HAL open science

Contributions à la traduction binaire dynamique : support du parallélisme d'instructions et génération de traducteurs optimisés

Luc Michel

▶ To cite this version:

Luc Michel. Contributions à la traduction binaire dynamique : support du parallélisme d'instructions et génération de traducteurs optimisés. Informatique et langage [cs.CL]. Université de Grenoble, 2014. Français. NNT : 2014GRENM101 . tel-01547198

HAL Id: tel-01547198
https://theses.hal.science/tel-01547198v1

Submitted on 26 Jun 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Luc Michel

Thèse dirigée par **Frédéric Pétrot**
et encadrée par **Nicolas Fournel**

préparée au sein **du Laboratoire TIMA**
et de **l'École Doctorale Mathématiques, Sciences et Techniques de l'Ingénieur, Informatique**

Contributions à la traduction binaire dynamique : support du parallélisme d'instructions et génération de traducteurs optimisés

Thèse soutenue publiquement le **18 décembre 2014**,
devant le jury composé de :

Mme Florence Maraninchi
Professeur des Universités, Grenoble INP, Présidente
M. Bernard Goossens
Professeur des Universités, Université de Perpignan, Rapporteur
M. Erven Rohou
Directeur de Recherche, Inria Rennes, Rapporteur
M. Paul Feautrier
Professeur des Universités, ENS Lyon, Examinateur
M. Benoît Dupont de Dinechin
Docteur, Kalray, Invité
M. Frédéric Pétrot
Professeur des Universités, Grenoble INP, Directeur de thèse
M. Nicolas Fournel
Maître de Conférences, Université Joseph Fourier, Encadrant

Remerciements

Je tiens en premier lieu à remercier les membres de mon jury de thèse. Merci à Bernard Goossens et Erven Rohou pour leurs retours riches, intéressants et constructifs sur mon travail. Merci aussi à Paul Feautrier et Benoît Dupont de Dinechin d'avoir examiner ma thèse. Ayant utilisé intensivement certains outils de Benoît, j'ai beaucoup apprécié ses retours lors de la soutenance. Finalement, merci à Florence Maraninchi, qui avant d'être présidente de mon jury, a été mon enseignante d'algorithmique à l'Ensimag, et m'a soutenu lorsque les mathématiques s'en prenaient à moi, petit admis sur titre que j'étais. Elle a alors prêté une oreille attentive et m'a fait découvrir le monde de la recherche.

Un immense merci à mon directeur, Frédéric Pétrot. Malgré ses nombreuses responsabilités et son emploi du temps chargé, il a su rester disponible et à l'écoute à la moindre occasion. C'est une chance que j'ai eu de pouvoir discuter avec lui à tout moment de mon avancement, des problèmes sur lesquels je bloquais, des solutions que j'apportais au fur et à mesure de ces trois années. Merci à Nicolas Fournel qui m'a encadré durant ma thèse. Je me souviens de ces *coding night fever*, que l'on pourrait traduire par *soirées code – pizza* pendant lesquelles on s'acharnait à peaufiner notre prototype pour la soumission d'un article.

Merci à tous les gens que j'ai rencontré dans l'équipe SLS du laboratoire TIMA et ailleurs. Olivier et les nombreuses discutions que nous avons eu. Sa grande expérience de l'enseignement m'a beaucoup aidé alors que je débutais dans ce domaine. Merci aussi à Matthieu qui a accepté d'endurer le rôle de "tuteur d'enseignement" et qui lui aussi m'a apporté sa précieuse expertise dans ce domaine. Merci à tous les doctorants, post-doc ou ingénieurs pour ces bons moments en votre compagnie : Adrien, Damien, Clément, Hamayun, Marius, Alban, Ashraf, Marcos et j'en oublie. Merci à Tadeusz qui m'a fait découvrir le monde de la recherche quand j'étais à l'IUT, et qui m'a aiguillé dans mon parcours universitaire.

Enfin, merci à ma famille. Si j'en suis arrivé là c'est grâce à mes parents et surtout à ma maman qui m'a tiré contre vents et marrés vers le haut alors que son fils préférait de loin jouer à taper des commandes MS-DOS sur un vieux 486 plutôt que de faire ses devoirs (en CE1 déja !). Par ailleurs, sa relecture attentive et courageuse de ce manuscrit, qualifié alors d'ésotérique, m'a permis d'éliminer nombre d'erreurs.

Finalement, une pensée toute particulière pour ma femme Sofia, qui a rédigé et soutenu dans les mêmes périodes que moi. On m'a souvent averti que, la rédaction n'étant pas une période particulièrement joyeuse, l'ambiance à la maison risquait d'être morose. Il me suffit aujourd'hui de regarder en arrière pour m'apercevoir que non seulement ces dires étaient faux, mais qu'en plus je garde un excellent souvenir de cette période pendant laquelle chacun à pris soin de l'autre. Cette épreuve est un signe de plus qui me montre que j'ai rencontré la femme de ma vie.

Table des matières

1 Introduction **1**

2 Problématique **5**
2.1 Les familles de jeux d'instructions . . . 5
2.1.1 Processeurs généralistes, RISC et CISC . . . 6
2.1.2 Jeux d'instructions SIMD . . . 6
2.1.3 Processeurs VLIW et DSP . . . 7
2.2 Les techniques de simulation d'ISA . . . 10
2.2.1 Détail des différentes familles . . . 11
2.2.2 La traduction binaire dynamique . . . 12
2.3 Générer automatiquement un simulateur . . . 14
2.3.1 Les ADL . . . 14
2.3.2 Génération de simulateurs à l'aide d'ADL . . . 14
2.4 Conclusion . . . 15

3 État de l'art **17**
3.1 La traduction binaire dynamique . . . 17
3.1.1 Dans les machines virtuelles . . . 17
3.1.2 Pour le support à la migration de code sur de nouvelles architectures . 18
3.1.3 Pour l'optimisation binaire . . . 18
3.1.4 Pour l'instrumentation et l'aide à la mise au point . . . 18
3.1.5 Pour la simulation rapide . . . 18
3.2 Simulation rapide d'architectures VLIW . . . 19
3.2.1 Par traduction statique . . . 19
3.2.2 Par simulation compilée . . . 20
3.3 Génération de simulateurs rapides à partir d'un ADL . . . 21
3.3.1 Générer des simulateurs basés sur la simulation compilée . . . 22
3.3.2 UQBT : traduction binaire statique reciblable . . . 24
3.3.3 UQDBT et Walkabout : vers la traduction binaire dynamique . . . 25
3.3.4 Bintrans : DBT optimisée pour le couple cible/hôte . . . 26
3.4 Conclusion . . . 29

4 Simulation d'architectures VLIW à l'aide de la traduction binaire dynamique **31**
4.1 Un algorithme pour simuler une architecture VLIW . . . 33
4.1.1 Exemple . . . 33
4.1.2 Conclusion sur l'algorithme . . . 35
4.2 Intégration de l'algorithme dans un traducteur binaire dynamique . . . 36
4.2.1 Adaptation de l'algorithme . . . 36

4.2.2 L'allocation des répliques en pratique . . . 37
4.2.3 Les problèmes introduits par la granularité du TB . . . 38
4.2.4 Gestion des instructions prédiquées . . . 45
4.2.5 Conclusion sur l'intégration dans un traducteur DBT . . . 51
4.3 Expérimentations . . . 51
4.3.1 Protocole expérimental . . . 51
4.3.2 Statistiques sur le code généré . . . 51
4.3.3 Mesure du temps d'exécution de la simulation . . . 53
4.4 Conclusion . . . 54

5 Génération de simulateurs basés sur la traduction binaire dynamique 57
5.1 Présentation du flot de génération . . . 57
5.2 MDS : un ADL comportemental . . . 58
5.2.1 Les espaces mémoires et les registres . . . 59
5.2.2 Les instructions et leurs formats . . . 59
5.2.3 Les comportements d'instructions . . . 60
5.2.4 Conclusion sur MDS . . . 61
5.3 Analyse et transformation des comportements d'instructions . . . 61
5.3.1 Vérification syntaxiques et premières transformations . . . 62
5.3.2 Expansion des nœuds d'accès aux opérandes . . . 62
5.3.3 Décoration des nœuds avec leurs types . . . 64
5.3.4 Uniformisation de l'arbre . . . 64
5.4 Extraction des signatures des instructions . . . 66
5.4.1 Exemple de l'instruction `div` avec plusieurs effets . . . 67
5.4.2 Exemple de l'instruction `j` avec registre de contrôle . . . 68
5.4.3 Exemple de l'instruction `lw` avec calcul d'adresse . . . 68
5.4.4 Exemple de l'instruction `beq` avec condition . . . 69
5.5 Mise en correspondance des instructions . . . 70
5.5.1 Correspondances entre nœuds . . . 71
5.5.2 La correspondance entre effets . . . 74
5.5.3 La correspondance entre signatures . . . 76
5.5.4 Algorithme de mise en correspondance . . . 76
5.5.5 Version non-exhaustive de l'algorithme . . . 83
5.5.6 Conclusion de la mise en correspondance . . . 83
5.6 Génération du simulateur . . . 83
5.6.1 Le *runtime* minimal . . . 84
5.6.2 Génération de l'IR spécialisée . . . 86
5.6.3 Génération du *frontend* et du *backend* . . . 86
5.6.4 Conclusion . . . 88
5.7 Expérimentation sur les correspondances . . . 88
5.7.1 Valeurs attribuées aux constantes d'évaluations pour les expérimentations . . . 89
5.7.2 Comportement de l'algorithme de mise en correspondance . . . 89
5.7.3 Limitations des transformations et de l'algorithme . . . 92
5.7.4 Statistiques sur les correspondances . . . 93
5.7.5 Conclusion . . . 96
5.8 Expérimentation sur la génération d'un *frontend* pour QEMU . . . 98
5.9 Conclusion . . . 98

6 Conclusion **101**

A Preuve de correction de l'algorithme de simulation VLIW **103**
A.1 Définitions préliminaires 103
A.2 L'algorithme de simulation d'une machine VLIW 103
A.3 Formalisation de l'état de la machine 104

B Résultat d'expérimentations du simulateur VLIW sur la suite Polybench **107**

C Génération automatique de décodeurs d'instructions **109**

Publications **119**

Chapitre 1

Introduction

La micro-électronique, omniprésente dans notre vie de tous les jours, est au fondement même de la société numérique et connectée d'aujourd'hui. Les progrès de ces trente dernières années ont permis de toujours plus miniaturiser les objets électroniques de notre quotidien, si bien qu'ils tiennent aujourd'hui dans notre poche. La densité d'intégration est telle que les concepteurs parviennent à placer, au sein d'une même puce, de nombreuses unités de calculs, de la mémoire et une foule de périphériques, formant ainsi un système complet. On parle alors de *Système sur la puce* ou *System on Chip (SoC)*.

Le nombre d'unités de calculs dans ces systèmes ne cesse de croître. Parmi ces unités, on trouve en général un ou plusieurs processeurs à usage général, et bien souvent des processeurs spécialisés comme des DSP, dédiés à des traitements précis. Un bon exemple d'un tel SoC est l'OMAP5 de Texas Instruments présenté par la figure 1.1. Ce dernier embarque un processeur double cœur à usage général ARM Cortex-A15, deux cœurs Cortex-M4, un DSP de la famille C64x, ainsi que de nombreuses unités dédiées à diverses applications.

Bien que la conception de telles puces soit complexe et demande du temps, elle n'est pas la seule étape dans la réalisation d'un système complet. Le développement du logiciel qui va s'exécuter sur ces puces représente une part sans cesse croissante du temps de conception global. Les fabricants doivent faire face à un marché extrêmement dynamique et en constante évolution, pour lequel le temps d'insertion et le cycle de vie d'un produit est de plus en plus réduit. Une de leurs problématiques est donc de développer le logiciel le plus rapidement possible et au plus tôt. Cependant, le matériel, nécessaire pour développer et mettre au point le logiciel, n'est pas disponible lors la première phase de la conception.

Le prototypage virtuel

Afin de pallier ce problème, une technique largement acceptée consiste à réaliser un prototype virtuel du matériel. Un tel prototype reproduit le comportement du matériel dans un environnement simulé s'exécutant sur une machine quelconque. Il permet le développement en amont du logiciel qui peut alors être testé, exécuté et mis au point sur la plateforme virtuelle. Il permet aussi d'aider à la conception du matériel, notamment pour évaluer les besoins et explorer l'espace de conception.

Cependant, réaliser un prototype virtuel est une tâche longue et complexe. La complexité et les cœurs de calculs toujours plus nombreux rendent leur simulation lente et les simulateurs difficiles à concevoir. Un prototype virtuel doit malgré tout conserver une vitesse de simulation acceptable pour être utilisable. Si un système d'exploitation, mettant quelques secondes à démarrer sur le vrai matériel, prend plusieurs heures en simulation,

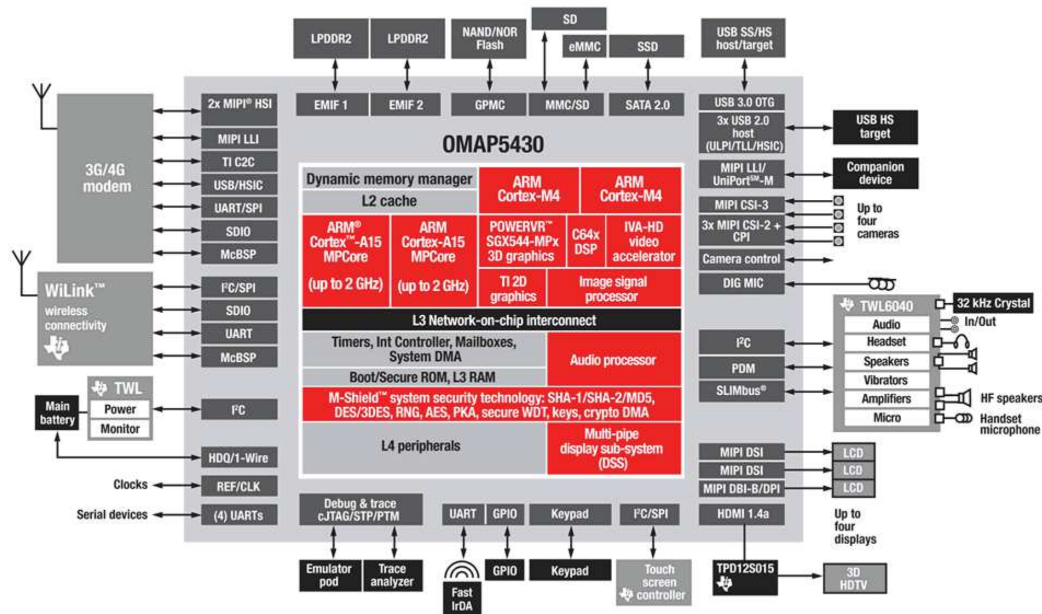


FIGURE 1.1 – Architecture de l'OMAP 5430 de Texas Instruments

la pertinence du prototype virtuel devient discutable. Une technique de simulation rapide de jeux d'instructions appelée la traduction binaire dynamique (*Dynamic Binary Translation (DBT)*) vient répondre à une partie de ce problème. Elle s'inscrit dans la lignée des techniques de simulation de jeux d'instructions en proposant de traduire les instructions du processeur simulé (la cible) en instructions de la machine exécutant la simulation (l'hôte). Elle est dynamique car la traduction est réalisée au fur et à mesure de la simulation, par opposition à une traduction statique faite une fois pour toute. De nombreux avantages découlent de cette technique, et notamment une vitesse de simulation des jeux d'instructions élevée, tout en gardant un haut niveau de flexibilité. Elle reste cependant difficile à développer et à mettre au point ce qui freine son adoption par rapport aux simulateurs interpréteurs classiques.

Objectifs de la thèse

Les travaux de cette thèse s'articulent autour de la DBT et explorent deux axes d'améliorations de cette technique. Le premier s'intéresse à la possibilité d'utiliser la technique de la DBT pour simuler un jeu d'instructions d'une architecture de type Very Long Instruction Word (VLIW) sur une architecture hôte scalaire classique. Les architectures VLIW sont souvent utilisées dans les processeurs spécialisés de type DSP comme le C64x dans l'exemple précédent. Elles ont la particularité d'exprimer explicitement le parallélisme entre instructions.

Le second se concentre sur l'aspect automatisation en proposant de générer un traducteur binaire dynamique en partant de la description architecturale du processeur à simuler et de la machine hôte dans l'optique de réduire la difficulté de conception d'un simulateur basé sur la DBT, et d'étudier les possibilités d'optimisations du traducteur lorsque l'on dispose de ces deux descriptions. Ces travaux sont donc une contribution au besoin de simuler

toujours plus vite et plus largement que ce qui est fait aujourd'hui.

Plan de la thèse

Le chapitre 2 présente de manière plus précise le contexte dans lequel s'inscrivent les travaux de cette thèse, et formule les questions auxquelles nous essaierons de répondre au fil du manuscrit.

Le chapitre 3 dresse un état de l'art de la simulation de processeurs et plus particulièrement de la technique de DBT. Il s'intéresse aux techniques de simulation rapide d'architectures à parallélisme explicite et à la génération de simulateurs.

Les chapitres 4 et 5 présentent les deux contributions de cette thèse. La première est une solution pour simuler un processeur de type VLIW à l'aide de la DBT, sur un processeur hôte scalaire. La seconde propose de générer un traducteur optimisé pour un couple cible – hôte à partir de leur description architecturale.

Finalement, le chapitre 6 conclut ce manuscrit en faisant un bilan des contributions et en proposant une ouverture vers des perspectives possibles et de futurs travaux.

Chapitre 2

Problématique

DANS ce chapitre, nous présentons les différents problèmes liés à la simulation rapide de processeurs. La pression sur le marché des ASIC, ainsi que la complexité toujours croissante des SoC dans lesquels les processeurs sont intégrés, poussent les constructeurs à opter pour des solutions de simulation. Cependant, les coûts de développement des simulateurs sont élevés, et conserver une vitesse de simulation correcte dans des environnements multiprocesseurs hétérogènes est un vrai défi.

Afin de comprendre les besoins en simulation, nous passerons en revue les différentes familles de jeux d'instructions (Instruction Set Architecture (ISA)), puis nous verrons les techniques classiques pour les simuler. Enfin, nous étudierons les solutions de description d' ISA haut-niveau pour nous intéresser au problème de la génération automatique de simulation de processeurs.

2.1 Les familles de jeux d'instructions

Il existe plusieurs grandes familles de jeux d'instructions que nous allons passer rapidement en revue.

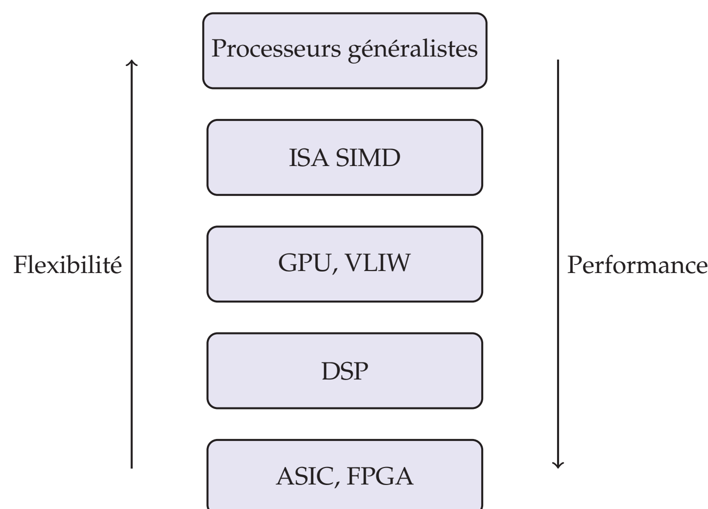


FIGURE 2.1 – Les grandes familles d'ISA

2.1.1 Processeurs généralistes, RISC et CISC

Ces processeurs que nous trouvons généralement dans nos machines de bureau et téléphones portables, sont pourvus d'un jeu d'instructions généraliste, capable d'exécuter de manière efficace le code des programmes courants (système d'exploitation, applications interactives, etc. . .) Ils sont généralement classés en deux catégories :

Les processeurs RISC pour Reduced Instruction Set Computer, ils sont composés d'instructions simples et peu nombreuses. Les instructions sont en général de type *registre-registre*, c'est à dire que les opérandes et la destination des instructions sont des registres. On y trouve aussi des instructions de chargement et écriture mémoire pour accéder à celle-ci. Les architectures *MIPS*, *ARM*, *PowerPC* et *SPARC* font partie de cette famille.

Les processeurs CISC pour Complex Instruction Set Computer, sont composés d'instructions pouvant réaliser des opérations complexes comme un chargement mémoire et une addition au sein de la même instruction. Les instructions acceptent donc une plus grande diversité d'opérandes (accès mémoire avec différents modes d'adressage, valeur immédiate. . .). On rencontre dans cette famille l'architecture *Intel IA-32* ou *x86*, équipant la majorité de nos ordinateurs personnels, ainsi que son évolution 64bit, l'architecture *Intel 64* ou *x86-64*.

2.1.2 Jeux d'instructions SIMD

Les jeux d'instructions Single Instruction Multiple Data (SIMD) viennent en général en supplément de l'ISA classique de nos processeurs. Ils ont la particularité de réaliser en parallèle la même opérations sur des vecteurs de données. Ces vecteurs sont souvent des registres de 64bits ou plus, l'instruction décidant de la granularité des données dans ces registres. Prenons par exemple l'instruction présentée au listing 2.1.

```
1   paddw %xmm0, %xmm1
```

Listing 2.1 – Exemple d'instruction SIMD MMX/SSE

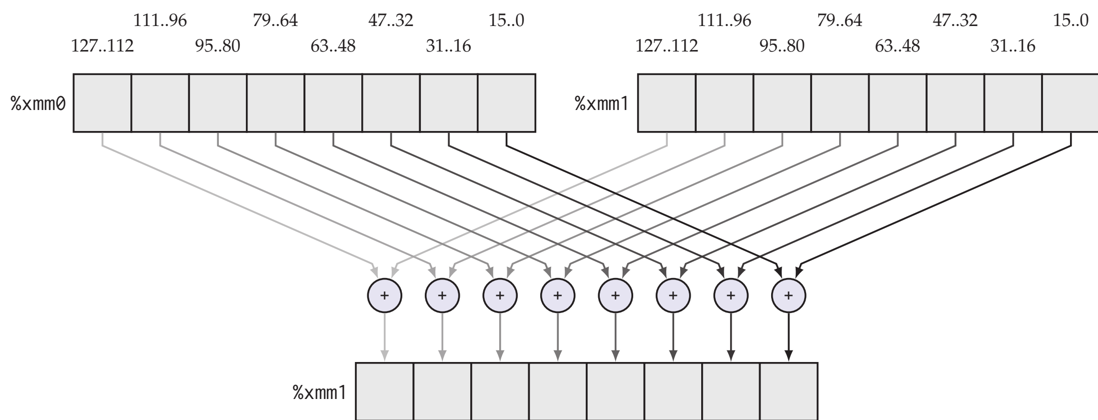


FIGURE 2.2 – Fonctionnement de l'instruction paddw

Elle fait partie du jeu d'instruction SIMD *MMX/SSE* du *x86*. La figure 2.2 décrit son fonctionnement. Les registres `%xmm0` et `%xmm1` sont des registres de 128bits. L'instruction considère ces registres comme étant des vecteurs de 8 éléments de 16bits. Elle additionne un à un les éléments de ces vecteurs et place le résultat dans `%xmm1`. Elle réalise donc 8 additions en parallèle.

Ces jeux d'instructions sont souvent utilisés dans les applications multimédia comme le traitement de flux audio ou vidéo, qui réalisent la même opération sur un grand nombre de données. C'est aussi le principe de base de fonctionnement des GPU.

2.1.3 Processeurs VLIW et DSP

Les architectures VLIW, contrairement au processeurs superscalaire classique, ont la particularité de laisser le contrôle de leurs pipelines au compilateur ou au programmeur. Elles exposent pleinement les ressources dont elles disposent, et donnent la possibilité d'exprimer l'ordonnancement des instruction au niveau de l'ISA.

On retrouve souvent ces architectures dans les processeurs de type Digital Signal Processor (DSP), qui sont des processeurs spécialisés dédiés à des tâches de calculs intensifs, comme le traitement d'un signal venant d'un capteur.

L'exemple du `TMS320C64x`

Pour mieux décrire le fonctionnement d'un tel processeur, nous prendrons en exemple dans ce document l'architecture `TMS320C64x`[Tex10a] de Texas Instrument, que nous abrégerons `c64x` dans la suite de ce document. Cette architecture est utilisée dans certains de leurs DSP. Elle est particulièrement adaptée à notre problème puisqu'elle possède les caractéristiques qui rendent difficile la simulation de VLIW. Ce sont ces caractéristiques que nous allons détailler.

Le `c64x` est une architecture VLIW disposant de deux fois 4 unités de traitement (ou unités fonctionnelles). On parle alors de processeurs VLIW 8 voies. Ces 8 unités sont découpées en deux chemins de données A et B, disposant chacun de leur banc de registres. Elles sont dénotées `.L1`, `.S1`, `.M1` et `.D1` pour le chemin de données A, et `.L2`, `.S2`, `.M2` et `.D2` pour le chemin de données B. La figure 2.3 illustre le pipeline du `c64x`.

Chacune de ces unités peut exécuter un sous-ensemble du jeu d'instructions du `c64x`. Par exemple, les unités `.D1` et `.D2` sont les seules ayant des instructions de chargement et écriture mémoire alors que les unités `.M1` et `.M2` prennent en charge les instructions de multiplication du processeur.

Les chemins de données A et B peuvent communiquer par l'intermédiaire d'un *cross path* dans chaque direction. Ainsi, une instruction s'exécutant dans A peut lire un registre du banc B et vice versa.

Une autre particularité importante du pipeline du `c64x` est l'absence des mécanismes classiques de résolution de dépendance de données inter-instructions. C'est un problème classique qui apparaît dans les processeurs pipelinés. Si une instruction a besoin d'une donnée se trouvant encore dans le pipeline au moment de son exécution, soit la donnée va être récupérée au moyen d'un court-circuit (*bypass*), soit le processeur va insérer une bulle dans le pipeline pour faire patienter l'instruction le nombre de cycles nécessaires. Ces mécanismes permettent de rendre transparent au programmeur ou au compilateur les aléas introduits par les architectures pipelinées.

Certaines architectures font le choix de ne pas résoudre tous les aléas. C'est le cas de l'architecture MIPS et de son *delay slot* pour les instructions de saut. Dans le cas de l'archi-

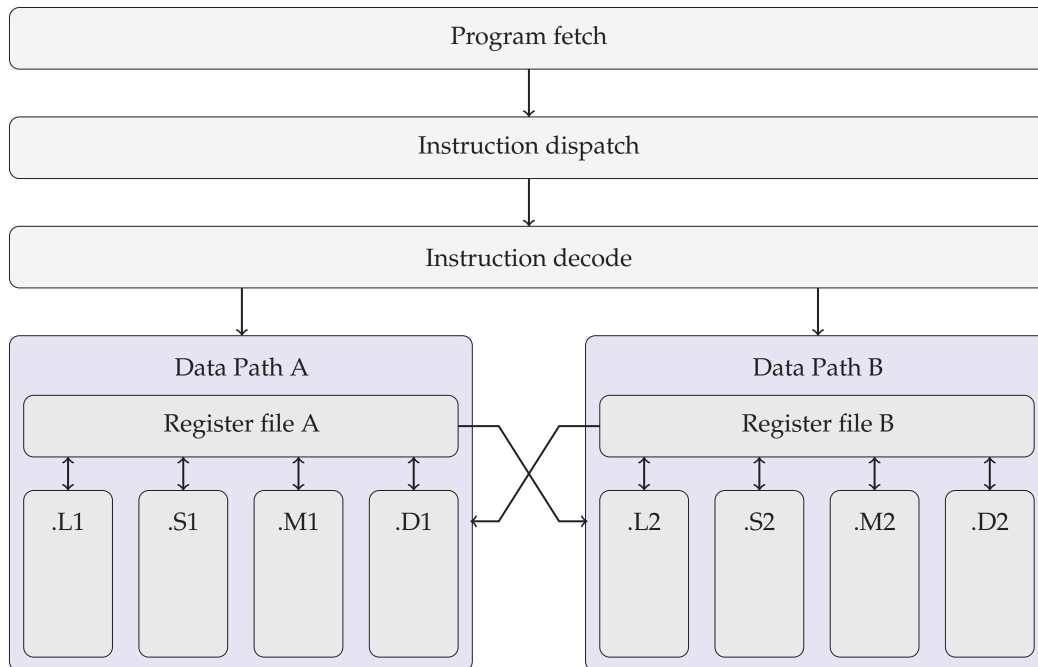


FIGURE 2.3 – Représentation du pipeline du c64x

tecture c64x, aucun de ces mécanismes n'est mis en place. C'est pourquoi le programmeur ou le compilateur doit connaître la latence de chaque instruction. Afin d'illustrer ce point, prenons l'exemple du code assembleur c64x donné au listing 2.2

```
1        add     .L1   a0, a3, a2       ; a2 <- a0 + a3
2     || mpy32   .M1   a0, a1, a2       ; a2 <- a0 * a1
3     || and     .S2   b0, b1, b2       ; b2 <- b0 & b1
4
5   [a0] add     .L1   a1, a2, a3       ; a3 <- a1 + a2
6     || or      .L2X  b0, a2, b4       ; b4 <- b0 | a2
7
8        nop
9
10       nop
11
12       sub     .L1   a2, a4, a5       ; a5 <- a2 - a4
```

Listing 2.2 – Exemple de code assembleur c64x

Le symbole || dénote l'exécution en parallèle d'une instruction avec celle qui la précède. Dans cet exemple, les instructions des lignes 1, 2 et 3 sont exécutées en parallèle. Il en va de même pour les instructions des lignes 5 et 6. Un groupe d'instructions exécutées en parallèle est parfois appelé *bundle*, ou *paquet d'exécution* (*execute packet*). Pour le premier bundle de l'exemple, les unités .L1, .M1 et .S2 sont utilisées. On pourrait donc encore compléter ce

bundle avec 5 autres instructions parallèles, pour un total de huit, qui correspond au nombre total d'unités. À la ligne 6, la présence du `X` à la fin de l'unité indique que le cross path doit être utilisé pour lire un registre du banc voisin (ici le registre `a2`). Finalement, à la ligne 5, la présence de `[a0]` signifie que l'instruction est prédiquée, elle ne sera exécutée que si le contenu du registre `a0` n'est pas nul.

Instructions et latences

Intéressons-nous maintenant à la latence de chaque instruction. Dans cet exemple, toutes les instructions sauf une ont une latence de 0. Leur résultat est donc directement disponible pour le bundle suivant. La seule instruction n'ayant pas une latence nulle est l'instruction `mpy32` à la ligne 2. Cette instruction de multiplication a une latence de 3, son résultat n'est donc pas immédiatement visible. Par exemple, l'instruction `add` à la ligne 5 lit le registre `a2`. Or, à ce moment, le registre ne contient pas encore le résultat de la multiplication entre `a0` et `a1`. C'est l'ancienne valeur de `a2` qui est lue. Le résultat est effectivement disponible à la ligne 12 pour l'instruction `sub`, les 4 cycles nécessaires à l'instruction s'étant écoulés.

Le même principe s'applique pour les instructions de branchement. Prenons l'exemple donné au listing 2.3.

```
1      b       .S1  some_label       ; instruction de branchement
2      add     .L1  a0, a1, a2       ; 1
3      nop                           ; 2
4      nop                           ; 3
5      nop                           ; 4
6      nop                           ; 5
7      sub     .L2   1, b2, b1       ; branchement effectué
```

Listing 2.3 – Un branchement en `c64x`

L'instruction de branchement à la ligne 1 a une latence de 5. Dans notre exemple, les instructions de la ligne 2 à 6 sont exécutées avant que le branchement ne soit effectivement pris. En revanche, l'instruction de la ligne 7 n'est pas exécutée.

Optimiser l'utilisation du pipeline

En exécution classique, il n'existe pas de cas où le processeur est susceptible de vider son pipeline. Une instruction qui est rentrée ressortira forcément au bout d'un nombre connu de cycles. Les compilateurs se servent de cette hypothèse pour optimiser au mieux l'utilisation du pipeline. L'exemple du listing 2.4 illustre ce dernier point.

Ce listing présente un programme qui réalise deux branchements consécutifs (ligne 1 et 2), patiente trois cycles, puis fait une multiplication (ligne 4). Le branchement ayant une latence de 5 cycles, et la multiplication une latence de 3 cycles, les instructions des ligne 2 et 4 sont toujours dans le pipeline lorsque le premier branchement est effectivement pris. Lorsque le saut à `lbl1` est réalisé, il reste un cycle avant que le deuxième ne survienne. L'instruction `add` de la ligne 10 est donc exécutée. En revanche au cycle suivant, le processeur branche à `lbl2`. Le `sub` de la ligne 11 n'est donc pas exécuté. Finalement, il faut encore patienter trois cycles pour obtenir le résultat de la multiplication de la ligne 4 dans `a2`.

```
1       b       .S1 lbl1            ;
2       b       .S1 lbl2            ; 1
3       nop     3                   ; 2,3,4
4       mpy32   .M1 a0, a1, a2      ; 5
5       add     .L1 a0, a2, a2      ; branchement vers lbl1
6       [...]                       ; (add non exécuté)
7
8   lbl1:
9       add     .L1 a2, a2, a3      ;
10      sub     .L1 a2, a1, a3      ; branchement vers lbl2
11      [...]                       ; (sub non exécuté)
12
13  lbl2:
14      nop                         ;
15      nop                         ;
16      and     .L1 a2, a0, a0      ; Résultat du mpy32 disponible
17                                  ; dans a2
```

Listing 2.4 – Gestion du pipeline et des latences

Pour résumer

Nous venons de voir quelques caractéristiques de l'architecture `c64x` :

- La gestion du pipeline est laissée au compilateur ou au programmeur. Le processeur expose ses ressources et permet d'exprimer le parallélisme entre instructions directement dans l'ISA.
- Chaque instruction a sa propre latence. Le pipeline n'étant pas équipé de mécanisme de résolution d'aléas, le programmeur et le compilateur doivent en tenir compte, et peuvent se servir de ces latences pour optimiser au mieux le flux d'instructions dans le pipeline.
- Le processeur dispose de deux chemins de données, possédant chacun leur banc de registres et unités fonctionnelles. Des *crosspath* sont disponibles pour échanger la valeur d'un registre d'un chemin de données à l'autre.
- Les instructions peuvent être prédiquées. Elles ne seront exécutées que si la condition est remplie.
- Le pipeline n'est jamais vidé en exécution normale.

L'exemple du `c64x` est intéressent car il présente les principales caractéristiques des architectures VLIW qui rendent la simulation non triviale.

2.2 Les techniques de simulation d'ISA

Comme nous l'avons évoqué précédemment, la simulation connaît un essor important dans le monde de la conception des systèmes sur puce. Elle accélère grandement le développement, tant matériel que logiciel des systèmes. Le débogage sur plateforme de simulation est bien plus aisé que sur un vrai système, du fait d'une meilleure observabilité du système simulé. Elle permet aussi d'obtenir des estimations de performance avant même que

le système réel ne soit conçu, ce qui permet de l'adapter en cas de besoin.

Il existe de nombreuses techniques de simulation d'ISA. Les simulateurs d'ISA sont appelés Instruction Set Simulator (ISS). Le choix de la technique dépend principalement des besoins de l'utilisateur. D'une manière générale, plus la simulation sera proche du modèle réel, plus elle sera précise mais lente. À l'opposé, plus elle s'élèvera en niveau d'abstraction, plus elle sera rapide mais imprécise[PGH+11].

La figure 2.4 présente trois familles de techniques, de la plus précise à la plus rapide. Nous détaillerons par la suite ces trois familles, puis nous nous intéresserons à une technique en particulier, la *traduction binaire dynamique*.

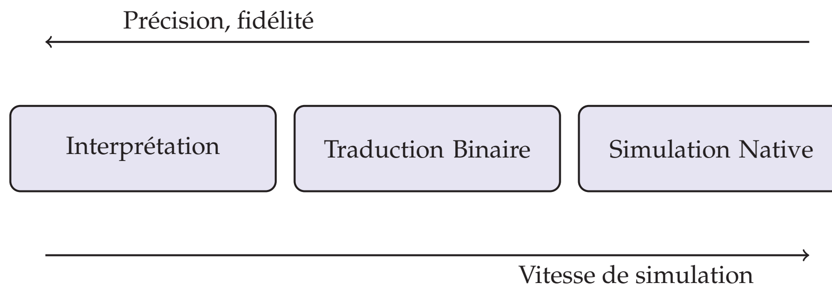


FIGURE 2.4 – Classification des techniques de simulation d'ISA

2.2.1 Détail des différentes familles

L'interprétation

Les techniques d'interprétation d'ISA simulent les unes après les autres les instructions du programme. Pour chaque instruction, ce processus passe par une étape de récupération, de décodage, et de simulation de l'instruction. L'algorithme 1 décrit ce processus.

Algorithme 1 Simulation d'ISA par interprétation

```
1  tant que simulation running faire
2      opcode ← mem(regs[pc])                          ▷ Récupération
3      dec ← decode(opcode)                            ▷ Décodage
4      selon dec.op faire
5          ...
6          cas ADD                                     ▷ Simulation de l'instruction ADD
7              regs[dec.dst] ← regs[dec.src1] + regs[dec.src2]
8          ...
9      fin selon
10 fin tant que
```

La première étape récupère, dans la mémoire de la machine simulée, le code opération (*opcode*) à l'adresse courante du compteur ordinal. Cet opcode est ensuite décodé pour déterminer le type d'instruction dont il s'agit, les opérandes d'entrées, de sorties, etc. Finalement, le simulateur reproduit le comportement de l'instruction de manière procédurale.

Cette technique reste proche du comportement réel d'un pipeline de processeur. Les simulateurs l'exploitant sont souvent *instruction accurate* voire *cycle accurate*. En revanche,

cette méthode reste très lente puisque pour chaque instruction, le processus présenté par l'algorithme 1 est exécuté.

Une optimisation consiste à stocker le résultat du décodage dans un cache afin de le réutiliser directement les fois suivantes si l'instruction est à nouveau exécutée (par exemple dans le cas d'une boucle). Si le simulateur modélise le cache d'instructions, une bonne technique consiste à stocker dans celui-ci, non pas les opcodes, mais la forme décodée des instructions[PGH+11].

La traduction binaire

La traduction binaire, dans le cas de la simulation d'ISA, permet de garder un niveau de précision raisonnable, tout en gagnant très significativement en vitesse de simulation. Le principe de base consiste à traduire le code de la machine simulée (la *cible*) en un code exécutable par la machine exécutant la simulation (*l'hôte*).

Il existe deux techniques qui se distinguent l'une de l'autre :

La traduction binaire statique (Static Binary Translation (SBT)), elle consiste à traduire une fois pour toutes le programme que l'on souhaite simuler. Cette technique est complexe et pose des problèmes de fond lors de la traduction. Il est par exemple délicat de traduire un branchement dont la cible n'est pas connue lors de la traduction, car l'espace d'adressage n'est pas le même entre les deux versions. De plus, il n'est pas possible de supporter simplement le code automodifiant.

La traduction binaire dynamique (DBT), alterne entre les phases de traductions et d'exécutions du code généré. Le fonctionnement de cette technique est décrit en détail dans la section 2.2.2

La simulation native

La simulation native est la technique qui s'éloigne le plus de la machine que l'on souhaite simuler. Le gain en vitesse est très important, au prix d'une simulation imprécise. En effet, cette technique ne permet pas d'extraire une quelconque information de performances de la machine simulée (cycles, consommation, ...). On parle alors de simulation fonctionnelle.

Le principe est de compiler le code que l'on souhaite simuler directement pour la machine hôte. Afin de faire l'interface entre le logiciel et le matériel, il est nécessaire d'implémenter une couche d'abstraction matérielle (Hardware Abstraction Layer (HAL)). Une implémentation de cette HAL spécifique au simulateur permet au programme simulé de s'exécuter de manière transparente sur la machine hôte. La HAL quant à elle communique avec des couches de simulation de plus bas niveau, comme par exemple une implémentation SystemC des périphériques de la plateforme simulée.

Une limite de cette méthode est la nécessité d'avoir les sources du programme que l'on souhaite simuler.

2.2.2 La traduction binaire dynamique

Nous allons maintenant détailler la technique de la DBT puisque c'est elle qui va nous intéresser dans le cadre de ces travaux.

Le principe général de cette technique de simulation d'ISA est de traduire le code de l'architecture cible que l'on cherche à simuler en un code de même comportement, exécutable

par la machine hôte. Le code cible est découpé en blocs appelés *blocs de traduction* (Translation block (TB)). Un TB commence à la valeur courante du compteur ordinal de la machine simulée, et se termine sur une instruction de branchement.

Il existe plusieurs manières de réaliser le cœur de traduction du simulateur. La manière la plus simple consiste à traduire directement le code cible en code hôte. Cependant, beaucoup de traducteurs dynamiques choisissent d'ajouter une étape supplémentaire dans la traduction, en utilisant une *représentation intermédiaire* (Intermediate Representation (IR)). Le code cible est d'abord traduit dans cette représentation intermédiaire, puis le code final est généré à partir de celle-ci. Cette représentation intermédiaire est composée d'instructions simples, parfois appelées micro-opérations, en nombre et expressivité suffisante pour simuler n'importe quelle architecture. Elle a plusieurs avantages par rapport à une traduction directe :

– Elle permet la décorrélation de la cible et de l'hôte, il est par la suite plus aisé d'ajouter une nouvelle cible ou un nouvel hôte au simulateur.
– Elle peut bénéficier d'optimisations génériques lors de la traduction, applicables à n'importe quel couple cible – hôte.

La figure 2.5 illustre le fonctionnement d'un tel simulateur. La première étape consiste

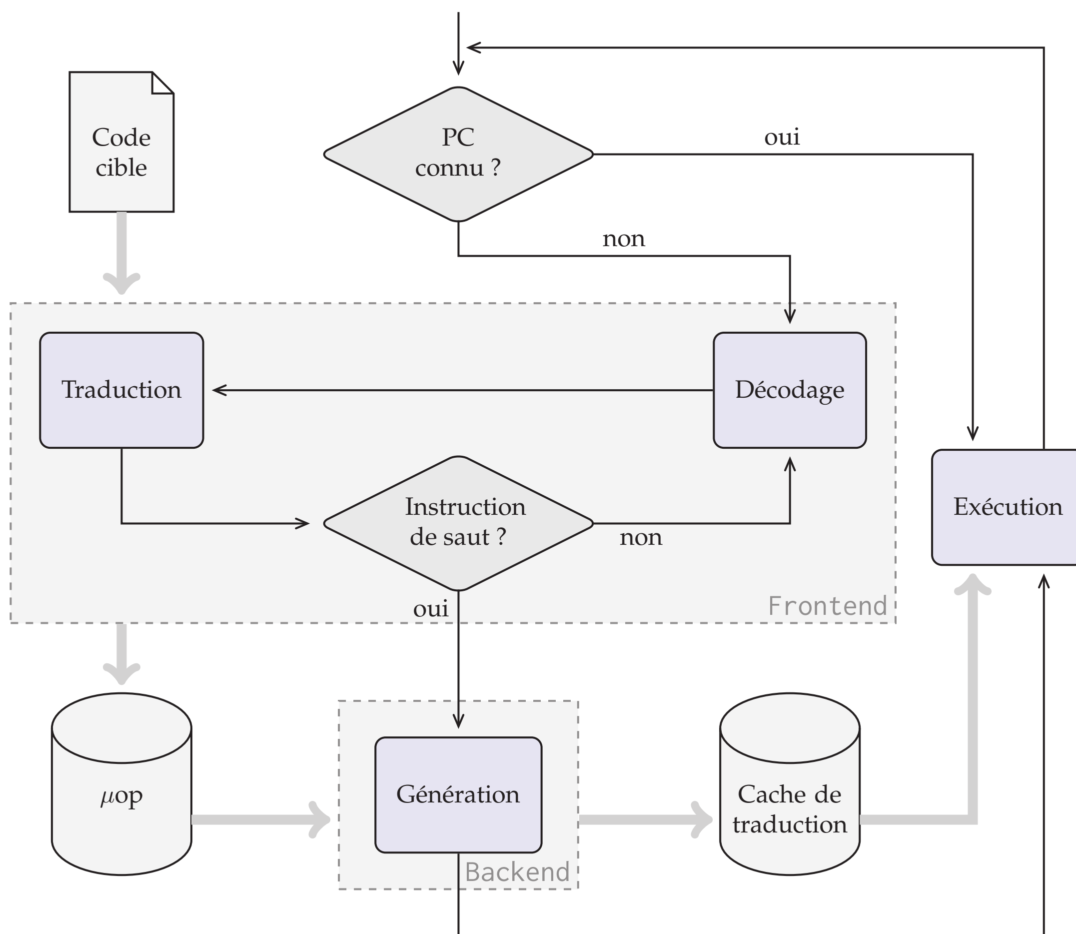


FIGURE 2.5 – Fonctionnement de la simulation par traduction binaire dynamique

à vérifier si la valeur courante du compteur ordinal a déjà été rencontrée. Si oui, on peut

immédiatement exécuter le code généré correspondant. Sinon, le simulateur rentre dans le processus de traduction. Il commence par traduire le code cible en micro-opérations de l'IR à l'aide du *frontend*. Celui-ci remplit le tampon de micro-opérations tant qu'il ne rencontre pas d'instruction de saut, marquant la fin du TB en cours de traduction. Lorsqu'une instruction de saut est rencontrée, le simulateur passe la main au *backend* qui traduit les micro-opérations présentes dans le tampon en code hôte, pour former le *bloc de traduction traduit* (Translated Translation block (TTB)). Ce dernier est placé dans le cache de traduction, et est finalement exécuté pour simuler le code cible. Ce processus est répété pour faire avancer la simulation.

