». In : *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, p. 269-279.

RAJESH, V et Rajat MOONA (1999). « Processor modeling for hardware software codesign ». In : *Proceedings Twelfth International Conference on VLSI Design.(Cat. No. PR00013)*. IEEE, p. 132-137.

RATSIAMBAHOTRA, Tahiry, Hugues CASSÉ et Pascal SAINRAT (2009). « A versatile generator of instruction set simulators and disassemblers ». In : *2009 International Symposium on Performance Evaluation of Computer & Telecommunication Systems*. T. 41. IEEE, p. 65-72.

REINEKE, Jan et Rathijit SEN (2009). « Sound and efficient WCET analysis in the presence of timing anomalies ». In : *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

REINEKE, Jan, Björn WACHTER, Stefan THESING, Reinhard WILHELM, Ilia POLIAN, Jochen EISINGER et Bernd BECKER (2006). « A definition and classification of timing anomalies ». In : *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

ROCHANGE, Christine et Pascal SAINRAT (2009). « A context-parameterized model for static analysis of execution times ». In : *Transactions on High-Performance Embedded Architectures and Compilers II*. Springer, p. 222-241.

SAINRAT, Pascal, Christine ROCHANGE et Sascha UHRIG (2014). *Time-predictable architectures*. John Wiley & Sons.

SCHNEIDER, Jörn et Christian FERDINAND (1999). « Pipeline behavior prediction for superscalar processors by abstract interpretation ». In : *ACM SIGPLAN Notices* 34.7, p. 35-44.

SCHOEBERL, Martin, Sahar ABBASPOUR, Benny AKESSON, Neil AUDSLEY, Raffaele CAPASSO, Jamie GARSIDE, Kees GOOSSENS, Sven GOOSSENS, Scott HANSEN, Reinhold HECKMANN et al. (2015). « T-CREST : Time-predictable multi-core architecture for embedded systems ». In : *Journal of Systems Architecture* 61.9, p. 449-471.

SCHRIJVER, Alexander (1998a). *Theory of linear and integer programming*. John Wiley & Sons.

— (1998b). *Theory of linear and integer programming*. John Wiley & Sons.

SCHUH, Matheus, Claire MAIZA, Joël GOOSSENS, Pascal RAYMOND et Benoît Dupont de DINECHIN (2020). « A study of predictable execution models implementation for industrial data-flow applications on a multi-core platform with shared banked memory ». In : *2020 IEEE Real-Time Systems Symposium (RTSS)*, p. 283-295. DOI : 10.1109/RTSS49844.2020.00034.

SHARIR, Micha, Amir PNUELI et al. (1978). *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences ...

STOCK, Gregory, Sebastian HAHN et Jan REINEKE (2019). « Cache Persistence Analysis : Finally Exact ». In : *2019 IEEE Real-Time Systems Symposium (RTSS)*, p. 481-494. DOI : 10.1109/RTSS46320.2019.00049.

STORNETTA, Tony et Forrest BREWER (1996). « Implementation of an efficient parallel BDD package ». In : *33rd Design Automation Conference Proceedings, 1996*. IEEE, p. 641-644.

THEILING, Henrik, Christian FERDINAND et Reinhard WILHELM (2000). « Fast and precise WCET prediction by separated cache and path analyses ». In : *Real-Time Systems* 18.2, p. 157-179.

THESING, Stephan (2004). « Safe and precise WCET determination by abstract interpretation of pipeline models ». In.

TOMASULO, Robert M (1967). « An efficient algorithm for exploiting multiple arithmetic units ». In : *IBM Journal of research and Development* 11.1, p. 25-33.

TOUZEAU, Valentin, Claire MAÏZA, David MONNIAUX et Jan REINEKE (2017). « Ascertaining uncertainty for efficient exact cache analysis ». In : *International Conference on Computer Aided Verification*. Springer, p. 22-40.

TOUZEAU, Valentin, Claire MAÏZA, David MONNIAUX et Jan REINEKE (2019). « Fast and Exact Analysis for LRU Caches ». In : *Proc. ACM Program. Lang.* 3.POPL. DOI : 10.1145/3290367. URL : https://doi.org/10.1145/3290367.

VARGHESE, Blesson, Nan WANG, Sakil BARBHUIYA, Peter KILPATRICK et Dimitrios S NIKOLOPOULOS (2016). « Challenges and opportunities in edge computing ». In : *2016 IEEE International Conference on Smart Cloud (SmartCloud)*. IEEE, p. 20-26.

WENZEL, Ingomar, Raimund KIRNER, Peter PUSCHNER et Bernhard RIEDER (2005). « Principles of timing anomalies in superscalar processors ». In : *Fifth International Conference on Quality Software (QSIC'05)*. IEEE, p. 295-303.

WILHELM, Reinhard, Jakob ENGBLOM, Andreas ERMEDAHL, Niklas HOLSTI, Stephan THESING, David WHALLEY, Guillem BERNAT, Christian FERDINAND, Reinhold HECKMANN, Tulika MITRA, Frank MUELLER, Isabelle PUAUT, Peter PUSCHNER, Jan STASCHULAT et Per STENSTRÖM (2008). « The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools ». In : *ACM Trans. Embed. Comput. Syst.* 7.3. ISSN : 1539-9087. DOI : 10.1145/1347375.1347389. URL : https://doi.org/10.1145/1347375.1347389.

WILHELM, Stephan (2007). « Efficient analysis of pipeline models for WCET computation ». In : *5th International Workshop on Worst-Case Execution Time Analysis (WCET'05)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

YUSTER, Raphael et Uri ZWICK (2005). « Fast sparse matrix multiplication ». In : *ACM Transactions On Algorithms (TALG)* 1.1, p. 2-13.

ZARUBA, Florian et Luca BENINI (2019). « The cost of application-class processing : Energy and performance analysis of a Linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology ». In : *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11, p. 2629-2640.