Les TTB étant placés dans le cache de traduction, le simulateur peut les exécuter plusieurs fois sans avoir à les retraduire. Cependant, lorsque ce cache est plein, il faut faire un choix pour y libérer de l'espace, le plus simple consistant à le vider complètement.

Cette technique de la DBT donne de très bons résultats en terme de vitesse de simulation mais reste compliquée à implémenter.

2.3 Générer automatiquement un simulateur

2.3.1 Les ADL

Il existe de nombreux moyens de décrire un système, du niveau le plus bas (niveau transistor) au plus élevé en abstraction (langages de haut niveau), chacun ayant ses avantages et inconvénients.

Les Architecture Description Language (ADL) sont une famille de langages spécialement conçus pour la description d'architectures. Ils sont utilisés comme support à la génération de modèles matériels, de validation, de tests et environnement de développement (compilateurs, simulateurs, débogueurs, ...).

Ils se découpent suivant deux grandes familles[MD08], les ADL structurels et les ADL comportementaux. Les ADL structurels sont plus adaptés pour le prototypage matériel puisqu'ils capturent la structure, parfois à grain fin, du processeur qui y est décrit. Les ADL comportementaux sont quant à eux orientés prototypage virtuel, compilation, simulation, puisque de plus haut niveau. Ils capturent le comportement du système plutôt que sa structure.

Dans le cadre de la génération automatique de simulateurs en partant d'une description haut-niveau, les ADL comportementaux seront donc plus adaptés. Ils permettent de spécifier l'ISA du processeur et de décrire le comportement de chaque instruction d'un point de vue comportemental, contrairement à un ADL de type structurel, duquel il est plus difficile d'extraire ces informations de manière systématique.

2.3.2 Génération de simulateurs à l'aide d'ADL

La génération de simulateurs à partir d'une description dans un ADL est une technique déjà bien établie[PHM00, BNH+04, RBMD03, CVER99, UC00, CVERL02, Pro01]. Le générateur exploite la sémantique de la description pour produire un simulateur plus ou moins précis suivant les possibilités offertes par l'ADL et les besoins de l'utilisateur. Les simulateurs peuvent fonctionner à base d'interprétation ou de traduction binaire, le premier cas étant le plus courant.

Un point important de cette génération est la manière dont est décrit le comportement des instructions. Dans de nombreux cas, une description de type impérative est utilisée, ce

qui rend difficile l'analyse de la sémantique du comportement des instructions par le générateur. Celui-ci doit se contenter de recopier ce comportement dans le simulateur généré, sans pouvoir faire d'optimisations dessus.

2.4 Conclusion

Nous venons d'aborder différents points autour de la simulation rapide de processeurs. Nous avons énoncé les spécificités de quelques familles d'architecture comme les jeux d'instructions SIMD et VLIW, puis nous avons présenté différentes techniques de simulation.

La traduction binaire dynamique est une technique intéressante puisqu'elle permet d'obtenir de bonnes performance en terme de vitesse de simulation, tout en gardant une précision acceptable.

Finalement, nous nous sommes intéressés à la génération automatique de simulateurs à partir d'une description dans un ADL de l'architecture cible.

Voici les questions auxquelles nous tenterons de répondre :

– Comment utiliser la traduction binaire dynamique pour simuler des jeux d'instructions VLIW sur des processeurs scalaires ?
– Comment générer des simulateurs d'ISA basés sur des techniques de DBT ?
– Comment optimiser le processus de DBT généré pour le couple cible/hôte donné ?

Chapitre 3

État de l'art

DANS ce chapitre, nous présentons un panorama de la simulation rapide de processeurs, principalement axé sur la traduction binaire dynamique. Cette technique consiste à traduire le code binaire exécutable que l'on souhaite simuler en un code binaire exécutable par la machine qui exécute la simulation, et ce de manière dynamique. Par ailleurs, elle est utilisée dans divers domaines, notamment à des fins de support et migration vers de nouvelles architectures, d'optimisation binaire, d'instrumentation et d'implémentation de machines virtuelles.

Nous explorerons ensuite les techniques de simulation d'architectures de type VLIW, en recherchant en particulier s'il existe des techniques qui permettent de simuler de manière efficace ce genre de processeurs autrement que par interprétation tout en étant fonctionnellement correct.

Finalement, nous étudierons les possibilités de génération automatique de simulateurs rapides en partant d'une description de type ADL des architectures cible et hôte.

3.1 La traduction binaire dynamique

La traduction binaire dynamique (Dynamic Binary Translation ou DBT) est une technique consistant à traduire le code binaire d'un jeu d'instruction vers un autre.

3.1.1 Dans les machines virtuelles

L'une des premières utilisations de la DBT se fit dans le cadre de l'optimisation des machines virtuelles. Deutsch et Schiffman[DS84] décrivent l'implémentation d'une machine virtuelle optimisée pour le langage Smalltalk-80. Parmi les optimisations décrites, la plus importante selon les auteurs est la traduction du *bytecode* Smalltalk en code binaire pour la machine hôte. Ils s'inspirent de [Rau78] qui semble être le premier à introduire la notion de traduction dynamique. Il décrit ce processus comme étant un bon compromis entre l'interprétation d'un langage intermédiaire peu gourmand en mémoire qui sera forcement coûteuse, et un code binaire directement sur un processeur.

Le monde des machines virtuelles continue d'exploiter cette technique, notamment pour le langage Java[LYBB13, YMP+99, BCF+99]. Elle est alors appelée Just-In-Time Compilation (JIT). Ces machines virtuelles possèdent plusieurs niveaux d'optimisation du code exécuté. Par défaut, le bytecode est interprété sans plus d'optimisation. Si une partie du code est exécutée suffisamment souvent, il est alors traduit en code binaire pour la machine hôte.

C'est le premier niveau d'optimisation. On parle alors de *hot spot* ou *hot path*. Si une partie de code dépasse un deuxième seuil en nombre d'exécutions, il est retraduit en un code de meilleur qualité. C'est le deuxième niveau d'optimisation.

3.1.2 Pour le support à la migration de code sur de nouvelles architectures

Un autre domaine utilisant la DBT est le support à la migration vers de nouvelles architectures. Il s'agit d'exécuter des programmes compilés pour d'anciennes architectures sur de nouveaux processeurs non compatibles d'un point de vue ISA[EA97, CHH+98, ZT00]. La traduction est faite de manière transparente pour l'utilisateur. Par exemple, DAISY[EA97] traduit des binaires PowerPC vers une architecture VLIW en essayant de tirer partie du parallélisme au niveau instruction (Instruction Level Parallelism (ILP)). FX !32[CHH+98] permet l'exécution de programmes x86 sur des architectures Alpha au dessus de Windows NT. Enfin, Aries[ZT00] traduit les programmes HP-PA vers l'architecture IA-64.

3.1.3 Pour l'optimisation binaire

Outre le support vers de nouvelles architectures, la DBT est utilisée pour optimiser à la volée un programme déjà compilé. On parle parfois de recompilation à la volée. Le but est souvent de tirer partie d'une nouvelle version de l'architecture, contenant par exemple de nouvelles instructions plus efficaces. Le traducteur peut aussi tirer partie d'informations propres à l'exécution comme la probabilité pour un branchement conditionnel d'être pris, pour optimiser le flux binaire, informations que le compilateur ne possède pas lors de la compilation. Dynamo[BDB00] optimise le flux binaire au dessus du système HP-UX, arrivant à une accélération entre 10% et 20%.

3.1.4 Pour l'instrumentation et l'aide à la mise au point

La DBT peut aussi être utilisée à des fins d'instrumentation des programmes exécutés. Valgrind[NS07] est un *framework* ainsi qu'un ensemble de modules permettant la mise au point de programmes. Le plus connu est *memcheck*, le module permettant le débogage des erreurs mémoires. Il remplace à la volée les accès mémoires afin de les tracer et de tester leur validité. Dans la même idée, Pin[LCM+05] fournit au programmeur un *framework* de traduction binaire pour réaliser simplement les instrumentations de son choix dans un programme. Le programmeur peut alors créer des outils s'appuyant sur Pin pour effectuer dynamiquement divers traitements sur les instructions d'un programme. Un bon exemple d'une utilisation de Pin est PerPI[GLPP12] qui calcule l'ILP d'un programme en étudiant les dépendances de chaque instruction exécutée.

3.1.5 Pour la simulation rapide

Finalement, on retrouve la DBT dans le domaine de la simulation rapide de processeurs. La première utilisation semble être Mimic[May87] qui simule l'architecture System/370 sur l'IBM PC RT à l'aide de la DBT. L'auteur introduit la notion de bloc de traduction (Translation block), qu'il nomme *code block*. Il distingue trois générations de simulateurs, la première étant la simulation par interprétation comme décrite au chapitre 2.2.1. La deuxième génération est la variante de l'interprétation consistant à stocker le résultat du décodage dans un cache. Finalement, il introduit la troisième génération comme étant la DBT, bien qu'il ne la

nomme pas ainsi. Les expérimentations fournies montrent le gain en terme de nombre d'instructions hôtes utilisées pour simuler une instruction cible. L'auteur part du constat que la première génération de simulateurs réalise un ratio de l'ordre de 100 instructions hôtes pour une instruction cible. Pour la deuxième génération, ce ratio est de l'ordre de dix pour un. Ses expérimentations montrent qu'il arrive à un ratio de l'ordre de quatre pour un avec Mimic, en ne prenant en compte que le code généré, et pas la phase de traduction.

Embra[WR96] simule un système complet à base de MIPS. Il est basé sur Shade[CK93] et est un bon exemple de simulateur fonctionnel abouti. Il peut simuler un système multi-processeurs ainsi que les caches de chacun.

Finalement, QEMU[Bel05] s'inscrit dans cette lignée en fournissant un simulateur riche et performant. C'est un logiciel libre s'appuyant sur la DBT pour simuler de nombreux systèmes. À l'heure actuelle, il supporte 16 architectures cibles (architectures qu'il peut simuler) et 7 architectures hôtes (architectures sur lesquelles il peut s'exécuter). Il simule aussi une grande variété de périphériques (cartes réseaux, framebuffers, ...).

Son cœur de traduction repose sur le Tiny Code Generator (TCG). Il est muni d'une représentation intermédiaire (IR) composées de micro-opérations. Ces dernières forment un jeu d'instructions simple, similaire à celui d'un processeur RISC élémentaire. La richesse de cette IR permet ou non d'exploiter des instructions spécialisées sur l'hôte, ce qui a un impact non négligeable sur les performances globales du simulateur, comme montré dans [RWY13, MFP11]. Le TCG utilise des *temporaries* comme valeurs intermédiaires dans l'IR lors de la phase de traduction dans le frontend. Un processus d'allocation de registres les remplace par des registres hôtes pendant la phase de génération dans le backend.

QEMU peut aussi fonctionner en mode *user*, mode dans lequel il ne simule pas un système complet. Il permet d'exécuter des binaires compilés pour un système GNU/Linux pour une architecture différente de la machine courante. Il se contente de traduire le code par DBT, et traduit les appels systèmes pour les transférer à l'hôte.

Finalement, il est le frontend officiel pour le système de virtualisation KVM[KKL+07] du noyau Linux. Dans ce mode, le cœur de traduction binaire n'est pas utilisé.

3.2 Simulation rapide d'architectures VLIW

De part leur nature, les architectures VLIW sont compliquées à simuler. Le pipeline du processeur doit être reproduit de manière relativement précise pour respecter les latences explicites de chaque instruction. Par interprétation, la conception du simulateur reste simple, mais son exécution résultante lente.

3.2.1 Par traduction statique

Moreno et al.[MME+97] proposent un ensemble d'outils pour aider au développement sur VLIW. Parmi eux, on trouve un traducteur statique de code assembleur VLIW vers l'architecture PowerPC.

Cette traduction a lieu en deux phases. La première consiste à traduire le code VLIW en une représentation intermédiaire correspondant à un ISA VLIW abstrait appelé ForestaPC. Cette représentation intermédiaire est ensuite traduite en code PowerPC. Le code ainsi produit peut être assemblé normalement pour simuler le programme. Le traducteur possède deux modes de fonctionnement. Un mode *exploration* qui permet d'exécuter rapidement le programme mais se contente de simuler seulement l'ISA, et un mode *évaluation* qui simule aussi le processeur et la mémoire, ce qui permet d'extraire des métriques de performances.

Les auteurs comparent la vitesse d'exécution de chacun des deux modes à la vitesse d'exécution de la version native PowerPC. Pour leurs différents programmes de test, le mode *exploration* est jusqu'à 10 fois plus lent que la version native, alors que le mode *évaluation* est entre 500 et 1500 fois plus lent.

Cette solution apporte des résultats satisfaisants en terme de vitesse de simulation en mode *exploration*, mais comporte les inconvénients de la traduction statique comme la difficulté à supporter les indirections calculées à l'exécution (branchements indirects, tableaux de pointeurs, ...), le chargement dynamique de code, le code auto-modifiant, etc.

3.2.2 Par simulation compilée

La simulation compilée est une technique proche de la traduction binaire statique (SBT). Plutôt que de traduire le code cible en code hôte, le programme à simuler est traduit en langage de haut niveau comme le C. C'est ensuite le compilateur de la machine hôte qui fait le reste du travail. Le programme résultant a le même comportement que le programme à simuler.

Farfeleder et al.[FKH07] utilisent la simulation compilée pour simuler une architecture VLIW. Le code est traduit à la granularité du bloc de base (*basic block*). Chaque bloc de base est traduit en une fonction C. Celle-ci simule le comportement de chaque instruction du bloc et retourne le numéro du bloc suivant à exécuter. Les adresses des fonctions correspondant à chaque bloc sont placées dans un tableau indexé par le numéro de bloc. Il suffit d'utiliser ce tableau à l'index renvoyé par la fonction précédente pour connaître l'adresse de la fonction suivante. La structure de la boucle principale est présentée au listing 3.1.

`bb_num` commence par prendre le numéro du bloc correspondant au point d'entrée du programme. La boucle s'exécute tant que le numéro de bloc est positif ou nul. Le cas où le numéro est négatif correspond à une destination inconnue et est discutée au paragraphe suivant. `bb_num` est ensuite utilisé pour indexer le tableau `bbs` contenant les adresses des fonctions simulant les blocs. La fonction correspondante est appelée, et `bb_num` est mis à jour avec le retour de celle-ci.

```
1  int bb_num = first_bb;
2  while(bb_num >= 0) {
3    bb_num = bbs[bb_num]();
4  }
```

Listing 3.1 – Structure de la boucle principale

Reconstruction du *control flow graph*

Pour connaître l'enchaînement des blocs de base, il faut regarder la destination du saut et la faire correspondre au numéro de bloc de base. Dans certains cas, la destination n'est pas connue statiquement. La correspondance [adresse $\rightarrow$ numéro de bloc] doit donc être conservée à l'exécution.

Finalement, lorsqu'à l'exécution, la correspondance ne peut pas être établie, cela signifie que les limites de la traduction statique ont été atteintes. Le programme essaie de se diriger vers une zone de code qui n'a pas été traduite statiquement. Cela peut être dû à du code auto-modifiant, à un chargement dynamique ou bien à une zone de code qui n'a pas été

découverte statiquement. Dans ce cas, les auteurs choisissent de passer la main à un interpréteur classique, lent mais à même de simuler les instructions de l'architecture cible.

Gestion des latences d'instructions

Pour gérer les latences des différentes instructions du VLIW simulé, les auteurs mettent en place plusieurs mécanismes.

Pour les instructions de saut, ils ajoutent à la fin du bloc de base les instructions présentes dans le *delay slot* du saut. Le bloc de base se termine donc au moment effectif du saut et non lorsque l'instruction de saut est rencontrée.

Un problème survient lorsque le bloc de base se termine alors qu'une instruction est toujours dans le pipeline. La fonction simulant le bloc de base suivant doit terminer l'exécution de cette instruction. Dans le cas où le bloc suivant est connu statiquement, il est possible de le générer de manière à prendre en compte cette instruction. Cependant, un bloc peut être la destination d'autres blocs n'ayant pas d'instruction à terminer. Les auteurs choisissent donc de répliquer le bloc destination, laissant une version générée normalement, et une autre autre prenant en compte l'instruction à terminer dans le prédécesseur concerné.

Lorsque le problème n'est pas soluble statiquement, le programme généré passe la main à l'interpréteur en lui fournissant une structure représentant l'état du pipeline à la fin du bloc. Cette structure permet à l'interpréteur de terminer correctement l'exécution d'éventuelles instructions encore dans le pipeline.

Performances et conclusion

Les auteurs rapportent une simulation 1000 à 3000 fois plus rapide grâce à leur implémentation de simulation compilée pour l'architecture xDSP, et ce comparée à un interpréteur classique.

Cependant, il n'y a pas d'indication quant au taux d'utilisation de l'interpréteur intégré dans leur programme généré. Les résultats présentés se basent essentiellement sur des benchmarks de calculs intensifs. La traduction statique montre ses limites dans les cas de chargement dynamique de code, cas omniprésent dans un système d'exploitation par exemple.

3.3 Génération de simulateurs rapides à partir d'un ADL

Développer un simulateur rapide est une tâche longue et compliquée, notamment lorsqu'on utilise des techniques avancées comme la DBT. La mise au point des parties spécifiques à l'architecture simulée est souvent délicate, et la réutilisation du code pour de futurs développements est en général très limitée.

Est-il possible d'aider à la conception de ces simulateurs rapides en en générant une partie ou la totalité ? Pour un simulateur à base de DBT, peut-on obtenir un simulateur performant en le générant à partir d'une description ADL de la cible ? Finalement, peut on prendre en compte l'hôte et sa description ADL lors de la génération pour tirer pleinement partie de son ISA ?

3.3.1 Générer des simulateurs basés sur la simulation compilée

La génération d'un simulateur utilisant la technique de simulation compilée semble à première vue envisageable. En effet, le résultat de la traduction statique d'un programme en simulation compilée est une représentation dans un langage de haut niveau comme le C. Pour générer un tel traducteur, il est donc nécessaire d'avoir la description de la machine cible seulement, la partie *backend* étant assurée par le compilateur de la machine hôte.

Pees et al. proposent d'utiliser le langage LISA[ZPM96, PHZM99] pour générer un simulateur utilisant la simulation compilée[PHM00]. L'une de leurs cibles de simulation est le `TMS320C62x`[Tex10b] (`c62x`), version plus ancienne du `c64x` présentée section 2.1.3. La simulation générée pour cette architecture est *cycle accurate*.

Le flot de traduction est décrit par la figure 3.1. Comme pour la SBT, le programme à simuler est traduit en programme exécutable sur l'hôte, avant la simulation.

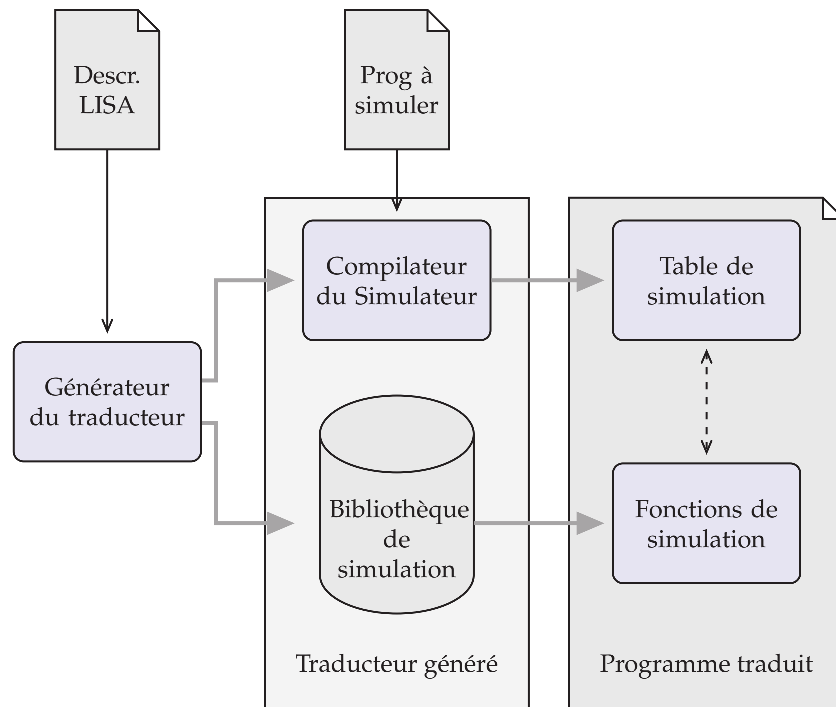


FIGURE 3.1 – Flot de génération de simulation compilée

Ici, le traducteur est lui-même généré à partir de la représentation LISA de la machine cible. Il est composé de deux parties :

Le compilateur du simulateur qui s'occupe d'extraire le flot de contrôle du programme à simuler et de créer la *table de simulation* dans le programme final. Celle-ci possède une entrée pour chaque instruction du programme cible. Elle est indexée par l'adresse de l'instruction en question et contient l'adresse des fonctions à appeler pour simuler cette instruction.

La bibliothèque de simulation qui comprend la description comportementale de toutes les instructions de la machine cible. Elle est extraite de la description LISA, et permet de générer les *fonctions de simulations* référencées par la table de simulation.

La table de simulation contenant une entrée par instruction, la simulation est réalisée à la granularité de l'instruction. La figure 3.2 donne un exemple de table de simulation générée.

Adresse	fonction de simu	fonction de simu	fonction de simu	...
`0x8000`	`fetch()`	`decode()`	`addi()`	...
`0x8004`	`fetch()`	`decode()`	`sw()`	...
`0x8008`	`fetch()`	`decode()`	`mult()`	...
...	...	...	...	...

FIGURE 3.2 – Exemple d'une table de simulation

Chaque instruction fait appel à une ou plusieurs fonctions de simulation. On peut voir ici que chaque fonction simule en fait une partie du pipeline du processeur. Ceci est dû à la manière dont sont décrites les instructions en LISA. Leur comportement est découpé par étage de pipeline, ce qui simplifie grandement la génération d'un simulateur *cycle accurate*.

Finalement, la génération de ces fonctions est directe puisque la description comportementale des instructions en LISA est faite dans le langage C++.

Les auteurs rapportent un facteur d'accélération de 36 à 169 par rapport au simulateur interpréteur de Texas Instrument.

Cependant, les simulateurs générés sont impactés par les limitations de la SBT et de la simulation compilée. Ils ne peuvent simuler du code dont des parties sont résolues dynamiquement ni du code auto-modifiant.

Pour pallier ce problème, Braun et al. propose JIT-CCS[BNH+04] qui remet en cause l'aspect statique de la table de simulation. Plutôt que de la générer *hors-ligne* lors de la phase de traduction, celle-ci est générée dynamiquement et placée dans un cache, à la manière de la DBT. Une partie du *compilateur du simulateur* est donc intégrée dans le programme généré, et est appelée dès qu'une instruction n'est pas trouvée dans le cache.

Lors de leurs expérimentations, les auteurs font varier la taille du cache et mesurent le taux de *miss* dans le cache ainsi que les performances en terme de vitesse de simulation, comparé à la version statique. Ils constatent que le cache n'a pas besoin d'être important pour obtenir des performances correctes. Pour un cache d'environ 2Mio, les performances atteignent 95% des performances de la version statique. Cette dernière a besoin de 23Mio pour stocker la table de simulation. Avec un tel cache, le taux de *miss* est de seulement 1%.

Cette approche est intéressante car elle converge vers la DBT. Elle garde en revanche les côtés négatifs de la simulation compilée, tout en en perdant les bienfaits. En effet, dans le cas de JIT-CCS, le compilateur n'est plus en mesure d'optimiser le flot de contrôle du programme généré puisqu'il n'est pas connu à la compilation. Pour chaque instruction, JIT-CCS devra donc faire plusieurs appels de fonctions pour simuler le pipeline. C'est cette granularité de l'instruction qui rend la simulation très lourde. Quitte à intégrer un processus de décodage et de génération dynamique, pourquoi ne pas travailler directement à générer des instructions pour la machine hôte ? Cela permet à la fois de se passer des appels de fonctions pour chaque instruction, et offre la possibilité d'obtenir une granularité au niveau bloc de base par exemple, ce qui donne des opportunités d'optimisations bien plus intéressantes.

Reshadi et al.[RBMD03] proposent un *framework* se basant sur un langage de description appelé EXPRESSION[HGG+99] pour décrire l'architecture cible. Ils s'appuient sur la technique de simulation compilée IS-CS[RMD03], similaire à JIT-CCS. Comme ce dernier, le programme final embarque un décodeur d'instructions pour supporter le code dynamique, mais traduit aussi tout le code statique en amont. Cela permet de ne pas générer de surcoût

lors de l'exécution, tout en supportant le cas du code auto-modifiant ou chargé dynamiquement.

3.3.2 UQBT : traduction binaire statique reciblable

Cifuentes et al.[CVER99] proposent UQBT, un *framework* de SBT reciblable. Son flot de traduction est complexe puisqu'il prend en entrée l'exécutable de la machine cible (notée M_S) et produit un exécutable pour la machine hôte (notée M_T). Les différentes étapes de ce flot sont présentées figure 3.3.

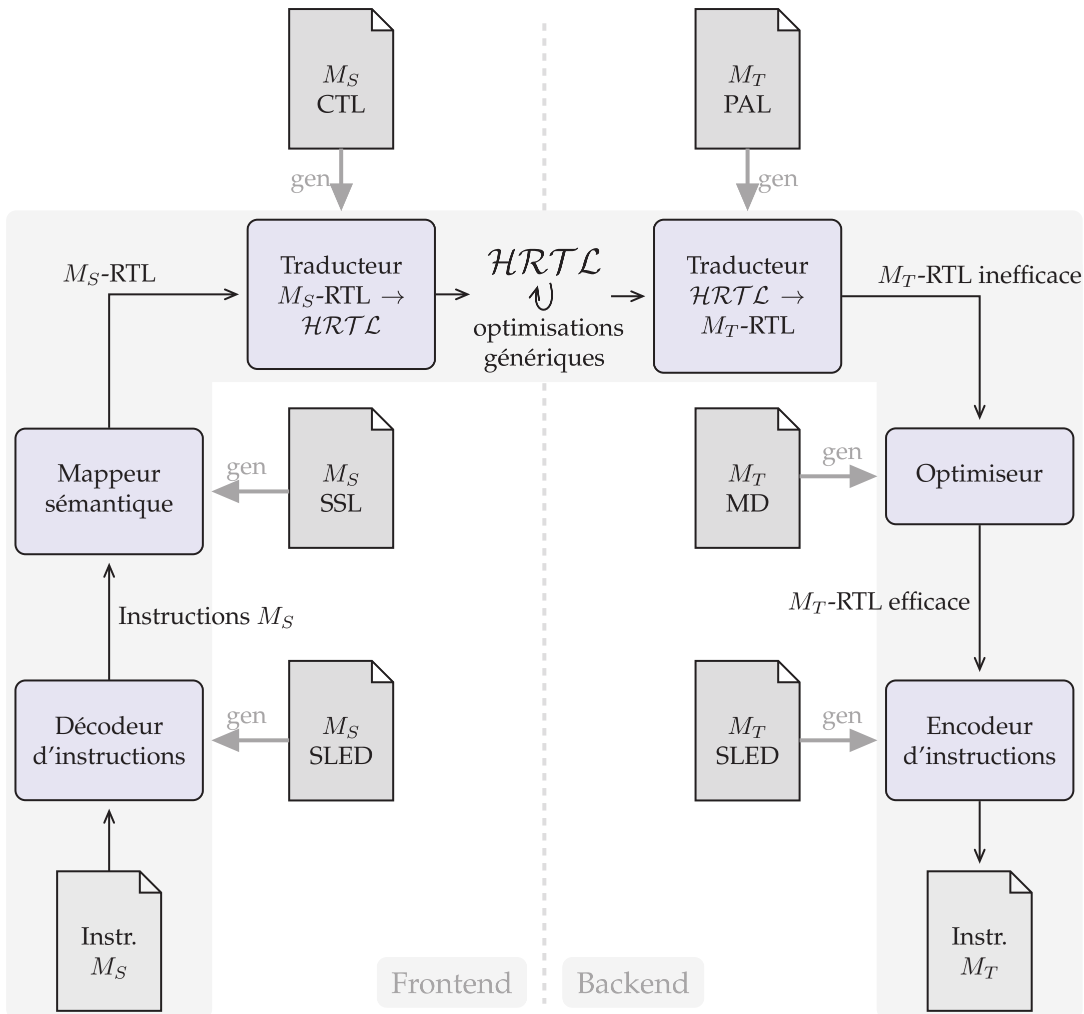


FIGURE 3.3 – Flot de génération d'UQBT

Pour décrire le comportement des instructions, UQBT utilise plusieurs niveaux de représentations intermédiaires, basés sur le langage RTL. Ce langage décrit une instruction en listant de manière séquentielle ses différents *effets*. Un effet assigne une expression à un emplacement (typiquement un registre ou la mémoire).

Le flot d'UQBT traduit les instructions de la machine M_S en M_S-RTL, le langage RTL dédié pour celle-ci. Cette représentation contenant encore les aspects propres à la machine

M_S, une deuxième phase traduit cette représentation en $\mathcal{HRTL}$. Ce langage RTL de haut niveau est indépendant de M_S-RTL et M_T-RTL. Cette phase élimine les aspects dépendants de M_S comme des branchements avec *delay slots*. Quelques optimisations génériques sont effectuées sur cette représentation de haut niveau, puis elle est à son tour traduite en M_T-RTL, représentation adaptée pour la machine M_T. Un optimiseur propre à M_T s'occupe d'améliorer le flot de M_T-RTL, puis il est finalement traduit en instructions de la machine M_T, et enfin assemblé pour former l'exécutable final.

Chacune des étapes du flot est générée en totalité ou en partie depuis un langage de description dédié :

– Pour générer le décodeur et l'encodeur d'instructions, le flot d'UQBT utilise le langage SLED[RF97] (Specification Language for Encoding and Decoding) qui décrit le format binaire de chaque instruction.
– Pour générer le mappeur sémantique qui traduit les instructions assembleurs de M_S en M_S-RTL, c'est le langage SSL[CS98] (Semantic Specification Language) qui est utilisé. Celui-ci associe chaque instruction de M_S à sa représentation M_S-RTL.
– Le langage CTL permet de générer le traducteur M_S-RTL vers $\mathcal{HRTL}$. Il traduit les comportements RTL spécifiques de M_S comme les branchements avec *delay slots* en $\mathcal{HRTL}$.
– Le langage PAL décrit l'Application Binary Interface (ABI) de la machine M_T, permettant par exemple de respecter les conventions d'utilisation des registres et d'appels de fonctions.

En pratique, UQBT ne génère pas les deux dernières étapes de son flot, mais utilise un outil existant (VPO[BD88]) pour optimiser le M_T-RTL et émettre les instructions assembleurs, et les outils standard de la machine M_T pour assembler le programme.

Performances et conclusion

Pour leurs tests, les auteurs génèrent un traducteur pour les architectures SPARC et x86, dans les 4 combinaisons possibles (SPARC $\rightarrow$ SPARC, SPARC $\rightarrow$ x86, x86 $\rightarrow$ SPARC et x86 $\rightarrow$ x86). Les traducteurs ayant les mêmes cible et hôte permettent d'évaluer la qualité du code produit comparé au code natif. De ce point de vue, le code est de bonne qualité puisqu'il n'y a pas de dégradation de performance notable entre le code natif et le code traduit. Selon les auteurs, c'est grâce à VPO qui est à même d'optimiser grandement le code traduit.

Aussi prometteur ce *framework* soit-il, il reste néanmoins d'importantes parties qui sont à ce jour écrites à la main, comme VPO. De plus, il souffre des problèmes classiques de la traduction binaire statique et doit embarquer un interpréteur si le flot d'exécution rencontre du code non traduit statiquement. Il n'est donc pas utilisable pour simuler un système complet, mais seulement des fichiers binaires déjà compilés pour un système d'exploitation donné (Solaris dans ce cas). Finalement, le grand nombre de langages de description et d'outils externes utilisés dans UQBT peuvent rendre le ciblage vers de nouvelles architectures fastidieux.

3.3.3 UQDBT et Walkabout : vers la traduction binaire dynamique

UQDBT[UC00, CVERL02] se base sur UQBT en adaptant son flot pour la DBT. Il utilise la granularité du bloc de base lors de la traduction, et stocke le résultat de la traduction dans un cache. Comme dans UQBT, les langages SLED et SSL sont utilisés pour générer certaines parties du flot (les auteurs rapportent que la génération du *backend* n'est pas implémentée). En revanche, les auteurs ont dû faire plusieurs choix de simplification du flot. En effet, les

contraintes de vitesse de traduction entre la SBT et la DBT ne sont pas les mêmes. Dans le cas de la DBT, les traductions étant faites à la demande au fur et à mesure de l'exécution, elles ne doivent pas prendre un temps trop important sous peine d'impacter les performances. Parmi les principales simplifications du flot, on trouve :

– L'abandon du langage de représentation intermédiaire $\mathcal{HRTL}$ au profit du I-RTL. Ce dernier est similaire mais n'inclut pas le processus complexe d'inférence des paramètres des fonctions traduites, chose que faisait $\mathcal{HRTL}$. À la place, les auteurs ont écrit à la main une partie du flot pour supporter l'ABI de M_S et M_T.
– L'abandon de la phase d'optimisation du M_T-RTL avec VPO. Ce dernier est avant tout destiné à être utilisé dans des compilateurs. Il est donc trop lourd pour être intégré dans un flot de DBT.

Les performances du code généré s'en trouvent ainsi impactées. Pour la traduction de SPARC vers SPARC, les auteurs rapportent un facteur de ralentissement de 2 à 10 fois par rapport à la version native de chaque benchmark. Les benchmarks sont les mêmes que ceux utilisés pour tester UQBT.

Afin d'améliorer les performances du code généré, les auteurs proposent de détecter et optimiser les *hotpaths*[UC01, UC06]. Un *hotpath* est défini comme une suite de blocs de base dépassant un certain seuil en nombre d'exécutions. Les *hotpaths* sont retraduits et placés dans un cache spécial (le *hot cache*). Les optimisations réalisées consistent principalement à améliorer la localité du code pour profiter des effets de cache de l'hôte. Les blocs de base sur un même chemin sont déplacés, fusionnés ou dupliqués pour optimiser le flot d'exécution.

Une autre optimisation améliore la gestion de l'endianness. Si la cible et l'hôte diffèrent de par leur endianness, une conversion est nécessaire à chaque simulation de chargement et d'écriture mémoire. Les auteurs proposent quelques techniques pour supprimer cette conversion lorsqu'elle n'est pas nécessaire (par exemple lors d'une copie mémoire).

Grâce à ces optimisations, les auteurs gagnent environ 15% de performance en temps d'exécution par rapport à la version non-optimisée. On reste donc loin des performances d'UQBT, ce qui montre la différence entre un flot SBT dans lequel des optimisations importantes sont permises, et un flot de DBT ou le temps de traduction doit rester négligeable pour ne pas impacter globalement le temps d'exécution.

Walkabout[CLU02] est un projet similaire dans ces objectifs puisqu'il promet un flot de DBT que l'on peut recibler à l'aide de descriptions de la cible et de l'hôte, avec optimisation des *hot paths*. Cependant la version présentée dans [CLU02] n'inclut pas la génération d'un flot de DBT mais seulement celle d'un interpréteur accompagnée d'un détecteur de *hotspot* et d'un optimiseur binaire du code cible.

3.3.4 Bintrans : DBT optimisée pour le couple cible/hôte

Bintrans[Pro01] propose un générateur de traducteur binaire dynamique en partant à la fois de la description de la machine cible et de la machine hôte. La description d'une machine est faite à l'aide d'un unique langage propre à Bintrans, issu de Lisp. Elle contient la descriptions des registres de la machine, ainsi que la description des formats et du comportement des instructions.

Description des comportements d'instruction

Les comportements sont décrits sous forme d'effets, chaque effet modifiant la mémoire ou un registre. Le listing 3.2 donne un exemple de description de l'instruction `addu` de l'architecture MIPS.

```
(set (reg rd gpr)
  (+
    (reg rs gpr)
    (reg rt gpr)))
```

Listing 3.2 – Description de l'instruction MIPS `addu` dans le langage de Bintrans

D'un point de vue de l'hôte, les effets sont classés en deux catégories, les effets primaires, et les autres. Les effets primaires modifient la mémoire ou un registre à usage général (General Purpose Register (GPR)) contrairement aux autres qui modifient un registre spécial comme le registre d'état sur l'architecture x86.

Les effets peuvent contenir des structures conditionnelles. Prenons l'exemple donné au listing 3.3.

```
(if (= pc 31)
  (nop)
  (set (reg rc gpr)
    (logor
      (if (= ra 31)
        0 (reg ra gpr))
      (if (= rb 31)
        0 (reg rb gpr)))))
```

Listing 3.3 – Description de l'instruction Alpha `bis`

Il s'agit de l'instruction `bis` de l'architecture Alpha. Elle stocke dans `rc` le résultat de `ra | rb` seulement si `rc` n'est pas le registre 31. De la même manière, elle lit `ra` et `rb` seulement si ils ne sont pas le registre 31. Sinon, 0 est lu.

Bintrans supprime les structures conditionnelles des effets en décomposant ces derniers. Il crée autant de sous-effets qu'il y a d'issues possibles aux différentes conditions. Pour l'instruction `bis` précédente, il crée 5 effets qui ne possèdent plus de structures conditionnelles :

1. `(nop)                                                  ; rc = 31`
2. `(set (reg rc gpr) (logor 0 0))                         ; ra = 31, rb = 31`
3. `(set (reg rc gpr) (logor (reg ra gpr) 0))              ; rb = 31`
4. `(set (reg rc gpr) (logor 0 (reg rb gpr)))              ; ra = 31`
5. `(set (reg rc gpr) (logor (reg ra gpr) (reg rb gpr)))   ; autres cas`

Génération du simulateur

Grâce à la description des machines cible et hôte, Bintrans peut générer le traducteur binaire dynamique propre au couple. Il lit les formats d'instruction de la machine cible pour générer le décodeur d'instructions correspondant.

Pour générer le cœur de traduction binaire, Bintrans réalise des correspondances entre les comportements des instructions pour trouver les instructions hôtes pouvant simuler les

instructions cibles. Pour cela, chaque effet de chaque instruction cible est mis en correspondance avec chaque effet de chaque instruction hôte. Si une instruction hôte possède plusieurs effets, seul l'effet primaire est pris en compte. Si elle a plusieurs effets primaires, alors elle est ignorée dans le processus de mise en correspondance.

Lors de la correspondance, l'auteur distingue trois cas :

Correspondance d'une expression cible avec un registre hôte Si une expression est mise en correspondance avec un registre ou sous-registre de l'hôte, alors la valeur de cette expression doit être placée dans ce registre. Pour cela, un effet temporaire est créé.

Correspondance d'une constante de traduction avec un opérande hôte Certaines valeurs sont connues lors de la phase de traduction. C'est le cas des valeurs immédiates des instructions cibles. Elles peuvent donc être considérées comme des constantes. En revanche, ces valeurs ne sont pas connues au moment de la génération du traducteur. La correspondance doit donc être reportée au moment de la traduction. Cette correspondance consiste à déterminer si la constante de traduction peut être représentée par l'opérande de l'instruction hôte. Par exemple si l'opérande est de taille 8 bits et subit une extension de zéro, alors la constante cible doit être comprise dans l'intervalle $[0; 2^8 - 1]$. Sinon, la correspondance est refusée.

Les autres cas Pour tous les cas ne répondant pas au deux critères précédents, la correspondance n'a lieu que si les deux expressions ainsi que leurs opérandes sont les mêmes.

En cas de multiples correspondances pour une même instruction cible, un classement en fonction du nombre d'instructions hôtes nécessaires est effectué. Les correspondances considérées comme les meilleurs sont celles qui génèrent le moins d'instructions hôtes. Les autres ne sont cependant pas éliminées car elle peuvent servir de solution de repli si la correspondance au moment de la traduction échoue.

L'auteur introduit la notion de *transformation*, qui sont des règles génériques qui peuvent être appliquées à des expressions pour les exprimer différemment, en conservant la même sémantique. L'exemple de transformation donné au listing 3.4 indique qu'une négation d'une sous-expression x peut être remplacée par une soustraction de 0 avec x.

```
1  (define-transformation
2    (neg (match x))
3    (- 0 x))
```

Listing 3.4 – Exemple d'une transformation

Ces transformations sont utilisées pendant le processus de correspondance pour augmenter les chances de trouver des instructions hôtes candidates pour traduire les instructions cibles.

Performances et conclusion

Pour ses tests, l'auteur génère deux traducteurs, PowerPC vers Alpha, et x86 vers Alpha. Il exécute deux programmes issus de la suite de benchmarks SPECINT95, `go` et `compress`, sur chacun des deux traducteurs.

Pour le traducteur PowerPC, le code généré est 3 à 5 fois plus lent que le code natif compilé pour l'Alpha. Le traducteur génère environ 6.5 instructions hôtes pour une instruction

cible décodée. Quant au traducteur x86, le ratio de vitesse est 2.5 à 3.5 fois plus lent que le code natif. Le ratio d'instructions générées est d'environ 7 instructions hôtes pour une instruction cible.

Cette approche semble prometteuse pour exploiter au mieux les capacités de l'hôte, à moindre coût de conception. L'auteur explique les mauvaises performances du simulateur PowerPC par le fait que les *flags* du registre de statut sont gérés de manière non-optimale. De plus, il semble renoncer à cette technique de génération automatique au profit de l'écriture manuelle du traducteur[Pro02], argumentant sur le fait que cette technique de correspondance fonctionne bien pour les instructions régulières, mais mal pour les instructions complexes comme les instructions conditionnelles ou modifiant un registre d'état.

3.4 Conclusion

La simulation rapide de processeurs est un domaine compliqué. Les familles de processeurs et d'ISA sont variées, et écrire des simulateurs efficaces est une tache ardue.

À ce jour, il ne semble pas exister de méthode pour simuler un processeur VLIW sur une architecture scalaire à l'aide de la DBT. Cette technique est pourtant très intéressante puisqu'elle s'intègre parfaitement dans une plateforme de simulation d'un système complet, contrairement aux techniques classiques comme la SBT ou la simulation compilée.

Quant à la génération de simulateurs rapides, les techniques explorées ont tendance à mettre la DBT de côté, principalement à causes de mauvaises performances du simulateur généré. Les pistes explorées par Bintrans[Pro01] pour optimiser la génération à un couple cible/hôte donné semblent pourtant prometteuses.

Chapitre 4

Simulation d'architectures VLIW à l'aide de la traduction binaire dynamique

COMME nous l'avons évoqué au chapitre 3, la DBT est une solution très intéressante dans le cadre de la simulation. Ce chapitre présente la première contribution de cette thèse, comment simuler un processeur VLIW sur une architecture scalaire à l'aide de la DBT.

L'algorithme d'un *frontend* classique de DBT tel que présenté à la figure 2.5 page 13 peut être décrit ainsi :

Algorithme 2 *Frontend* d'un traducteur binaire dynamique classique

```
1 tant que not end of translation block faire
2     opcode ← get_next_instruction()          ▷ Récupération
3     dec ← decode(opcode)                     ▷ Décodage
4     translate(dec)                           ▷ Traduction
5 fin tant que
```

Cet algorithme ne tient pas compte des spécificités d'une architecture VLIW puisque les notions de paquet d'exécution et de latence d'instructions n'apparaissent pas. La figure 4.1 donne la traduction d'un code `c6x` en représentation intermédiaire QEMU avec un tel algorithme.

Dans ce listing, chaque instruction `c6x` est mise en correspondance avec sa traduction en micro-opérations de l'IR QEMU. Pour faciliter la lecture, les listings d'IR QEMU de ce document sont simplifiés par rapport à la réalité. Dans son fonctionnement normal, QEMU ne travaille pas sur les registres de l'architecture cible directement. Il passe par des valeurs intermédiaires appelées *temporaries*. Par exemple, la traduction complète de la première instruction `c6x` est illustrée par la figure 4.2

La notation simplifiée consiste à travailler directement sur les registres de l'architecture cible, cela permettant de supprimer les micro-opérations de mouvement dans les *temporaries*.

La figure 4.1 nous montre que l'algorithme classique de DBT ne fonctionne pas dans le cadre de la traduction d'un ISA VLIW. Trois problèmes de natures différentes apparaissent ici :

– Tout d'abord, les instructions 1, 2 et 3 font partie du même paquet d'exécution, elles

EP		L	Code c6x	Micro-opérations QEMU
1	1	0	`add .L1 a2, a1, a2`	`add_i32 a2, a2, a1`
	2	0	`|| sub .D1 a1, a4, a6`	`sub_i32 a6, a1, a4`
	3	3	`|| mpy32 .M1 a2, a1, a3`	`mul_i32 a3, a2, a1`
2	4	5	`b .S1 0xbeef`	`mov_i32 pce1, $0xbeef`
	5			`exit_tb 0x0`
3	6	0	`sub .D1 a6, a3, a3`	`; sub_i32 a3, a6, a3`
4	7	0	`add .L1 a2, a3, a4`	`; add_i32 a4, a2, a3`
5	8	3	`mpy32 .M1 a2, a3, a5`	`; mul_i32 a5, a2, a3`
6	9		`nop 2`	`; nop 2`
7	10	0	`add .L1 a2, a3, a5`	`; add_i32 a5, a2, a3`

FIGURE 4.1 – Traduction naïve de code c6x avec l'algorithme de DBT classique

EP		L	Code c6x	Micro-opérations QEMU
1	1	0	`add .L1 a2, a1, a2`	`mov_i32 tmp0, a1`
	2			`mov_i32 tmp1, a2`
	3			`add_i32 tmp1, tmp1, tmp0`
	4			`mov_i32 a2, tmp1`

FIGURE 4.2 – Traduction complète d'une instruction cible en micro-opérations

sont vouées à être exécutées en parallèle sur le processeur cible. En revanche, le traducteur produit un code séquentiel ce qui pose des problèmes de violation de dépendances de type *écriture après lecture* (Write after Read (WAR)). En effet, la première instruction écrit son résultat dans le registre a2, registre qui est lu par la troisième instruction. Or, l'instruction 3 s'attend à lire la valeur de a2 avant que celle-ci soit écrasée par le résultat de l'instruction 1 ce qui se traduit par la violation d'une dépendance WAR sur la figure 4.1.

– De plus, les latences d'instructions ne sont pas prises en compte. L'instruction 3 a une latence de 3 cycles. Son résultat dans a3 est normalement visible à partir de l'instruction 7 mais le code généré ne tient pas compte de cette latence. Il écrase directement le contenu de a3.
– Finalement, l'instruction 4 de branchement n'est pas correctement gérée. Le TB se termine immédiatement et ne respecte pas la latence de 5 cycles de l'instruction b. Les instructions des lignes 6 à 10 ne sont donc pas traduites. Les micro-opérations correspondantes sont données sur la figure en commentaire à titre indicatif.

Cet algorithme a donc besoin d'être adapté pour prendre en compte les spécificités des architectures VLIW.

4.1 Un algorithme pour simuler une architecture VLIW

Nous proposons un algorithme à même de prendre en compte les spécificités des architectures VLIW dans le cadre de la simulation. Dans un premier temps, cet algorithme est suffisamment général pour être utilisé dans n'importe quel type de simulateur et pas seulement dans le cadre de la DBT. Il repose sur l'idée suivante : plutôt que d'utiliser un seul emplacement mémoire pour stocker la valeur d'un registre, un nouvel emplacement est créé à chaque écriture du registre. On se rapproche ainsi de la forme Static Single Assignment (SSA)[CFR+91] utilisée en compilation. Nous appellerons ces emplacements les *répliques* du registre.

À tout moment, le simulateur a connaissance de la réplique à utiliser lors de la lecture d'un registre. C'est sa *réplique courante*. Il sait aussi à quel moment il doit changer de réplique courante pour un registre. Ce principe est illustré par l'algorithme 3. Il traite les paquets d'exécutions indépendamment les uns des autres. Pour chaque instruction dans le paquet d'exécution courant, il récupère sa latence et son opérande de sortie, et crée une nouvelle réplique de ce dernier. Il indique à la fonction de création la latence de l'instruction pour permettre la future mise à jour au moment voulu. Il simule ensuite l'instruction, en indiquant au simulateur d'utiliser cette réplique pour stocker le résultat. Lorsque toutes les instructions du paquet ont été traitées, il met à jour les indirections sur les registres de la machine simulée. C'est à ce moment que le simulateur est susceptible de changer de réplique courante pour un registre.

Algorithme 3 Utilisation de répliques pour simuler une architecture VLIW

```
1  tant que not end of the simulation faire
2      ep ← get_next_execute_packet()
3      pour all insn ∈ ep faire
4          d ← get_delay(insn)                          ▷ Récupération de la latence de l'instruction
5          output_op ← get_output_operand(insn)         ▷ Récupération de l'opérande de sortie
6          r ← new_replicate(output_op, d)              ▷ Création de la réplique
7          simulate(insn, r)                            ▷ Simulation de l'instruction en utilisant la réplique
8      fin pour
9      do_end_of_cycle_updates()                        ▷ Mise à jour des indirections
10 fin tant que
```

4.1.1 Exemple

Pour illustrer le fonctionnement de l'algorithme 3, prenons les quelques instructions `c6x` de la figure 4.3. Les répliques en attente sont schématisées par une file d'attente qui progresse d'un cycle à chaque appel de `do_end_of_cycle_updates()`.

État initial

L'état initial du simulateur est illustré par la figure 4.4. Chaque registre `ax` possède sa réplique initiale ax_0, utilisée en cas de lecture. La file d'attente est vide puisqu'aucune instruction n'a encore créé de nouvelle réplique.

EP		L	Code c6x
1	1	0	`add .L1 a0, a1, a2 ; a2 <- a0 + a1`
	2	0	`|| sub .D1 a3, a1, a0 ; a0 <- a3 - a1`
	3	3	`|| mpy32 .M1 a2, a1, a2 ; a2 <- a2 * a1`
2	4	0	`add .L1 a2, a2, a3 ; a3 <- a2 + a2`
3	5		`nop 2 ; no-op pendant 2 cycles`
4	6	0	`add .L1 a2, a3, a3 ; a3 <- a2 + a3`

FIGURE 4.3 – Exemple de code c6x

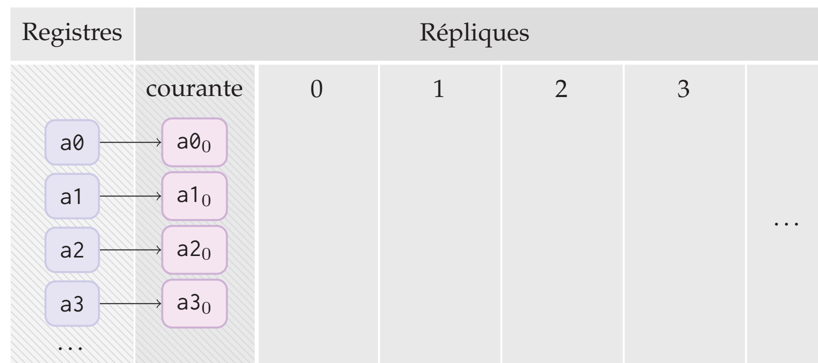


FIGURE 4.4 – État initial

Après le premier paquet d’exécution

Après l’exécution du premier paquet (les instructions des lignes 1 à 3 sur la figure 4.3), trois répliques ont été créées, une pour `a0` et deux pour `a2`. L’état correspondant est donné par la figure 4.5. On voit ici que le même registre peut être la destination de deux instructions dans le même paquet, pourvu que ces instructions n’aient pas la même latence. Les répliques $\texttt{a0}_1$ et $\texttt{a2}_1$ sont placées dans la case 0 de la file d’attente car les instructions des lignes 1 et 2 ont une latence de 0. La multiplication quant à elle, a une latence de 3. On voit ici l’intérêt de passer l’information de latence en paramètre à la fonction `new_replicate()`. À ce moment précis, la fonction `do_end_of_cycle_updates()` de l’algorithme n’a pas encore été appelée.

Après la mise à jour des indirections

Après l’appel à la fonction `do_end_of_cycle_updates()`, la file d’attente progresse d’un cycle. Les répliques $\texttt{a0}_1$ et $\texttt{a2}_1$ deviennent les répliques courantes pour leur registre respectif. Les anciennes répliques sont supprimées car devenues obsolètes. La réplique $\texttt{a2}_2$ contenant le résultat de la multiplication a avancé d’un cycle dans la file d’attente. L’état correspondant est donné par la figure 4.6.

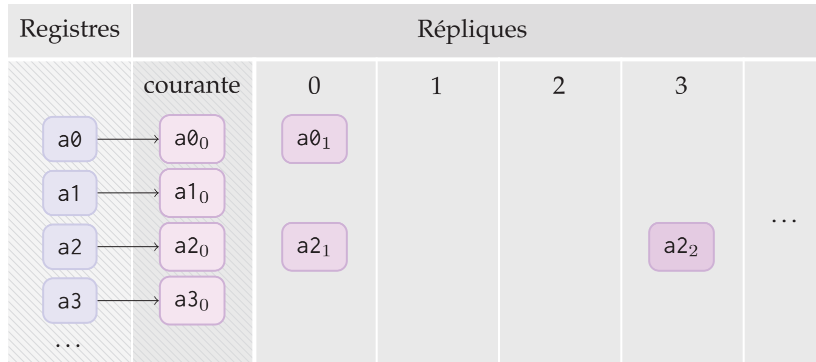


FIGURE 4.5 – État après la simulation du premier paquet d'exécution

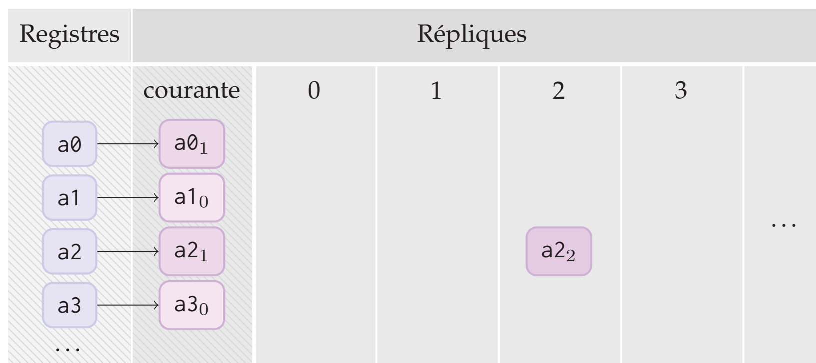


FIGURE 4.6 – État après la mise à jour des indirections

Après les paquets d'exécution 2 et 3

Après la simulation des paquets 2 et 3, et la mise à jour des indirections, le résultat de la multiplication est disponible dans a2. En effet, trois cycles se sont écoulés. L'instruction add à la ligne 6 lira donc ce résultat et non l'ancienne valeur, ce qui se traduit sur la figure 4.7 par l'utilisation de la réplique $a2_2$ pour le registre a2.

4.1.2 Conclusion sur l'algorithme

Cet algorithme permet à un simulateur de respecter facilement les contraintes d'un processeur VLIW, même s'il simule les instructions de manière séquentielle. Il s'inspire de SSA puisqu'une même réplique ne peut subir qu'une seule écriture. En revanche, plusieurs répliques d'un même registre peuvent exister en même temps, ce qui n'est pas possible avec SSA. La preuve du bon fonctionnement de l'algorithme par rapport au comportement du matériel est donnée par l'annexe A. Celle-ci montre qu'un simulateur utilisant cet algorithme produit le même comportement que le matériel réel.

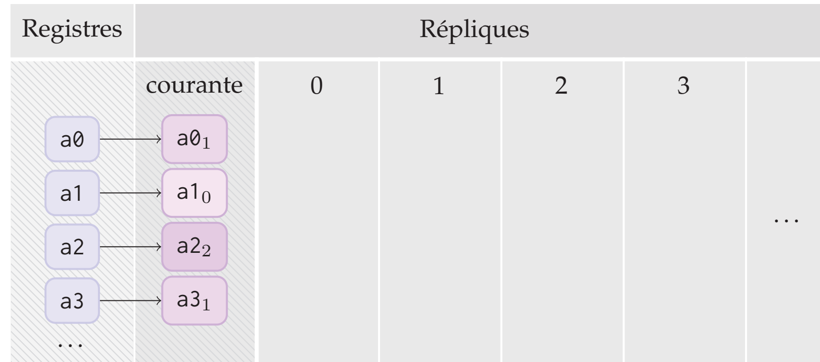


FIGURE 4.7 – État après la simulation du `nop 2` et la mise à jour des indirections

4.2 Intégration de l'algorithme dans un traducteur binaire dynamique

Utiliser cet algorithme dans un *frontend* de DBT demande à prendre en compte les spécificités de cette technique de simulation. Dans le cadre de ces travaux, nous avons réalisé un *frontend* `c6x` dans le logiciel QEMU. Dans cette section, nous nous appuierons sur cet exemple pour décrire les problèmes clés d'une telle implémentation.

4.2.1 Adaptation de l'algorithme

La version adaptée pour la DBT de l'algorithme 3 est donné par l'algorithme 4.

Algorithme 4 Version adaptée pour un frontend DBT

```
1  tant que not end of the translation block faire
2      ep ← get_next_execute_packet()
3      pour all insn ∈ ep faire
4          d ← get_delay(insn)                          ▷ Récupération de la latence de l'instruction
5          output_op ← get_output_operand(insn)         ▷ Récupération de l'opérande de sortie
6          r ← new_replicate(output_op, d)              ▷ Création de la réplique
7          translate(insn, r)                           ▷ Traduction de l'instruction en utilisant la réplique
8      fin pour
9      do_end_of_cycle_updates()                        ▷ Mise à jour des indirections
10 fin tant que
```

La différence majeure par rapport aux simulateurs classiques vient du fait que cet algorithme est exécuté au moment de la traduction dans le *frontend* du simulateur. Lors de cette phase, le contexte d'exécution, qui contient par exemple l'état du processeur et des registres, n'est pas connu. En effet, un TB est généré en dehors de tout contexte. Il est indépendant des autres TB et ne fait aucune hypothèse quant à ses prédécesseurs. C'est une propriété clé de la DBT pour pouvoir réutiliser les TB déjà traduits.

4.2.2 L'allocation des répliques en pratique

Lorsque QEMU simule un processeur, il maintient son état dans une structure appelée *l'environnement*. Le contenu de cette structure est défini par le concepteur du *frontend*. Elle contient entre autres les registres du processeur simulé. Cette structure étant fixe lors de l'exécution du simulateur, l'algorithme 4 semble inadapté puisqu'il crée sans cesse de nouvelles répliques. Cependant, ces répliques ont une durée de vie courte, qui n'excède jamais quelques cycles. De plus, ce nombre de cycles est facile à borner puisqu'il suffit de trouver dans le jeu d'instruction simulé l'instruction ayant la latence p la plus grande. Dans le cas du `c6x`, il s'agit de l'instruction de branchement, ayant une latence de $p = 5$ cycles. Pour un registre donné, le nombre de répliques vivantes à un instant t est lui aussi facile à borner. Dans le pire des cas, le paquet d'exécution courant contiendra une instruction ayant une latence de p, une de $p-1$, $p-2$, etc. jusqu'à 0, toutes écrivant le même registre. Dans ce cas, il faudra en plus de la réplique courante, p répliques du même registre, soit $p+1$ répliques. Une autre manière d'atteindre ce pire-cas est d'enchaîner les paquets d'exécutions contenant une instruction écrivant le même registre avec la latence p. Au bout de p paquets, $p+1$ répliques seront en cours d'utilisation.

Notre nombre de répliques étant borné, nous pouvons utiliser un tampon circulaire pour les stocker dans *l'environnement*. La taille de ce tampon est $n \times (p+1)$, n étant le nombre de registres de la machine cible. Lors de la traduction, le simulateur utilise les répliques de ce tampon comme opérandes pour les micro-opérations. La figure 4.8 donne un exemple de traduction correcte.

EP		L	Code c6x	Micro-opérations QEMU
1	1	0	`add .L1 a2, a1, a2`	`add_i32 a2_1, a2_0, a1_0`
	2	0	`|| sub .D1 a1, a4, a6`	`sub_i32 a6_1, a1_0, a4_0`
	3	3	`|| mpy32 .M1 a2, a1, a3`	`mul_i32 a3_1, a2_0, a1_0`
2	4	5	`b .S1 0xbeef`	`mov_i32 pce1_1, $0xbeef`
3	5	0	`sub .D1 a6, a3, a3`	`sub_i32 a3_2, a6_1, a3_0`
4	6	0	`add .L1 a2, a3, a4`	`add_i32 a4_1, a2_1, a3_2`
5	7	3	`mpy32 .M1 a2, a3, a5`	`mul_i32 a5_1, a2_1, a3_1`
6	8		`nop 2`	`; nop 2`
	9			`exit_tb 0x0`
7	10	0	`add .L1 a2, a3, a5`	`; add_i32 a5_2, a2_1, a3_1`

FIGURE 4.8 – Traduction correcte de code `c6x`

On peut y voir l'utilisation des répliques $\mathtt{ax}_y$ (notées `ax_y` sur la figure), avec une allocation pour chaque écriture et une gestion correcte des latences. Par exemple, l'instruction `mpy32` à la ligne 3 ayant une latence de 3, sa réplique de destination `a3_1` n'est pas utilisée comme réplique courante avant la micro-opération de la ligne 7. De plus, le branchement de la ligne 4 est correctement traduit puisque le calcul de la destination du saut est effectué à la ligne 4 au moment de l'instruction de branchement mais le TB se termine bien à la ligne

9, lorsque les 5 cycles de latence se sont écoulés. L'instruction à la ligne 10 n'est donc pas traduite.

4.2.3 Les problèmes introduits par la granularité du TB

Dans le contexte d'un TB, cet algorithme semble à première vue adapté. Il génère simplement du code ayant un comportement correct d'un point de vue du processeur simulé. Cependant, la granularité de traduction au niveau du TB imposée par la DBT amène de nouveaux problèmes qu'il nous faut régler pour conserver une simulation fonctionnellement correcte.

L'état canonique entre les blocs de traduction

Comme décrit à la section 4.2.2, l'allocation de répliques est réalisée statiquement au moment de la traduction. Par conséquent, la correspondance registre $\rightarrow$ réplique ne fait pas partie de l'état du processeur simulé et n'est donc pas connue au moment de l'exécution du code généré. De plus, lorsque le traducteur prend la main pour traduire un TB, il le fait indépendamment de tout contexte puisque les TB doivent être indépendants les uns des autres. Sur l'exemple de la figure 4.8, lorsque le TB se termine, il laisse les valeurs des registres `a2` à `a6` dans leur réplique n°1 respective. Or, le TB suivant n'a pas de moyen de récupérer cette information. Il est donc nécessaire d'introduire un état connu entre chaque TB.

C'est pourquoi nous introduisons la notion *d'état canonique*. Cet état correspond pour chaque registre `r` de la machine cible, à l'utilisation de la réplique $\texttt{r}_0$. Pour respecter cet état canonique, le traducteur doit, avant de terminer un TB, s'assurer que la réplique courante de chaque registre est la réplique 0. Si ce n'est pas le cas, il doit générer une copie de la réplique courante vers la réplique 0. La figure 4.9 reprend l'exemple précédent en ajoutant dans l'épilogue du TB le retour à l'état canonique.

Les lignes 10 à 14 effectuent le retour à l'état canonique pour les registres `a2`, `a3`, `a4` et `a6`. De cette manière, l'indépendance des TB est assurée. Ils commencent et finissent tous dans le même état quant à l'utilisation des répliques.

Les sauvegardes de contexte nécessaires durant l'exécution

Dans l'exemple de la figure 4.9, l'instruction `mpy32` de la ligne 7 écrit son résultat dans le registre `a5`. Une nouvelle réplique `a5_1` est donc allouée pour ce résultat. Or, dans le retour à l'état canonique, aucun mouvement ne concerne le registre `a5`. C'est le comportement voulu puisque le *delay slot* de l'instruction `mpy32` n'est pas terminé. Si `a5_1` était copié dans `a5_0`, le TB suivant verrait le résultat du `mpy32` un cycle trop tôt.

Cependant, les informations du tampon circulaire étant connues seulement à la traduction du TB courant, dans la situation de l'exemple précédent, la réplique `a5_1` sera considérée comme non-allouée lors de la traduction du TB suivant et le résultat du `mpy32` perdu. Ce problème survient lorsqu'il reste des répliques dans la file d'attente à la fin du TB. Pour résoudre ce problème, et dans un souci de conserver l'indépendance inter-TB, il est nécessaire d'introduire un mécanisme à l'exécution permettant la sauvegarde et la restauration des informations de la file d'attente *entre* les TB. Ce mécanisme doit modifier l'état du processeur simulé puisque l'information doit se propager entre les TB au moment de l'exécution. Nous appellerons ce mécanisme la *file d'attente dynamique* puisqu'elle prend place au moment de

EP		L	Code c6x	Micro-opérations QEMU
1	1	0	`add .L1 a2, a1, a2`	`add_i32 a2_1, a2_0, a1_0`
	2	0	`|| sub .D1 a1, a4, a6`	`sub_i32 a6_1, a1_0, a4_0`
	3	3	`|| mpy32 .M1 a2, a1, a3`	`mul_i32 a3_1, a2_0, a1_0`
2	4	5	`b .S1 0xbeef`	`mov_i32 pce1_1, $0xbeef`
3	5	0	`sub .D1 a6, a3, a3`	`sub_i32 a3_2, a6_1, a3_0`
4	6	0	`add .L1 a2, a3, a4`	`add_i32 a4_1, a2_1, a3_2`
5	7	3	`mpy32 .M1 a2, a3, a5`	`mul_i32 a5_1, a2_1, a3_1`
6	8		`nop 2`	`; nop 2`
7	9		`; Épilogue TB`	`; Retour état canonique`
	10			`mov_i32 a2_0, a2_1`
	11			`mov_i32 a3_0, a3_1`
	12			`mov_i32 a4_0, a4_1`
	13			`mov_i32 a6_0, a6_1`
	14			`mov_i32 pce1_0, pce1_1`
	15			`exit_tb 0x0`

FIGURE 4.9 – Retour dans l'état canonique

l'exécution, contrairement à la version *statique* présentée précédemment qui est utilisée au moment de la traduction.

QEMU propose un mécanisme de *helpers* pour réaliser des opérations complexes dans le code généré. Un *helper* est une fonction C qui peut être appelée depuis le code généré, donc à l'exécution. Nous choisissons d'implémenter ce mécanisme de file d'attente dynamique à l'aide de *helpers*. Le prototype du *helper* de sauvegarde de contexte est donné par le listing 4.1

```
1 void HELPER(save_context)(uint32_t reg, uint32_t val,
2                                 uint32_t delay);
```

Listing 4.1 – *Helper* de sauvegarde de contexte

Il prend en paramètre le numéro du registre concerné par la modification, la valeur à lui appliquer, et le nombre de cycles avant de l'appliquer. Il sauvegarde ces informations dans la file d'attente dynamique, dans l'*environnement*. Cependant, cette file d'attente dynamique se différencie de la version statique utilisée pendant la traduction par le fait qu'elle ne stocke pas l'information d'une réplique à utiliser pour un registre au bout d'un certain nombre de cycles. Au lieu de cela, elle sauvegarde la *valeur* à appliquer au registre au bout d'un nombre donné de cycles. En effet, ce processus prenant place à l'exécution, la correspondance registre $\rightarrow$ réplique n'existe pas. En revanche, les résultats des instructions sont

connus.

La figure 4.10 complète l'exemple précédent avec la sauvegarde de contexte à la ligne 10. Le traducteur, constatant que le résultat de l'instruction `mpy32` la ligne 7 est toujours en

EP		L	Code c6x	Micro-opérations QEMU
1	1	0	`add .L1 a2, a1, a2`	`add_i32 a2_1, a2_0, a1_0`
	2	0	`|| sub .D1 a1, a4, a6`	`sub_i32 a6_1, a1_0, a4_0`
	3	3	`|| mpy32 .M1 a2, a1, a3`	`mul_i32 a3_1, a2_0, a1_0`
2	4	5	`b .S1 0xbeef`	`mov_i32 pce1_1, $0xbeef`
3	5	0	`sub .D1 a6, a3, a3`	`sub_i32 a3_2, a6_1, a3_0`
4	6	0	`add .L1 a2, a3, a4`	`add_i32 a4_1, a2_1, a3_2`
5	7	3	`mpy32 .M1 a2, a3, a5`	`mul_i32 a5_1, a2_1, a3_1`
6	8		`nop 2`	`; nop 2`
7	9		`; Épilogue TB`	`; Sauvegarde contexte`
	10			`call save_context, 5, a5_1, 0`
	11			`; Retour état canonique`
	12			`mov_i32 a2_0, a2_1`
	13			`mov_i32 a3_0, a3_1`
	14			`mov_i32 a4_0, a4_1`
	15			`mov_i32 a6_0, a6_1`
	16			`mov_i32 pce1_0, pce1_1`
	17			`exit_tb 0x0`

FIGURE 4.10 – Sauvegarde de contexte

file d'attente statique à la fin du TB, génère une sauvegarde de contexte. Cette sauvegarde concerne le registre numéro 5 dans l'état du processeur (`a5`), la valeur à y inscrire est contenue dans la réplique `a5_1`, et le nombre de cycle restant est 0 (soit un paquet d'exécution).

La restauration de contexte

L'opération antagonique à la sauvegarde de contexte à la fin d'un TB est la restauration de contexte au début du suivant. Là aussi c'est un *helper* qui l'effectue. Cependant, la restauration de contexte apporte de nouvelles difficultés. Le *helper* de restauration, dont le prototype est donné par le listing 4.2, ne prend pas de paramètre. Il se base uniquement sur l'état de la file d'attente dynamique dans l'*environnement* pour connaitre les modifications à appliquer au cycle courant.

```
1 void HELPER(restore_context)(void);
```

Listing 4.2 – *Helper* de sauvegarde de contexte

L'appel à ce *helper* doit donc être entrelacé avec l'exécution des paquets d'exécution. À la fin d'un paquet, le *helper* est appelé pour vérifier s'il y a une valeur en attente à appliquer à un registre. Le premier problème à résoudre est de connaître le nombre minimum de cycles pendant lequel on doit faire appel à ce *helper* en début de TB pour garantir d'appliquer toutes les valeurs en attente. La réponse à cette question est simple puisqu'il s'agit ici encore de la latence p la plus grande dans l'ISA simulé. En effet, à la fin d'un TB lors d'une sauvegarde de contexte, la valeur de latence la plus grande que l'on puisse introduire dans la file d'attente est $p - 1$. Il faudra donc p cycles (de 0 à $p - 1$) pour voir cette valeur appliquée à son registre de destination. Par conséquent, il faut réaliser p appels à ce *helper* au début de chaque TB, entrelacé avec les p premiers paquets d'exécution, chaque appel étant équivalent à l'avancée d'un cycle dans la file d'attente. [1]

Le deuxième problème à résoudre vient de la non connaissance de la correspondance registre $\rightarrow$ réplique au moment de l'exécution. Lorsque le *helper* applique une valeur à un registre, quelle réplique doit-il écraser ? Cette question n'est pas solvable puisqu'il ne sait pas quelle réplique est en cours d'utilisation pour un registre donné. Là encore ce problème est réglé grâce à l'état canonique. Nous choisissons d'étendre l'utilisation de l'état canonique aux p premiers cycles de chaque TB. Pour se faire, à la fin de chaque paquet, le traducteur remet la valeur courante des registres dans la réplique 0. Ainsi, le *helper* considère toujours que l'état courant est l'état canonique, et applique ses modifications dans la réplique 0 des registres concernés. La figure 4.11 donne le code résultant d'une telle stratégie. Aux lignes 6, 8, 11, 15 et 17, on peut voir les 5 appels au *helper* de restauration de contexte. Pendant ces 5 premiers cycles, on peut aussi constater la conservation de l'état canonique. Les lignes 4 et 5 placent les résultats des instructions des lignes 1 et 2 dans la réplique 0 de leurs destinations respectives. C'est aussi le cas pour l'instruction de la ligne 3, mais à la ligne 14 seulement puisqu'elle a une latence de trois cycles.

Le cas particulier des branchements

Grâce à ce mécanisme de sauvegarde et restauration de contexte, les latences inter-TB sont correctement gérées. Cependant, le cas des branchements reste problématique. En effet, d'un point de vue du simulateur, un branchement n'a pas pour seul effet de modifier un registre, il termine aussi le TB en cours. Cela pose problème dans le cas d'un branchement encore dans la file d'attente à la fin d'un TB. La destination du branchement est sauvegardée normalement par le *helper* `tb_context_save`. En revanche, lors de la restauration, la présence d'un branchement doit provoquer dynamiquement la fin du TB. QEMU permet d'arrêter l'exécution d'un TB déjà généré à l'aide d'une fonction dédiée (`cpu_loop_exit`) ce qui permet à première vue de résoudre le problème. De plus, le traducteur reste dans l'état canonique pendant les p premiers cycles. L'état des répliques devrait donc être cohérent pour le TB suivant.

Ici encore, le problème restant est dû aux valeurs en file d'attente. Prenons l'exemple de l'instruction `mpy32` à la ligne 3 de la figure 4.11. Si un branchement en attente est effectué lors d'une restauration de contexte prenant place après l'exécution de cette instruction, par exemple pendant la restauration à la ligne 6, alors le résultat du `mpy32` en attente dans `a3_1` sera perdu. Comme ce genre d'évènements n'est pas prévisible au moment de la traduction, il est nécessaire de générer une sauvegarde de contexte pour le résultat de cette instruction.

1. En réalité, notre implémentation réalise $p + 1$ appels à ce *helper* car QEMU est susceptible de stopper le processus de traduction en plein milieu d'un paquet d'exécution. Dans ce cas, le cycle se termine au début du TB suivant, avec un appel supplémentaire au *helper*.

Cette sauvegarde est particulière car elle intervient pendant les p premiers cycles du TB. Il n'est pas possible d'utiliser le *helper* `save_context` dans ce cas car placer cette information dans la file d'attente dynamique n'est correct que si un branchement est en attente dans celle-ci. Dans le cas contraire, c'est la suite du TB qui gèrera normalement les répliques. Elle utilise donc un *helper* à part, `save_ctx_prolog`, qui vérifie la présence d'un branchement dans la file d'attente dynamique avant d'y insérer la sauvegarde.

Conclusion sur la granularité du TB

La figure 4.12 présente la version finale fonctionnelle du code généré par notre implémentation. On peut voir aux lignes 4, 9 et 19 la sauvegarde de contexte dans le prologue pour les trois instructions ayant une latence supérieure à 0. L'ensemble de ces techniques permet d'appliquer l'algorithme de simulation VLIW à la DBT malgré sa granularité du TB au prix d'une taille de code généré plus importante.

EP		L		Code c6x			Micro-opérations QEMU
1	1	0		add	.L1	a2, a1, a2	`add_i32 a2_1, a2_0, a1_0`
	2	0	||	sub	.D1	a1, a4, a6	`sub_i32 a6_1, a1_0, a4_0`
	3	3	||	mpy32	.M1	a2, a1, a3	`mul_i32 a3_1, a2_0, a1_0`
	4						`mov_i32 a6_0, a6_1 ; État canonique`
	5						`mov_i32 a2_0, a2_1 ; État canonique`
	6						`call restore_context ;Rest. ctx`
2	7	5		b	.S1	0xbeef	`mov_i32 pce1_1, $0xbeef`
	8						`call restore_context ;Rest. ctx`
3	9	0		sub	.D1	a6, a3, a3	`sub_i32 a3_2, a6_1, a3_0`
	10						`mov_i32 a3_0, a3_2 ; État canonique`
	11						`call restore_context ;Rest. ctx`
4	12	0		add	.L1	a2, a3, a4	`add_i32 a4_1, a2_1, a3_2`
	13						`mov_i32 a4_0, a4_1 ; État canonique`
	14						`mov_i32 a3_0, a3_1 ; État canonique`
	15						`call restore_context ;Rest. ctx`
5	16	3		mpy32	.M1	a2, a3, a5	`mul_i32 a5_1, a2_1, a3_1`
	17						`call restore_context ;Rest. ctx`
6	18			nop	2		`; nop 2`
7	19			;	Épilogue TB		`; Sauvegarde contexte`
	20						`call save_context, 5, a5_1, 0`
	21						`; Retour état canonique`
	22						`mov_i32 a2_0, a2_1`
	23						`mov_i32 a3_0, a3_1`
	24						`mov_i32 a4_0, a4_1`
	25						`mov_i32 a6_0, a6_1`
	26						`mov_i32 pce1_0, pce1_1`
	27						`exit_tb 0x0`

FIGURE 4.11 – Restauration de contexte

EP		L	Code c6x	Micro-opérations QEMU
1	1	0	`add .L1 a2, a1, a2`	`add_i32 a2_1, a2_0, a1_0`
	2	0	`|| sub .D1 a1, a4, a6`	`sub_i32 a6_1, a1_0, a4_0`
	3	3	`|| mpy32 .M1 a2, a1, a3`	`mul_i32 a3_1, a2_0, a1_0`
	4			`call save_ctx_prolog, 3, a3_1, 3`
	5			`mov_i32 a6_0, a6_1 ; État canonique`
	6			`mov_i32 a2_0, a2_1 ; État canonique`
	7			`call restore_context ;Rest. ctx`
2	8	5	`b .S1 0xbeef`	`mov_i32 pce1_1, $0xbeef`
	9			`call save_ctx_prolog, "pce1", pce1_1, 5`
	10			`call restore_context ;Rest. ctx`
3	11	0	`sub .D1 a6, a3, a3`	`sub_i32 a3_2, a6_1, a3_0`
	12			`mov_i32 a3_0, a3_2 ; État canonique`
	13			`call restore_context ;Rest. ctx`
4	14	0	`add .L1 a2, a3, a4`	`add_i32 a4_1, a2_1, a3_2`
	15			`mov_i32 a4_0, a4_1 ; État canonique`
	16			`mov_i32 a3_0, a3_1 ; État canonique`
	17			`call restore_context ;Rest. ctx`
5	18	3	`mpy32 .M1 a2, a3, a5`	`mul_i32 a5_1, a2_1, a3_1`
	19			`call save_ctx_prolog, 5, a5_1, 3`
	20			`call restore_context ;Rest. ctx`
6	21		`nop 2`	`; nop 2`
7	22		`; Épilogue TB`	`; Sauvegarde contexte`
	23			`call save_context, 5, a5_1, 0`
	24			`; Retour état canonique`
	25			`mov_i32 a2_0, a2_1`
	26			`mov_i32 a3_0, a3_1`
	27			`mov_i32 a4_0, a4_1`
	28			`mov_i32 a6_0, a6_1`
	29			`mov_i32 pce1_0, pce1_1`
	30			`exit_tb 0x0`

FIGURE 4.12 – Sauvegarde de contexte pendant les p premiers cycles

4.2.4 Gestion des instructions prédiquées

Les instructions prédiquées ou conditionnelles ne sont exécutées que si une certaine condition est remplie, en général la valeur d'un registre étant nulle ou non. Pour l'architecture `c6x`, la condition peut reposer sur la valeur de l'un des 6 registres `a0...a2`, `b0...b2`. Il est possible de tester l'égalité ou non à zéro. La figure 4.13 présente un paquet d'exécution avec les trois cas possibles. L'instruction de la ligne 1 est une instruction classique,

EP		L	Code c6x
1	1	0	`add .L1 a0, a1, a2`
	2	0	`|| [a0] sub .D2 b3, b2, b4`
	3	5	`|| [!b2] b .S2 some_label`

FIGURE 4.13 – Les instructions prédiquées en `c6x`

non prédiquée. Celle de la ligne 2 ne sera exécutée que si `a0` est différent de 0, alors que le branchement de la ligne 3 ne sera pris que si `b2` vaut 0.

Pour prendre en charge les instructions prédiquées, il est possible de générer des structures de contrôle de type `if...then` en représentation intermédiaire QEMU. Cependant, la valeur de la condition n'étant pas connue au moment de la traduction, les instructions prédiquées posent problème lors de l'allocation des répliques. Si on traduit naïvement le code de l'exemple précédent, on obtient les micro-opérations telles que décrites par la figure 4.14. Les micro-opérations aux lignes 2 et 5 permettent de tester les conditions. Pour la première,

EP		L	Code c6x	Micro-opérations QEMU
1	1	0	`add .L1 a0, a1, a2`	`add_i32 a2_1, a0_0, a1_0`
	2	0	`|| [a0] sub .D2 b3, b2, b4`	`brcond_i32 a0_0, $0x0, eq, lbl1`
	3			`sub_i32 b4_1, b3_0, b2_0`
	4			`lbl1:`
	5	5	`|| [!b2] b .S2 some_label`	`brcond_i32 b2_0, $0x0, ne, lbl2`
	6			`mov_i32 pce1_1, some_label`
	7			`lbl2:`
2	8	0	`add .L2 b4, b4, b0`	`add_i32 b4_1, b4_1, b0_0`

FIGURE 4.14 – Traduction naïve des instructions prédiquées

un saut à `lbl1` est effectué si `a0_0` vaut 0 (`a0_0 eq $0x0`). C'est donc la condition inverse qui est testée pour sauter la micro-opération qui simule l'instruction en cas de condition fausse. Pour la deuxième, le branchement à `lbl2` est pris si `b2_0` est différent de 0 (`b2_0 ne $0x0`).

Pour le résultat de chaque instruction, une réplique a été statiquement allouée. Cependant, que se passe-t-il lorsque la condition est fausse au moment de l'exécution ? Si l'instruction de la ligne 2 n'est pas exécutée, la réplique `b4_1` ne sera pas initialisée. Or, elle est utilisée par l'instruction à la ligne 8 dans le paquet suivant. On voit ici que le traducteur doit aussi gérer le cas où la condition est fausse.

Il existe plusieurs cas de conditions qui rendent la génération non triviale :

– le cas classique d'une instruction prédiquée ayant une latence de 0 cycle comme nous venons de le voir ;
– celui d'une instruction prédiquée ayant une latence de plus de 0 cycle ;
– le cas de deux instructions prédiquées, parallèles, l'une ayant la condition inverse de l'autre (par exemple `[a0]` et `[!a0]`) et les deux écrivant le même registre résultat. Ce cas est d'ailleurs le seul pour lequel il est autorisé dans la spécification du `c6x` d'avoir deux instructions écrivant le même registre au même cycle puisqu'il y en aura forcément une et une seule qui sera exécutée ;
– le cas des branchements prédiqués.

Nous passerons chaque cas en revue en mettant en évidence les difficultés et donnerons une solution pour chacun d'entre eux.

L'instruction prédiquée sans latence

Ce cas est celui de l'instruction `sub` à la ligne 2 sur la figure 4.14. Le plus simple pour ne pas perturber l'algorithme d'allocation statique de répliques est de recopier la valeur de l'ancienne réplique dans la nouvelle en cas de condition fausse. La figure 4.15 illustre cette solution. Ici, une structure de contrôle de type `if...then...else` est générée. Si la

EP		L	Code `c6x`	Micro-opérations QEMU
1	1	0	`[a0] sub .D2 b3, b2, b4`	`brcond_i32 a0_0, $0x0, eq, lbl1`
	2			`sub_i32 b4_1, b3_0, b2_0`
	3			`br lbl2`
	4			`lbl1:`
	5			`mov_i32 b4_1, b4_0`
	6			`lbl2:`

FIGURE 4.15 – Traduction d'une instruction prédiquée sans latence

condition n'est pas remplie, alors la valeur de `b4_0` est propagée dans `b4_1`.

L'instruction prédiquée avec latence

Pour une instruction ayant une latence supérieure à 0 cycle, la solution précédente ne convient pas. La figure 4.16 met en avant le problème en insérant une instruction dans le *delay slot* de l'instruction `mpy32` de la ligne 1. L'instruction `add` à la ligne 7 provoque l'allocation de la réplique `a1_2` pour le registre `a1`. Si la condition n'est pas vérifiée à l'exécution, alors le registre `a1` utilisera la réplique `a1_1` contenant une valeur obsolète après le *delay slot* de l'instruction `mpy32`. Le résultat du `add` sera perdu.

Dans ce cas, la solution consiste à déporter la ligne 5 de propagation au moment où le *delay slot* de l'instruction conditionnelle se termine. Cependant pour appliquer cette solution, il faut être capable de connaître le résultat, vrai ou faux, de la condition plus tard dans le code généré. Pour se faire, nous introduisons dans l'*environnement* des drapeaux de condition (`cf_x`), utilisables par le traducteur au même titre que les répliques des registres, afin de sauvegarder le résultat des conditions. Sur la figure 4.17, le drapeau `cf_0` est écrit à la ligne 3 ou 6, suivant le résultat de la condition. Il est ensuite testé à la fin du *delay slot* de l'instruction `mpy32` à la ligne 10. Si la condition se révèle fausse, il faut propager la valeur

EP		L	Code c6x	Micro-opérations QEMU
1	1	3	`[a0] mpy32 .M1 a3, a2, a1`	`brcond_i32 a0_0, $0x0, eq, lbl1`
	2			`mul_i32 a1_1, a3_0, a2_0`
	3			`br lbl2`
	4			`lbl1:`
	5			`mov_i32 a1_1, a1_0`
	6			`lbl2:`
2	7	0	`add .L1 a2, a2, a1`	`add_i32 a1_2, a2_0, a2_0`
3	8		`nop 2`	`; nop 2`
4	9	0	`add .L1 a1, a3, a2`	`add_i32 a2_1, a1_1, a3_0`

FIGURE 4.16 – Traduction incorrecte d'une instruction prédiquée avec latence

de l'ancienne réplique dans la nouvelle, comme le fait la micro-opération à la ligne 11. À ce moment de la génération, le traducteur connaît la réplique en cours d'utilisation (a1_2), et peut générer la micro-opération de propagation correspondante.

La double condition inverse, avec même destination

Le cas de deux instructions prédiquées écrivant au même cycle la même destination, avec des conditions inverses doit être pris en compte lors de la génération. La figure 4.18 illustre ce cas. L'instruction sub de la ligne 1 est traduite comme expliqué précédemment. En revanche, pour l'instruction add de la ligne 7, le traducteur se rend compte qu'il existe déjà une réplique en attente pour le registre b4, et que celle-ci provient d'une instruction prédiquée, dont la condition est l'inverse de la condition courante. Dans ce cas, il n'alloue pas de nouvelle réplique pour le résultat de l'instruction mais réutilise celle allouée pour la première instruction prédiquée. De plus, il ne génère pas de propagation en cas de condition fausse puisque à ce stade, nous sommes sûrs que la nouvelle réplique sera initialisée correctement par l'une des deux instructions.

À ce propos, la propagation à la ligne 5 n'est pas nécessaire non plus. Cependant, dans son implémentation actuelle, le traducteur traite les instructions une par une, et n'est pas capable d'anticiper la présence de l'instruction add ici. Il n'est donc pas possible de détecter que l'on se trouve dans un cas de double condition inverse au moment de la traduction de l'instruction sub à la ligne 1. De plus, QEMU ne permet pas de revenir sur le code généré. Il s'agirait donc d'une optimisation que d'étendre le contexte du traducteur à plusieurs instructions pour anticiper ce genre de cas.

Les branchements conditionnels

Les branchements conditionnels ajoutent la contrainte de terminaison du TB courant. D'une manière générale, peu importe le résultat de la condition, il est préférable de terminer le TB au moment où le branchement devrait être pris. Cela permet d'éviter de traduire le code qui suit le branchement puisque ce dernier ne sera peut-être jamais exécuté. C'est la stratégie employée par notre traducteur. En cas de branchement conditionnel, la fin du TB

EP		L	Code c6x	Micro-opérations QEMU
1	1	3	`[a0] mpy32 .M1 a3, a2, a1`	`brcond_i32 a0_0, 0, eq, lbl1`
	2			`mul_i32 a1_1, a3_0, a2_0`
	3			`movi_i32 cf_0, 1`
	4			`br lbl2`
	5			`lbl1:`
	6			`movi_i32 cf_0, 0`
	7			`lbl2:`
2	8	0	`add .L1 a2, a2, a1`	`add_i32 a1_2, a2_0, a2_0`
3	9		`nop 2`	`; nop 2`
	10			`brcond_i32 cf_0, 0, ne, lbl3`
	11			`mov_i32 a1_1, a1_2`
	12			`lbl3:`
4	13	0	`add .L1 a1, a3, a2`	`add_i32 a2_1, a1_1, a3_0`

FIGURE 4.17 – Traduction correcte d'une instruction prédiquée avec latence

survient à la fin du *delay slot* du branchement. La figure 4.19 décrit la stratégie employée. Le prologue du TB contient la sauvegarde de contexte et le retour à l'état canonique à la fois dans le cas du branchement (lignes 11 à 13), et dans le cas contraire (lignes 15 à 18). La seule différence entre les deux cas est la copie de l'adresse de l'instruction suivante dans `pce1` à la ligne 15 si le branchement n'est pas pris. Cela peut paraître sous-optimal puisque la sauvegarde de contexte est la même dans les deux cas. Cependant, séparer les deux est nécessaire pour gérer un cas particulier décrit à la figure 4.20. Dans ce cas, la dernière instruction est un `nop` qui dure 4 cycles. Or, si le branchement est pris, il survient juste après le premier cycle du `nop`. Le comportement du processeur dans ce cas est de stopper l'instruction pour prendre le branchement et d'annuler les 3 cycles restants. Notre méthode de génération permet de gérer correctement ce cas. Si le branchement est pris, alors la sauvegarde de contexte est effectuée comme si l'instruction `nop` n'avait duré qu'un cycle. On peut donc voir la sauvegarde dynamique de contexte aux lignes 14 et 15, avec 0 cycle restant pour le résultat du `mpy32` et 2 cycles restants pour le `ldw` (chargement mémoire). En revanche, si le branchement n'est pas pris, les 3 cycles du `nop` laissent le temps aux répliques en attente d'être appliquées statiquement. Il n'y a donc pas de sauvegarde dynamique mais seulement un retour à l'état canonique (lignes 20 à 22).

EP	#	L	Code c6x	Micro-opérations QEMU
1	1	0	`[a0] sub .D2 b3, b2, b4`	`brcond_i32 a0_0, $0x0, eq, lbl1`
	2			`sub_i32 b4_1, b3_0, b2_0`
	3			`br lbl2`
	4			`lbl1:`
	5			`mov_i32 b4_1, b4_0`
	6			`lbl2:`
	7	0	`|| [!a0] add .L2 b3, b2, b4`	`brcond_i32 a0_0, $0x0, ne, lbl3`
	8			`add_i32 b4_1, b3_0, b2_0`
	9			`lbl3:`

FIGURE 4.18 – Traduction de deux instructions parallèles avec conditions inverses

EP	#	L	Code c6x	Micro-opérations QEMU
1	1	5	`[a0] b .S2 0x80ec`	`brcond_i32 a0_0, 0, eq, lbl1`
	2			`mov_i32 pce1_1, some_label`
	3			`movi_i32 cf_0, 1`
	4			`br lbl2`
	5			`lbl1:`
	6			`movi_i32 cf_0, 0`
	7			`lbl2:`
2	8	0	`add .L1 a0, a0, a0`	`add_i32 a0_1, a0_0, a0_0`
3	9		`nop 4`	`; nop 4`
4	10		`; Prologue TB`	`brcond_i32 cf_0, 0, eq, lbl3`
	11			`mov_i32 a0_0, a0_1`
	12			`mov_i32 pce1_0, pce1_1`
	13			`exit_tb $0x0`
	14			`lbl3:`
	15			`movi_i32 pce1_1, next_insn`
	16			`mov_i32 a0_0, a0_1`
	17			`mov_i32 pce1_0, pce1_1`
	18			`exit_tb $0x0`

FIGURE 4.19 – Traduction d'un branchement prédiqué

EP		L	Code c6x	Micro-opérations QEMU
1	1	5	`[a0] b .S2 some_label`	`brcond_i32 a0_0, $0x0, eq, lbl1`
	2			`mov_i32 pce1_1, some_label`
	3			`movi_i32 cf_0, 1`
	4			`br lbl2`
	5			`lbl1:`
	6			`movi_i32 cf_0, 0`
	7			`lbl2:`
2	8		`nop 2`	`; nop 2`
3	9	3	`mpy32 .M1 a0, a1, a2`	`mul_i32 a2_1, a0_0, a1_0`
4	10	4	`ldw .D1T1 *+a0(0), a1`	`add_i32 tmp0, a0_0, $0x0`
	11			`qemu_ld32 a1_1, tmp0`
5	12		`nop 4`	`; nop 4`
6	13		`; Prologue TB`	`brcond_i32 cf_0, 0, eq, lbl3`
	14			`call save_ctx, 2, a2_1, 0`
	15			`call save_ctx, 1, a1_1, 2`
	16			`mov_i32 pce1_0, pce1_1`
	17			`exit_tb $0x0`
	18			`lbl3:`
	19			`movi_i32 pce1_1, next_insn`
	20			`mov_i32 a1_0, a1_1`
	21			`mov_i32 a2_0, a2_1`
	22			`mov_i32 pce1_0, pce1_1`
	23			`exit_tb $0x0`

FIGURE 4.20 – Cas particulier d'un branchement prédiqué avec un nop en fin de TB

4.2.5 Conclusion sur l'intégration dans un traducteur DBT

L'ensemble des techniques présentées dans cette section permet l'intégration de l'algorithme de simulation VLIW sur architecture scalaire au sein d'un traducteur binaire dynamique. La granularité de traduction de la DBT étant le TB, l'algorithme est adapté en déportant une partie de la file d'attente au moment de l'exécution. Après la mise en place de mécanismes pour gérer les cas limites apparaissant à cause de la traduction par bloc, le traducteur est à même de produire du code correct pour simuler une architecture VLIW, au prix de TB générés plus conséquents.

4.3 Expérimentations

Cette section présente les résultats d'expérimentations sur notre implémentation de traducteur `c6x` dans QEMU. Nous nous comparons au simulateur officiel fourni par Texas Instrument, intégré dans la suite Code Composer Studio[CCS] (CCS) qui utilise la technique de l'interprétation pour simuler le `c6x`.

4.3.1 Protocole expérimental

L'implémentation du traducteur a été vérifiée à l'aide de tests unitaires non présentés ici, et grâce à des tests d'intégrations plus conséquents pour lesquels la trace d'exécution générée par QEMU était comparée à celle de CCS pour en vérifier la conformité.

Tous les tests ont été compilés à l'aide de GCC 4.9.1, et exécutés sur une machine équipée de 8 processeurs Intel(R) Xeon(R) E5-2665 cadencés à 2.40Ghz et contenant 4 cœurs chacun, et de 256GiO de mémoire vive.

Pour nos différentes évaluations, nous nous appuierons sur trois tests de natures différentes :

fibo qui calcule le nombre de la suite de Fibonacci pour un indice donné en paramètre. L'algorithme utilisé dans ce programme est une version récursive naïve qui réalise un grand nombre d'appels récursifs de la même fonction.

matrix qui calcule le produit de deux matrices carrées de taille donnée en paramètre du test. Il s'agit ici encore d'une version naïve, effectuant une multiplication et une addition dans une triple boucle. Ce test exécute beaucoup d'instructions de contrôle, et quelques instructions arithmétiques.

idct qui calcule la transformée en cosinus discrète inverse de n blocs de 16 bits, avec n donné en paramètre. Ce test implémente une version optimisée de l'algorithme qui réalise beaucoup d'instructions arithmétiques et logiques, et peu de contrôle ce qui implique des TB de grandes tailles.

4.3.2 Statistiques sur le code généré

Afin d'étudier l'impact de notre solution sur le code généré, nous instrumentons notre implémentation dans QEMU pour en extraire des statistiques sur les micro-opérations d'un TB. Pour chaque TB, nous extrayons le nombre d'instructions cibles `c6x` traduites, ainsi que des informations précises sur les micro-opérations générées parmi lesquelles :

- le nombre de micro-opérations effectuant le retour à l'état canonique,
- le nombre de micro-opérations gérant les instructions prédiquées,

– le nombre de micro-opérations prenant en charge la sauvegarde et restauration de contexte,
– le reste des micro-opérations simulant effectivement les instructions cibles. Il s'agit ici de toutes les micro-opérations générées par QEMU pour simuler le comportement des instructions du TB.

La figure 4.21 donne les résultats de ces statistiques sur les trois programmes de tests décrits section 4.3.1. Pour chaque test, elle donne le nombre moyen d'instructions cibles traduites dans un TB et le nombre moyen de micro-opérations générées en conséquence. Les micro-opérations sont découpées comme décrit précédemment. Les tests `fibo` et `matrix` ont des TB relativement petits en moyenne. Cela est dû à la nature de ces tests qui réalisent principalement des branchements et peu de calculs. Pour ces tests, la sauvegarde et restauration de contexte représentent une part importante des micro-opérations générées. En effet, pour des TB avec 7 à 9 instructions cibles en moyenne, le coût fixe de la restauration de contexte durant les 6 premiers cycles n'est pas amorti. En revanche, la part de retour à l'état canonique est négligeable puisque pour un nombre d'instructions faible dans le TB, l'allocateur de répliques n'est presque pas sollicité. Ce coût est d'environ 3 micro-opérations pour `fibo` et 4 pour `matrix`.

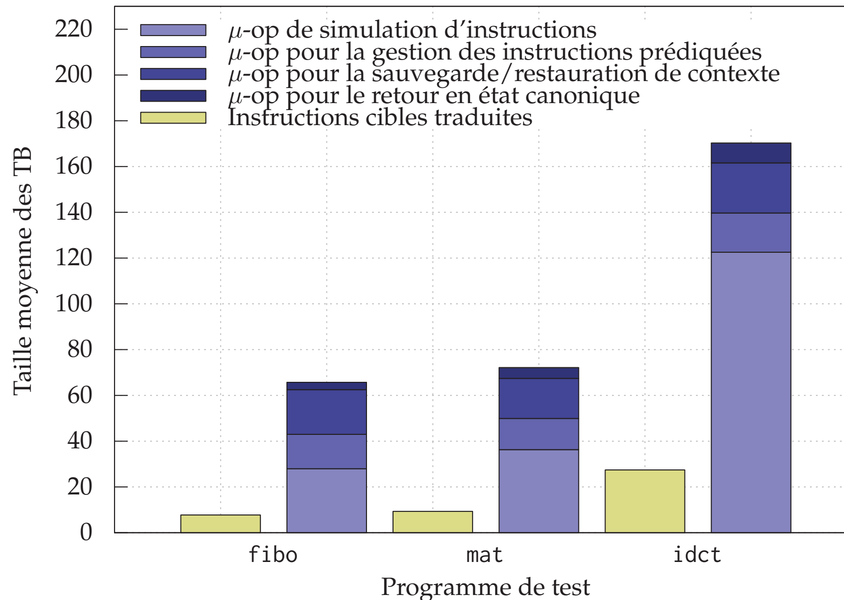


FIGURE 4.21 – Statistique sur les micro-opérations générées dans les trois tests

Pour le test `idct`, la taille moyenne des TB est bien plus importante puisqu'elle vaut en moyenne une trentaine d'instructions cibles traduites. Le ratio de micro-opérations pour simuler effectivement les instructions cibles est ici beaucoup plus important que dans les cas précédents. Elles représentent environ 72% du nombre total de micro-opérations. Les coûts de la sauvegarde/restauration de contexte et de retour à l'état canonique sont donc nettement amortis dans ce cas.

Le ratio micro-opérations générées/instructions cibles est d'environ 8.45 pour `fibo`, 7.74 pour `matrix`, et 6.21 pour `idct`, ce qui semble confirmer les observations précédentes. Pour vérifier l'amortissement de ce ratio en fonction de la taille d'entrée du TB, nous avons me-

suré le ratio moyen de chaque TB dans tous les tests dont nous disposions. La figure 4.22 illustre cette expérience. Elle trace le ratio moyen d'un TB en fonction de sa taille d'entrée,

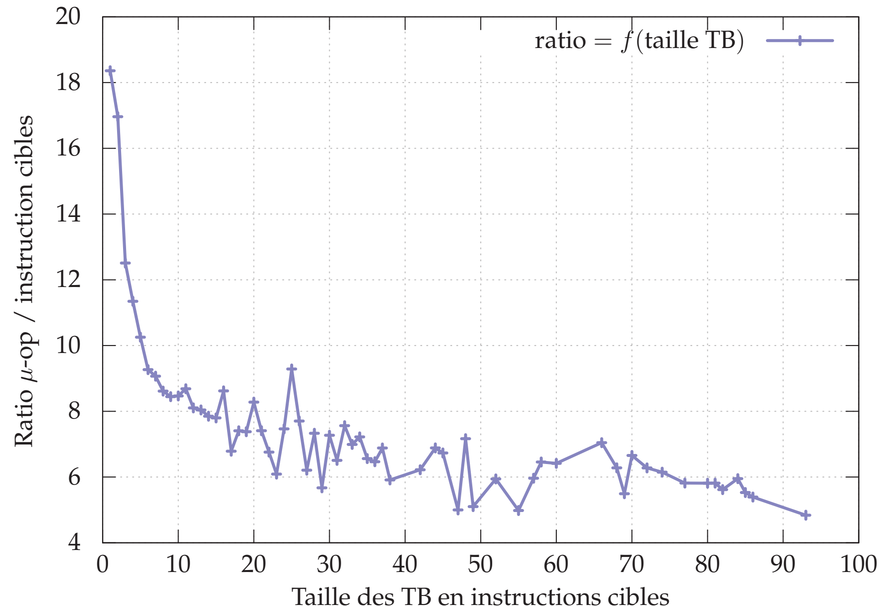


FIGURE 4.22 – Évolution du ratio de taille des TB en fonction de sa taille d'entrée

c'est-à-dire sa taille en nombre d'instructions cibles traduites. On peut voir que ce ratio décroit très largement au début, puis se stabilise autour de 6 micro-opérations pour une instruction cible. Cela confirme la nécessité d'avoir des TB de tailles raisonnables pour amortir le coup des mécanismes de traduction décrits précédemment.

4.3.3 Mesure du temps d'exécution de la simulation

Afin d'évaluer les performances en terme de vitesse d'exécution de notre simulateur, nous mesurons le temps d'exécution des différents tests sur notre implémentation et comparons ce temps à celui de CCS. Chaque test de performance est réalisé lorsque les simulateurs ont été initialisés. Pour QEMU, les mesures sont effectuées autour de la boucle principale. Pour CCS, la machine virtuelle Java est déjà chargée et prête à simuler la première instruction. Pour chaque test, nous extrayons la moyenne et l'écart-type du temps d'exécution sur 120 lancements. Cela nous permet de considérer les temps d'exécutions comme étant distribués selon une loi normale, et de pouvoir calculer un intervalle de confiance pour la moyenne.

Chacun des tests présentés section 4.3.1 a un paramètre que l'on fait varier pour réaliser une série de mesures de performance. Les figures 4.23, 4.24, et 4.25 donnent les résultats des tests `fibo`, `matrix` et `idct` respectivement.

En abscisse, on retrouve la valeur du paramètre du test. En ordonnée est indiqué le temps d'exécution du test dans le simulateur en ms, sur une échelle logarithmique. Pour QEMU, le temps de traduction est extrait du temps total et donné à part sur chaque figure. On constate un très net avantage pour la DBT par rapport à la simulation interprétée. Dès lors que les tests deviennent raisonnablement longs, les deux simulateurs se démarquent très clairement. Une différence d'environ deux ordres de grandeur apparaît dès l'indice 20 pour

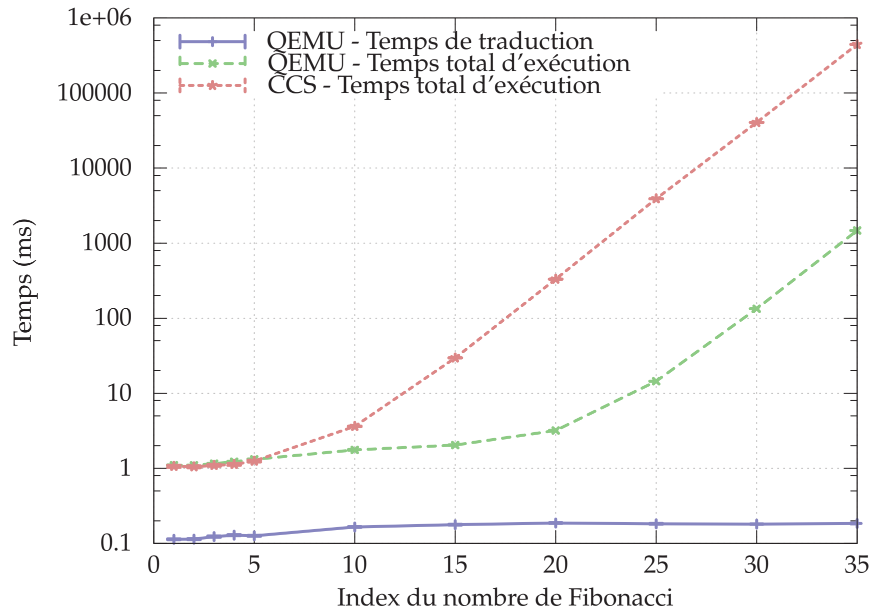


FIGURE 4.23 – Temps d'exécution du test `fibo`

`fibo` et avant l'indice 50 pour `matrix` et `idct`. Pour QEMU, le temps de traduction est stable en fonction du paramètre, et reste négligeable par rapport au temps total d'exécution. Le coût de traduction est donc très vite amorti grâce à la réutilisation du code traduit.

Ces résultats montrent que malgré le surcoût introduit par les mécanismes de traduction de l'architecture VLIW, les performances du simulateur sont excellentes. Afin de montrer la reproductibilité de ces résultats, ainsi que la maturité de notre implémentation dans QEMU, nous fournissons dans l'annexe B d'autres mesures de performances réalisées sur la suite Polybench[Pou11].

4.4 Conclusion

Dans ce chapitre, nous avons proposé une méthode pour simuler une architecture de type VLIW sur une architecture scalaire à l'aide de la DBT. Nous avons donné un algorithme général pour prendre en charge les instructions parallèles et leur éventuelle latence, puis nous avons adapté cet algorithme pour l'introduire dans un flot de DBT. La granularité de traduction classique d'un flot de DBT induit des effets de bords aux frontières des TB, effets que nous avons gérés en déportant une partie de l'algorithme au moment de l'exécution du code généré.

Les résultats expérimentaux nous montrent que malgré la part importante de micro-opérations générées pour gérer ces effets de bords, les performances sont excellentes comparées à un simulateur fonctionnant sur le principe de l'interprétation.

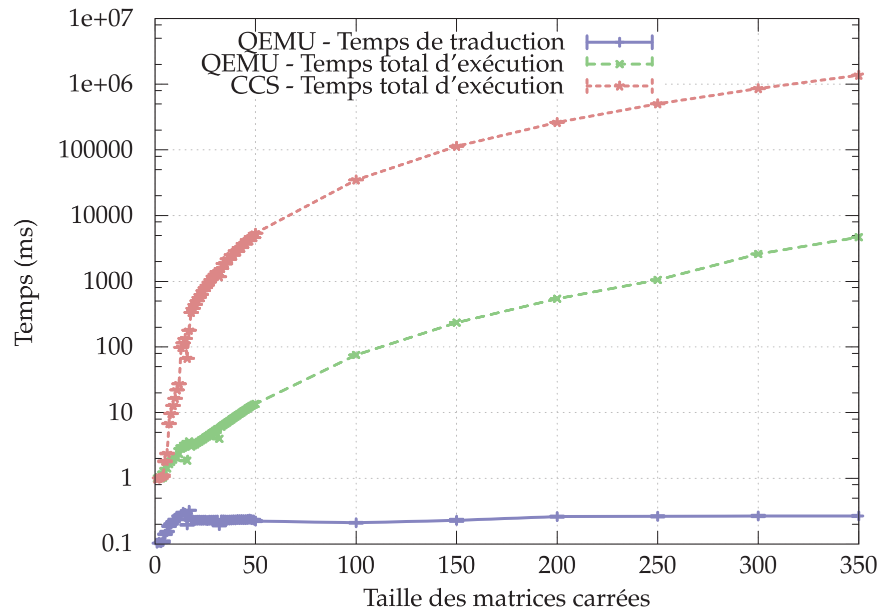


FIGURE 4.24 – Temps d'exécution du test `matrix`

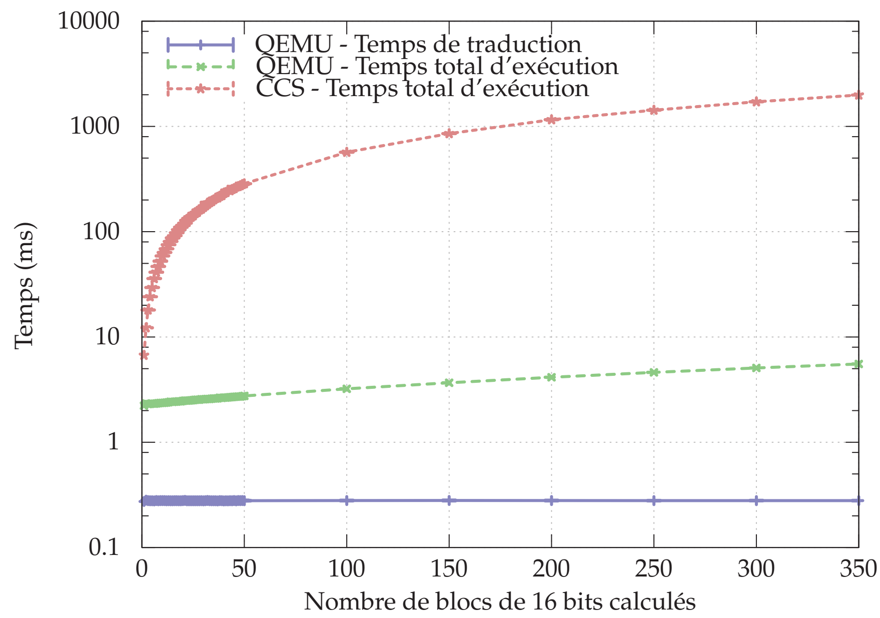


FIGURE 4.25 – Temps d'exécution du test `idct`

Chapitre 5

Génération de simulateurs basés sur la traduction binaire dynamique

DANS ce chapitre, nous présentons notre flot de génération automatique de simulateurs basés sur la traduction binaire dynamique. Un simulateur est généré à partir de la description de l'architecture à simuler (l'architecture cible) ainsi que celle de l'architecture exécutant la simulation (l'architecture hôte). L'intérêt de disposer de ces deux descriptions est la possibilité de générer un simulateur optimisé pour le couple cible – hôte.

Ce chapitre s'organise comme suit. Nous présenterons tout d'abord le flot général de génération et en décrirons les principales briques. Nous présenterons ensuite rapidement le langage de description d'architecture sur lequel nous nous baserons pour la génération. Dans les sections suivantes, nous décrirons les stratégies d'analyses de comportement d'instructions de la machine cible et hôte pour ensuite les mettre en correspondance. Finalement, nous présenterons nos méthodes de génération du simulateur et conclurons par quelques expérimentations.

5.1 Présentation du flot de génération

Le flot que nous proposons pour la génération de simulateurs basés sur la DBT est illustré par la figure 5.1. Il repose sur une étape essentielle de sélection des instructions de la machine hôte pour simuler celles de la machine cible, ainsi que sur des étapes de génération du simulateur en lui-même. Il se découpe en trois parties principales :

– la mise en correspondance des instructions, qui est l'étape consistant à trouver pour chaque instruction cible, une ou plusieurs instructions hôtes candidates pour la simuler. Nous obtenons en sortie de cette étape, en plus des informations de correspondance entre les instructions, l'ensemble des micro-opérations spécialisées pour le couple cible – hôte ;
– le générateur du *frontend*, qui produit la première partie du simulateur, responsable de la traduction des instructions cibles en micro-opérations. Cette étape génère aussi l'*environnement* du processeur simulé, contenant par exemple l'état des registres de ce dernier. Cette étape se sert de la description de la machine cible pour générer l'environnement et le décodeur d'instructions, et des informations de mise en correspondance de l'étape précédente pour générer le traducteur ;
– le générateur du *backend*, qui produit la dernière partie du simulateur en charge de traduire les micro-opérations en instructions pour la machine hôte. Il produit l'enco-

deur binaire pour la machine hôte à partir de la description de cette dernière et la correspondance entre les micro-opérations générées et les instructions hôtes.

Certaines parties du flot ne sont pas générées automatiquement. Il s'agit de la partie *Support hôte* dans le *backend*, ainsi que l'ensemble minimal des micro-opérations. Ces micro-opérations ne sont pas créées à l'issue du processus de mise en correspondance. En effet, elles sont fixes quelque soit le couple cible – hôte. Ces parties non auto-générées sont appelées le *runtime minimal* et seront présentées en détail section 5.6.1.

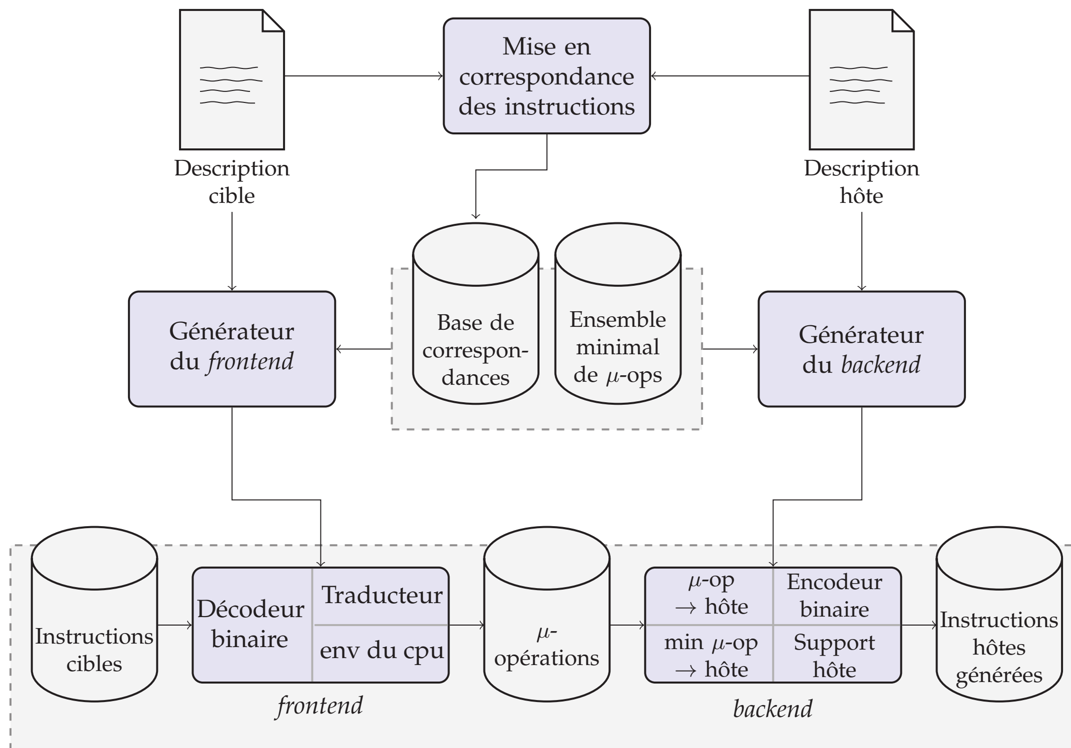


FIGURE 5.1 – Flot de génération du simulateur

À l'issue de ce flot, nous obtenons un *frontend* et un *backend* à même de traduire les instructions de la machine cible en instructions de la machine hôte. Nous conservons une IR dans le but de pouvoir appliquer facilement des optimisations génériques au moment de la traduction sur les micro-opérations générées comme la propagation de constantes ou l'élimination de code mort. En effet, il est toujours plus aisé de réaliser ce genre d'opérations sur une représentation abstraite connue que sur le code assembleur de l'hôte.

5.2 MDS : un ADL comportemental

Machine Description System (MDS)[DdD] est un ADL comportemental visant le support à la génération de simulateurs, de *backend* de compilateurs, de jeux de tests hauts niveaux, etc. La description de la machine peut être faite à l'aide du langage YAML[BKEI01] (YAML Ain't Markup Language), qui est ensuite analysée et traduite dans une représentation propre à MDS.

Une description MDS se découpe en plusieurs parties. Nous décrirons celles d'intérêt

pour le support à la génération de simulateurs.

5.2.1 Les espaces mémoires et les registres

MDS décrit les espaces adressables par le processeur à différents niveaux de hiérarchie. La première section concernée par cette description est la section `Storage`. On y retrouve les bancs de registres ainsi que la mémoire. Le listing 5.1 donne un extrait de la description des `Storage` de l'architecture MIPS en MDS.

```
Storage:
  - ID: GPRS
    kind: Register
    width: 32
    count: 32
  - ID: MEM
    kind: Memory
    width: 8
```

Listing 5.1 – Extrait de la description des `Storage` pour l'architecture MIPS

On peut y voir le banc de registres à usage général, composé de 32 registres de taille 32 bits chacun, ainsi que la mémoire, dont la granularité d'accès est 8 bits et la taille non spécifiée.

Les registres eux-mêmes sont ensuite décrits dans la section `Regfile` s'ils font partie d'un banc de registres ou `Register` s'il s'agit de registres isolés. Chacun fait référence à une partie de `Storage` pour indiquer leur localisation dans le processeur. Le listing 5.2 donne un extrait de la description des registres du MIPS. On peut y voir les trois premiers registres placés dans le `Storage` `GPRS`, chacun à son emplacement respectif.

```
Regfile:
  - ID: GPRF
    what: General Purpose Register File
    registers:
      - {ID: GR0, location: [GPRS, 0], names: "zero", stuckat: 0}
      - {ID: GR1, location: [GPRS, 1], names: "at"}
      - {ID: GR2, location: [GPRS, 2], names: "v0"}
      [...]
```

Listing 5.2 – Extrait de la description des registres du MIPS

5.2.2 Les instructions et leurs formats

La description du jeu d'instructions se découpe en deux parties, les formats d'instructions et les instructions elles-mêmes. Les formats regroupent plusieurs instructions qui ont en commun :

– un certain motif dans l'encodage ;

– leurs opérandes ;
– leur syntaxe en langage d'assemblage ;
– une partie de leur comportement.

Vient ensuite la description des instructions comprenant principalement le comportement de chacune.

Les listings 5.3 et 5.4 donnent la description de l'instruction addu et de son format.

```
Instruction:
  - ID: ADDU
    formats: [ R_TYPE ]
    behavior: |
      (WRITE res
        (ADD
          (ZX 32 (READ tmp0))
          (ZX 32 (READ tmp1))))
```

Listing 5.3 – Description de l'instruction MIPS addu

```
Format:
  - ID: R_TYPE
    fields:
      - func: {[...], "100001": ADDU, "100011": SUBU, [...]}
      - sa: "-----"
      - rd: "-----"
      - rt: "-----"
      - rs: "-----"
      - op: "000000"
    operands: [{singleReg: rd}, {singleReg: rs}, {singleReg: rt}]
    syntax: "%0 %1, %2, %3"
    behavior: |
      (SEQ
        (WRITE tmp0 (ACCESS 0 %2))
        (WRITE tmp1 (ACCESS 0 %3))
        (COMMIT 3 %1 (READ res))
      )
```

Listing 5.4 – Description du format de addu

On voit ici que le format décrit l'encodage de l'instruction (sous-section fields), et que addu n'est pas la seule à posséder ce format. Ce format décrit trois opérandes : rd, rs et rt.

5.2.3 Les comportements d'instructions

Les comportements d'instructions sont décrits dans la sous-section behavior des instructions et des formats. Le comportement d'une instruction est donné par la concaténation

du comportement de son format et de son propre comportement. Les comportement sont décrits dans un langage parenthésé proche de Lisp.

Le comportement de l'instruction addu est rassemblé dans le listing 5.5

```
1  (SEQ
2    (WRITE tmp0 (ACCESS 0 %2))
3    (WRITE tmp1 (ACCESS 0 %3))
4    (COMMIT 3 %1 (READ res)))
5  (WRITE res
6    (ADD
7      (ZX 32 (READ tmp0))
8      (ZX 32 (READ tmp1))))
```

Listing 5.5 – Description de l'instruction MIPS addu

Décrivons quelques opérations remarquables dans ce comportement :

– Les opérations (READ) et (WRITE) permettent l'utilisation de variables intermédiaires, comme tmp0, tmp1 et res dans cet exemple. Elles n'ont pas d'effet du point de vue du matériel. Elles facilitent cependant l'écriture des comportements et sont nécessaires pour séparer le comportement du format de celui de l'instruction.
– Les opérations (ACCESS) et (COMMIT) permettent respectivement de lire et d'écrire un emplacement désigné par un opérande de l'instruction. Dans notre exemple, tmp0 se voit attribuer la valeur de l'opérande 2 (rs), alors que tmp1 reçoit celle de l'opérande 3 (rt). La valeur de res quant à elle est écrite dans l'opérande 1 (rd).
– L'opération (ZX) réalise une extension de zéros sur 32 bits. Dans ce cas, elle est inutile puisque les opérandes sont déjà des registres de 32 bits. Cependant, une instruction pouvant avoir plusieurs formats, la taille de l'opérande peut ne pas être connu à l'avance. Cette écriture permet de s'assurer que l'addition est réalisée sur 32 bits, peu importe les opérandes de l'instruction. L'opération (SX) non présente dans cet exemple permet de réaliser une extension de signe plutôt qu'une extension de zéros.

5.2.4 Conclusion sur MDS

Le langage MDS est bien adapté au but recherché dans ces travaux, à savoir la génération automatique de simulateurs basés sur la DBT. Il est plus proche de la catégorie des ADL comportementaux que de celle des ADL structurels dans le sens où il décrit le comportement des instructions en termes fonctionnels plutôt que structurels. De plus, il décrit toutes les ressources nécessaires à la génération d'un simulateur d'ISA.

5.3 Analyse et transformation des comportements d'instructions

Dans le but de pouvoir les mettre en correspondance, les comportements des instructions doivent être analysés et transformés. Ces transformations vont permettre d'obtenir une forme facilement exploitable des comportements pour la mise en correspondance. Le but est d'arriver à une représentation la plus étendue et développée possible pour que les correspondances entre les comportements similaires d'un point de vue sémantique puissent être faites, même si leurs descriptions diffèrent à l'origine. En ce sens, nous essayons de nous

rapprocher d'une représentation canonique, bien que nous n'ayons de preuve formelle de la canonicité de celle-ci.

Voici la liste des principales étapes de transformations que subit l'arbre de comportement d'une instruction. Elles sont données dans l'ordre de leur application sur l'arbre :

`check_sym`	Vérifications de syntaxe sur l'arbre du comportement.
`rewrite_rw_syms`	Réécriture des symboles utilisés dans les nœuds (READ) et (WRITE) sous la forme SSA.
`merge_seqs`	Fusion des nœuds (SEQ) consécutifs.
`rw_sustitutes`	Substitution des nœuds (READ) par leur définition dans le nœud (WRITE) correspondant.
`remove_writes`	Suppression des nœuds (WRITE).
`access_commit_to_op`	Expansion des nœuds (ACCESS) et (COMMIT) en leur équivalent d'accès aux opérandes de l'instruction.
`type_deco`	Décoration des nœuds avec les types inférés depuis les feuilles de l'arbre.
`expand_op`	Passe d'uniformisation de l'arbre et d'expansion des nœuds en vue de la mise en correspondance avec d'autres comportements.

5.3.1 Vérification syntaxiques et premières transformations

La première passe consiste à analyser l'arbre pour s'assurer qu'il respecte la syntaxe de description comportementale définie dans MDS. Les passes de `rewrite_rw_syms` jusqu'à `remove_writes` transforment l'arbre en un ensemble cohérent, en retirant les nœuds (SEQ) redondants, et en remplaçant chaque nœud (READ) par sa définition (WRITE) correspondante. Les nœuds (WRITE) peuvent être ensuite supprimés. Le listing 5.6 reprend l'instruction addu décrite précédemment et donne son arbre de comportement après lui avoir appliqué les premières passes jusqu'à `remove_writes` incluse.

```
1  (SEQ
2    (COMMIT 3 %1
3      (ADD
4        (ZX 32 (ACCESS 0 %2))
5        (ZX 32 (ACCESS 0 %3)))))
```

Listing 5.6 – Le comportement de l'instruction addu après la passe `remove_writes`

5.3.2 Expansion des nœuds d'accès aux opérandes

Avec la passe `access_commit_to_op`, l'arbre est privé de ses nœuds (ACCESS) et (COMMIT) puisqu'ils sont remplacés par les sous-arbres d'accès aux emplacements désignés par les opérandes de l'instruction. Le listing 5.7 donne l'arbre de comportement de l'instruction après cette passe.

```
1  (SEQ
2    (ABSTRACT_DST Operand-mips-rd op_nat:register
3      (I2F 32
4        (ADD
5          (ZX 32
6            (F2I 32
7              (ABSTRACT_OP Operand-mips-rs
8                           type:[field, 32]
9                           op_nat:register)))
10         (ZX 32
11           (F2I 32
12             (ABSTRACT_OP Operand-mips-rt
13                          type:[field, 32]
14                          op_nat:register)))))))
```

Listing 5.7 – Le comportement de l'instruction addu après la passe access_commit_to_op

On voit apparaître ici deux nouvelles notions. Tout d'abord, les nœuds (I2F) et (F2I) qui permettent de changer de mode de représentation. En effet, MDS supporte plusieurs modes de représentation pour les données manipulées :

- Le mode field, qui représente un vecteur de bits de taille connue sans signification particulière. C'est le mode de représentation utilisé lorsque une donnée est lue ou écrite dans un emplacement.
- Le mode integer, qui représente un entier signé ou non, de taille connue. Ce mode est utilisé par la plupart des opérations arithmétiques et logiques disponibles dans MDS.
- Le mode boolean utilisé pour le résultat des opérations conditionnelles.

Les nœuds (F2I) et (I2F) permettent de passer respectivement du mode de représentation field au mode integer et inversement.

De plus, les nœuds (ACCESS) et (COMMIT) sont remplacés par des nœuds (ABSTRACT_OP) et (ABSTRACT_DST). Ces nœuds représentent des opérandes d'entrées ou de sorties de l'instruction. Ils sont une abstraction par rapport aux nœuds standards de MDS car ils ne spécifient pas explicitement comment accéder à la donnée désignée par l'opérande. Cependant, ils contiennent toute l'information nécessaire pour retrouver cette donnée plus tard. Cette notation permet d'uniformiser l'accès aux opérandes, peu importe leur nature (registre, valeur immédiate, accès mémoire, ...). Par ailleurs, ces nœuds sont déjà décorés de leur type, qui est utilisé dans la passe suivante, et de leur nature. La nature d'un opérande, désignée par l'attribut op_nat dans l'arbre, définit la nature de l'objet auquel l'opérande accède. Elle peut prendre les valeurs suivantes :

register qui désigne un registre à usage général du processeur, que nous nommerons *registre de travail* dans la suite ;

special qui désigne un registre ayant un statut particulier dans le processeur. Il sert souvent à configurer une partie du système ou à en lire l'état. Une écriture dans un tel registre provoque parfois des effets de bord, comme la modification de la configuration de certains sous-systèmes (par exemple l'unité de gestion mémoire (MMU) du processeur). Ces registres ne peuvent être utilisés pour y stocker des valeurs arbitraires ;

control qui désigne un registre de compteur ordinal, unique dans la plupart des processeurs ;

memory qui désigne un opérande présent en mémoire ;

immediate qui désigne une constante disponible dans le codage d'une instruction, notée *valeur immédiate* dans la suite ;

constant qui désigne une valeur constante dans la description de l'instruction, fixe quelque soient les autres paramètres.

5.3.3 Décoration des nœuds avec leurs types

Les nœuds sont ensuite décorés avec leur types. Le type d'un nœud représente le type de la donnée à la sortie de ce nœud. Les types sont inférés depuis ceux des nœuds fils, et des modification éventuelles apportées par le nœud courant. Par exemple, le nœud (ADD) reçoit deux entiers non signés de 32 bits, et produit donc un résultat du même type. En revanche, les nœuds (ZX) reçoivent un entier signé sur 32 bits, mais produisent un entier non signé de la même taille. Le résultat de cette passe sur l'arbre de l'instruction addu est donné par le listing 5.8.

```
1  (SEQ
2    (ABSTRACT_DST Operand-mips-rd type:[field, 32] op_nat:register
3      (I2F type:[field, 32]
4        (ADD type:[unsigned, 32]
5          (ZX type:[unsigned, 32]
6            (F2I type:[signed, 32]
7              (ABSTRACT_OP Operand-mips-rs
8                           type:[field, 32]
9                           op_nat:register)))
10         (ZX type:[unsigned, 32]
11           (F2I type:[signed, 32]
12             (ABSTRACT_OP Operand-mips-rt
13                          type:[field, 32]
14                          op_nat:register)))))))
```

Listing 5.8 – Le comportement de l'instruction addu après la passe type_deco

5.3.4 Uniformisation de l'arbre

Pour faciliter la mise en correspondance des comportements, et pour se rapprocher d'une forme canonique, la dernière passe expand_op simplifie et effectue quelques modifications sur les nœuds de l'arbre. À l'issue de cette passe, nous obtenons la représentation *étendue* du comportement de l'instruction. Nous la nommons ainsi car tous les nœuds complexes sont explosés en nœuds plus simples. Voici quelques exemples d'opérations effectuées par cette passe :

- Les nœuds (ZX) et (SX) sont supprimés s'ils n'effectuent aucune opération d'un point de vue du matériel. Dans notre exemple, les deux nœuds (ZX) sont supprimés car ils ne réalisent pas d'extension de zéros. Ils modifient juste le type des données dans l'arbre. Maintenant que l'information de type a été propagée, ils ne sont plus nécessaires. Lors de leur suppression, leur type est propagé dans leur nœud fils.

– En revanche, si ils effectuent effectivement une opération d'extension de zéro ou de signe, ils sont remplacés par leur équivalent en opérations logiques. Un nœud (ZX) est remplacé par un nœud (AND) entre le nœud fils de (ZX) et la constante (1 << n) - 1, où n est la taille du nœud fils. Un nœud (SX) est lui remplacé par un nœud (SHL) (décalage à gauche) entre le fils de (SX) et la constante n, et par un (SHRA) (décalage arithmétique à droite) entre le résultat du (SHL) et n. L'opération réalisée au final est donc data = ((data << n) >> n);
– Un nœud (NOT) qui effectue l'opération "non binaire" sur un entier, est remplacé par un nœud (XOR) entre l'entier et la constante -1;
– Un nœud (NEG) qui inverse le signe d'un entier est remplacé par un nœud (SUB) entre la constante 0 et l'entier ;
– Si un nœud (F2I) est directement fils d'un nœud (I2F), un nouveau nœud (IOR) (opération "ou logique") est inséré entre les deux. Le nœud (F2I) devient le premier fils de (IOR), et la constante 0 est insérée comme deuxième fils de (IOR) ;
– Certaines opérations comme le décalage à droite ou la division ne sont pas les mêmes suivant le type des données sur lesquelles elles travaillent. Dans ce cas, les nœuds sont modifiés pour indiquer explicitement quelle opération est réalisée. Ainsi, si le type d'un nœud (SHR) (décalage à droite) est signé, alors il est remplacé par un nœud (SHRA) (décalage arithmétique à droite). De la même manière, un nœud (DIV) non signé est remplacé par un nœud (DIVU) (division non signée).

```
1  (SEQ
2    (ABSTRACT_DST Operand-mips-rd type:[field, 32] op_nat:register
3      (I2F type:[field, 32]
4        (ADD type:[unsigned, 32]
5          (F2I type:[unsigned, 32]
6            (ABSTRACT_OP Operand-mips-rs
7                         type:[field, 32]
8                         op_nat:register))
9          (F2I type:[unsigned, 32]
10           (ABSTRACT_OP Operand-mips-rt
11                        type:[field, 32]
12                        op_nat:register))))))
```

Listing 5.9 – Le comportement de l'instruction addu après la passe expand_op

Les arbres ainsi obtenus sont décris sous la forme étendue et ce afin de faciliter la future mise en correspondance. Ils ne sont cependant pas encore idéaux pour celle-ci car :
– Le mode de représentation field n'est pas utile car il n'apporte pas d'information supplémentaire ;
– Les informations de manipulation de données et de contrôle de flot sont mélangées ; En effet, si une instruction est conditionnelle, son comportement va alors mélanger nœuds de contrôle et de manipulation de données. Nous verrons section 5.5 qu'il est souhaitable d'isoler les traitements sur les données des actions de contrôle dans le contexte de la traduction binaire dynamique ;
– Si l'instruction réalise plusieurs opérations indépendantes, celles-ci sont mélangées au sein de l'arbre.

Pour toutes ces raisons, il est nécessaire de réaliser une dernière étape sur les comportements afin d'en extraire la forme la plus simple du traitement réalisé par l'instruction sur les données.

5.4 Extraction des signatures des instructions

Cette dernière étape avant la mise en correspondance consiste à extraire la *signature* de chaque instruction. Chaque signature est composée d'un ou plusieurs *effets* qui représentent chacun une modification unitaire de l'état de la machine. Dans un effet, chaque nœud correspond à une opération réalisée sur une donnée par l'instruction. Par ailleurs, le mode de représentation `field` est supprimé pour ne garder que l'information de lecture ou d'écriture d'un opérande. Enfin, chaque résultat intermédiaire est stocké dans un opérande intermédiaire qui sera utile lors de la mise en correspondance.

La signature de l'instruction addu directement extraite de son arbre de comportement est donnée au listing 5.10. On peut y voir les deux opérandes d'entrée de nature registre de travail, l'opération d'addition, et l'opérande de sortie lui aussi de nature registre de travail.

```
1 (op_out Reg Storage-mips-GPRS Operand-mips-rd
2        type:[unsigned, 32] op_nat:register
3     (ADD type:[unsigned, 32]
4         (op_in Reg Storage-mips-GPRS Operand-mips-rs
5                type:[unsigned, 32] op_nat:register)
6         (op_in Reg Storage-mips-GPRS Operand-mips-rt
7                type:[unsigned, 32] op_nat:register)))
```

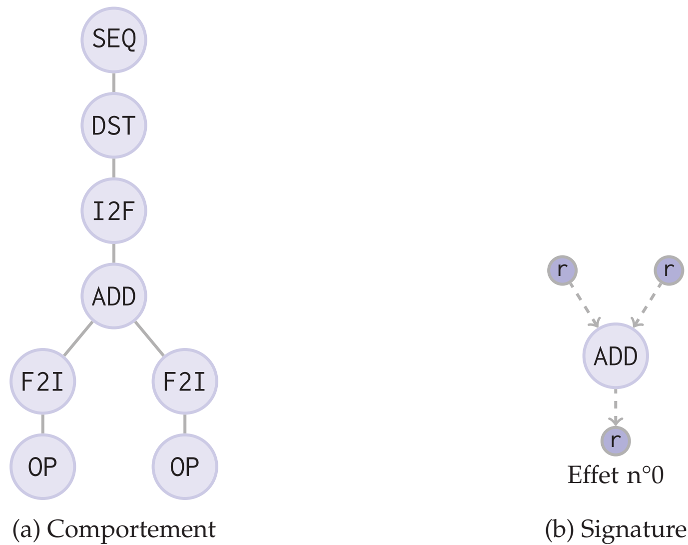
Listing 5.10 – La signature de l'instruction addu avec son seul effet

Cependant pour plus de clarté, nous choisissons dans les exemples suivant, de représenter les signatures sous forme d'arbres. La légende utilisée pour la nature des opérandes dans ces arbres est donnée par la figure 5.2. On y retrouve les natures d'opérandes présentées précédemment, avec l'opérande intermédiaire en plus.

- (r) Opérande de nature registre de travail
- (s) Opérande de nature registre spécial
- (c) Opérande de nature registre de contrôle
- ( ) Opérande intermédiaire
- (i) Opérande de nature immédiate
- (m) Opérande de nature accès mémoire
- ( ) Opérande de nature constante

FIGURE 5.2 – Légende des nœuds des signatures

La représentation sous forme d'arbre du comportement et la signature de l'instruction addu sont données par la figure 5.3. Le comportement de la figure 5.3a est le même que celui du listing 5.6. Il en va de même pour la signature de la figure 5.3b qui est la représentation sous forme d'arbre du listing 5.10. La représentation des arbres a été simplifiée au maximum dans un souci de lisibilité. Seule la nature des nœuds est indiquée, leur type est masqué. Pour les signatures, le code couleur de l'opérande donne sa nature. On voit ainsi



(a) Comportement (b) Signature

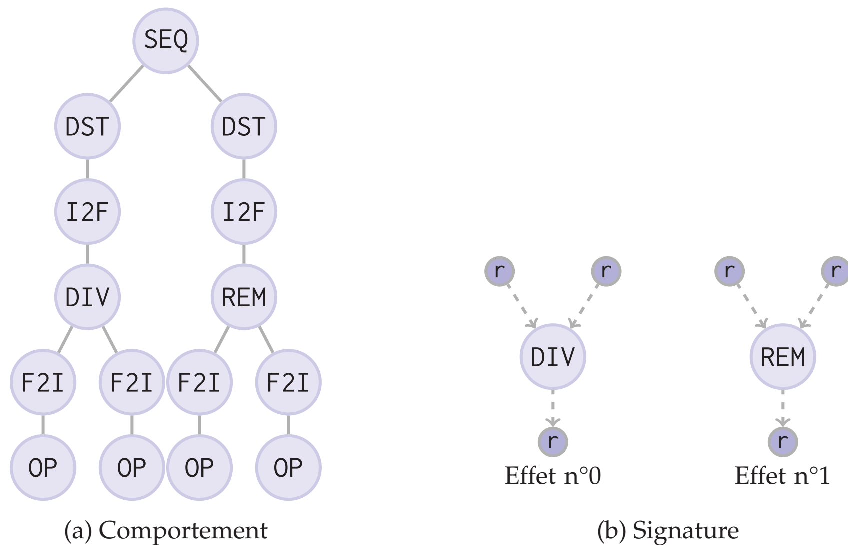
FIGURE 5.3 – Instruction `addu`

pour l'instruction `addu` qu'elle lit deux registres à usage général et écrit son résultat dans un troisième.

Nous allons maintenant passer en revue quelques exemples de signatures d'instructions MIPS afin de mettre en avant certaines particularités de l'extraction des signatures.

5.4.1 Exemple de l'instruction `div` avec plusieurs effets

L'instruction `div` du MIPS réalise à la fois une division et un calcul de reste. Elle stocke le résultat de la division dans le registre `lo` et le reste dans `hi`. Le comportement après analyse



(a) Comportement (b) Signature

FIGURE 5.4 – Instruction `div`

est donné par la figure 5.4a. On peut y voir la division et le calcul du reste. La signature

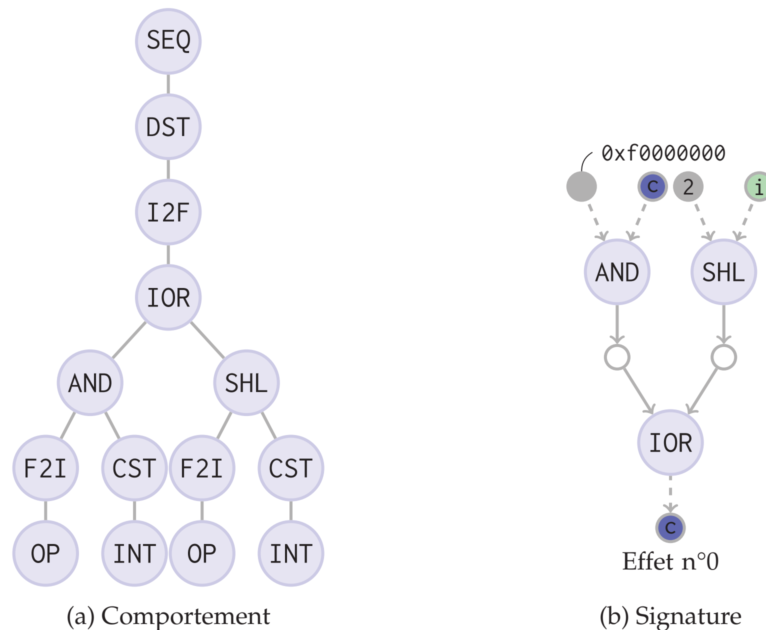
donnée par la figure 5.4b possède donc deux effets indépendants, un pour chaque opération.

5.4.2 Exemple de l'instruction j avec registre de contrôle

L'instruction j du MIPS réalise un saut absolu inconditionnel. Il possède une dynamique de saut de 2^{28} bits, les 4 bits de poids fort du compteur ordinal avant branchement étant conservés. La destination du saut peut donc être calculée ainsi :

$$\texttt{pc} \leftarrow (\texttt{target} \ll 2) \vee (\texttt{pc} \wedge \texttt{0xf0000000})$$

Avec `target` l'adresse du saut sur 26 bits, sans les deux bits de poids faible. La figure 5.5



(a) Comportement

(b) Signature

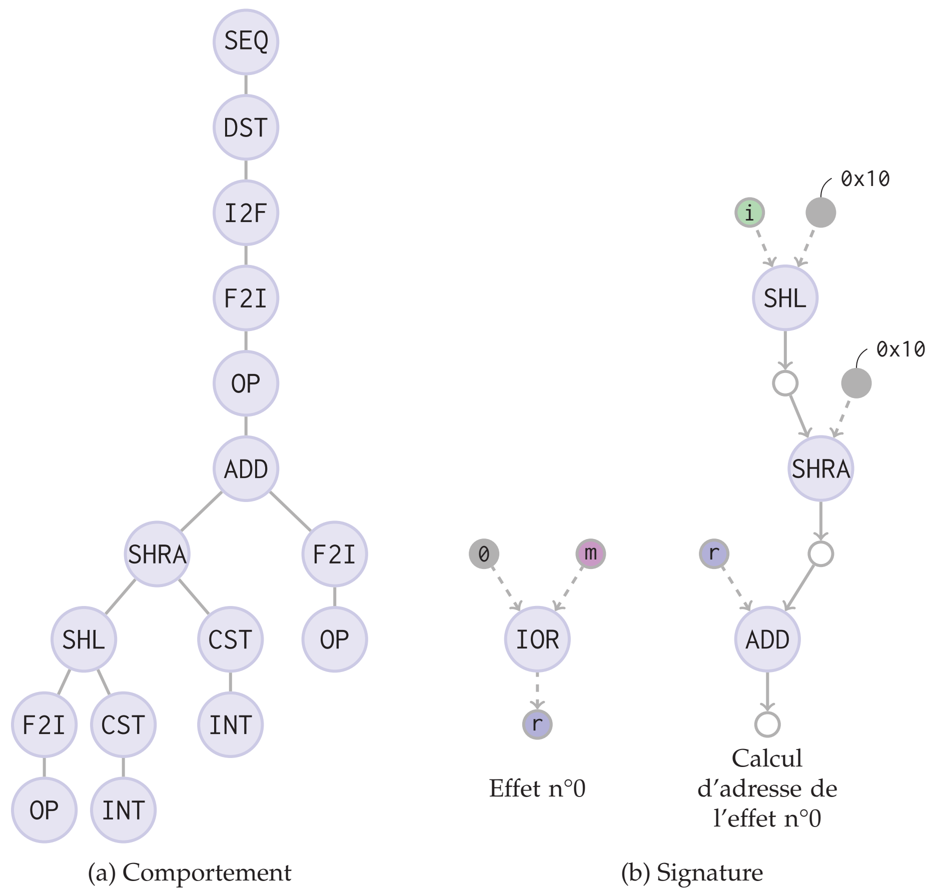
FIGURE 5.5 – Instruction j

donne le comportement de j après analyse, et la signature correspondante. Ici, on voit que l'opérande écrit est un registre de contrôle, puisque cette instruction est une instruction de saut.

5.4.3 Exemple de l'instruction lw avec calcul d'adresse

L'instruction `lw` de chargement mémoire du MIPS fonctionne selon un adressage de type *indirect + déplacement*. Par exemple, l'instruction `lw $5, 8($3)` réalise un chargement mémoire à l'adresse `sign_ext`(8) + `$3` et stocke le résultat dans `$5`.

Lorsqu'un comportement effectue un chargement ou une écriture mémoire, un opérande de type mémoire est créé dans la signature correspondante. Cependant, les opérandes mémoires sont gérés de manière particulière dans une signature. En effet, le calcul d'adresse est considéré comme étant un effet à part, ne faisant pas partie de l'effet principal. La figure 5.6 illustre le comportement après analyse et la signature de l'instruction `lw`. La signature possède deux effets, l'un étant le chargement mémoire en lui-même (le nœud `(IOR)` étant introduit par la passe `expand_op` pour signifier qu'un simple mouvement de données a lieu ici), et l'autre le calcul d'adresse de l'opérande mémoire.



(a) Comportement (b) Signature

FIGURE 5.6 – Instruction `lw`

Cette séparation entre l'effet et le calcul d'adresse permet de garder une cohérence d'un point de vue des opérandes. Pour tous les autres opérandes, l'information d'emplacement de la donnée est présente statiquement dans le nœud opérande et ne demande aucun calcul dynamique (c.-à-d. à l'exécution). En revanche, pour un accès mémoire, l'adresse doit souvent être calculée au moment de l'exécution. C'est pourquoi le nœud opérande possède son propre effet pour ce calcul. Ce n'est pas un effet classique dans le sens où il ne modifie pas l'état observable de la machine en tant que tel, mais vient "paramétrer" le nœud opérande d'accès mémoire.

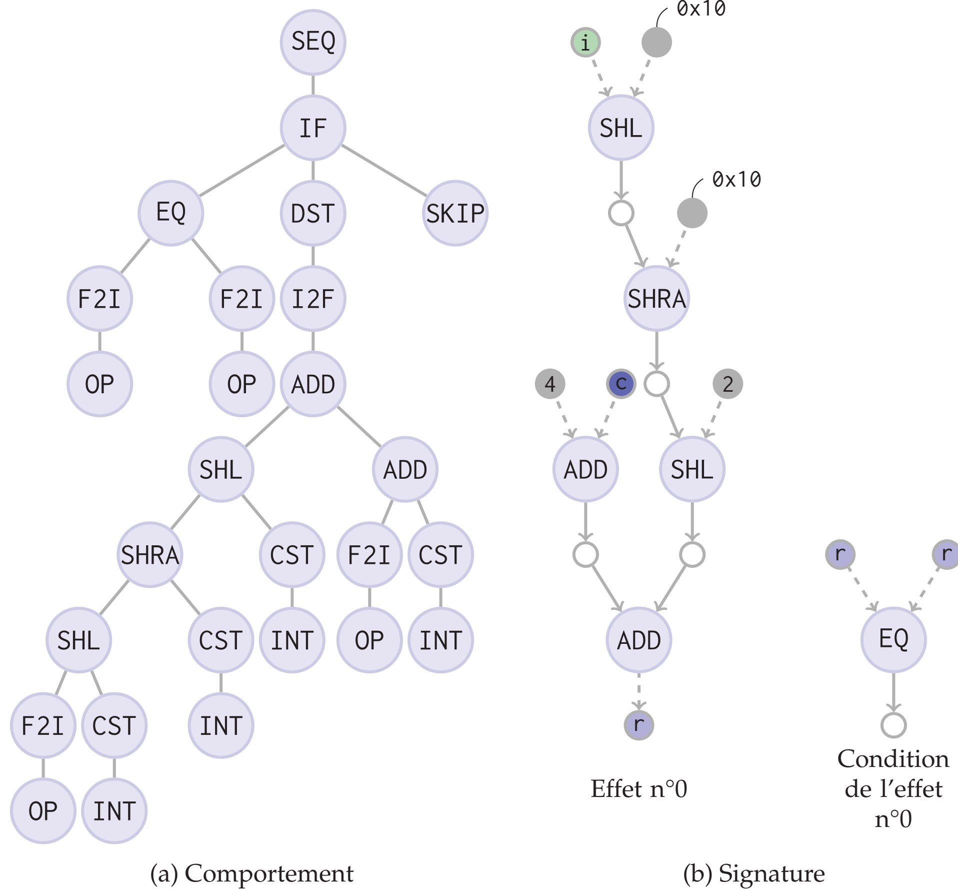
5.4.4 Exemple de l'instruction beq avec condition

Le dernier exemple est l'instruction `beq` du MIPS réalisant un branchement relatif conditionnel. La destination du saut est calculée ainsi :

$$\texttt{pc} \leftarrow (\texttt{sign_ext}(\texttt{target}) \ll 2) + (\texttt{pc} + 4)$$

Avec `target` une valeur immédiate de 16 bits, étendue de signe sur 32 bits. Elle est conditionnée par l'égalité de deux registres de travail. Par exemple, l'instruction `beq $3, $4, foo` n'effectuera un saut à l'adresse désignée par `foo` que si `$3` et `$4` sont égaux.

Son comportement après analyse et sa signature sont donnés par la figure 5.7. Le com-



(a) Comportement (b) Signature

FIGURE 5.7 – Instruction beq

portement décrit la condition grâce au nœud (IF). La première branche de ce nœud correspond à la condition testée. La deuxième branche contient le sous-arbre du comportement si la condition est vraie et la troisième le sous-arbre du comportement si la condition est fausse. Dans le cas présent, la troisième branche ne possède qu'un nœud (SKIP) signifiant que l'instruction ne fait rien dans ce cas.

Pour obtenir la signature correspondante, les conditions sont extraites des effets. Dans cet exemple, l'effet n'est appliqué que s'il remplit la condition. Il est donc marqué comme étant conditionnel, et sa condition est décrite à part. Si la troisième branche possédait un autre effet, celui-ci apparaîtrait dans la signature en tant que tel, conditionné par la même condition mais inversée.

Les effets ainsi obtenus ne contiennent aucun nœud de contrôle de flot. En traitant ainsi l'ensemble des instructions d'une architecture, on obtient une base de signatures que l'on peut mettre facilement en correspondance avec une autre base.

5.5 Mise en correspondance des instructions

Après avoir extrait les signatures de l'ensemble des instructions des architectures cible et hôte, il est possible de mettre ces deux bases de signatures en correspondance. L'idée est de

trouver, pour chaque instruction cible, une ou plusieurs instructions hôtes pour la simuler.

Avant d'exposer l'algorithme de correspondance, il est nécessaire de définir des règles qui vont régir ces correspondances. En effet, la cible et l'hôte n'ont pas une relation symétrique en DBT. Si l'on prend deux instructions i_0 et i_1 appartenant respectivement aux architectures a_0 et a_1, le fait que i_0 soit capable de simuler i_1 n'implique pas nécessairement que i_1 peut simuler i_0.

Cela est dû aux trois contraintes suivantes dues à la DBT :

– La machine cible est simulée dans son intégralité. Il faut donc simuler les instructions cibles privilégiées, qui modifient l'état des fonctions "systèmes" du processeur comme la gestion de la mémoire virtuelle, des interruptions, etc. Cependant, la simulation d'une instruction cible privilégiée ne va en aucun cas se traduire en une instruction hôte privilégiée, cela n'aurait pas de sens. Par exemple, ce n'est pas parce que le processeur simulé modifie sa table des pages que le processeur hôte doit en faire de même. Le simulateur s'exécutant en général en tant que programme utilisateur au sein d'un système d'exploitation, il ne pourrait de toute façon pas effectuer de telles actions sur le processeur hôte.
– De la même manière, un accès mémoire réalisé par le processeur cible ne peut être traduit directement. L'espace d'adressage de la machine cible étant simulé, le simulateur doit intervenir pour faire correspondre l'adresse de l'accès à l'emplacement mémoire sur la machine hôte. Cette correspondance n'est jamais directe. De plus, si le simulateur modélise la MMU du processeur cible, il doit consulter l'état de celle-ci pour effectuer la traduction d'adresse.
– Finalement, la granularité du TB fait que la traduction d'une instruction de saut ne peut se traduire directement par un branchement du côté de l'hôte. La première raison est la même que pour les accès mémoire, l'adresse de destination du branchement ne peut correspondre directement avec l'adresse à laquelle est stocké le code traduit. De plus, un saut provoque une recherche du TB suivant, TB qui peut être non encore traduit. Toute cette mécanique est gérée par le simulateur mais cela implique qu'un branchement ne peut être traduit directement.

Il est donc nécessaire d'établir des règles précises concernant la correspondance entre instructions pour respecter ces contraintes imposées par la DBT et plus généralement par la simulation d'un processeur.

5.5.1 Correspondances entre nœuds

Nous commençons par décrire la correspondance au grain le plus fin : la correspondance entre les nœuds qui composent les effets.

Tout d'abord, nous introduisons la notion de *simulation* d'un nœud par un autre. Si un nœud cible peut être mis en correspondance avec un nœud hôte, alors on dira que le nœud hôte *simule* le nœud cible.

Une règle commune à toutes correspondances entre nœuds

Cette première règle est valable pour toutes correspondances entre deux nœuds, quelles que soient leurs natures. Elle est énoncée ainsi :

Pour qu'un nœud n_0 simule un nœud n_1, il est nécessaire que la taille du type associé à n_0 soit supérieure ou égale à celle du type associé à n_1. (Règle 1)

Cette règle assure que le nœud hôte sera toujours suffisamment “spacieux” pour accueillir ou traiter la ou les données simulées. Elle est nécessaire pour qu'un nœud en simule un autre, mais pas suffisante. Le reste des règles dépend de la nature des nœuds.

La correspondance entre opérandes

La correspondance entre nœuds de nature opérande est régie par plusieurs règles. La règle suivante régit la correspondance avec un opérande de nature registre de travail :

Un nœud opérande de nature registre de travail simule n'importe quel nœud opérande sous réserve de respect de la règle 1. (Règle 2)

Cette règle signifie qu'un opérande cible pourra toujours être mis en correspondance avec un opérande hôte de nature registre de travail, si ce dernier est suffisamment grand. Cela illustre le fait qu'en DBT, les manipulations de données sont effectuées dans des registres hôtes, le *backend* ayant préalablement effectué une allocation de registres.

La règle suivant régie la correspondance entre nœuds intermédiaires :

Un nœud opérande de nature intermédiaire simule un nœud opérande de nature intermédiaire sous réserve de respect de la règle 1. (Règle 3)

Cette règle autorise les correspondances entre nœuds intermédiaires à condition que le nœud hôte soit suffisamment spacieux pour accueillir la donnée du nœud cible.

Vient ensuite la règle interdisant l'utilisation des opérandes mémoires, des registres spéciaux et de contrôle sur l'hôte :

Un nœud opérande de nature accès mémoire, registre spécial ou registre de contrôle ne simule aucun nœud. (Règle 4)

Cette règle illustre le fait que les accès mémoires sur l'hôte ne sont jamais provoqués par la traduction directe d'une instruction pour les raisons de traduction d'adresses exprimées précédemment. Un opérande mémoire hôte n'est donc jamais considéré pour la correspondance. De plus, la machine cible n'ayant pas de raison de modifier l'état de la machine hôte, les registres spéciaux et de contrôle de l'hôte ne sont pas utilisés lors de la mise en correspondance.

Finalement, les opérandes valeurs immédiates et les constantes de l'hôte peuvent être utilisées sous certaines conditions définies par les deux dernières règles :

Un nœud opérande de nature immédiate simule un nœud opérande de nature immédiate ou constante sous réserve de respect de la règle 1. (Règle 5)

Un nœud opérande de nature constante simule un nœud opérande de nature constante si les constantes associées à ces deux nœuds sont égales. (Règle 6)

Ces règles sont résumées par la figure 5.8. Sur cette dernière, une flèche entre deux opérandes signifie que l'un est candidat pour simuler l'autre, sous réserve de respect des règles indiquées.

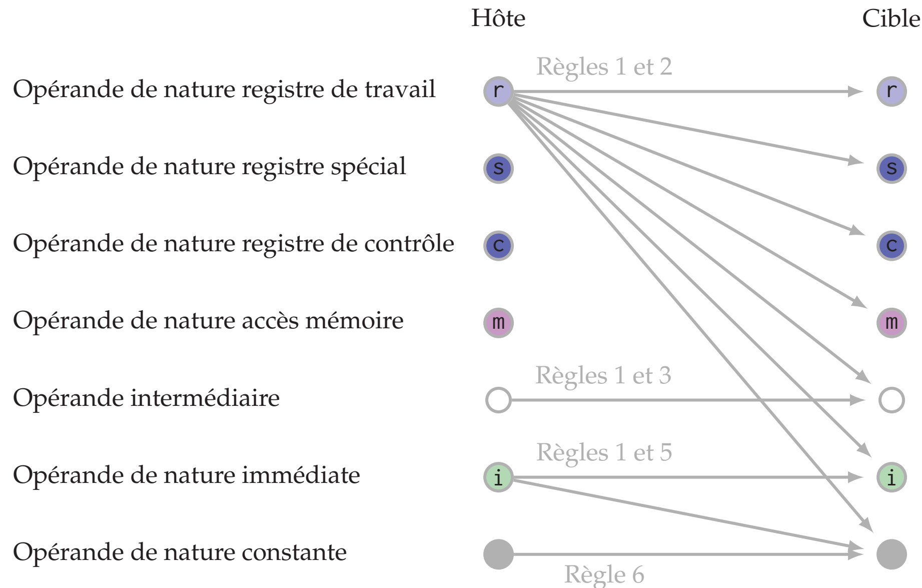


FIGURE 5.8 – Opérandes hôtes candidats pour simuler les opérandes cibles sous réserve de respect des règles de correspondance

La correspondance entre nœuds opérations

La forme étendue des nœuds opérations dans les comportements nous permet de simplifier au maximum la correspondance entre ces nœuds :

un nœud opération simule un autres nœud opération si la règle 1 est respectée et si l'opération est la même pour les deux nœuds. (Règle 7)

Cette règle fait l'hypothèse qu'un nœud opération peut simuler un autres nœud opération de même nature mais avec un type plus petit. C'est par exemple le cas pour une addition sur des données de 32 bits, qui peut être utilisée pour simuler une addition sur des données de 16 bits. Si cette propriété est fausse pour une opération, alors cela signifie que la taille de l'opération fait pleinement partie de sa nature, et la passe *expand_op* se sera chargée de spécialiser l'opération pour rendre cette information explicite dans sa nature. Par exemple, l'opération `(CLZ)` qui compte le nombre de bits de poids forts consécutifs à 0 dans un mot est dépendante de la taille du mot. Une opération `(CLZ)` sur un mot de 16 bits ne peut être simulée par une opération `(CLZ)` sur un mot de 32 bits. Ces opérations sont donc transformées respectivement en `(CLZ16)` et `(CLZ32)` par la passe *expand_op* pour signifier explicitement qu'il s'agit d'opérations différentes.

Pondération des correspondances entre nœuds

En prévision de la mise en correspondance des instructions, nous définissons une évaluation des correspondances entre nœuds. Cette évaluation permet de classer qualitativement les correspondances, et de pouvoir choisir la plus intéressante en terme de performance de simulation en cas de multiples correspondances hôtes pour la même instruction cible.

Si un nœud n_0 simule un nœud n_1, alors on définit la fonction $eval_{noeud}$ de la manière suivante :

$$eval_{noeud}(n_0, n_1) = \alpha(\ taille(n_0) - taille(n_1)\ ) + \beta(\ dist(n_0, n_1)\ )$$

Elle associe au couple (n_0, n_1) un entier positif ou nul reflétant la qualité de la correspondance du couple. Plus le résultat est proche de 0, meilleure est la correspondance. Elle est composée de deux termes :

- la différence entre les tailles des types associés à n_0 et n_1. Ce terme pénalise les correspondances entre nœuds de tailles éloignées. Cela permet par exemple de mettre en avant une instruction hôte d'addition sur 32 bits pour simuler une instruction cible d'addition de la même taille, plutôt qu'une addition 64 bits, et ce afin d'éviter les éventuelles conversions de types, extensions de signe ou de zéros, et autres troncatures ;
- la distance entre n_0 et n_1. Cette distance pénalise la correspondance entre certains opérandes, en suivant la règle présentée par la figure 5.9. Ainsi, plus la distance entre deux opérandes est grande, plus la correspondance est coûteuse. Pour toutes les natures non spécifiées sur la figure, la distance est nulle. Ici, la distance entre un registre de travail et un opérande valeur immédiate vaut 1, alors que celle entre un registre de travail et une constante vaut 2.

Nature	Registre de travail Registre spécial Registre de contrôle Accès mémoire	Immédiate	Constante
Distance	0	1	2

FIGURE 5.9 – Distance entre les opérandes

Chaque terme est pondéré par une constante (α et β) pour permettre de privilégier l'un ou l'autre des termes.

5.5.2 La correspondance entre effets

Maintenant que nous avons défini les règles de correspondance entre nœuds, nous pouvons définir les règles de correspondance entre effets. Pour ces derniers, nous définissons deux types de correspondances, les *correspondances totales* et les *correspondances partielles*.

Correspondance totale

Un effet e_0 simule totalement un effet e_1 si pour chaque nœud n_i de e_0, e_1 possède un nœud n_j positionné de la même manière dans e_1 que n_i dans e_0, et n_i simule n_j. (Règle 8)

Cette règle impose que les deux effets soient semblables en terme de structure. Si les deux effets n'ont pas le même nombre de nœuds, ou si ces nœuds sont positionnés différemment dans l'un et dans l'autre, ils ne pourront être mis en correspondance. Si leurs structures sont similaires, alors il faut que les nœuds puissent être mis en correspondance deux à deux. On parle alors de correspondance totale entre deux effets. Si e_0 est en correspondance totale avec e_1, alors e_0 peut être utilisé pour simuler e_1. La figure 5.10 illustre une correspondance totale entre deux effets.

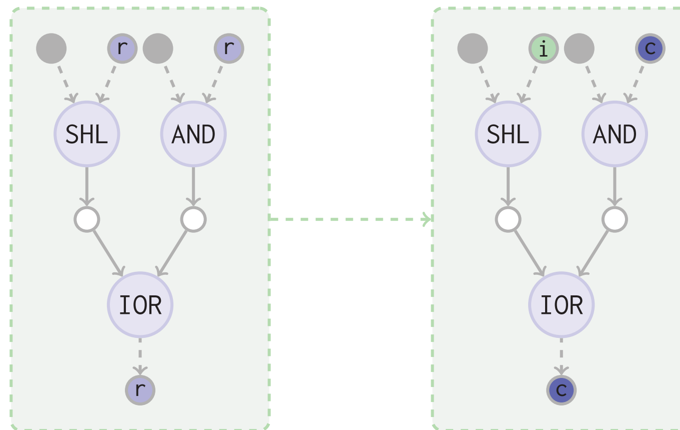


FIGURE 5.10 – Correspondance totale entre deux effets

Pour évaluer une correspondance totale entre deux effets, on définit la fonction $eval_{\text{totale}}$ sur deux effets e_0 et e_1 comme suit :

$$eval_{totale}(e_0, e_1) = \sum_{(n_i, n'_i) \in (e_0, e_1)} eval_{noeud}(n_i, n'_i)$$

Elle est définie comme la somme de toutes les évaluations des nœuds en correspondance deux à deux dans e_0 et e_1.

Correspondance partielle

Un effet e_0 simule partiellement un effet e_1 si e_0 simule totalement un sous-arbre de e_1. (Règle 9)

Une correspondance est donc partielle lorsqu'un effet e_0 ne suffira pas pour simuler un effet e_1 en entier. Il pourra néanmoins être utilisé pour en simuler une partie. Sur la figure 5.11, l'effet de gauche simule partiellement l'effet de droite.

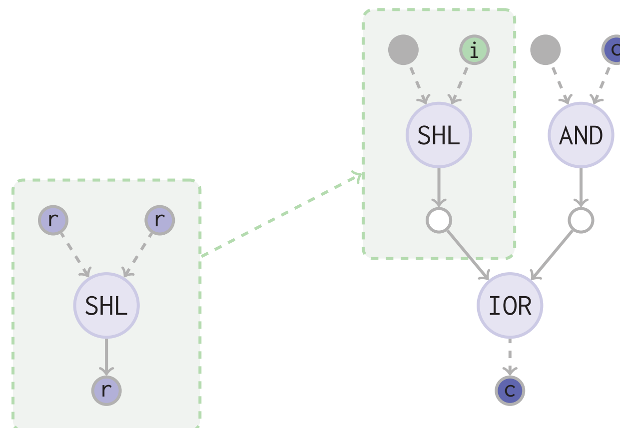


FIGURE 5.11 – Correspondance partielle entre deux effets

La fonction d'évaluation est similaire à celle de la correspondance totale mais rajoute un coût constant γ pour pénaliser les correspondances partielles devant les correspondances totales. Elle est définie comme suit :

$$eval_{partielle}(e_0, e_1) = \gamma + \sum_{(n_i, n'_i) \in (e_0, e_1)} eval_{noeud}(n_i, n'_i)$$

5.5.3 La correspondance entre signatures

Une signature pouvant être composée de plusieurs effets, il faut aussi définir des règles de correspondance entre les signatures. D'un point de vue de la signature d'une instruction cible, tous les effets doivent être simulés pour simuler complètement l'instruction. Si une signature d'une instruction cible possède un calcul d'adresse, celui-ci est mis en correspondance au même titre que les autres effets. Si elle possède un effet conditionnel, ce dernier est mis en correspondance sans prendre en compte sa condition. La condition sera gérée à l'aide du *runtime* minimal décrit section 5.6.1.

Pour qu'une signature en simule une autre, chacun des effets peut être utilisé pour simuler un autre effet à condition que l'instruction ne cause pas d'effets de bord indésirables sur l'hôte. C'est la dernière règle de correspondance que nous définissons :

Une signature s_0 simule une signature s_1 si et seulement si au moins un effet de s_0 simule un effet de s_1 et aucun des effets de s_0 n'écrit dans un opérande de nature registre spécial, registre de contrôle ou accès mémoire. (Règle 10)

Avec cette règle, toutes les instructions de type "systèmes" ainsi que les accès mémoire et les branchements de l'hôte sont éliminés des correspondances.

5.5.4 Algorithme de mise en correspondance

Toutes ces règles de correspondance étant définies, nous pouvons exposer l'algorithme qui va réaliser la base de correspondances entre une architecture cible et une architecture hôte. Il prend en entrée l'ensemble des signatures de la cible et de l'hôte, et produit des correspondances entre ces signatures. Pour chaque signature cible, il peut produire zéro, une ou plusieurs correspondances. Une correspondance peut être composée d'une signature hôte si la correspondance est totale, ou de plusieurs signature hôtes si elles sont partielles. Si l'algorithme produit zéro correspondance pour une signature cible, cela signifie qu'il n'est pas parvenu à trouver d'instruction hôte pouvant simuler l'instruction cible concernée. Ce cas ne doit pas arriver dans un processus de génération complètement automatisé. Si plusieurs correspondances sont trouvées pour la même instruction cible, l'évaluation de chacune d'entre elles permettra de sélectionner la meilleure selon les règles d'évaluations définies précédemment. Ce processus de correspondances est décrit en trois parties données par les algorithmes 5, 6 et 7.

Notations

Avant de présenter l'algorithme, nous introduisons quelques notations :

- Si e est un effet, alors il est représenté par un arbre dont le nœud racine seul est désigné par $e|_{root}$;

– Si e_i et e_j sont deux effets et que e_j est un sous-arbre de e_i, alors la différence entre eux notée $e_i - e_j$ produit un ensemble d'effets. Le résultat est l'ensemble des sous-arbres de e_i produit lorsqu'on prive ce dernier du sous-arbre e_j mais en conservant les nœuds intermédiaires de e_i s'ils correspondent à la racine et aux feuilles de e_j. Si $e_i = e_j$, alors cet ensemble est vide. Si e_j n'est pas un sous-arbre de e_i alors cette opération n'est pas définie. La figure 5.12 illustre cette opération entre deux arbres ;
– Une sous-correspondance est un couple (e_i, e_j) avec e_i et e_j deux effets en correspondance totale ;
– Une correspondance m est un triplet de la forme $(m|_{final}, m|_{exp}, m|_{weight})$ tel que :
  – $m|_{final}$ est l'ensemble des sous-correspondances finales pour e_t. Il s'agit de couples (e', e_h) avec e_h un effet hôte et e' un sous-arbre de e_t avec lequel e_h est en correspondance totale ;
  – $m|_{exp}$ est l'ensemble des sous-arbres de e_t restant à explorer. Il est vide lorsque l'algorithme est terminé si ce dernier est parvenu à couvrir e_t totalement ;
  – $m|_{weight}$ est le poids de la correspondance.

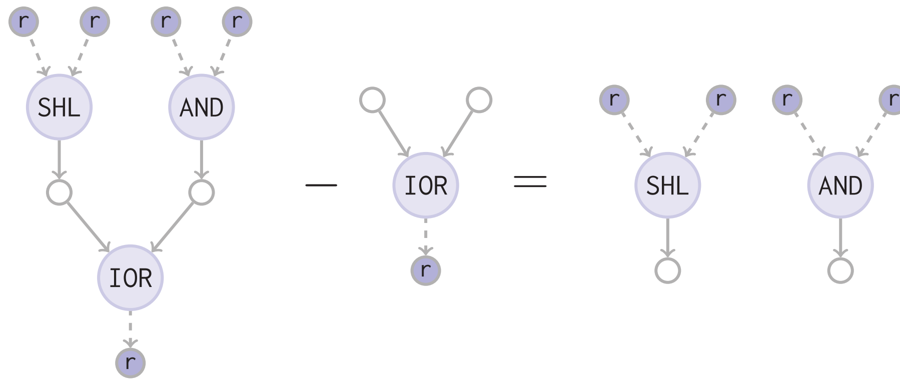


FIGURE 5.12 – Différence entre deux effets

Point d'entrée de l'algorithme

L'algorithme 5 est le point d'entrée de la mise en correspondance. Il reçoit l'effet cible e_t à mettre en correspondance, et la base des signatures de la machine hôte H. Il produit l_{final} comme étant l'ensemble des correspondances entre e_t et H.

Cet algorithme travaille avec une pile de correspondances à compléter l_{exp}. Chaque correspondance m dans l_{exp} possède donc un ensemble $m|_{exp}$ non vide. Cette pile est initialisée avec la correspondance m_0 contenant un ensemble de sous-correspondances finales vide, un ensemble de sous-arbres à explorer contenant e_t tout entier, et un poids nul (ligne 1). Chaque itération retire une correspondance m de l_{exp} et tente de compléter un élément e de $m|_{exp}$ avec les effets des signatures de H (lignes 4 et 5). Lors du parcours de H, si la signature hôte s_h courante contient un effet qui modifie l'état de la machine hôte, elle n'est pas considérée pour la mise en correspondance et l'algorithme passe à la signature hôte suivante (ligne 7). Cela est dû à la règle 10 qui interdit l'utilisation d'instructions hôtes modifiant l'état de la machine hôte dans une correspondance. Chaque effet e_h est mis en correspondance avec e grâce à l'algorithme 6 MATCH (ligne 9). Ce sous-algorithme retourne le sous-arbre e' de e qu'il a réussi à mettre en correspondance avec e_h, ainsi que le poids de la correspondance w. Si e' n'est pas vide, cela signifie que la mise en correspondance a réussi. Dans ce cas,

Algorithme 5 Recherche de correspondances hôtes pour une signature cible

Entrée : e_t un effet d'une signature cible.
Entrée : H la base des signatures de l'hôte
Sortie : l_{final} la liste des correspondances pour l'effet e_t

```
1  m_0 ← (∅, {e_t}, 0)              ▷ Créer une nouvelle correspondance partielle m_0 de poids 0
2  l_exp ← {m_0}                    ▷ Ajouter m_0 à la liste des correspondances partielles l_exp
3  tant que l_exp ≠ ∅ faire
4      m ← l_exp.pop()                         ▷ Retirer une correspondance partielle m de l_exp
5      e ← m|_exp.pop()                        ▷ Retirer un sous-arbre à explorer e de m
6      pour chaque s_h ∈ H faire
7          si s_h modifie l'état de la machine hôte alors continuer
8          pour chaque e_h ∈ s_h faire
9              (s, e', w) ← MATCH(e, e_h)                ▷ Mettre en correspondance e et e_h
10                                                       ▷ e' est le sous-arbre de e simulé par e_h
11                                                       ▷ w est le poids de la correspondance
12             si e' ≠ ∅ alors                 ▷ Si e_h simule totalement ou partiellement e
13                 m' ← m                                ▷ Créer m' comme une copie de m
14                 m'|_final.push((e', e_h))    ▷ Ajouter la correspondance entre e' et e_h à m'
15                 m'|_weight ← m'|_weight + w  ▷ Mettre à jour le poids de la correspondance
16                 m'|_exp.push(e − e')      ▷ Ajouter les sous-arbres de e non-couverts par e_h
17                                                 ▷ à la liste des sous-arbres à explorer de m'
18                 si m'|_exp = ∅ alors
19                     l_final.push(m')         ▷ Ajouter m' à la liste des corresp. finales l_final
20                 sinon
21                     m'|_weight ← m'|_weight + γ ∗ |e − e'|      ▷ Pénalité de correspondance
                                                       ▷ partielle valant γ fois le nombre
                                                       ▷ de sous-arbres non-couverts par e'.
22                     l_exp.push(m')         ▷ Ajouter m' à la liste des corresp. partielles l_exp
23                 fin si
24             fin si
25         fin pour
26     fin pour
27 fin tant que
28 retourne l_final
```

Algorithme 6 Correspondance entre les nœuds d'une signature cible et hôte

Entrée : e_t un nœud de type opérande
Entrée : e_h un nœud de type opérande
Sortie : s l'état de la correspondance
Sortie : e' le sous-arbre de e_t en correspondance avec e_h
Sortie : w le poids de la correspondance

```
1  fonction MATCH(e_t, e_h)
2      si ¬e_h.simule(e_t) alors retourne ("échec", ∅, 0)          ▷ échec si e_h ne simule pas e_t
3      w ← eval_noeud(e_t, e_h)

4      selon le type d'opérande de e_t faire
5          cas opérande d'entrée
6              si e_h est un opérande d'entrée alors retourne ("succès", e_t|root, w)
7              sinon retourne ("échec", ∅, 0)
8          cas opérande intermédiaire
9              si e_h est un opérande d'entrée alors retourne ("partielle", e_t|root, w)
10         cas opérande de sortie
11             si e_h n'est pas un opérande de sortie alors retourne ("échec", ∅, 0)
12     fin selon
13     e' ← e_t|root                          ▷ e' sera le sous-arbre retourné par l'algorithme.
                             ▷ Il n'est pour l'instant composé que du nœud e_t sans ses fils.
14     op_t ← fils unique de e_t
15     op_h ← fils unique de e_h
16     si ¬op_h.simule(op_t) alors retourne ("échec", ∅, 0)
17     w_op ← eval_noeud(op_t, op_h)

18     op' ← op_t|root
19     Ajouter op' comme nœud fils à e'          ▷ e' est maintenant le sous-arbre de e_t
                             ▷ composé du nœud racine et de son nœud opération fils.
20     w ← w + w_op

21     l_t ← liste de l'ensemble des nœuds fils de op_t
22     l_h ← liste de l'ensemble des nœuds fils de op_h
23     (s, l', w') ← SUBMATCH(l_t, l_h)

24     si l'opération de op_t est commutative alors
25         l_t ← liste de l'ensemble des nœuds fils de op_t pris dans le sens inverse
26         l_h ← liste de l'ensemble des nœuds fils de op_h
27         (s_com, l_com, w_com) ← SUBMATCH(l_t, l_h)
28         si s_com ≠ "échec" et w' > w_com alors       ▷ si cette correspondance est meilleure
29             (s, l', w') ← (s_com, l_com, w_com)           ▷ que la version non commutée.
30         fin si
31     fin si

32     Ajouter le contenu de l' comme nœuds fils à op'      ▷ e' est maintenant le sous-arbre
33     retourne (s, e', w + w')            ▷ de e_t en correspondance avec e_h (si s ≠ "échec")
34 fin fonction
```

Algorithme 7 Propagation de la récursion aux sous-nœuds

Entrée : l_t et l_h les listes de sous-arbres à mettre en correspondance deux à deux
Sortie : s l'état de la sous-correspondance
Sortie : l_e la liste des sous-arbres produits par les appels à MATCH
Sortie : w le poids de la sous-correspondance

```
1  fonction SUBMATCH(l_t, l_h)
2      s ← "succès"
3      l_e ← ∅
4      pour chaque nœud e_{t,c} dans l_t et chaque nœud e_{h,c} dans l_h faire
5          (s_c, e_c, w_c) ← MATCH(e_{t,c}, e_{h,c})
6          si s_c = "échec" alors retourne ("échec", ∅, 0)
7          Ajouter e_c à l_e
8          si s = "succès" alors s ← s_c        ▷ L'état "partielle"est prioritaire sur "succès"
9          w ← w + w_c
10     fin pour
11     retourne (s, l_e, w)
12 fin fonction
```

l'algorithme crée une nouvelle correspondance m' comme une copie de m (ligne 13), lui ajoute cette nouvelle sous-correspondance entre e' et e_h (ligne 14), et met son poids total à jour (ligne 15). Si e' est différent de e, cela signifie que MATCH n'est parvenu à mettre que partiellement en correspondance e et e_h. Il reste donc un ou plusieurs sous-arbres de e à explorer qui sont obtenus par la différence entre e et e' et qui sont ajoutés à $m'|_{exp}$ (ligne 16). Finalement, si $m'|_{exp}$ est vide, m' est une correspondance totale qui peut être ajoutée à l_{final}. Sinon elle est placée dans l_{exp} pour être terminée lors des prochaines itérations.

Le cœur de la mise en correspondance avec le sous-algorithme MATCH

Le sous-algorithme 6 parcourt deux effets e_t et e_h récursivement pour vérifier leur correspondance et le poids associé. Il procède en quatre étapes par niveau de récursion :

Correspondance entre opérandes La première étape (lignes 2 à 12) vérifie la correspondance entre les deux opérandes e_t et e_h. Ces deux nœuds sont les paramètres d'entrées de l'algorithme et sont garantis d'être de type opérande. Cette garantie vient du fait qu'un effet commence forcément par un opérande de sortie à la racine de son arbre, et alterne ensuite des nœuds opérations avec des nœuds opérandes, et ce jusqu'aux feuilles. L'algorithme commence par vérifier que e_h simule e_t (ligne 2). Si cette condition est fausse, la correspondance est un échec. Viennent ensuite des vérifications suivant le type d'opérandes de e_t et e_h :

- Si e_t est un opérande d'entrée, alors la récursion est arrivée à une feuille de l'effet cible et la correspondance est un succès si e_h est aussi un opérande d'entrée (ligne 6). Sinon, la correspondance est un échec car un opérande d'entrée cible ne peut pas être mis en correspondance avec autre chose qu'un opérande d'entrée hôte.
- Dans le cas où e_t est un opérande intermédiaire, le parcours s'arrête ici si e_h est un opérande d'entrée. Cela signifie que nous sommes arrivés à une feuille de la signature hôte. Il s'agit donc d'une correspondance partielle puisqu'il reste des nœuds à explorer du côté de e_t. Dans les cas où e_h n'est pas un opérande d'entrée, l'algorithme continue son parcours normalement.

– Finalement, le cas où e_t est un opérande de sortie correspond au cas où l'algorithme en est à sa première itération. e_h est forcément un opérande de sortie ici et l'algorithme continue normalement. La vérification de la ligne 11 n'est pas strictement nécessaire si on fait l'hypothèse que les effets passés en paramètres sont correctement formés.

Correspondance entre opérations La deuxième étape vérifie la correspondance entre les nœuds opérations op_t et op_h sous-jacents à e_t et e_h. Si op_h ne simule pas op_t, l'algorithme s'arrête ici (ligne 16). Sinon, il met à jour ses valeurs de retour et parcourt récursivement les fils de op_t et op_h.

Propagation de la récursion Le parcours récursif est fait dans le sous-algorithme 7 SUBMATCH qui prend en paramètres la liste des fils de op_t et celle des fils de op_h. La stratégie ici est de considérer que la correspondance est un échec si elle échoue dans au moins un fils. Sinon, elle est partielle si au moins l'une des sous-correspondances est partielle (ligne 8 de SUBMATCH). Les poids des sous-correspondances sont accumulés et retournés par SUBMATCH. Ce sous-algorithme retourne aussi la liste des sous-arbres retournés par les appels récursifs à MATCH.

Finalement, l'algorithme MATCH refait appel à SUBMATCH si op_t est une opération binaire commutative. La même opération est répétée mais en prenant cette fois-ci les nœuds fils de op_t dans le sens inverse (ligne 25).

Réunion des résultats L'algorithme MATCH se termine par la réunion des différents résultats de l'itération courante, et des itérations sur les fils. Si op_t est commutative, et que les deux versions ont pu être mises en correspondance, l'algorithme conserve le résultat avec le poids le plus faible (lignes 28 et 29). À la ligne 33, il retourne l'état courant de la correspondance (partielle ou totale), le sous-arbre mis en correspondance jusqu'à maintenant et le poids de la correspondance.

Complexité en temps de l'algorithme dans le pire-cas

Afin de déterminer la complexité temporelle de l'algorithme 5, il est nécessaire de caractériser le nombre d'appels récursifs faits à MATCH. Les paramètres de ce calcul de complexité sont :

– L'effet cible e_t contenant n nœuds opérations ;
– L'ensemble des effets hôtes H contenant m effets.

Pour pouvoir caractériser plus facilement le nombre d'appels à MATCH, nous faisons les hypothèses suivantes :

(i) Toutes les opérations sont binaires et produisent un et un seul résultat. Elles ont donc exactement deux nœuds fils qui représentent leurs deux données en entrée. Cette hypothèse est applicable car il est toujours possible d'effectuer des transformations sur un arbre de comportement pour le ramener à cette forme. Grâce à cette hypothèse, on peut affirmer que le nombre de nœuds opérandes dans un arbre à n nœuds opérations est $2n + 1$, quel que soit l'arbre ;

(ii) Tous les nœuds simulent tous les nœuds. C'est une première étape pour nous placer dans le pire-cas puisque l'algorithme s'arrête lorsqu'un nœud hôte ne simule pas le nœud cible avec lequel il est mis en correspondance et fait donc moins d'appels récursifs.

Théorème 1. *Sous les hypothèse (i) et (ii), la complexité temporelle en pire cas de notre algorithme est en* $\mathcal{O}(m^n)$.

Pour démontrer ce théorème, nous commençons par donner le lemme suivant :

Lemme 1. *Sous les hypothèses (i) et (ii), pour e_t et $e_h \in H$ fixés, l'appel* MATCH*(e_t, e_h) provoque un nombre d'appels récursifs à* MATCH *égal à*

$$g(e_t, e_h) := 2op(e_t \cap e_h) + 1 \tag{5.1}$$

avec $e_t \cap e_h$ le sous-arbre commun entre e_t et e_h, et $op(e)$ le nombre de nœuds opérations dans e.

Démonstration. Examinons le nombre d'appels récursifs provoqué par MATCH(e_t, e_h) en fonction de la forme de e_t et e_h

- si e_t et e_h ont la même forme, MATCH retournera l'état "succès" et la correspondance sera totale. Dans ce cas, le nombre d'appels récursifs à MATCH est égal au nombre de nœuds opérandes de e_t, et $e_t = e_t \cap e_h$;
- si e_h est un sous-arbre de e_t, MATCH retournera l'état "partielle". Dans ce cas, le nombre d'appels récursifs à MATCH est égal au nombre de nœuds opérandes dans e_h, et $e_h = e_t \cap e_h$;
- dans les autres cas, MATCH retournera l'état "échec". Le nombre d'appels à MATCH est alors déterminé par le nombre de nœuds opération du sous-arbre $e_t \cap e_h$.

Ainsi, dans tous les cas, le nombre d'appels à MATCH est égal au nombre de nœuds opération du sous-arbre $e_t \cap e_h$. L'hypothèse (i) donne alors l'expression (5.1). □

Nous pouvons maintenant prouver le théorème 1.

Démonstration. À l'issue d'un appel à MATCH, si la correspondance est partielle, le sous-arbre restant de e_t à explorer est remis dans la pile l_{exp} (ligne 22). Il subira à nouveau m appels à MATCH lors d'une prochaine itération de la boucle principale (ligne 3). Commençons par donner l'expression générale de la fonction f donnant le nombre total d'appels à MATCH pour une liste d'effets cibles l_{exp} et pour un ensemble d'effets hôtes H :

$$f(l_{exp}, H) := \sum_{e_t \in l_{exp}} \sum_{e_h \in H} (g(e_t, e_h) + f(e_t - e_h, H)).$$

Il s'agit, pour chaque e_t dans l_{exp} et chaque e_h dans H, de la somme du nombre d'appels à MATCH entre e_t et e_h à l'itération courante (donné par le lemme 1), et des appels aux itérations futures. Ainsi, f est récursive et se rappelle avec e_t privé de sa partie déjà mise en correspondance.

Lorsque l'algorithme effectue sa première itération, l_{exp} contient seulement e_t. La fonction f s'écrit alors

$$f(\{e_t\}, H) = \sum_{e_h \in H} \left(g(e_t, e_h) + f(e_t - e_h, H)\right). \tag{5.2}$$

Pour obtenir la complexité en pire cas, nous devons maximiser le nombre d'appels récursifs à MATCH, ce qui revient à maximiser le nombre correspondances partielles provoquées par MATCH(e_t, e_h) pour e_h dans H. C'est le cas lorsque e_h a un seul nœud opération. Considérons donc l'ensemble d'effets H_p de cardinal m tel que

$$\forall e_p \in H_p,\ op(e_p) = 1 \tag{5.3}$$

Vu la forme des éléments de H_p et l'hypothèse (ii), nous remarquons que

- Pour tout e_t et e_h dans H_p, $e_t \cap e_h = e_h$ et donc $op(e_t \cap e_h) = 1$ et $g(e_t, e_h) = 3$;

– Pour tout e_t contenant n nœuds opération, et tout e_h dans H_p, le sous-arbre $e_t - e_h$ contient $n-1$ nœuds opération ;
– Pour toutes l_{exp} et l'_{exp} listes d'effets telles que $op(l_{exp}) = op(l'_{exp})$, nous avons que $f(l_{exp}, H_p) = f(l'_{exp}, H_p)$. Cette propriété signifie que $f(\cdot, H_p)$ est entièrement déterminée par le nombre de nœuds opération dans l_{exp}. Ainsi, on peut définir la fonction f_{op} telle que $f(l_{exp}, H_p) = f_{op}(op(l_{exp}), H_p)$.

Par conséquent, pour e_t ayant n nœuds opération, nous pouvons écrire :

$$\begin{aligned} f(\{e_t\}, H_p) &= f_{op}(n, H_p) \\ &= 3m + mf(n-1, H_p) \\ &= 3m + 3m^2 + \ldots + 3m^n \\ &= 3\sum_{i=1}^{n} m^i \\ &= 3\left(\frac{m^{n+1}-1}{m-1} - 1\right) \underset{m\to+\infty}{\sim} 3m^n \end{aligned}$$

Lorsque m tend vers l'infini, $3m^n$ est un équivalent de $f(e_t, H_p)$. Nous pouvons donc en déduire que la complexité temporelle en pire-cas de notre algorithme est en $\mathcal{O}(m^n)$. □

5.5.5 Version non-exhaustive de l'algorithme

Cette complexité n'est pas acceptable si l'algorithme est utilisé sur des architectures réelles possédant de nombreuses instructions. Cependant, il s'agit ici de la version exhaustive de l'algorithme, qui trouve toutes les correspondances entre toutes les instructions. Bien qu'elle soit intéressante lors d'expérimentations, lors de la génération d'un simulateur, cette exhaustivité est inutile. Seule la meilleure correspondance d'après notre fonction d'évaluation nous intéresse, les autres étant de toute façon ignorées.

Afin de réduire drastiquement la complexité en pratique de l'algorithme, nous lui adjoignons une étape de *séparation et évaluation* (branch&bound) pour ne pas explorer toutes les solutions. La fonction d'évaluation simplifie grandement cette modification. Il suffit de conserver le poids w_{best} de la meilleurs correspondance trouvée jusqu'ici, et de n'ajouter une nouvelle correspondance partielle à l_{exp} que si son poids courant est inférieur strictement à w_{best}. Toute correspondance partielle dépassant déjà ce meilleur poids sera de toute façon moins intéressante si elle devient totale. Ces correspondances sont donc ignorées au plus tôt de la recherche dans cette version non-exhaustive.

5.5.6 Conclusion de la mise en correspondance

À la suite de ces opérations, nous obtenons une base de correspondance pour les instructions de la machine cible. Grâce à cette base, il est possible de sélectionner la ou les instructions hôtes pour simuler une instruction de la machine cible.

5.6 Génération du simulateur

La dernière étape du flot consiste à générer le simulateur. Cette génération passe par la production de l'IR spécialisée pour le couple cible – hôte, fondée sur les correspondances obtenues précédemment, ainsi que du *frontend* et du *backend* du simulateur.

Comme nous l'avons vu dans la section 5.1, le flot de génération s'appuie sur un *runtime* minimal que nous allons décrire plus en détail ici.

5.6.1 Le *runtime* minimal

Ce *runtime* représente une machine minimale que le générateur suppose avoir à disposition pour générer son *frontend*. Il est essentiellement composé des briques qui sont soit difficiles à inférer et générer depuis la description de l'architecture hôte, soit pour lesquelles la génération automatique n'apporte pas de valeur ajoutée en terme de temps de conception et/ou de qualité du code.

La figure 5.13 met en exergue les briques composant ce *runtime* minimal. Il est principalement composé de deux parties, un ensemble minimal de micro-opérations qui seront présentes quel que soit le couple cible – hôte, et un support de l'hôte composé de briques nécessaires au processus de DBT.

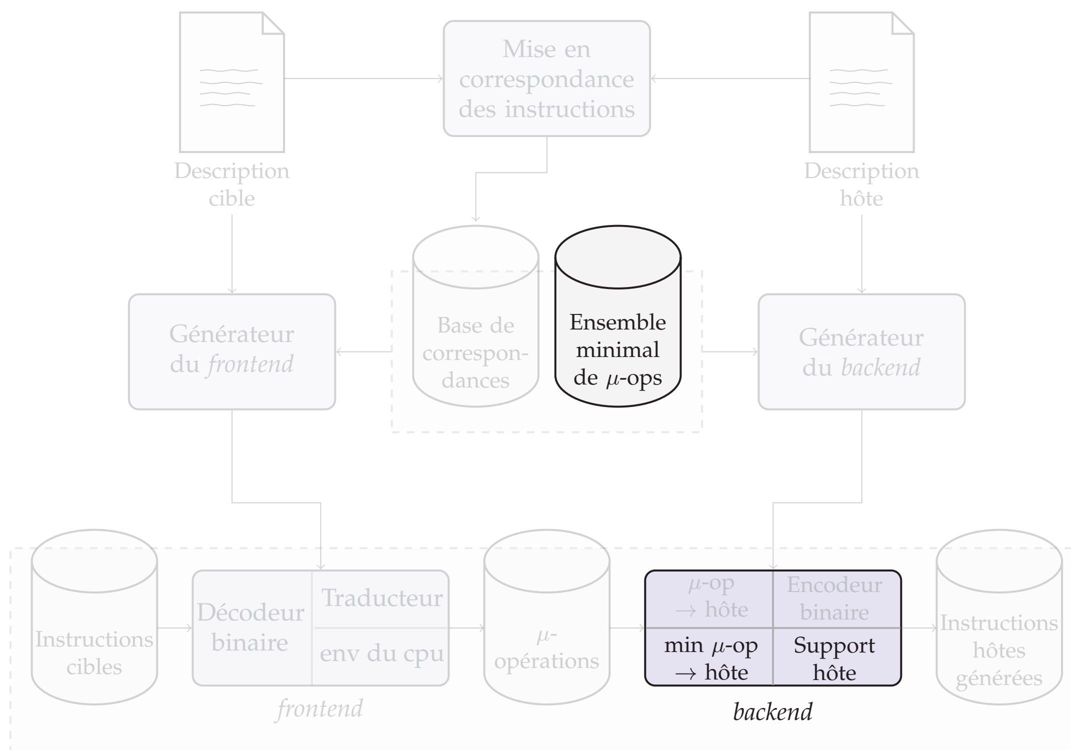


FIGURE 5.13 – Briques composant le *runtime* minimal

L'ensemble minimal des micro-opérations

L'ensemble minimal des micro-opérations contient le nécessaire pour effectuer des branchements conditionnels ou non à l'intérieur du TB courant, des chargements et des écritures dans l'environnement ou dans la mémoire de la machine cible, et pour provoquer la fin d'un TB.

Les branchements permettent de prendre en charge les effets conditionnels de la machine cible. Toute condition peut ainsi être évaluées grâce à ces micro-opérations et à des

constructions de la forme `if...else...`;

Les accès à l'environnement de la machine cible permettent notamment de lire et d'écrire les registres simulés de la machine cible et donc de prendre en charge les nœuds opérandes de nature registre dans les signatures ;

Les accès à la mémoire cible permettent la prise en charge des opérandes mémoires dans les signatures de la machine cible. Ces micro-opérations permettent des accès à différentes granularités suivant la taille du nœud ;

La fin d'un TB est provoquée lorsque le compteur ordinal de la machine cible est modifié.

Le support de l'hôte

Le support de l'hôte rassemble les briques qui sont difficiles à inférer à partir de la description hôte, ou à générer de manière efficace. Elles sont donc implémentées manuellement par le concepteur. Parmi ces briques, on trouve la traduction de certaines micro-opérations de l'ensemble minimal, comme les branchements conditionnels, l'accès à la mémoire cible et la fin d'un TB.

Concernant les branchements conditionnels, nous avons fait le choix de les implémenter manuellement car certaines architectures rendent la recherche de ces comportements difficile. Par exemple en ARM ou en x86, il est nécessaire d'exécuter deux instructions pour réaliser un branchement conditionnel. La première est une instruction de comparaison entre deux registres qui vient mettre à jour les drapeaux du processeur. La deuxième instruction réalise le branchement en fonction de l'état de ces drapeaux. Prenons l'exemple du branchement conditionnel en Intel 64 présenté par le listing 5.11.

```
1        cmp %rax, %rcx
2        jge foo
```

Listing 5.11 – Un branchement conditionnel en Intel 64

L'instruction `jge` de la ligne 2 effectuera le branchement si le contenu du registre `%rax` est plus grand ou égal au contenu du registre `%rcx`. Le lien entre les instructions `jge` et `cmp` est ici implicite puisqu'il passe par la modification et la lecture des drapeaux contenus dans le registre `rflags`. La manière dont sont décrites ces instructions ne rend pas leur détection facile. L'instruction `jge` réalise son branchement si le test `SF xor OF` est vrai, `SF` étant le drapeau de signe, et `OF` le drapeau de dépassement de capacité. Les nœuds du type `(GE)` n'apparaîtront donc pas explicitement dans la description de l'instruction.

Une solution pour palier ce problème est d'enrichir la description de l'architecture avec la notion de drapeaux, ce qui n'est actuellement pas possible dans MDS. Si la description contient la sémantique des drapeaux du processeur, il est alors possible de faire correspondre un test entre ceux-ci avec la condition testée qui en découle (comme `SF xor OF` correspond à `(GE)` dans cet exemple).

Les micro-opérations de chargement et écriture mémoire de la machine cible sont laissées à la charge du concepteur pour des raisons d'optimisation et de réutilisation. Au delà de la prise en charge de la mémoire virtuelle du processeur cible, le code correspondant aux accès mémoire dans un simulateur peut être décorrélé du couple cible – hôte. Une bonne partie de la traduction de ces micro-opérations est donc fixe quel que soit la cible et l'hôte. La génération automatique n'a donc que peu de sens ici. On préfèrera un code écrit une fois

pour toute et optimisé pour le cas d'utilisation du simulateur, ou la réutilisation de briques en s'intégrant par exemple dans un simulateur existant.

Il en va de même pour la micro-opération de fin de TB qui fait partie du cœur du moteur de DBT, et qui reste la même quel que soit le couple cible – hôte.

Au delà des micro-opérations de l'ensemble minimal, il reste quelques briques concernant principalement l'interfaçage de la machine hôte avec le moteur de DBT. On y trouve par exemple la gestion du contexte dans un TB, avec son prologue et son épilogue, les registres hôtes à sauvegarder avant d'exécuter un TB, etc.

5.6.2 Génération de l'IR spécialisée

L'IR spécialisée est générée à partir des informations contenues dans la base de correspondances. Elle vient compléter l'IR minimal pour former une machine virtuelle avec un pouvoir d'expression suffisant pour simuler l'ensemble des instructions de la machine cible.

Plusieurs stratégies sont possibles pour produire automatiquement cette IR spécialisée. L'une de ces techniques consiste à générer une micro-opération par instruction cible. La traduction des instructions cibles est alors directe puisque chacune possède sa micro-opération la représentant. En revanche, la génération de code dans le *backend* est complexe puisqu'il faudra potentiellement générer plusieurs instructions hôtes pour une micro-opération. Cette solution est souvent évitée en DBT car la liberté d'action dans le *backend* est en général plus limitée que dans le *frontend*. Cela est principalement dû au processus d'allocation de registres que l'on souhaite conserver le plus simple possible pour ne pas alourdir la traduction. Si cette technique est utilisée, plusieurs registres intermédiaires sont potentiellement nécessaires pour traduire une micro-opération, ce que l'on souhaite éviter pour des raisons de performance.

En revanche, créer une micro-opération par instruction hôte utilisée au moins une fois dans une correspondance règle ce problème. C'est alors au *frontend* de générer plusieurs micro-opérations pour une instruction si nécessaire. La flexibilité dans le *frontend* est en général plus grande, puisqu'on peut par exemple supposer le nombre de registre de la représentation intermédiaire infini (c'est en pratique l'hypothèse qui est faite dans l'IR de QEMU).

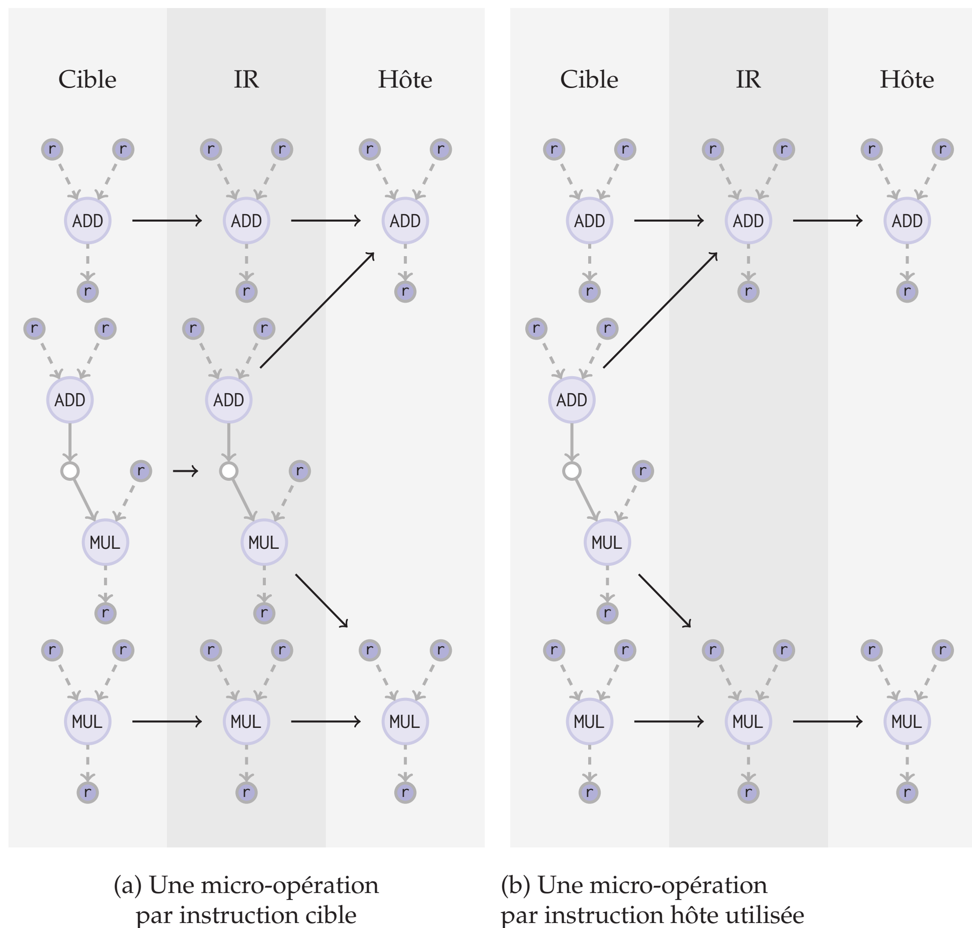
La figure 5.14 illustre ces deux stratégies par un exemple. La machine cible possède trois instructions et la machine hôte seulement deux. La figure 5.14a donne les micro-opérations générées si la première stratégie est utilisée. Nous obtenons trois micro-opérations, dont une qui se traduit en deux instructions hôtes. Si la deuxième stratégie est utilisée, seulement deux micro-opérations sont créées comme indiqué sur la figure 5.14b. L'instruction cible qui nécessite deux instructions hôtes est donc traduite en deux micro-opérations.

5.6.3 Génération du *frontend* et du *backend*

La phase de génération du *frontend* se découpe en trois étapes principales, la génération du décodeur binaire, du traducteur, et de l'environnement du processeur simulé. Elle prend en entrée la description de l'architecture cible, ainsi que la base de correspondances et l'ensemble minimal de micro-opérations.

La génération du décodeur binaire

Le décodeur binaire est chargé de décoder les instructions cibles et de les transmettre au traducteur. Il existe plusieurs stratégies pour générer un décodeur à partir de la descrip-



(a) Une micro-opération par instruction cible

(b) Une micro-opération par instruction hôte utilisée

FIGURE 5.14 – Stratégies de génération de l'IR

tion d'un jeu d'instructions. Nous avons développé une technique générant des décodeurs efficaces décrite dans [FMP13] et fournie par l'annexe C.

La génération de l'environnement du processeur

L'environnement du processeur simulé contient l'état de ce dernier lors de la simulation. On y trouve notamment la valeur de chaque registre. Cet environnement est généré à partir de la description de la machine cible.

La génération du traducteur

La génération du traducteur s'appuie essentiellement sur la base de correspondances. Pour chaque instruction cible, sa correspondance dans la base nous donne la ou les micro-opérations à générer. Les opérandes des instructions sont chargés et écrits à l'aide de fonctions spécifiques, générées en amont. Par exemple, la fonction chargée de lire un registre est générée de telle sorte qu'elle accède à l'environnement du processeur, alors que celle qui effectue un accès mémoire génère la micro-opération correspondante présente dans l'ensemble minimal. Les conditions des effets sont eux aussi traduits en utilisant les micro-

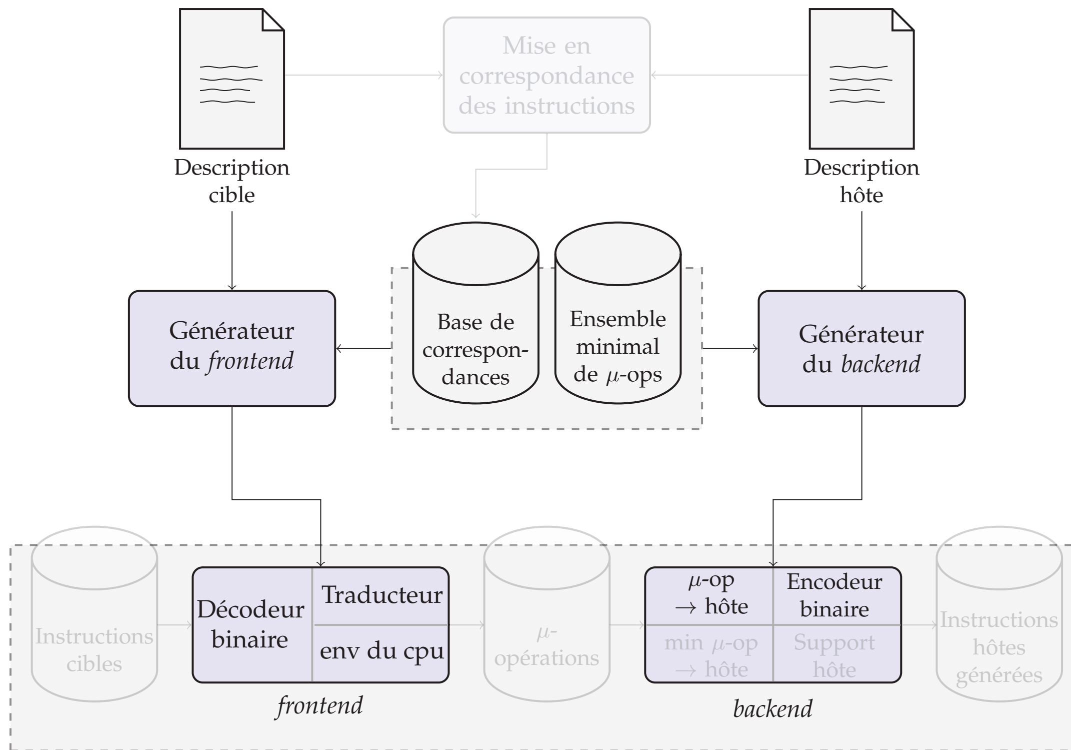


FIGURE 5.15 – Phase de génération du *frontend*et du *backend*

opérations conditionnelles de l'ensemble minimal. Lorsque le traducteur s'aperçoit qu'une instruction modifie la valeur du compteur ordinal de la machine cible, il considère que cette instruction est un branchement et provoque la fin du TB courant.

La génération du *backend*

La génération du *backend* se résume principalement en la production du générateur de code, chargé d'émettre les instructions hôtes correspondant aux micro-opérations générées par le *frontend*. Cette étape fait intervenir la base de correspondance ainsi que la description de l'hôte pour récupérer le codage des instructions.

5.6.4 Conclusion

Toutes ces briques générées plus quelques autres écrites manuellement s'assemblent pour former un simulateur pour la machine cible basé sur la DBT et optimisé pour le couple cible – hôte.

5.7 Expérimentation sur les correspondances

Afin d'observer le comportement de notre algorithme de mise en correspondance d'instructions, nous réalisons quelques expérimentations sur différents couples. Les descriptions d'architecture en MDS dont nous disposons sont les suivantes :

- La description d'un sous-ensemble de l'architecture MIPS32 composé de 127 instructions. Ce sous-ensemble est choisi de manière à couvrir les instructions produites classiquement par un compilateur C.
- La description d'un sous-ensemble limité de l'architecture ARMv7, principalement utilisée pour mettre en avant quelques points intéressants dans les correspondances.
- La description de l'architecture `k1` du processeur de Kalray.
- La description de la représentation intermédiaire du TCG de QEMU. Cette description n'a pas beaucoup d'intérêt ni de sens en tant que telle puisque le TCG est une machine abstraite qui ne possède pas toutes les propriétés d'une vraie machine (nombre de registres non borné, pas de compteur ordinal explicite, ...) et possède des micro-opérations propre à la gestion du processus de DBT (comme la micro-opération de fin d'un TB par exemple). Cependant, elle fournit un cas simple à décrire car chaque micro-opération n'effectue qu'une opération, et complet car elle fait le lien entre n'importe quel couple cible – hôte dans QEMU. Elle n'a donc d'intérêt que pour tester notre algorithme puisque dans le flot final, la représentation intermédiaire est automatiquement générée.

5.7.1 Valeurs attribuées aux constantes d'évaluations pour les expérimentations

Les fonctions d'évaluations $eval_{noeud}$, $eval_{totale}$ et $eval_{partielle}$ définissent trois constantes α, β et γ que nous choisissons de fixer comme décrit dans le tableau 5.1. Pour les deux

Constante	Description	Valeur
α	Pondération de la différence entre les tailles de type	1
β	Pondération de la distance entre les opérandes	10
γ	Pénalité de correspondance partielle	100

TABLE 5.1 – Valeurs attribuées aux constantes d'évaluations

premiers termes, cette répartition permet de ne pas privilégier un terme plus que l'autre car la différence entre les tailles de types est souvent une puissance de 2 comme 16 ou 32, alors que la distance entre les nœuds est comprise entre 0 et 2. En revanche, nous souhaitons au maximum privilégier les correspondances avec le moins d'instructions hôtes possibles. Nous appliquons donc un fort coefficient γ.

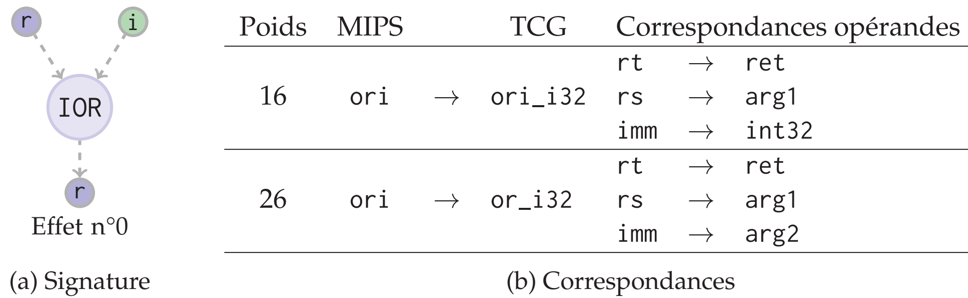
5.7.2 Comportement de l'algorithme de mise en correspondance

Afin d'illustrer le comportement de l'algorithme, prenons quelques exemples de correspondances principalement extraites des couples MIPS – TCG et MIPS – ARM, après leur avoir appliqué l'algorithme de recherche exhaustive des correspondances.

L'instruction `ori`

L'instruction `ori rt, rs, imm` effectue l'opération "ou logique" entre le registre `rs` et la valeur immédiate sur 16 bits `imm` et écrit le résultat dans `rt`. Pour cette instruction, deux correspondances sont trouvées du côté du TCG. La première est la micro-opération `ori_i32` avec un poids de 16, qui correspond presque parfaitement avec l'instruction MIPS sauf que sa valeur immédiate est de taille 32 bits. La valeur immédiate `imm` subira donc une extension de zéros lors de la génération du code. La deuxième est la micro-opération `or_i32` avec un

poids de 26. Elle est moins intéressante que la première puisque la valeur immédiate de l'instruction MIPS doit d'abord être placée dans un registre TCG avant d'être utilisée.



(a) Signature

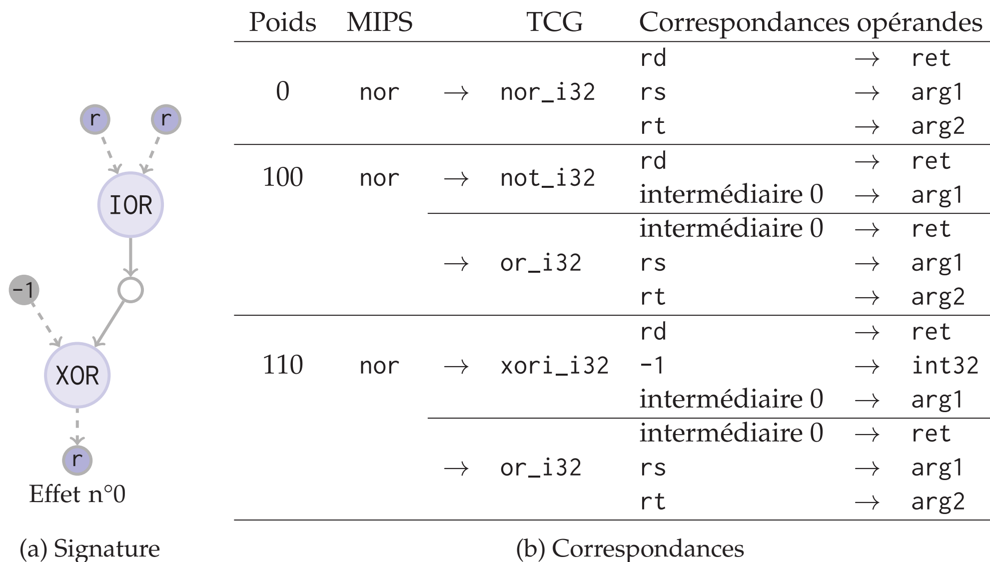
Poids	MIPS		TCG	Correspondances opérandes		
16	ori	→	ori_i32	rt	→	ret
				rs	→	arg1
				imm	→	int32
26	ori	→	or_i32	rt	→	ret
				rs	→	arg1
				imm	→	arg2

(b) Correspondances

FIGURE 5.16 – Signatures et correspondances pour l'instruction `ori`

L'instruction nor

L'instruction `nor rd, rs, rt` effectue l'opération logique "non ou" entre les registres `rs` et `rt`, et écrit le résultat dans `rd`. Cette instruction est intéressante du point de vue de la mise en correspondance car elle est composée de deux opérations de base, un nœud `(XOR)` et un nœud `(IOR)` comme décrit par la signature de la figure 5.17a. Ici, la passe d'uniformisation a remplacé dans la signature le nœud `(NOT)` par un nœud `(XOR)` avec la constante `-1`.



(a) Signature

Poids	MIPS		TCG	Correspondances opérandes		
0	nor	→	nor_i32	rd	→	ret
				rs	→	arg1
				rt	→	arg2
100	nor	→	not_i32	rd	→	ret
				intermédiaire 0	→	arg1
		→	or_i32	intermédiaire 0	→	ret
				rs	→	arg1
				rt	→	arg2
110	nor	→	xori_i32	rd	→	ret
				-1	→	int32
				intermédiaire 0	→	arg1
		→	or_i32	intermédiaire 0	→	ret
				rs	→	arg1
				rt	→	arg2

(b) Correspondances

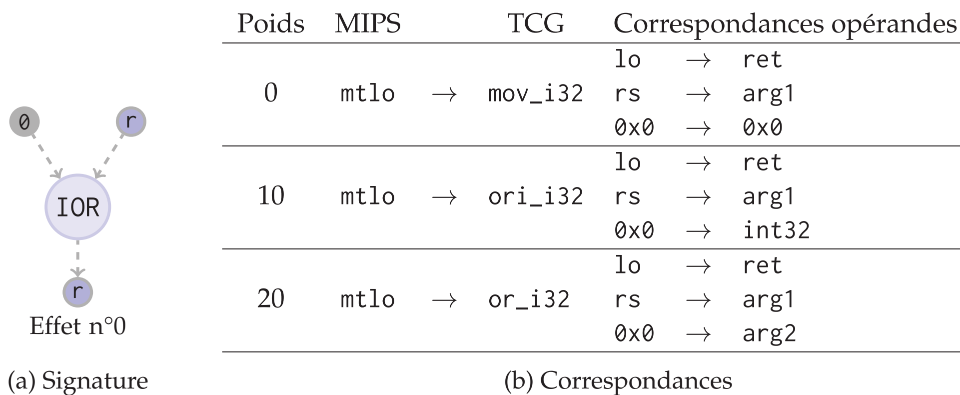
FIGURE 5.17 – Signatures et correspondances pour l'instruction `nor`

La représentation intermédiaire de QEMU possédant une micro-opération au comportement similaire (`nor_i32`), la correspondance est directe et le poids est nul. Mais l'algorithme trouve aussi une correspondance avec deux micro-opérations (`not_i32` et `or_i32`) avec un poids plus important mais qui pourrait être choisie si l'hôte n'était pas équipé d'une instruction similaire à `nor`. Finalement, une troisième correspondance est trouvée, utilisant une micro-opération `xori_i32` pour réaliser le non logique sur le résultat du ou logique. On voit

ici que la passe d'uniformisation permet de se passer de la micro-opération `not_i32` dans le cas où l'hôte ne serait pas équipé d'une telle instruction.

L'instruction `mtlo`

L'instruction `mtlo rs` permet de copier le contenu du registre `rs` dans le registre `lo`. L'opération réalisée par cette instruction est donc un simple mouvement de données. Cependant, grâce à notre représentation des signatures sous la forme étendue, cette instruction est décrite avec un nœud (`IOR`) entre `rs` et la constante 0. Cela permet à une architecture hôte ne possédant pas d'instruction de mouvement de données de pouvoir simuler cette instruction.



Poids	MIPS		TCG	Correspondances opérandes		
0	mtlo	→	mov_i32	lo	→	ret
				rs	→	arg1
				0x0	→	0x0
10	mtlo	→	ori_i32	lo	→	ret
				rs	→	arg1
				0x0	→	int32
20	mtlo	→	or_i32	lo	→	ret
				rs	→	arg1
				0x0	→	arg2

(a) Signature (b) Correspondances

FIGURE 5.18 – Signatures et correspondances pour l'instruction `mtlo`

Cette propriété se reflète dans les correspondances puisque même si la micro-opération `mov_i32` est la meilleure candidate pour simuler l'instruction `mtlo`, l'algorithme trouve aussi les micro-opérations `ori_i32` et `or_i32` comme de possibles substitutions mais avec un poids supérieur.

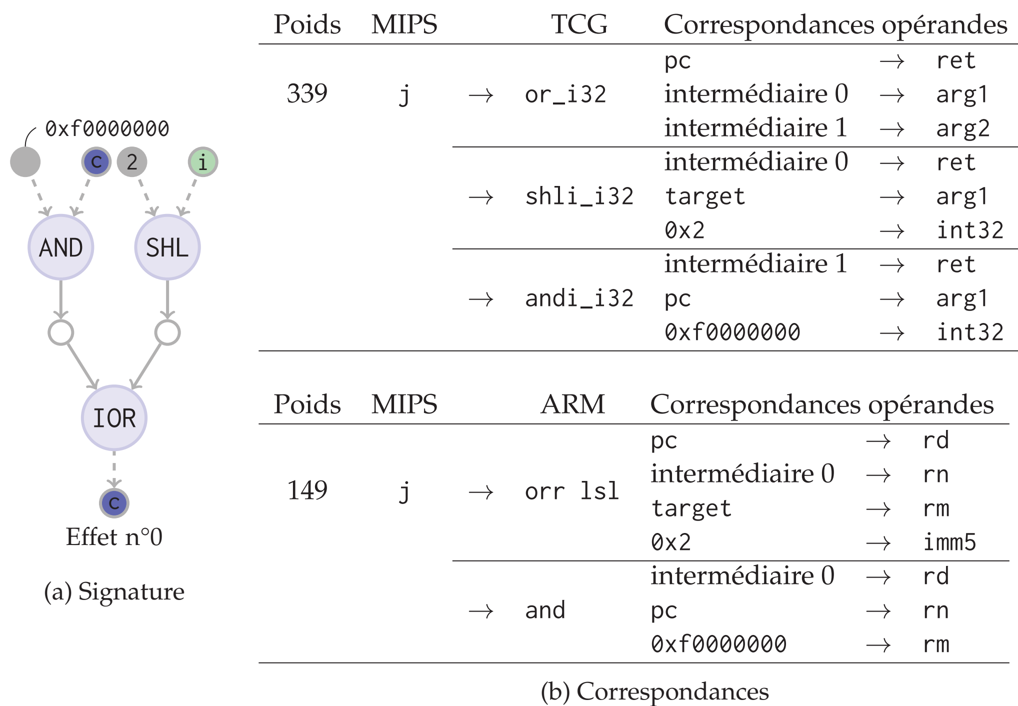
L'instruction `j`

L'instruction `j` du MIPS calcule l'adresse de son saut comme décrit à la section 5.4.2 page 68.

La meilleure correspondance trouvée par l'algorithme ici utilise trois micro-opérations TCG et a un poids de 339. En revanche, lorsqu'on met le MIPS en correspondance avec l'ARM, la meilleure correspondance est plus intéressante car elle n'utilise que deux instructions ARM pour simuler le saut MIPS. C'est grâce aux instructions ARM pour lesquelles on peut spécifier un décalage optionnel sur l'opérande `rm`, en plus de l'opération réalisée par l'instruction. Dans cet exemple, c'est l'instruction `orr` qui effectue ces deux opérations comme illustré par le listing 5.12

```
1  orr r1, r2, r3, lsl #2      ; r1 <- r2 | (r3 << 2)
```

Listing 5.12 – L'instruction ARM `orr` avec décalage



(a) Signature

Poids	MIPS		TCG	Correspondances opérandes		
339	j	→	or_i32	pc	→	ret
				intermédiaire 0	→	arg1
				intermédiaire 1	→	arg2
		→	shli_i32	intermédiaire 0	→	ret
				target	→	arg1
				0x2	→	int32
		→	andi_i32	intermédiaire 1	→	ret
				pc	→	arg1
				0xf0000000	→	int32

Poids	MIPS		ARM	Correspondances opérandes		
149	j	→	orr lsl	pc	→	rd
				intermédiaire 0	→	rn
				target	→	rm
				0x2	→	imm5
		→	and	intermédiaire 0	→	rd
				pc	→	rn
				0xf0000000	→	rm

(b) Correspondances

FIGURE 5.19 – Signatures et correspondances pour l'instruction `j`

Cette instruction décale le contenu du registre `r3` de 2 vers la gauche avant de réaliser l'opération "ou logique" avec `r2`. Elle écrit le résultat final dans `r1`.

Ce résultat est doublement intéressant car il montre d'une part que l'algorithme s'adapte et arrive à exploiter au mieux les spécificités du jeu d'instruction hôte, mais il montre aussi l'intérêt de générer une représentation intermédiaire adaptée pour le couple puisqu'ici, une IR générique comme celle du TCG ne serait pas parvenue à exploiter ce genre d'instructions sur l'hôte.

5.7.3 Limitations des transformations et de l'algorithme

Certains cas posent encore des problèmes pour la mise en correspondance. L'un d'eux survient lorsque l'on cherche à simuler un opérande cible sur un opérande hôte alors que ce dernier est trop petit. La règle 1 de mise en correspondance décrite section 5.5.1 interdit ce cas. Cependant, cela pose un problème lorsque l'architecture cible travaille avec des registres plus grands que ceux de l'hôte. Par exemple, l'architecture cible peut posséder des instructions SIMD travaillant avec des registres de 128 bits. Pour que l'hôte puisse malgré tout les simuler, il faut rajouter une passe de transformation supplémentaire sur les effets des instructions cibles pour découper ces derniers en opérations plus petites de manière à ce que les données tiennent dans les registres hôtes. De cette manière, l'algorithme pourra trouver des correspondances sans violer la règle 1.

Par ailleurs, certaines instructions peuvent être décrites à l'aide de nœuds `(EFFECT)` et `(APPLY)`. Ces nœuds servent à indiquer qu'une partie du comportement de l'instruction n'est pas décrite en MDS, mais par exemple à l'aide d'un *helper*. Dans ce cas, nous ne possédons pas la sémantique cachée derrière de tels nœuds et l'algorithme ne peut trouver de correspondances.

5.7.4 Statistiques sur les correspondances

Afin d'évaluer la qualité des correspondances trouvées par l'algorithme, nous extrayons des statistiques sur quelques couples. Nous prenons le MIPS, le TCG et le k1 comme architecture cible, et le MIPS et le TCG comme hôte.

Avec l'architecture MIPS comme hôte

Les tableaux 5.2, 5.3 et 5.4 montrent les statistiques de la mise en correspondance entre les cibles MIPS, k1 et TCG et l'hôte MIPS. Notre description de l'architecture MIPS comporte 127 instructions et 131 effets.

L'algorithme parvient à simuler totalement le MIPS sur lui-même puisqu'il trouve au moins une correspondance pour toutes les instructions. Cela montre bien l'efficacité de la forme étendue des signatures grâce à laquelle des instructions comme `mthi`, `mtlo`, etc... peuvent être simulées avec l'instruction `ori` ou `or`. En effet, l'architecture MIPS ne possède pas d'instruction dédiée à déplacer une donnée d'un registre à un autre. Par ailleurs, environ un quart du jeu d'instructions hôte est utilisé pour simuler l'ensemble des instructions cibles. Le poids moyen des correspondances est d'environ 70, avec 1.5 instructions en moyenne pour simuler un effet cible. Cela reflète la nature Reduced Instruction Set Computer (RISC) de l'architecture MIPS, avec une majorité d'instructions simples réalisant une opération, et quelques instructions plus complexes à simuler comme les chargements et écritures mémoire avec leur calcul d'adresse, ou bien les sauts sauvegardant la valeur du compteur ordinal avant branchement (`jal`, ...).

Lorsque le TCG est pris comme cible, on constate un taux de couverture d'environ 42% ce qui signifie que pour 58% des micro-opérations, aucune correspondance n'est trouvée. Cela se justifie par les limitations évoquées précédemment, et notamment par le fait que les micro-opérations TCG sont composées pour moitié d'opérations sur des données de 64 bits, ce que notre description architecturale du MIPS ne peut simuler. Pour les micro-opérations qu'il parvient à simuler, il utilise environ 15% de son jeu d'instructions avec un poids moyen d'environ 28 et un nombre moyen d'instructions pour un effet cible proche de 1. Le TCG étant composé de micro-opérations très simples, ne possédant qu'un effet à chaque fois, la probabilité de trouver une correspondance directe dans le jeu d'instructions hôte est forte, ce qui explique ces résultats.

Finalement, nous répétons l'opération avec l'architecture k1. Cette dernière étant conséquente (664 instructions, 985 effets), nous n'effectuons pas une recherche exhaustive des correspondances. Les statistiques sont donc moins précises car la recherche non-exhaustive ne cherche que la meilleure correspondance pour un effet cible et élimine les correspondances moins bonnes au plus tôt dans la recherche. L'ensemble final des correspondances est donc bien moins important que dans le cas exhaustif mais le temps d'exécution de l'algorithme reste ainsi acceptable. Ici, l'algorithme parvient à couvrir environ 28% des effets du k1. Ceci est encore une fois dû aux opérations sur plus de 32 bits, ainsi que les nombreuses instructions décrites à l'aide de nœuds `(EFFECT)` et `(APPLY)` dans le k1 (environ 40% des effets). L'algorithme ne possédant la sémantique de ces instructions, il ne peut déterminer de candidat hôte pour les simuler. Dans ce cas, ces instructions cibles devraient être simulées à l'aide de *helpers*. Les instructions du k1 étant pour beaucoup complexes, le poids moyen est d'environ 244, avec 2.69 instructions MIPS pour simuler un effet cible en moyenne.

Ces résultats montrent que dans un processus de DBT, le jeu d'instruction hôte n'est que peu exploité en terme de diversité d'instructions. Dans le cas du MIPS, entre 15% et 25% des instructions sont utilisées malgré sa nature RISC. Toutes les instructions de type "réservée"

ou “systèmes”, alors qu’elles sont simulées côté cible, ne sont jamais utilisées côté hôte, ce qui participe grandement à ce faible ratio.

Temps d’exécution de l’algorithme	2s
Nombre d’instructions cibles	127
Nombre d’effets cibles	131
Nombre d’instructions hôtes	127
Nombre total de correspondances	914
Pourcentage d’instructions cibles simulées	100.00%
Poids moyen des meilleures correspondances	70.55
Poids moyen de la totalité des correspondances	862.12
Nombre moyen de correspondances trouvées pour un effet cible	15.76
Nombre moyen d’instructions hôtes pour simuler un effet cible	1.52
Pourcentage d’instructions hôtes utilisées	23.62%
Pourcentage d’instructions hôtes utilisées (meilleures correspondances)	22.83%

TABLE 5.2 – Statistiques des correspondances MIPS – MIPS

Temps d’exécution de l’algorithme	3s
Nombre d’instructions cibles	98
Nombre d’effets cibles	98
Nombre d’instructions hôtes	127
Nombre total de correspondances	570
Pourcentage d’instructions cibles simulées	42.86%
Poids moyen des meilleures correspondances	28.07
Poids moyen de la totalité des correspondances	1108.75
Nombre moyen de correspondances trouvées pour un effet cible	13.57
Nombre moyen d’instructions hôtes pour simuler un effet cible	1.17
Pourcentage d’instructions hôtes utilisées	16.54%
Pourcentage d’instructions hôtes utilisées (meilleures correspondances)	15.75%

TABLE 5.3 – Statistiques des correspondances TCG – MIPS

Avec le TCG comme hôte

La description du TCG se compose de 98 micro-opérations composées chacune d’un effet. Les tableaux 5.5, 5.6 et 5.7 exposent à leur tour les statistiques de la mise en correspondance, mais en utilisant cette fois-ci le TCG comme hôte. En raison du plus grand nombre d’effets hôtes utilisables pour la simulation dans le TCG que dans le MIPS, les temps d’exécution de l’algorithme sont plus important. Ils varient d’environ 2 minutes en recherche exhaustive pour les cibles MIPS et TCG, et montent jusqu’à 4 minutes en recherche non-exhaustive avec le k1 comme cible.

L’exécution de l’algorithme avec le MIPS comme cible produit une couverture de 100% des effets. Il faut 1.52 micro-opérations TCG en moyenne pour simuler un effet MIPS avec un poids moyen d’environ 65. Ces résultats sont très similaires au couple MIPS – MIPS et mettent en avant la nature RISC de cette architecture. En revanche ici, 54% des micro-opérations sont utilisées. Ce nombre reflète bien le fait que ces micro-opérations ont conçues

Temps d'exécution de l'algorithme en recherche non exhaustive	20s
Nombre d'instructions cibles	664
Nombre d'effets cibles	985
Nombre d'instructions hôtes	127
Nombre total de correspondances	506
Pourcentage d'instructions cibles simulées	27.92%
Poids moyen des meilleures correspondances	244.79
Nombre moyen de correspondances trouvées pour un effet cible	1.84
Nombre moyen d'instructions hôtes pour simuler un effet cible	2.69
Pourcentage d'instructions hôtes utilisées (meilleures correspondances)	14.96%

TABLE 5.4 – Statistiques des correspondances k1 – MIPS (recherche non-exhaustive)

pour servir de représentation intermédiaire dans un processus de DBT.

La correspondance TCG – TCG produit un résultat intéressant qui appuie l'observation précédente. L'algorithme parvient à faire correspondre chaque micro-opération avec elle-même, ce qui produit un poids nul dans tous les cas. Ce résultat vient du fait que nous n'avons pas inclus les micro-opérations réservées à la gestion du processus de DBT comme celle qui provoque la fin d'un TB dans la description. En effet, ces micro-opérations n'interviennent de toutes façon pas dans le processus de mise en correspondance, et sont difficilement descriptibles en MDS puisqu'elles manipulent des notions qui ne représentent pas le comportement d'une machine mais celui d'un traducteur binaire dynamique. La description est donc exclusivement composée d'opérations sur des données en vue de la simulation d'instructions cibles. Toutes les micro-opérations sont donc utilisables côté hôte, d'où cette correspondance TCG – TCG "parfaite".

Finalement, la correspondance avec le k1 comme cible produit une couverture d'environ 60% des effets. Cette dernière est meilleure que dans le cas du couple k1 – MIPS car le TCG gère les données sur 64 bits. En revanche la limitation des nœuds `(EFFECT)` et `(APPLY)` reste la même.

Ces résultats montrent que la représentation intermédiaire de QEMU est bien adaptée pour simuler les jeux d'instructions cibles.

Temps d'exécution de l'algorithme	1m52s
Temps d'exécution de l'algorithme en recherche non exhaustive	2.2s
Nombre d'instructions cibles	127
Nombre d'effets cibles	131
Nombre d'instructions hôtes	98
Nombre total de correspondances	134755
Pourcentage d'instructions cibles simulées	100.00%
Poids moyen des meilleures correspondances	65.34
Poids moyen de la totalité des correspondances	1681.80
Nombre moyen de correspondances trouvées pour un effet cible	2323.36
Nombre moyen d'instructions hôtes pour simuler un effet cible	1.52
Pourcentage d'instructions hôtes utilisées	58.16%
Pourcentage d'instructions hôtes utilisées (meilleures correspondances)	34.69%

TABLE 5.5 – Statistiques des correspondances MIPS – TCG

Temps d'exécution de l'algorithme	1m46s
Temps d'exécution de l'algorithme en recherche non exhaustive	1.9s
Nombre d'instructions cibles	98
Nombre d'effets cibles	98
Nombre d'instructions hôtes	98
Nombre total de correspondances	131423
Pourcentage d'instructions cibles simulées	100.00%
Poids moyen des meilleures correspondances	0.00
Poids moyen de la totalité des correspondances	1706.22
Nombre moyen de correspondances trouvées pour un effet cible	1341.05
Nombre moyen d'instructions hôtes pour simuler un effet cible	1.00
Pourcentage d'instructions hôtes utilisées	100.00%
Pourcentage d'instructions hôtes utilisées (meilleures correspondances)	100.00%

TABLE 5.6 – Statistiques des correspondances TCG – TCG

Temps d'exécution de l'algorithme en recherche non exhaustive	4m11s
Nombre d'instructions cibles	664
Nombre d'effets cibles	985
Nombre d'instructions hôtes	98
Nombre total de correspondances	4491
Pourcentage d'instructions cibles simulées	59.19%
Poids moyen des meilleures correspondances	461.05
Nombre moyen de correspondances trouvées pour un effet cible	7.70
Nombre moyen d'instructions hôtes pour simuler un effet cible	4.46
Pourcentage d'instructions hôtes utilisées (meilleures correspondances)	63.27%

TABLE 5.7 – Statistiques des correspondances k1 – TCG (recherche non-exhaustive)

Les instructions et micro-opérations les plus utilisées

Les figures 5.20 et 5.21 montrent les instructions et micro-opérations les plus utilisées lors de la correspondance entre respectivement les couples k1 – MIPS et k1 – TCG.

Du côté du MIPS, l'instruction `or` est la plus utilisée puisqu'elle prend en charge les mouvements de données, en plus de l'opération "ou logique". Viennent ensuite `sra` et `sll` qui simulent les extensions de signe des données lues en plus des décalages. En effet, l'architecture MIPS ne possède pas d'instructions dédiées à l'extension de signe.

Quant au TCG, les mouvements de données sont gérés par la micro-opération `mov_i32` et les extensions de signe par les micro-opérations du type `ext32s_i64`, `ext16s_i32`, etc. La distribution est donc plus étalée que dans le cas du MIPS.

5.7.5 Conclusion

Ces expérimentations montrent que l'algorithme est à même de trouver des correspondances entre les instructions d'une cible et d'un hôte et ce même lorsque les comportements de ces dernières ne semblent pas correspondre au premier abord. C'est principalement grâce aux transformations effectuées *à priori* sur les signatures, afin de les ramener dans la forme étendue.

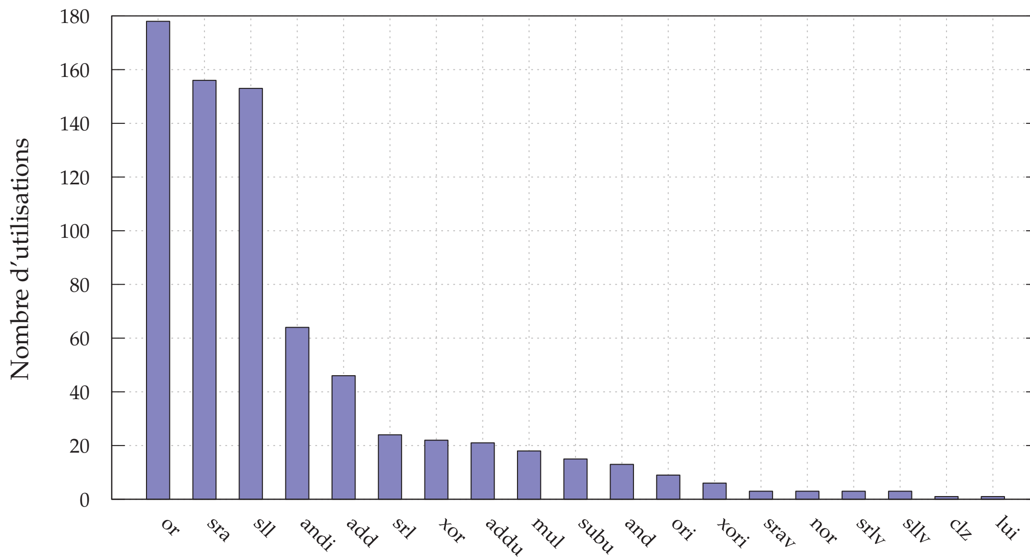


FIGURE 5.20 – Les instructions MIPS les plus utilisées dans les correspondances avec le k1

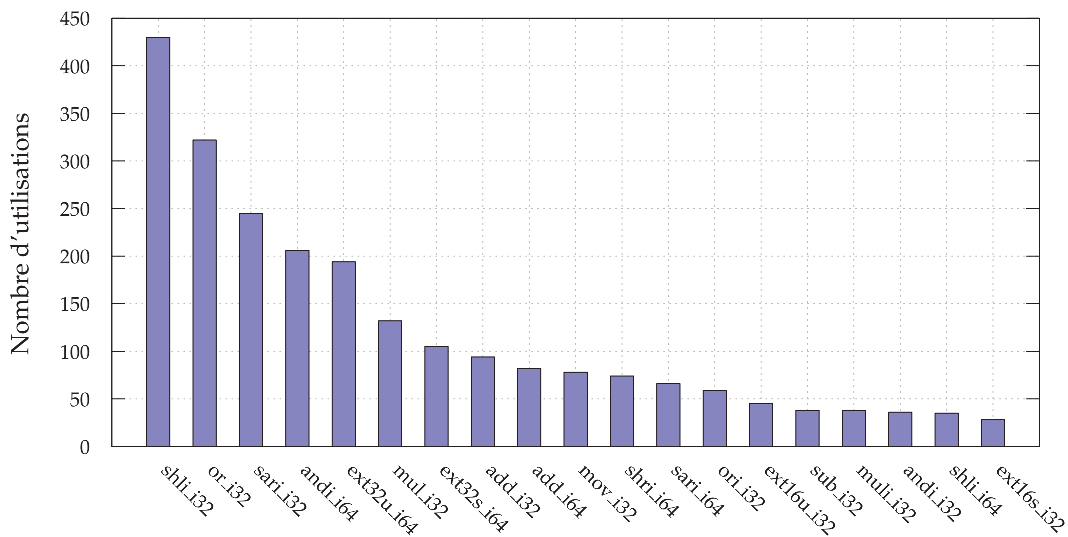


FIGURE 5.21 – Les micro-opérations TCG les plus utilisées dans les correspondances avec le k1

Le système d'évaluation des correspondances permet dans tous les cas de sélectionner la ou les meilleures instructions hôtes pour simuler une instruction cible. Ce système permet aussi de réduire grandement le temps de recherche grâce à une étape de *séparation et évaluation* qui en fait l'usage.

Les transformations sur les signatures ont encore besoin d'être travaillées, notamment pour prendre en charge des opérations sur des données de taille supérieure aux registres de travail de l'hôte. Une fois cette modification effectuée, l'algorithme sera en mesure de trou-

ver des correspondances pour la grande majorité des instructions si celles-ci sont exprimées explicitement dans la description architecturale.

5.8 Expérimentation sur la génération d'un *frontend* pour QEMU

Dans cette section, nous nous proposons de générer un *frontend* s'intégrant directement dans le simulateur QEMU. L'intérêt d'utiliser QEMU ici est de pouvoir réutiliser une grande partie de son infrastructure. Cela permet de combler les manques de notre prototype qui est pour le moment uniquement capable de générer la partie *frontend* du flot.

L'idée est de générer un *frontend* MIPS et de réutiliser les *backends* existants de QEMU ainsi que sa représentation intermédiaire. Le couple que nous utilisons pour générer le *frontend* est donc MIPS – TCG. Cela nous permet de mettre en correspondance chaque instruction MIPS avec des micro-opérations de l'IR QEMU, et de laisser ensuite QEMU traduire ces micro-opérations en instructions hôtes.

Le décodeur d'instructions est généré grâce à la technique décrite dans [FMP13]. Une fois une instruction décodée, il fait appel à la fonction générée pour la simuler en lui fournissant les détails du décodage. La fonction de simulation de l'instruction génère les micro-opérations nécessaires d'après la correspondance. Elle accède aux opérandes grâce à des fonctions elles-aussi générées. Ces dernières viennent lire ou modifier l'état du processeur simulé si l'instruction accède à des registres ou bien génèrent une micro-opération d'accès mémoire dans le cas d'un chargement ou d'une écriture mémoire.

Les conditions des effets sont générées à l'aide des micro-opérations de branchement conditionnels de QEMU, et faisant partie du *runtime* dans le flot complet.

Notre *frontend* généré est capable d'exécuter des programmes complexes, compilés depuis un langage de haut niveau comme le C. Nous l'avons testé sur des tests unitaires et sur de nombreux cas pour en vérifier le bon fonctionnement. Il produit cependant un traducteur de qualité médiocre puisqu'aucune optimisation n'est réalisée pendant la phase de génération du *frontend*.

Nous réalisons quelques mesures de performance de ce *frontend* sur les programmes de la suite Polybench[Pou11], en comparant notre temps de simulation à celui du *frontend* MIPS original de QEMU. La figure 5.22 illustre ces temps totaux de simulation alors que la figure 5.23 ne montre que le temps de traduction dans le *frontend*.

Les résultats sont très encourageants. Les temps de simulation, bien qu'un peu supérieurs avec notre simulateur, sont très similaires. L'écart entre les deux simulateurs est de 7.92% en moyenne. L'écart se creuse avec le temps passé dans le *frontend*, ce qui était prévisible étant donné la qualité du code généré par notre prototype, et ce comparé à un code écrit à la main.

Cependant, QEMU parvient ensuite à optimiser le flux de micro-opérations générées de manière satisfaisante, rendant les TB générés par notre traducteur quasiment aussi rapides que ceux du *frontend* original.

5.9 Conclusion

Dans ce chapitre, nous avons présenté un flot de génération automatique de simulateurs basés sur la technique de traduction binaire dynamique. Le simulateur est généré à partir des descriptions architecturales de la machine cible et de la machine hôte et est optimisé pour ce couple. Une phase de mise en correspondance des instructions cibles et hôtes permet

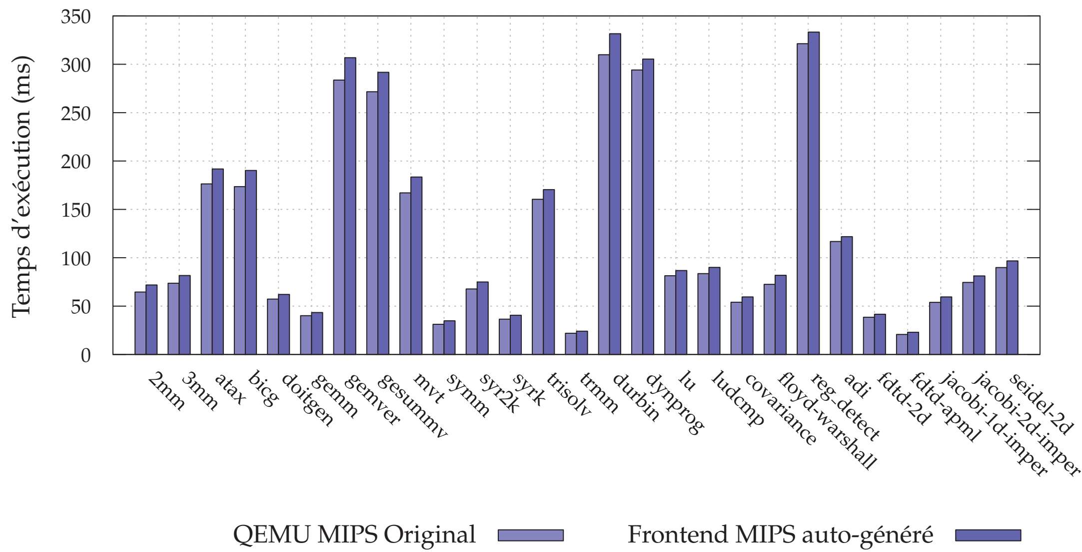


FIGURE 5.22 – Temps d'exécution de la simulation de la suite Polybench

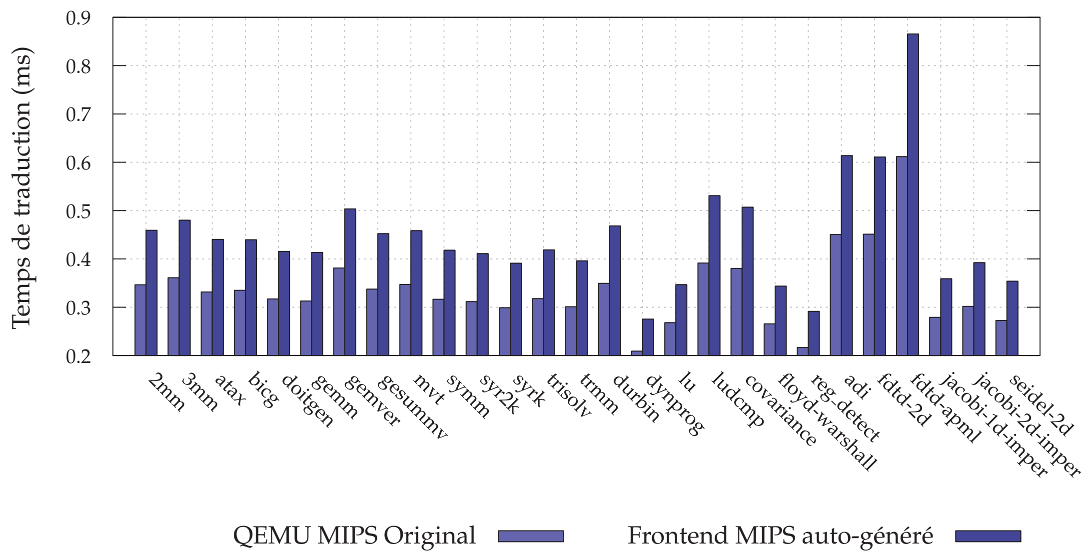


FIGURE 5.23 – Temps passé à décoder et traduire les instructions cibles dans le *frontend*

de sélectionner la ou les meilleures instructions hôtes pour simuler une instruction cible. Une représentation intermédiaire spécialisée pour le couple est générée à partir de cette base de correspondances. Conserver une IR dans le simulateur final lui permet d'effectuer des optimisations dynamiques sur les TB générés, ce que serait difficilement réalisable sans IR.

La forme étendue des comportements d'instructions permet à l'algorithme de correspondance de trouver facilement des instructions hôtes pour la plupart des instructions cibles. Il est capable d'exploiter les particularités du jeu instructions hôte, ce qui confère un avantage non négligeable à l'IR générée par rapport à une IR classique comme celle de QEMU.

Les résultats de notre premier prototype en terme de performance de simulation sont très encourageants. Bien que seulement la partie *frontend* du simulateur soit générée, et que l'IR spécialisée ne soit pas encore utilisée, il arrive déjà à des performances qui sont similaires à celles d'un simulateur écrit manuellement alors que le code généré est naïf et non optimisé.

Le temps pour concevoir un nouveau simulateur est ainsi fortement réduit, une description d'une architecture étant plus simple à écrire qu'un *frontend* ou un *backend* pour un traducteur binaire dynamique. De plus, une partie de la description peut souvent être extraite directement de la documentation de l'architecture. La puissance de notre flot s'exprimera complètement lorsque la forme étendue sera à même de re-découper les opérations trop grandes pour les registres de la machine hôte, et lorsque notre prototype génèrera son propre *backend*.

Un autre point d'amélioration concerne la gestion des drapeaux du processeur cible. À terme, nous souhaitons enrichir MDS avec une description explicite de la sémantique des drapeaux. En plus de permettre d'inférer les instructions conditionnelles de l'architecture hôte pour la traduction des micro-opérations de l'ensemble minimal, cela permettrait de gérer efficacement les instructions cibles modifiant les drapeaux. En effet, il existe des techniques d'évaluations paresseuses des drapeaux en simulation qu'il serait ainsi possible de mettre en place de manière automatique.

Chapitre 6

Conclusion

L'OBJECTIF global de cette thèse est de proposer des solutions pour simuler les systèmes numériques intégrés plus largement et plus rapidement grâce à la technique de traduction binaire dynamique. Dans la problématique détaillée à laquelle nous nous intéressons, nous avons posé trois questions autour desquelles ces travaux se sont articulés. Nous avons tenté d'y répondre au fil de ce manuscrit et les reprenons maintenant afin de faire le bilan des réponses que nous y avons apportées.

La première question vise la simulation d'architectures de processeurs à parallélisme explicite et était ainsi formulée :

Comment utiliser la traduction binaire dynamique pour simuler des jeux d'instructions VLIW sur des processeurs scalaires ?

La solution que nous proposons au chapitre 4 se base sur une technique de réécriture des registres employés dans les instructions simulées pour prendre en compte leur parallélisme et leur latence. L'algorithme proposé est applicable à toute technique de simulation et pas seulement à la DBT. Cependant, l'utilisation naïve de cette stratégie au sein d'un traducteur binaire dynamique soulève un certain nombre de problèmes liés aussi bien à la sémantique des instructions elles-mêmes qu'à la technique de DBT.

Les frontières de TB doivent être gérées avec attention puisque le contexte de traduction est perdu entre la génération de deux blocs, et ce pour conserver leur indépendance. Un mécanisme à l'exécution doit donc être introduit pour conserver les parties importantes de l'état implicite du processeur, et notamment les instructions encore dans pipeline à la fin du TB. De plus, pour assurer cette indépendance entre les TB, un état connu des répliques utilisées doit être assuré entre chacun d'eux. Un certain nombre de déplacements de données est donc effectué en fin de TB pour garantir cet état.

Les instructions conditionnelles posent aussi problème. Le traducteur ne connaît pas le résultat d'une condition puisque la traduction a lieu en dehors de tout contexte d'exécution. Il doit cependant garantir que la bonne valeur du registre se trouve dans la réplique courante à tout moment, que la condition soit vraie ou non à l'exécution. Il est donc parfois obligé de conserver un drapeau pour connaître la valeur de la condition plus tard dans un TB afin de prendre les mesures nécessaires si la condition se révèle fausse.

Les deux dernières questions abordent le problème de la complexité de conception des simulateurs basés sur la DBT, et sur la possibilité d'optimiser le processus pour un couple cible – hôte donné.

Comment générer des simulateurs d'ISA basés sur des techniques de DBT ?

Le flot présenté au chapitre 5 permet de générer facilement un traducteur en partant de la description architecturale de la cible et de l'hôte. Le *frontend* et le *backend* du simulateur peuvent être produits automatiquement en minimisant l'intervention manuelle du concepteur. Les descriptions sont bien plus faciles à écrire et à mettre au point qu'un cœur de traduction binaire dynamique. Une partie de ces descriptions peut même souvent être extraite directement de la documentation de l'architecture.

Cependant, il est difficile d'effectuer des optimisations à l'exécution du code généré sans conserver une représentation intermédiaire sur laquelle s'appuyer pendant la phase de traduction. C'est pourquoi la dernière question aborde le sujet de l'optimisation du couple :

Comment optimiser le processus de DBT généré pour le couple cible/hôte donné ?

Nous proposons une solution de génération d'un traducteur en partant de la description architecturale de la cible et de l'hôte. Un processus de mise en correspondance des instructions de ces deux architectures permet de sélectionner la ou les instructions hôtes pour simuler au mieux une instruction cible. Ce processus est grandement facilité par la forme étendue des nœuds dans les arbres de comportement des instructions. L'algorithme exhaustif possède une complexité pire-cas élevée mais s'avère en pratique relativement rapide, et a été expérimenté sur plusieurs cas. La représentation intermédiaire spécialisée pour le couple qui en découle est très prometteuse pour exploiter au mieux les particularités du jeu d'instructions hôte.

Perspectives

En plus des améliorations techniques à apporter pour faire d'un prototype un outil effectivement exploitable, les directions à creuser que nous avons identifié sont les suivantes :

- La technique de retour à l'état canonique dans les petits TB peut s'avérer coûteuse car elle génère des mouvements de données. Une solution de réécriture *a posteriori* de la génération de TB permettrait d'allouer intelligemment les répliques de manière à se trouver dans l'état canonique à la fin du TB sans autres mouvements de données ;
- Le support à l'exécution de mémorisation de l'état du pipeline du processeur gagnerait à être allégé puisqu'il génère un nombre d'appels de fonctions non négligeable ;
- Concernant la génération de traducteurs, un support explicite des drapeaux des processeurs décrits en MDS permettrait de réduire la partie “Support hôte” du flot, écrite manuellement, et d'ouvrir la voie à des techniques de simulation optimisées des drapeaux du processeur cible ;
- Par ailleurs, un support des instructions cibles utilisant des registres plus larges que ceux disponibles sur l'hôte finirait de couvrir les instructions cibles lors du processus de mise en correspondance.

Annexe A

Preuve de correction de l'algorithme de simulation VLIW

A.1 Définitions préliminaires

Nous considérons deux machines VLIW, une réelle, et l'autre simulée, toutes deux possédant r registres numérotés de 0 à $r-1$. L'unité de temps est le cycle processeur, avec t_0 le cycle initial.

L'état s d'une machine représente l'état de chacun des registres de la machine, c'est-à-dire leur valeur. La notation $s(i)$ dénote l'état du registre i dans s. Par ailleurs, les deux machines sont supposées démarrer dans le même état initial s_0.

E est l'ensemble des paquets d'exécutions du programme à exécuter sur les deux machines. $E(s)$ représente le prochain paquet d'exécution à exécuter d'après l'état s. Un paquet d'exécution ep contient des instructions `insn` exécutées en parallèle. Pour toute instruction `insn`, notons

- $\texttt{out}(\texttt{insn})$ l'indice du registre dans lequel `insn` écrit son résultat ;
- $t_{\texttt{in}}(\texttt{insn})$ la date à laquelle l'instruction `insn` entre dans le pipeline d'exécution ;
- $t_{\texttt{out}}(\texttt{insn})$ la date à laquelle le résultat de l'instruction `insn` devient visible dans $\texttt{out}(\texttt{insn})$;
- $d(\texttt{insn})$ la latence de `insn`. Elle est définie comme $d(\texttt{insn}) = t_{\texttt{out}}(\texttt{insn}) - t_{\texttt{in}}(\texttt{insn})$;
- $\texttt{res}(\texttt{insn}, s)$ le résultat produit par l'instruction `insn` à partir de l'état s.

A.2 L'algorithme de simulation d'une machine VLIW

Grâce à ces notations, nous pouvons réécrire l'algorithme 4 page 36. Cette nouvelle écriture est donnée par l'algorithme 8.

On note $\texttt{tmp}(i,t)$ la variable temporaire destinée à accueillir le résultat d'une instruction écrivant dans i à la date t. Si une telle instruction n'existe pas, $\texttt{tmp}(i,t)$ vaut `null`. À la ligne 2, on récupère le paquet courant dans ep. On simule ensuite chaque instruction, le résultat étant stocké dans la variable temporaire qui lui est attribué. Après la simulation du paquet, chaque registre i du processeur simulé est mis à jour avec l'éventuelle variable temporaire $\texttt{tmp}(i,t)$ et t avance d'un cycle.

Algorithme 8 Réécriture de l'algorithme de simulation VLIW avec les précédentes notations

```
1  tant que la simulation n'est pas terminée faire
2      ep ← E(s_m)
3      pour chaque insn ∈ ep faire
4          tmp(out(insn), t_out(insn)) ← res(insn, s)
5      fin pour
6      pour chaque i ∈ {0, ..., r − 1} faire
7          si tmp(i, t) ≠ null alors
8              s(i) ← tmp(i, t)
9          fin si
10     fin pour
11     t ← t + 1
12 fin tant que
```

A.3 Formalisation de l'état de la machine

À partir de cet algorithme, il est possible d'exprimer s en fonction du temps, noté $s_e(t)$. Cette notation représente l'état de la machine simulée à la date t. La notation $s_e(i,t)$ quant à elle représente l'état du registre i à la date t. De la même manière, on note $s_m(t)$ l'état de la machine réelle à la date t.

D'après l'algorithme 8, $s_e(i,t)$ peut s'exprimer ainsi :

$$s_e(i,t) = \begin{cases} s_0(i) & \text{si } t = t_0 \\ \texttt{tmp}(i,t) & \text{si } t > t_0 \text{ et } \texttt{tmp}(i,t) \neq \texttt{null} \\ s_e(i,t-1) & \text{sinon.} \end{cases} \tag{A.1}$$

En effet, à $t = t_0$, $s_e(i,t_0)$ vaut la valeur initiale $s_0(i)$ du registre i. Autrement, s'il existe une variable temporaire $\texttt{tmp}(i,t)$, elle est appliquée à $s_e(i,t)$. Sinon, la valeur du cycle précédent $s_e(i,t-1)$ est conservée.

De la même manière, l'état s_m de la machine réelle peut s'écrire :

$$s_m(i,t) = \begin{cases} s_0(i) & \text{si } t = t_0 \\ \texttt{res}(\texttt{insn}, s_m(t_{\texttt{in}}(\texttt{insn}))) & \text{si } \exists \texttt{insn} : t_{\texttt{out}}(\texttt{insn}) = t \text{ et } \texttt{out}(\texttt{insn}) = i \\ s_m(i,t-1) & \text{sinon.} \end{cases} \tag{A.2}$$

La différence majeure entre $s_e(i,t)$ et $s_m(i,t)$ est que $s_m(i,t)$ se limite à décrire le comportement du processeur VLIW sans spécifier les mécanismes sous-jacents à la mise à jour des registres alors que $s_e(i,t)$ utilise explicitement $\texttt{tmp}(i,t)$ pour stocker de manière temporaire les résultats des instructions.

Pour prouver que le simulateur a le même comportement que la machine réelle, il nous faut alors prouver le théorème suivant :

Théorème 2. *Pour tout instant $t \geq t_0$ et pour tout registre $i \in \{0,\ldots,r-1\}$, $s_e(i,t) = s_m(i,t)$.*

Démonstration. Commençons par caractériser $\texttt{tmp}(i,t)$. D'après l'algorithme 8, si à la date t, il existe une variable temporaire $\texttt{tmp}(i,t)$, c'est qu'il existe une instruction `insn` dans le programme telle que $\texttt{out}(\texttt{insn}) = i$ et $t_{\texttt{out}}(\texttt{insn}) = t$. Par ailleurs, on connait le résultat de cette instruction grâce à la ligne 4 de l'algorithme :

$$\texttt{tmp}(\texttt{out}(\texttt{insn}), t_{\texttt{out}}(\texttt{insn})) = \texttt{res}(\texttt{insn}, s_e(t)) \tag{A.3}$$

On peut donc en déduire l'expression de $\texttt{tmp}(i,t)$ lorsque insn existe :

$$\begin{array}{rrcl} & \texttt{tmp}(i, t + d(\texttt{insn})) & = & \texttt{res}(\texttt{insn}, s_e(t)) \\ \Leftrightarrow & \texttt{tmp}(i,t) & = & \texttt{res}(\texttt{insn}, s_e(t - d(\texttt{insn}))) \\ \Leftrightarrow & \texttt{tmp}(i,t) & = & \texttt{res}(\texttt{insn}, s_e(t_{\texttt{in}}(\texttt{insn}))) \end{array} \tag{A.4}$$

Finalement, dans le cas général, $\texttt{tmp}(i,t)$ s'écrit :

$$\texttt{tmp}(i,t) = \begin{cases} \texttt{res}(\texttt{insn}, s_e(t_{\texttt{in}}(\texttt{insn}))) & \text{si } \exists \texttt{insn} : t_{\texttt{out}}(\texttt{insn}) = t \text{ et } \texttt{out}(\texttt{insn}) = i \\ \texttt{null} & \text{sinon} \end{cases} \tag{A.5}$$

Nous pouvons donc réécrire $s_e(i,t)$ comme suit :

$$s_e(i,t) = \begin{cases} s_0(i) & \text{si } t = t_0 \\ \texttt{res}(\texttt{insn}, s_e(t_{\texttt{in}}(\texttt{insn}))) & \text{si } \exists \texttt{insn} : t_{\texttt{out}}(\texttt{insn}) = t \text{ et } \texttt{out}(\texttt{insn}) = i \\ s_e(i, t-1) & \text{sinon} \end{cases} \tag{A.6}$$

Aux vues des expressions (A.2) et (A.6), par récurrence immédiate, on obtient que $s_e(i,t) = s_m(i,t)$ pour tout $t \geq t_0$ et $i \in \{0, \ldots, r-1\}$. □

Puisque les deux états sont les mêmes à tout moment, nous pouvons en déduire que l'algorithme 3 de simulation d'un processeur VLIW produit le même comportement que le processeur réel, sous l'hypothèse que les deux démarrent dans le même état.

Annexe B

Résultat d'expérimentations du simulateur VLIW sur la suite Polybench

Dans cette annexe, nous présentons d'avantage de tests de performances réalisés sur notre implémentation de simulateur `c6x` dans QEMU, et ce comparé au simulateur CCS fourni par Texas Instruments. Ces tests s'appuient sur la suite Polybench[Pou11] qui fournie un ensemble de noyaux d'applications réalistes de divers natures. Les résultats sont présentées par la figure B.1.

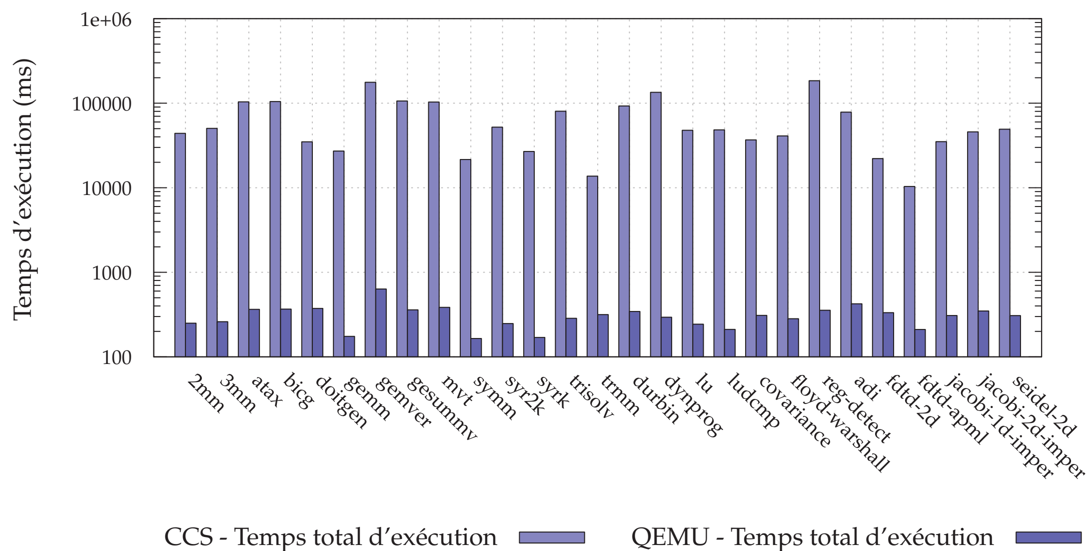


FIGURE B.1 – Temps d'exécution de la suite Polybench

Ils confirment les expérimentations de la section 4.3.3 page 53. Notre implémentation est plus rapide de deux à trois ordres de grandeur par rapport à CCS. Ces résultats montre aussi la maturité de notre simulateur, les tests réalisés étant variés et complexes.

Annexe C

Génération automatique de décodeurs d'instructions

Lors des travaux sur la génération automatique de simulateurs présentés au chapitre 5, nous avons été amené à travailler sur un problème qui en découle, la génération de décodeurs d'instructions. À partir de la description du jeu d'instructions d'une architecture, il s'agit de générer le code du décodeur qui sera utilisé par le simulateur.

Dans cette annexe, nous joignons l'article exposant notre solution, publié dans la conférence internationale ICCAD en 2013[FMP13]. Il présente deux solutions, une première générant rapidement un décodeur valide pour l'architecture, et une deuxième cherchant à générer un décodeur optimal en terme de nombre moyen d'opérations effectuées pour décoder une instruction. Pour cela, elle se base sur des statistiques d'utilisation des instructions dans des programmes divers pour connaitre les instructions à privilégier en terme de simplicité de décodage. Ces deux méthodes utilisent une représentation du jeu d'instructions innovante basée sur les arbres de décision binaire (Binary Decision Diagram (BDD)). Cette représentation permet de prendre en compte les motifs compliqués des jeux d'instructions d'aujourd'hui, qu'une représentation ternaire du type $(0, 1, -)$ ne peut capturer simplement.

Les deux méthodes produisent des résultats très satisfaisants puisque les décodeurs générés sont dans la plupart des cas plus rapides que des décodeurs écrits à la main. La deuxième méthode est mise en œuvre dans le prototype présenté au chapitre 5 pour générer le décodeur d'instructions de l'architecture MIPS.

Automated Generation of Efficient Instruction Decoders for Instruction Set Simulators

Nicolas Fournel, Luc Michel and Frédéric Pétrot
TIMA Lab, CRNS/Grenoble-INP/UJF, Grenoble, France
{Nicolas.Fournel,Luc.Michel,Frederic.Petrot}@imag.fr

***Abstract*—Fast Instruction Set Simulators (ISS) are a critical part of MPSoC design flows. The complexity of developing these ISS combined with the ability to extend instruction sets tend to make automated generation of ISS a need. One important part of every ISS is its instruction decoder, but as the encoding of instruction sets becomes less orthogonal because of the incremental addition of instructions, the generation of a decoder is not anymore an obvious task. In this paper, we present two automated decoder generation strategies that are able to handle non-orthogonal instruction encodings. The first one builds a decision tree that does not consider the instruction's occurrences while the second considers these frequencies. In both cases, we use binary decision diagrams to represent the instructions encodings and the complex conditions due to the non-orthogonality of the encodings in order to generate the decoders. Our experiments on the MIPS and ARM (including VFP and Neon extensions) instruction sets show that both algorithms produce efficient decoders, and that it is beneficial to consider instruction frequencies.**

I. Introduction

Instruction Set Simulators (ISS) have been used for decades in computer architecture studies, and their automated generation, usually linked with actual hardware generation, has been the subject of many works. This topic has reached a level of maturity such that several textbooks on the subject are now available, [1], [2] among others, and several companies (*i.e.* Asip Solutions, Synopys, Target, Tensilica to name a few) are selling industrial level tools for that purpose. Among the necessary steps to automate their production, instruction decoding is an issue that has seldom been described in detail. Indeed, instruction decoding can be considered simple when looking at early instruction sets, for which encodings where *orthogonal*, *i.e.* the distinction between any pair of instructions could be merely done in a single step using a mask and a comparison. However, because of the quest for upward binary compatibility while having the architecture become more efficient by enhancing its ISA, the coding pressure on the unused bits combinations is becoming very high. This applies to both hardwired processors, with the addition of floating point units and/or SIMD extensions [3], [4], and to extensible processors (ASIPs) whose instruction set is extended to fit a particular application class, either by the user or automatically [5]. As a result, more and more instructions must fit into the instruction word, leading to the use of specific values or value exclusions in bit fields otherwise dedicated to parameters (registers numbers or flags) to code the new instructions. Even though the linear time complexity traversal of a sorted list of mask/comparison pairs can be used for identifying unambiguously an instruction, we are not aware of any published work that has presented more efficient strategies to solve this issue.

The goal of this paper is to present two fully automated strategies for the generation of instruction decoders for processor simulators. The approaches use reduced ordered binary decision diagrams (RO-BDD) to represent the instruction set in a canonical and compact manner. The first strategy takes only the instruction set as input, while the second one also handles instruction frequencies.

This work was partially funded by the French Ministry of Industry though the BGLE ACOSE project.

The paper is organized as follows. Section II formulates the problem and presents the relevant related works. Section III defines the instruction decoder construction problem. Section IV details our RO-BDD based solution. Section V presents the application of our solution on the full MIPS 4k and ARM V7 instruction sets. Section VI summarizes our contributions and draws follow-ups of this work.

II. Related work

Even though the problem of building instruction decoders has been existing for decades, to the best of our knowledge very few detailed works have been published on this topic. The common solution, used *e.g* in [6], [7], [8] and generalized in the *opcodes* library [9] used by gdb and the GNU binutils, is to traverse a list of encodings representation covering all instructions, with possibly more than one representation identifying a given instruction. The list needs to be sorted in such a way that an encoding of an instruction in the list cannot be absorbed by the encoding of an instruction that occurs before in the list. The algorithmic complexity of the decoder is in $O(c)$, where c is the number of instructions in the ISA, which is acceptable for disassembling or step-by-step debugging, but not for intensive instruction decoding programs like ISS.

The modelling of the problem of generating an instruction decoder is covered in [10], in which the authors show that is it equivalent to find a decision tree for decoding a binary sequence. The algorithm they propose to produce the decision tree is working on the set of instruction encodings, and recursively tries to split the set by choosing tests bits until the set reaches a size of one. Some other works, like [11], even propose to build an optimal decision tree based on the occurrences of instructions. The goal is to reduce the average number of test made to actually decode often used instructions. The problem can then be seen as building a decision tree and minimizing the average path length.

These previous works and some others, like [12], rely on a representation of the instruction encoding in which each bit can take one of the three values {`0`,`1`,`-`}, where '`-`' stands for *"don't care"*.

To illustrate this instruction encoding representation, Table I is depicting a simple instruction set made of three 8-bit instructions. The c, v, imm_4 and x fields represent symbols (flags or register numbers), which are parameters of the behavior of the instruction and are usually not significant bits for matching the encoding. They are replaced by '`-`' in the representation.

The simple instruction set will then be represented as follows: (A, `------00`), (C, `----0-01`) and (D, `----1-01`).

As noted, new instruction encodings are added in modern instruction sets by filling unexploited bit combinations, leading to *non-orthogonal* instruction encodings. Assuming that the behavior of A is unpredictable if c equals to 111, we can add a new instruction B in our simple instruction set. The result is depicted in Table II. The instructions A and B are then subsumed. The straightforward representation can hardly catch these cases.

TABLE I
SIMPLE INSTRUCTION SET EXAMPLE

Instruction	7	6	5	4	3	2	1	0
A	c			v	r_0		0	0
C	imm$_4$				0	x	0	1
D	imm$_4$				1	x	0	1

TABLE II
EXTENDED INSTRUCTION SET EXAMPLE

Instruction	7	6	5	4	3	2	1	0
A	c			v	r_0		0	0
B	1	1	1	v	r_0		0	0
C	imm$_4$				0	x	0	1
D	imm$_4$				1	x	0	1

A first solution consists of handling the subsumed cases as a single instruction encoding (in our case the A encoding) and making the distinction between the two instructions in the behaviour description, as suggested in [6]. This mix of encoding and behavior can definitely not be applied in the extensible processor scope and the complete ISS generation process where encoding and behavior should be independent information.

An other solution is to order the representation evaluation to ensure that the more constrained one (B in our case) is evaluated/tested before the less constrained one (A). This solution is used for most of the GNU *opcodes* library CPU target supports. Unfortunately, this leads to testing the less frequent instructions before the most frequent ones.

A last solution to handle these subsumed cases, proposed in [10], is to consider the less constrained one as default case. All the common significant bits of subsumed pairs are consumed and then, if one encoding has no more significant bits (A), the identification is made on the significant bits of the other (B) and the first one (A) is chosen as default case. In such a solution, the two instructions are handled together in the generation algorithm and can only be mutually excluded at the last inner node of the decision tree.

The last two solutions cannot be used in the optimal decision tree generation algorithms of [11], since they fix the order in which these subsumed instructions may be decoded. This constrain reduces the efficiency of the generated decision tree.

On top of these simple subsumed cases which found more or less efficient solutions in the literature, the insertion of new instructions can be even more tricky. This is for example the case with the 6-bit immediate variante of the VCVT instruction in the ARM Neon SIMD extension, which collides with a VMOV for specific values of the immediate. These complex cases can only be represented with conditions expressed on the symbols that are usually *"don't care"* values in the representation. Table III offers an update of the synthetic instruction set summarizing these cases with instruction A.

TABLE III
FULL INSTRUCTION SET EXAMPLE

Instruction	7	6	5	4	3	2	1	0	Condition
A	c			v	r_0		0	0	$c \neq 111$ $\neg(v = 1 \wedge c \neq 000)$
B	1	1	1	v	r_0		0	0	
C	imm$_4$				0	x	0	1	
D	imm$_4$				1	x	0	1	

The only way of matching one instruction of this kind is to either enumerate all the matching cases or make sure that all unmatching cases are expressed in the encoding list as other instruction encodings or undefined encodings. As our solution wants to be comprehensive, it is necessary to work on a compact and efficient representation of the encoding set, along with effective ways of performing operations on it to catch complex conditions of complex instruction set encodings and allow optimal decision tree generation.

III. PROBLEM DEFINITION

A. Instruction decoder definition

An instruction can be defined as a bitstring $\zeta \in \mathbb{B}^m$, where $\mathbb{B}$ is the boolean domain and m is the size of the instruction. For embedded processor it is usually 32-bit, but theoretically, choosing m as the upper bound over all instruction sizes and using *don't care* where necessary allows to handle variable size instruction sets.

Decoding an instruction ζ can be done using a decision tree [10] whose set of nodes V is partitioned in an inner node set N and a leaf nodes set L. Inner nodes ($\nu \in N$) are testing parts of ζ and leaves ($\lambda \in L$) mark the matching of ζ with a specific instruction of the targeted processor architecture. Building an instruction decoder consists of building this decision tree using a description of the instruction encodings.

Instruction encodings are usually represented using a couple (r, l) named decoding entry, with r representing the instruction encoding and l is a label for this encoding [10]. Occurrences of instruction encodings can be taken into account to tune the instruction decoder to get better performances in the decoding task at run time. If so, decoding entries are triples (r, l, o) where $o \in [0, 1]$ is the occurrence information usually expressed as a probability of matching the encoding in the instruction decoder (*e.g.* in [11]).

For our purpose, an instruction set is defined as a set of decoding entries E which is used as the input of the instruction decoder generator. Thus the decision tree is in charge of mapping any bitstring ζ (instruction to be decoded) to a unique decoding entry (instruction encoding) $e \in E$. For this reason, the decoding entry set has to ensure the absence of collision between decoding entries to guaranty that a valid instruction decoder can be built. Indeed, collisions would lead to multiple encoding entry matches for some input bitstrings.

The instruction encoding description should be adapted to allow this verification, and this must not be neglected since it fits into a fully automated code generation process.

B. Instruction encoding representation

As encodings increase in complexity, having a representation of the instruction expressive enough to allow the correctness verification becomes difficult.

1) Standard encoding representation: In existing works, the representation of instruction encodings is handled by using bit patterns. A bit pattern is defined as $p \in \{0, 1, -\}^m$.

With this encoding representation, the operation of matching a bitstring with a decoding entry is defined as a bitstring ζ matches a bit pattern p iff $\forall i \in \{0, \ldots, m-1\}, p[i] = '-' \vee p[i] = \zeta[i]$.

The naive representation of the instruction set described in III (A, `------00}`),(B, `111---00}`),(C, `----0-01}`)and (D, `----1-01}`)leads to a mal-formed decoding entry set, since this representation is unable to catch the exclusion cases. As a consequence, the bitstring `11110000}`can eventually match the two first instruction encodings (A and B).

With this representation of instruction encodings, the only way to catch the exclusion cases is to exhaustively enumerate all the bit patterns corresponding to the complex instruction encodings (like

instruction A in Table III), or ensure that all exclude bit pattern are expressed through other instruction encoding bit patterns or by inserting undefined instruction encoding bit patterns. This solution implies a combinatorial explosion of bit patterns to be placed in the decoding entry set and forces an order for their evaluation.

On top of that, in the case of occurrences based instruction decoder construction, this explosion opens questions. If an instruction requires the enumeration of all its encodings, the occurrences of instruction encoding must either be spread over bit patterns or all bit pattern occurrences must be provided by the user.

C. Expressive representation of instruction encodings

We explore another way to represent these formats with a representation widely used in logic synthesis and formal verification, the BDD.

1) Binary Decision Diagrams: The binary decision diagram (BDD) aims at representing a boolean function in a convenient form [13]. A BDD is a rooted directed acyclic graph. A node represents a variable of the function. Each node has two edges, one that considers the variables to be false, and another one that considers it to be true. Two special terminal nodes give the final answer on the function value. Once one of them is reached, we know whether the function is true or false, and the path we took in the diagram to arrive to this conclusion. The figure 1 gives a BDD of the function $f(a,b,c) = (a \vee b) \wedge c$. On graphical representations, a dashed (resp. plain) edge means the previous variable is false (resp. true). We can deduce from the diagram that there are two

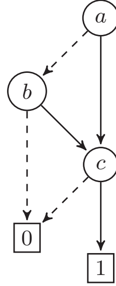


Fig. 1. BDD for the function $f(a,b,c) = (a \vee b) \wedge c$

paths that satisfy the function, $\{(a, false), (b, true), (c, true)\}$ and $\{(a, true), (c, true)\}$. When a variable does not appear in a path like variable b in the second path, it means that it can take any values without changing the final result. Note that a satisfying path can also be represented with a vector of elements in {`0`,`1`,`-`}, sized with the number of variables in the function. The variables are ordered so we know which vector element represents which variable.

With the previous example, if we choose the order (`abc`), the two satisfying paths become (`011`) and (`1-1`)

2) Instruction encoding representation using BDDs: In order to characterize instruction encodings in a more expressive way, we propose to use the BDD data structure. In that way, the conditions of *non-orthogonal* instructions can be represented canonically and more conveniently, *i.e.* in a non exhaustive way.

Figures 2(a), 2(b), 2(c), 2(d) respectively give the BDD for the instructions A, B, C and D presented in Table III.

BDD representation of instructions B, C and D encodings show that only one path is satisfying each of this encodings. As for the encoding of instruction A, 4 paths satisfy the BDD. These 4 paths are due to the complex conditions existing on this encoding.

IV. Decoder generation for complex ISA

The input of the instruction decoder generator is the set of decoding entries provided by the user as matching and exclusion cases. This information is similar to what has been presented in Table III. From these informations, the generator produces the instruction encoding representation based on BDD.

A. Generation principles

The process of instruction decoder generation consists of building the decision tree *i.e.* defining the set of nodes V and the tree structure. On top of that, each inner node ν needs to be labeled with the few tested bits, called decision function ϕ, and each leaf node λ needs to be labeled with the matched decoding entry or as an undefined instruction encoding.

1) Decision functions: As proposed in previous works [10], [11], the decision functions ϕ of each inner node of the produced decision tree are limited to very simple functions. The idea is to limit the computational complexity of the decision functions to $O(1)$.

These functions are of two different types.

- bit patterns: this type of decision functions tests if a subset of bits of the instruction ζ matches specified values. This type of decision functions is usually implemented as bitwise boolean functions whose operands are (mask, value) pairs,
- tables: this type of decision functions indices a table using a subset of bits of ζ. This type of decision functions can be implemented as switch statements, function pointer tables, ...

2) Decoder tree structure building: The decoder tree structure building process starts with the complete set of decoding entries from which it defines the node set V and the associated decision functions. This process is intrinsically recursive and tries to split the set of decoding entries E into decoding entries subsets E_i using a decision function ϕ. Algorithm 1 gives an high-level view of the tree construction. At each level of recursion, the algorithm works on the remaining decoding entries. These entries are the only decodings entries matching the sequence of decision functions traversed from the root to the current node. he remaining instructions will necessarily be decoded by the subtree grafted on the current node.

Algorithm 1 Decoding tree construction principle

```
    function DEC_GEN(E)
2:      if |E| = 1 then
            return leaf labeled with the unique entry e ∈ E
4:      end if
        select a decision function φ
6:      split E into k subsets E_i using φ
        for each E_i ≠ ∅ do
8:          call DEC_GEN(E_i)
        end for
10: end function
```

The number of recursive calls depends on the number of children and thus on the selected decision function. With bit pattern decision functions, there are two children for the current node whereas with table decision functions there are 2^q potential children for a index encoded on q bits.

The most important step in this algorithm is the split of the decoding entry set E. It is straightforward with the bitpattern representation of instruction encodings but needs a new definition of the

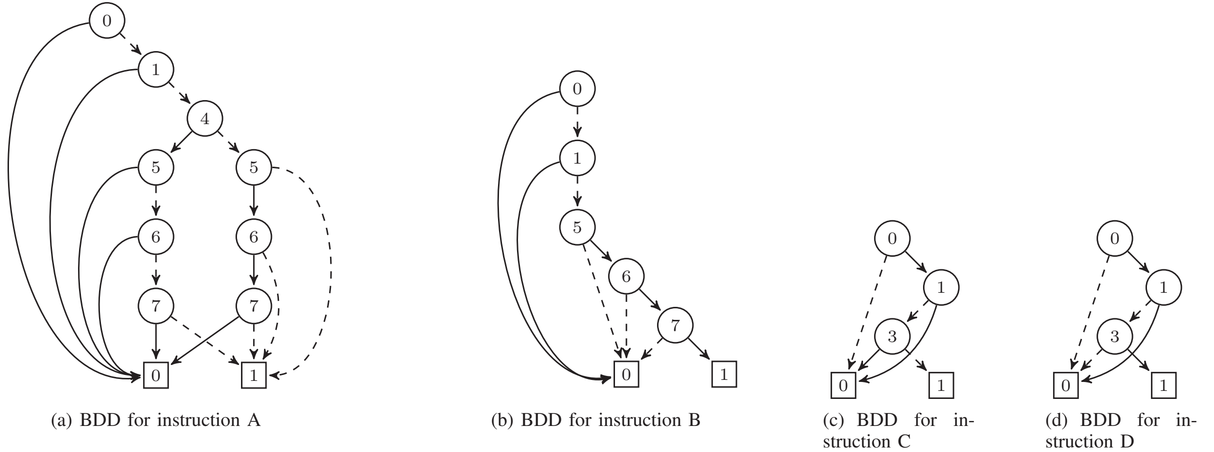


Fig. 2. BDD representation of instructions A (a), B (b), C (c) and D (d), the node label being the instruction bit number

match operation for the BDD representation of instruction encodings. Indeed, the splitting step can then result in two different situations. Either the decision function induces a unique match on a decoding entry or it is inducing multiple matches, which is the case when non-significant bits are involved in the tested bits of the function. In the first case the decoding entry can be added to the corresponding subset E_i. In the second case it is added to all matching subsets E_i but if occurrences are used for the generation, the added decoding entries are (r, l, o_i) where $\sum_i o_i = o$.

We propose two different solutions in the two following algorithms. The first one fallbacks on the standard bit pattern match operation, whereas the second one defines an *ad-hoc* a match operation for the BDD based representation.

B. *Equiprobable partitioning algorithm* (PART)

The algorithm is composed of two separate parts, the first preprocesses the encoding representations and the second builds the tree.

Preprocessing consists of extracting the satisfying paths from all the instructions BDD representations to fallback on the standard bit pattern representation. To this end, we use a specific operation on BDD called *sat*. The resulting *solution set* S is a set of pairs composed of an instruction and a satisfying path. An instruction can appear more than once in S, since it can have multiple satisfying paths (like instruction A in Table III).

The second step recursively builds the decision tree using S. Algorithm 2, initially called with S computed by preprocessing and $d_p = 0$, details the sequence of operations of each recursion. The satisfying paths are represented using the {0,1,-} notation. For the sake of readability, in the algorithm, they are split into binary values pairs (*mask*, *value*) where a '1' bit in the *mask* means that the corresponding bit in *value* is significant and a '0' is equivalent to the - value. We use the operations mask(s) and val(s) on a satisfying path s to retrieve its mask and value respectively.

Lines 5 and 6 are the specialized versions of the *select* and *split* operations introduced in Algorithm 1. The *select* operation consists in computing the *decision bits* d which is composed of the common bits in the mask solutions. All the bits selected by previous iterations are also removed from d. Then, the *split* operation creates the new subsets by grouping the solutions with the same *decision value*. The solution decision value v_j is the result of the disjunction between the solution value val(s) and d.

Algorithm 2 PART algorithm

```
    function PART(S, d_p)
 2:     if |S| = 1 then
            return leaf labeled with the unique entry s ∈ S
 4:     end if
        d ← ¬d_p ∧ (⋀_{s_i ∈ S} mask(s_i))
 6:     T = {S_{v_j} / s_i ∈ S_{v_j} ⇔ val(s_i) ∧ d = v_j}
        for each S_{v_j} ∈ T do
 8:         call PART(S_{v_j}, d_p ∨ d )
        end for
10: end function
```

For our instruction set example given in Table III, the preprocessing step produces a solutions set containing the following pairs:

(A, 0001--00)	(B, 111---00)
(A, --00--00)	(C, ----0-01)
(A, -010--00)	(D, ----1-01)
(A, 0110--00)	

The first iteration gives us $d =$ (00000011). The solution set is then split the following way:

Subset 1, $v_1 =$ ------00	Subset 2, $v_2 =$ ------01
(A, 0001--00)	(C, ----0-01)
(A, --00--00)	(D, ----1-01)
(A, -010--00)	
(A, 0110--00)	
(B, 111---00)	

As we can see, all solutions with the same value on the two least significant bits are in the same subset. This value corresponds to the *decision value* for the first iteration.

The algorithm stops when the solutions subset contains only one solution. The resulting node is a leaf in the final decoding tree. However, if all significant bits of the instruction encoding are not used, a new child node is created and labeled with the remaining bits. Indeed, without this node, some illegal instructions can be wrongly matched.

The resulting decoding tree is depicted in Figure 3, in which each node contains its corresponding instruction subset and its decision bits mask, if the node is not a leaf. Each edge carries also the decision bits value. For example, for the C instruction, the decoding tree first tests the two lower bits (masked with 0x03). C instruction has a

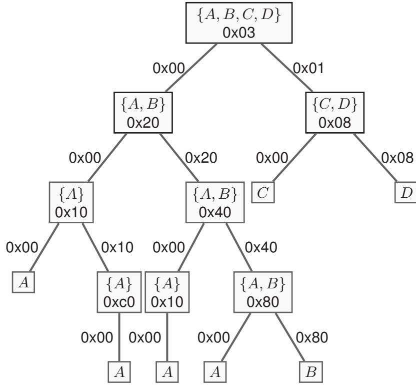


Fig. 3. Resulting decoding tree

`0x01` value for these two bits. At the second level, the decoding continues with the mask `0x08`, where `C` has this bit to `0`, conversely to instruction `D`. Decoding of instruction `C` is then completed since by following edge `0x00`, we end in a leaf of the decoding tree. In this decoding tree, instruction A appears in four leaves, matching exactly the 4 paths that lead to true in the BDD of Fig. 2(a).

C. *Efficient occurrence aware decoding tree algorithm (*EFF*)*

The previous algorithm shows that using BDD for instructions encoding helps in catching the complex conditions of these encodings and gives a first way to solve the match operation with BDD representation. However, this algorithm does not handle instructions occurrences since it builds the straightforward decoding tree. In the goal of building an efficient decoder or even trying to build the near-optimal one for a given set of occurrences, the construction process turns into finding the decision tree with the minimal average path length.

As the matching is obtained in leaves, the cost of decoding is related to the number of nodes traversed in the path from root to the matching leaf (*i.e.* to the number of decision functions tested). With the $O(1)$ constraint for decision functions, it is equal to the height of its matching leaf (path length).

The average cost of an instruction decoder tree is thus given by its average height H_{avg} defined in (1), where D_i is the height of the matching leaf for the (r_i, l_i, o_i) decoding entry.

$$H_{avg} = \sum_i o_i D_i \tag{1}$$

In the recursive construction process, the evaluation of the current subtree cost can then be basically obtained with H^c (2), where O_i is the cumulative occurrence of the subset i.

$$H^c = 1 + \sum_i O_i H_i^c \tag{2}$$

As can be seen from (2), the evaluation of a subtree involves the complete recursive children subtrees construction. Due to the high complexity of this solution, the cost of children subtrees is estimated using Huffman tree height ($\widetilde{H}_i^c$)[14]. In that condition, the recursive calls are only made once, on the subsets created by decision function having the lowest H^c. The construction algorithm for occurrence aware instruction decoder construction, using Algorithm 1's template, is presented in Algorithm 3.

The estimation of the cost of the subtree can optionally be adapted to take into account the memory usage of the decoder, as suggested in [11], by adding an extra member to the H^c evaluation.

Algorithm 3 EFF algorithm

```
    function DEC_GEN(E)
2:      if |E| = 1 then
            return leaf labeled with the unique entry e ∈ E
4:      end if
        populate decision function set Φ
6:      H_min ← ∞
        initialize empty node ν
8:      for each φ ∈ Φ do
            split E into k subsets E_i using φ
10:         evaluate H^c ← 1 + Σ_i O_i H̃_i^c
            if H_min > H^c then
12:             label ν with φ
                H_min ← H^c
14:         end if
        end for
16:     for each E_i ≠ ∅ do
            call DEC_GEN(E_i)
18:     end for
    end function
```

In that context, the match operation solution proposed in the first algorithm is not convenient for two main reasons. First the decoding entries combinatorial explosion due to the expression of all excluding/matching bitpatterns is very costly in term of generation time due to the large number of decoding entry set in the split operation. Second, handling the occurrence information is complicated, either the occurrence is split among bit patterns, which is inaccurate, or user should provide them all, which does make much sense. A second solution is to manipulate the BDD representation in the algorithm steps.

This split operation, more precisely the match operation between a decision function and a decoding entry, cannot be performed with current node information. Indeed, the BDD representation helps to catch the complex instruction encodings and to handle them with a unique formalism. To check if complex conditions are verified or not, the match operation needs to catch the complex sequence of decision functions. For that, the construction algorithm maintains a BDD representation of the sequence of decision functions from decoder root.

The match operation is then handled as described in Algorithm 4. The complete sequence of decision functions is build from upward functions sequence and the current one by performing a boolean AND on the two corresponding BDD using the *apply* BDD operation. The match operation per say is accomplished by applying an AND on the BDD resulting of the previous operation and the instruction encoding BDD. If the resulting BDD is only composed of the false node (denoted $\perp$ in the Algorithm), the decoding entry is not matching.

This basic operation is then used in the decoding entry set splitting step. If the decision function is a pattern function, the match operation is called with a *decfunc* corresponding to the bit pattern tested for the $true$ subset, and with a *decfunc* corresponding to the complemented bit pattern for the *false* subset. If the decision function is a table function, it is called with a *decfunc* corresponding to the index bit pattern of the subset.

The second step in Algorithm 3 to be highly dependent on the

Algorithm 4 BDD based match algorithm

```
  function MATCH(upwardfunc, decfunc, decentry)
2:    matchedfunc ← apply(upwardfunc, AND, decfunc)
      matched ← apply(decentry, AND, matchedfunc)
4:    if matched = ⊥ then
          return false
6:    else
          return true
8:    end if
  end function
```

BDD representation of instruction encoding is the populate operation, which is defining all possible decision functions for the current decoding entry set. This operation is handled by using the variables, *i.e.* bits, involved in the BDD representing instruction encoding. To limit the number of bits involved in the decision function generation for the subsequent recursion level, a given recursion level computes an updated version of the instruction encoding BDD based on the selected decision function. This update is made by applying the BDD *simplify*[15] function on the encodings BDD in order to remove from it the parts that have already been matched. From these updated BDD, the populate operation can infer the decision functions by building a more restricted list of all possible bits of the decoding entry set and reducing the number of generated decision functions.

In practice, the decision function generation and evaluation is an iterative process, in order to limit the search space. For example, the patterns are grown 1-bit at a time by selecting the best one at each step. Likewise, the table are also grown 1-bit at a time from the best 2-bit table. On top of that, the decision function generation can be helped by extracting some special pattern or table from the input instruction set description.

V. Experiments

In order to assess the correctness and evaluate the quality of both algorithms, we have implemented them using the Alliance CAD System BDD package [16] for all BDD matters, and also wrote a C threaded code [17] generator which realises the actual instruction decoding.

We choose as target processor architectures the MIPS 4k (163 instructions) and the ARMv7 (442 instructions, among which many are due to the Neon extension). We selected these ISA not only because they are widely used in embedded systems, but also because optimized handmade versions of the instruction decoders are readily available. We use two benchmark suites, PolyBench [18] and MiBench [19], to evaluate the static and dynamic metrics of the decoders.

A. Decoder Generation

Instruction decoders have been generated for MIPS and ARM, and some resulting statistics are presented in Table IV. This table gives the generated tree depth in terms of its minimum, maximum, average and standard deviation. It also gives an idea of table sizes generated in the tree, by giving the average number of children nodes. Finally, it gives the execution time of the generator which is producing the instruction decoder source code (in seconds).

For the PART algorithm, the decoding entry sets E based on BDD instruction encoding representation are converted into satisfying paths. The result of the preprocessing gives 272 satisfying paths out of 163 instructions encoding for the MIPS architecture, excluding invalid instructions. For the ARM ISA, the 442 instruction encodings give 4362 satisfying paths (excluding invalid instructions). These numbers clearly show that ARM ISA uses a lot more complex encodings than the MIPS one.

TABLE IV
GENERATED DECODING TREES STATISTICS

Algo	ISA	min depth	avg depth	max depth	depth stddev	avg nd dg	gen time(s)
PART	mips	1	7.84	26	8.17	3.13	0.001
PART	arm	2	7.07	14	1.11	11.26	0.397
EFF	mips	1	3.66	8	4.15	4.41	1.656
EFF	arm	2	7.89	18	10.44	4.67	26.175
EFF_OCC	mips	1	2.80	6	3.49	4.62	1.858
EFF_OCC	arm	1	3.74	18	4.71	4.09	79.113

The EFF algorithm is run with two different data sets. First it is run with a uniform data set, which means that all instructions have exactly the same probability, the results of this particular generation are mentioned as (EFF). Second it is trained on PolyBench based occurrences, which gives the decoder mentioned as EFF_OCC. Table V gives an overview of the ten most occurring instructions in the aggregated PolyBench traces for each architecture.

TABLE V
MOST OCCURRING INSTRUCTIONS IN POLYBENCH TRACES

ARM		MIPS	
insn	o_i (%)	insn	o_i (%)
MOV	25.59	ADDIU	17.18
LDR	10.97	SW	15.51
TEQ	08.20	ADDU	14.09
BL	06.77	LW	13.45
LSL	06.33	SLL	11.13
AND	05.80	JAL	06.82
STMDB	04.17	BNE	04.32
MOV	04.00	JR	02.49
ADD	03.92	SRL	02.41
B	03.44	SLTIU	01.65
others	20.75	others	10.89

The statistics of decoders generated using the first algorithm (PART) are presented on the first two rows of Table IV. For this generator, the average decoder tree depth for the ARM is around 7, which seems reasonable due to the instruction set size. The average tree depth of the MIPS is comparable which is a little more surprising. The median value and the standard deviation show that the MIPS tree is highly unbalanced. The maximum depth, 26, is due to a degenerated subtree for the MIPS `COPz` instruction. Indeed, this instruction is part of a subsumed set of instructions distinguished by special values of a 25-bit-field. This situation is also present in the ARM decoder, but as the fields are smaller, the impact on the tree is more limited. We learn from this analysis that a future improvement of the algorithm consists of a post processing step which refolds the degenerated subtrees generated in this kind of situations.

The EFF results can be compared to the one obtained with the first algorithm since all instructions encodings have the same probability. The average depth for the MIPS architecture is clearly better (3.66 against 7.84). This result is mainly due to a better handling of the `COPz` instructions, since the maximum depth is 8 instead of 26. This better handling of the complex cases can be explained by the larger search space exploration of the second algorithm, since it tries all possible bits and can even split the instruction encodings in multiple sets. However the average depth for the ARM architecture is slightly worse (7.89 against 7.07) mainly due to the fact that maximum depth is worst (18 against 14). However, PART has a larger number of children per nodes than EFF (around 11 against 4), which means that memory footprint of this decoder is bigger. The generation time is largely greater for EFF than for PART algorithm. This difference

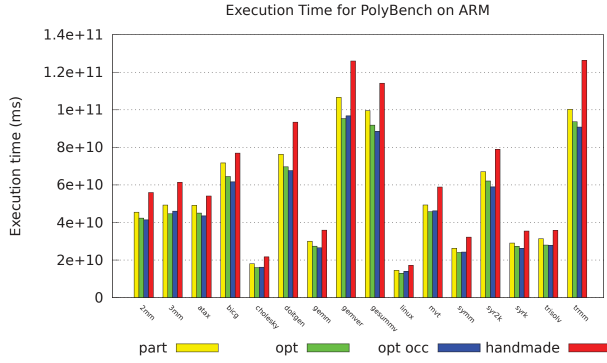


Fig. 4. Polybench ARM

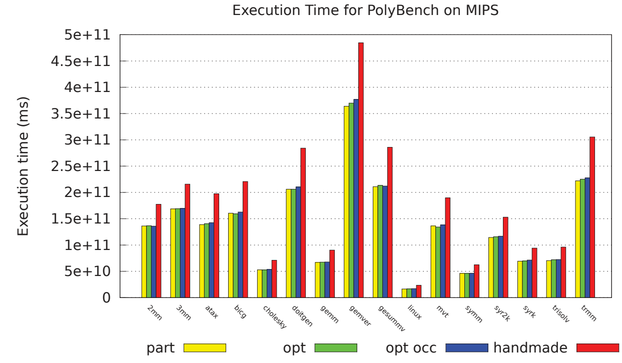


Fig. 5. Polybench MIPS

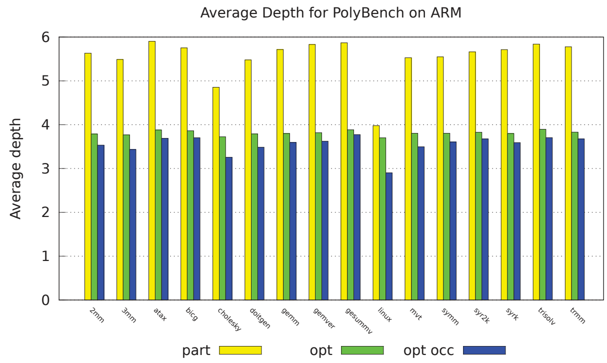


Fig. 6. Polybench ARM(depth)

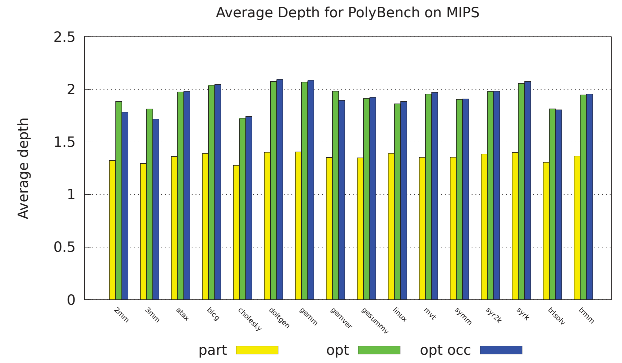


Fig. 7. Polybench MIPS(depth)

is mainly due to the iterative search for a better decision function, which makes space exploration larger for EFF.

The remaining results for this second algorithm (EFF_OCC) show that the decoder generated with realistic probabilities reaches better statistics. In fact, since the most often used instructions are matched earlier (with a shorter path), these instructions are heavier in the tree and the algorithm isolates them faster. The MIPS decoder is better, 2.8 average height against 3.6, but the ARM version is far better with a 3.7 average height against the 7.9 from the uniform version.

B. Generated decoder runtime

To test the generated instructions decoder, we have collected execution traces of the PolyBench benchmarks. These traces are dumps of the execution of the benchmarks on the QEMU [20] system simulator for MIPS and ARM architectures. These traces are then read by the instruction decoders generated previously. These instruction decoders do not emulate any instruction behavior, but decode all operands of the instructions and execute a simple dummy payload (negligible in the results).

In order to have a point of comparison for the performance of our decoders, we extracted handmade well-written decoders for the MIPS (in-house decoder) and the ARM (QEMU ARM system simulator instruction decoder). As we cannot directly compare the performance of our solution to the one of previously published algorithms ([7], [10], [11]), because they are not able to handle most subsumed cases, we made a first experiment comparing the runtimes of our algorithms (PART, EFF and EFF_OCC) with the handmade decoder and Qin's own implementation of his algorithm on a subset of the PolyBench on which we filtered out all subsumed instructions. For both MIPS and ARM, our algorithms generate decoders that are always slightly faster (between 3% to 12%) than the ones generated by Qin (using Qin's own implementation), which itself is a bit faster (between 0% to 7%) than the handmade ones. The second experiment compares our algorithms with their handmade counterparts for all traces of the PolyBench. All runs are performed 120 times and the average execution times are presented in Figures 4 and 5. In all cases, these execution times represent only the instruction decoding task, no initialization time is taken into account in these figures. The effective instruction decoding height for each specific trace is given in Figures 6 and 7. This effective instruction decoding height is obtained by averaging path lengths of all instructions of the corresponding trace, and is not available for the handmade decoders.

For full instruction sets, the execution times between ARM and MIPS decoders are quite similar. It can look surprising since MIPS decoder is simpler than ARM, however MIPS traces are about 3 times longer than ARM ones for the same test (*e.g.* the `2mm` program requires 24M instructions on ARM and 74M instructions on MIPS).

First of all, all generated decoders perform better than handmade ones for the exact same task. The comparison between the different generated decoder is slightly different for ARM and MIPS.

In the ARM case, the EFF is on the paper a little worse than PART. However, in practice, EFF is performing a little faster. On top of that, EFF_OCC is still performing better. They are however performing similarly in some case like `3mm` or `gemver`. The average

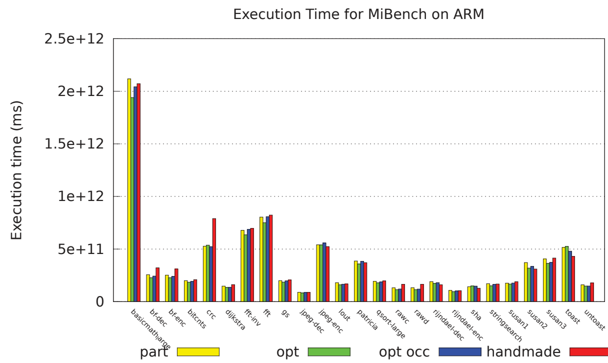


Fig. 8. Mibench ARM

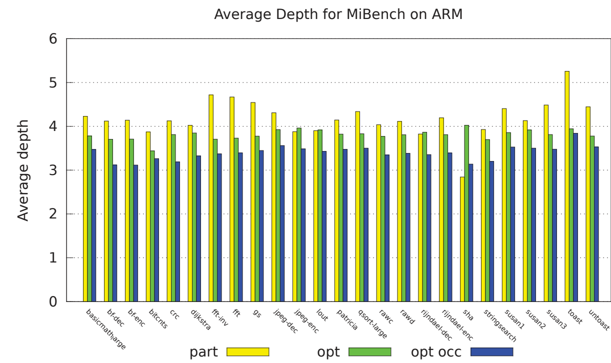


Fig. 9. Mibench ARM(depth)

depth for these traces (Fig 6) is confirming these results since EFF and EFF_OCC have a smaller average path length.

In the MIPS case, the situation is different. The statistics of the decoding trees made the EFF and EFF_OCC decoders better than the PART one. The execution time are however quite similar with a slight advantage to PART in `gemver` and `trmm` benches. These surprising results can be explained thanks to the smaller average path length in the PART decoder. These traces are exploiting only a subset of the instruction set, which is easily decodable at first steps in the decoder. The PART places them directly whereas EFF and EFF_OCC try to balance the cost of decoding across the complete ISA.

To evaluate the generated decoders on another set of benches, we also run them on traces of the MiBench suite for the ARM architecture. The results in term of execution time and average path length are presented in Fig. 8 and 9. As far as average depth is concerned, the occurrence aware decoder EFF_OCC is still better in almost all the cases. In one specific case, `sha`, PART is performing better. However, even if the average path length is better, the performance differences is not important. Compared to handmade decoder in term of execution time, the generated decoder have very similar performances, and have worse execution time only in few cases (`toast` and `jpeg-enc`). The EFF decoder, even though it has been trained on a different trace set, still reaches a good performance level.

VI. SUMMARY AND PERSPECTIVES

In this paper, we have proposed the usage of BDD to represent the encodings of instruction in order to generate instruction decoders. Our approach allows to handle the complex conditions that are due to the incremental addition of instructions in legacy processors. We have introduced two algorithms that take benefit of this representation, and, as opposed to the ones found in the litterature, are able to cover all subsumed cases. Experiments, done on the full MIPS 4k ISA and the ARMv7 ISA including the VFP and NEON extensions, have shown that both algorithms produce a decoding tree whose depth is appropriate for fast decoding as required for ISS. Slighlty less deeper trees are obtained when considering instruction frequencies, which lead to better performance if the code being executed is well characterized.

Further investigations can be made. The first one consists of adding a post-processing phase to fold the degenerated cases that occur in the current algorithm. This is not likely to improve the performances drastically, but would generate more compact decoders. The second one is more forward looking, and is to try to generate the decoding functions directly from the instruction encodings BDD instead of trying to recursively split the search space.

REFERENCES

[1] A. Hoffmann, H. Meyr, and R. Leupers, *Architecture Exploration for Embedded Processors with LISA*. Dordrecht, NL: Kluwer Academic Press, dec 2002.

[2] P. Mishra and N. Dutt, Eds., *Processor description languages: applications and methodologies*. Morgan Kaufmann, 2008.

[3] A. Peleg and U. Weiser, "MMX technology extension to the intel architecture," *IEEE Micro*, vol. 16, no. 4, pp. 42–50, 1996.

[4] J. Goodacre and A. Sloss, "Parallelism and the arm instruction set architecture," *Computer*, vol. 38, no. 7, pp. 42–50, 2005.

[5] A. Wang, E. Killian, D. Maydan, and C. Rowen, "Hardware/software instruction set configurability for system-on-chip processors," in *Proceedings of the 38th annual Design Automation Conference*. IEEE/ACM, 2001, pp. 184–188.

[6] R. Takken, "Sequential instruction set simulator generator," Master's thesis, Eindhoven University of Technology, 1992.

[7] T. E. Jeremiassen, "Sleipnir. an instruction-level simulator generator," in *Proceedings of the International Conference on Computer Design*. IEEE, 2000, pp. 23–31.

[8] M. Abbaspour and J. Zhu, "Retargetable binary utilities," in *Proceedings of the 39th annual Design Automation Conference*, 2002, pp. 331–336.

[9] "The GNU binutils," http://www.gnu.org/software/binutils/.

[10] H. Theiling, "Generating decision trees for decoding binaries," in *Proceeding of LCTES '01*, 2001, pp. 112 – 120.

[11] W. Qin and S. Malik, "Automated synthesis of efficient binary decoders for retargetable software toolkits," in *Proceedings of Design Automation Conference, 2003*, 2003, pp. 764 – 769.

[12] M. Reshadi, P. Mishra, and N. Dutt, "Hybrid-compiled simulation: An efficient technique for instruction-set architecture simulation," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 8, no. 3, 2009.

[13] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 100, no. 8, pp. 677–691, 1986.

[14] D. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[15] O. Coudert and J. Madre, "A unified framework for the formal verification of sequential circuits," in *Proceeding of the International Conference on Computer-Aided Design*. IEEE, 1990, pp. 126–129.

[16] L. Jacomme, "The ro-bdd package of the alliance cad system," 2007, http://www-soc.lip6.fr/recherche/cian/alliance.

[17] J. R. Bell, "Threaded code," *Communication of the ACM*, vol. 16, no. 6, pp. 370–372, 1973.

[18] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite (2011)," http://www-roc.inria.fr/~pouchet/software/polybench.

[19] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization*, 2001, pp. 3–14.

[20] F. Bellard, "QEMU, a fast and portable dynamic translator," in *the USENIX Annual Technical Conference*, 2005, pp. 41–46.

Publications

Les travaux réalisés au cours de cette thèse ont donné lieu à plusieurs publications répertoriées ici.

Conférences internationales

1. L. Michel, N. Fournel, and F. Pétrot. Speeding-up simd instructions dynamic binary translation in embedded processor simulation. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–4, March 2011
2. Luc Michel, Nicolas Fournel, and Frédéric Pétrot. Fast simulation of systems embedding vliw processors. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, pages 143–150, New York, NY, USA, 2012. ACM
3. Nicolas Fournel, Luc Michel, and Frédéric Pétrot. Automated generation of efficient instruction decoders for instruction set simulators. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '13, pages 739–746, Piscataway, NJ, USA, 2013. IEEE Press

Workshops

1. Luc Michel, Nicolas Fournel, Frédéric Pétrot. Dynamic binary translation of VLIW architectures on classical architectures. In *Workshop Dynamic Compilation Everywhere*, Hipeac 2012.
2. Luc Michel, Nicolas Fournel, Frédéric Pétrot. Automatic Generation of Efficient Dynamic Binary Translators. In *Workshop Dynamic Compilation Everywhere*, Hipeac 2013.

Glossaire

ABI Application Binary Interface.

ADL Architecture Description Language.

ASIC Application Specific Integrated Circuit.

BDD Binary Decision Diagram.

CISC Complex Instruction Set Computer.

DBT Dynamic Binary Translation.

DSP Digital Signal Processor.

GPR General Purpose Register.

GPU Graphics Processing Unit.

HAL Hardware Abstraction Layer.

ILP Instruction Level Parallelism.

IR Intermediate Representation.

ISA Instruction Set Architecture.

ISS Instruction Set Simulator.

JIT Just-In-Time Compilation.

MDS Machine Description System.

MMU Memory Management Unit.

RISC Reduced Instruction Set Computer.

SBT Static Binary Translation.

SIMD Single Instruction Multiple Data.

SoC System on Chip.

SSA Static Single Assignment.

TB Translation block.

TCG Tiny Code Generator.

TTB Translated Translation block.

VLIW Very Long Instruction Word.

WAR Write after Read.

Bibliographie

[BCF+99] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The jalapeño dynamic optimizing compiler for java. In *Proceedings of the ACM 1999 Conference on Java Grande*, JAVA '99, pages 129–141, New York, NY, USA, 1999. ACM.

[BD88] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. *SIGPLAN Not.*, 23(7) :329–338, June 1988.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo : A transparent dynamic optimization system. *SIGPLAN Not.*, 35(5) :1–12, May 2000.

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[BKEI01] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml ain't markup language (yaml™) version 1.1. *Working Draft 2008*, 5 :11, 2001.

[BNH+04] G. Braun, A Nohl, A Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr. A universal technique for fast and flexible instruction-set architecture simulation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(12) :1625–1639, Dec 2004.

[CCS] Code composer studio.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4) :451–490, October 1991.

[CHH+98] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. Fx !32 : A profile-directed binary translator. *IEEE Micro*, 18(2) :56–64, March 1998.

[CK93] Robert F. Cmelik and David Keppel. Shade : A fast instruction-set simulator for execution profiling. Technical Report SMLI 93-12, UWCSE 93-06-06, Sun Microsystems, 1993.

[CLU02] Cristina Cifuentes, Brian Lewis, and David Ung. Walkabout : A retargetable dynamic binary translation framework. Technical report, Sun Microsystems, Inc. Mountain View, CA, USA ©2002, Mountain View, CA, USA, 2002.

[CS98] C. Cifuentes and S. Sendall. Specifying the semantics of machine instructions. In *Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on*, pages 126–133, Jun 1998.

[CVER99] C. Cifuentes, M. Van Emmerik, and N. Ramsey. The design of a resourceable and retargetable binary translator. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, pages 280–291, Oct 1999.

[CVERL02] Cristina Cifuentes, Mike Van Emmerik, Norman Ramsey, and Brian Lewis. Experience in the design, implementation and use of a retargetable static binary translation framework. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 2002.

[DdD] Benoît Dupont de Dinechin. Machine description system (mds).

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM.

[EA97] Kemal Ebcioğlu and Erik R. Altman. Daisy : Dynamic compilation for 100 *SIGARCH Comput. Archit. News*, 25(2) :26–37, May 1997.

[FKH07] Stefan Farfeleder, Andreas Krall, and Nigel Horspool. Ultra fast cycle-accurate compiled emulation of inorder pipelined architectures. *Journal of Systems Architecture*, 53(8) :501 – 510, 2007. Architectures, Modeling, and Simulation for Embedded Processors.

[FMP13] Nicolas Fournel, Luc Michel, and Frédéric Pétrot. Automated generation of efficient instruction decoders for instruction set simulators. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '13, pages 739–746, Piscataway, NJ, USA, 2013. IEEE Press.

[GLPP12] Bernard Goossens, Philippe Langlois, David Parello, and Eric Petit. Perpi : A tool to measure instruction level parallelism. In Kristján Jónasson, editor, *Applied Parallel and Scientific Computing*, volume 7133 of *Lecture Notes in Computer Science*, pages 270–281. Springer Berlin Heidelberg, 2012.

[HGG+99] A Halambi, P. Grun, V. Ganesh, A Khare, N. Dutt, and A Nicolau. Expression : a language for architecture exploration through compiler/simulator retargetability. In *Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings*, pages 485–490, March 1999.

[KKL+07] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm : the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin : Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6) :190–200, June 2005.

[LYBB13] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification*. Addison-Wesley, 2013.

[May87] C. May. Mimic : A fast system/370 simulator. *SIGPLAN Not.*, 22(7) :1–13, July 1987.

[MD08] Prabhat Mishra and Nikil Dutt. *Processor Description Languages*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[MFP11] L. Michel, N. Fournel, and F. Pétrot. Speeding-up simd instructions dynamic binary translation in embedded processor simulation. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–4, March 2011.

[MFP12] Luc Michel, Nicolas Fournel, and Frédéric Pétrot. Fast simulation of systems embedding vliw processors. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS ’12, pages 143–150, New York, NY, USA, 2012. ACM.

[MME+97] J.H. Moreno, M. Moudgill, K. Ebcioglu, E. Altman, C. B. Hall, R. Miranda, S. K Chen, and A. Polyak. Simulation/evaluation environment for a vliw processor architecture. *IBM Journal of Research and Development*, 41(3) :287–302, May 1997.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind : A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6) :89–100, June 2007.

[PGH+11] F. Petrot, M. Gligor, M.-M. Hamayun, Hao Shen, N. Fournel, and P. Gerin. On mpsoc software execution at the transaction level. *Design Test of Computers, IEEE*, 28(3) :32–43, May 2011.

[PHM00] Stefan Pees, Andreas Hoffmann, and Heinrich Meyr. Retargetable compiled simulation of embedded processors using a machine description language. *ACM Trans. Des. Autom. Electron. Syst.*, 5(4) :815–834, October 2000.

[PHZM99] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. Lisa - machine description language for cycle-accurate models of programmable dsp architectures. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC ’99, pages 933–938, New York, NY, USA, 1999. ACM.

[Pou11] Louis-Noël Pouchet. Polybench : The polyhedral benchmark suite, 2011. `http://www-roc.inria.fr/~pouchet/software/polybench`.

[Pro01] Mark Probst. Fast machine-adaptable dynamic binary translation. In *Workshop on Binary Translation - PACT’01*. Citeseer, 2001.

[Pro02] Mark Probst. Dynamic binary translation. In *UKUUG Linux Developer’s Conference*, volume 2002. sn, 2002.

[Rau78] B. Ramakrishna Rau. Levels of representation of programs and the architecture of universal host machines. *SIGMICRO Newsl.*, 9(4) :67–79, November 1978.

[RBMD03] M. Reshadi, N. Bansal, P. Mishra, and N. Dutt. An efficient retargetable framework for instruction-set simulation. In *Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on*, pages 13–18, Oct 2003.

[RF97] Norman Ramsey and Mary F. Fernández. Specifying representations of machine instructions. *ACM Trans. Program. Lang. Syst.*, 19(3) :492–524, May 1997.

[RMD03] M. Reshadi, P. Mishra, and N. Dutt. Instruction set compiled simulation : a technique for fast and flexible instruction set simulation. In *Design Automation Conference, 2003. Proceedings*, pages 758–763, June 2003.

[RWY13] Erven Rohou, Kevin Williams, and David Yuste. Vectorization technology to improve interpreter performance. *ACM Trans. Archit. Code Optim.*, 9(4) :26 :1–26 :22, January 2013.

[Tex10a] Texas Instrument. *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide (SPRU732J)*, 2010.

[Tex10b] Texas Instruments. *TMS320C62x DSP CPU and Instruction Set Reference Guide (SPRU731A)*, 2010.

[UC00] David Ung and Cristina Cifuentes. Machine-adaptable dynamic binary translation. *SIGPLAN Not.*, 35(7) :41–51, January 2000.

[UC01] David Ung and Cristina Cifuentes. Optimising hot paths in a dynamic binary translator. *SIGARCH Comput. Archit. News*, 29(1) :55–65, March 2001.

[UC06] David Ung and Cristina Cifuentes. Dynamic binary translation using run-time feedbacks. *Science of Computer Programming*, 60(2) :189 – 204, 2006. Special Issue on Software Analysis, Evolution and, Re-engineering.

[WR96] Emmett Witchel and Mendel Rosenblum. Embra : Fast and flexible machine simulation. *SIGMETRICS Perform. Eval. Rev.*, 24(1) :68–79, May 1996.

[YMP+99] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioglu, and Erik Altman. Latte : A java vm just-in-time compiler with fast and efficient register allocation. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, PACT ’99, pages 128–, Washington, DC, USA, 1999. IEEE Computer Society.

[ZPM96] V. Zivojnovic, S. Pees, and H. Meyr. Lisa-machine description language and generic machine model for hw/sw co-design. In *VLSI Signal Processing, IX, 1996., [Workshop on]*, pages 127–136, Oct 1996.

[ZT00] Cindy Zheng and Carol Thompson. Pa-risc to ia-64 : Transparent execution, no recompilation. *Computer*, 33(3) :47–52, March 2000.

Résumé Les unités de calculs qui composent les systèmes intégrés numériques d'aujourd'hui sont complexes, hétérogènes, et en nombre toujours croissant. La simulation, largement utilisée tant dans les phases de conception logicielle que matérielle de ces systèmes devient donc un vrai défi. Lors de la simulation du système, la performance est en grande partie édictée par la stratégie de simulation des jeux d'instructions des processeurs. La traduction binaire dynamique (DBT) est une technique qui a fait ses preuves dans ce contexte. Le principe de cette solution est de traduire au fur et à mesure les instructions du programme simulé (la cible), en instructions compréhensibles par la machine exécutant la simulation (l'hôte). C'est une technique rapide, mais la réalisation de simulateurs fondée sur cette technologie reste complexe. Elle est d'une part limitée en terme d'architectures cibles supportées, et d'autre part compliquée dans sa mise en œuvre effective qui requiert de longs et délicats développements.

Les travaux menés dans cette thèse s'articulent autour de deux contributions majeures. La première s'attaque au support des architectures cibles de type Very Long Instruction Word (VLIW), en étudiant leurs particularités vis-à-vis de la DBT. Certaines de ces spécificités, tel le parallélisme explicite entre instructions, rendent la traduction vers un processeur hôte scalaire non triviale. La solution que nous proposons apporte des gains en vitesse de simulation d'environ deux ordres de grandeur par rapport à des simulateurs basés sur des techniques d'interprétation. La seconde contribution s'intéresse à la génération automatique de simulateurs basés sur la DBT. À partir d'une description architecturale de la cible et de l'hôte, nous cherchons à produire un simulateur qui soit optimisé pour ce couple. L'optimisation est faite grâce au processus de mise en correspondance des instructions du couple afin de sélectionner la ou les meilleures instructions hôtes pour simuler une instruction cible. Bien qu'expérimental, le générateur réalisé donne des résultats très prometteurs puisqu'il est à même de produire un simulateur pour l'architecture MIPS aux performances comparables à celles d'une implémentation manuelle.

Abstract Computing units embedded into modern integrated systems are complex, heterogeneous and numerous. Simulation widely used during both software and hardware design of these systems is becoming a real challenge. The simulator performance is mainly driven by the processors instruction set simulation approach, among which Dynamic Binary Translation (DBT) is one of the most promising technique. DBT aims at translating on the fly instructions of the simulated processor (the target) into instructions that can be understood by the computer running the simulation (the host). This technique is fast, but designing a simulator based on it is complex. Indeed, the number of target architectures is limited, and furthermore, implementing a simulator is a complicated process because of long and error prone development.

This PhD contributes to solve two major issues. The first contribution tackles the problem of supporting Very Long Instruction Word (VLIW) architectures as simulation targets, by studying their architecture peculiarities with regards to DBT. Some of these specificities, like explicit instruction parallelism make the translation to scalar hosts nontrivial. The solutions we propose bring simulation speed gains of two orders of magnitude compared to interpreter based simulators. The second contribution addresses the problem of automatic generation of DBT based simulators. With both target and host architectural descriptions, we produce a simulator optimised for this pair. This optimisation is done with an instructions matching process that finds host instruction candidates to simulate a target instruction. Although being experimental, our generator gives very promising results. It is able to produce a simulator for the MIPS architecture whose performances are close to a hand written implementation.