Simulation fonctionnelle native pour des systèmes many-cœurs

Guillaume Sarrazin

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Guillaume Sarrazin

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

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

Simulation fonctionnelle native pour des systèmes many-cœurs

Thèse soutenue publiquement le **23 mai 2016**,
devant le jury composé de :

M. David Defour
Maître de conférences HDR, Université de Perpignan, Rapporteur
M. François Pêcheux
Professeur des Universités, UPMC, Rapporteur
M. Matthieu Moy
Maître de conférences HDR, Grenoble INP, Examinateur
M. Benoît Dupont de Dinechin
Docteur, Kalray, Examinateur
M. Robert de Simone
Directeur de Recherche, Inria Sophia-Antipolis, Président
M. Frédéric Pétrot
Professeur des Universités, Grenoble INP, Directeur de thèse

Remerciements

Je tiens tout d'abord à remercier l'ensemble des membres de mon jury de thèse. Merci à David Defour et François Pêcheux d'avoir accepter d'être rapporteur pour ma thèse et pour leur retour constructif sur mon travail.Merci à Matthieu Moy pour le grand intérêt qu'il a porté à ma thèse. Je remercie Benoît Dupont de Dinechin, qui a été également mon responsable au sein de la société Kalray, pour ses retours intéressants et positifs sur mon manuscrit de thèse. Merci à Robert de Simone d'avoir présidé mon jury de thèse.

Je tiens à remercier tout particulièrement mon directeur de thèse, Frédéric Pétrot, pour avoir toujours été là quand j'avais besoin d'expliquer les difficultés techniques que j'ai rencontrées, pour ses conseils et le temps qu'il a consacré à m'aider dans la rédaction des articles malgré le peu de réussite finale concernant les publications. Je remercie également Nicolas Fournel grâce à qui j'ai pu faire ma thèse au sein de la société Kalray et qui m'a ensuite encadré et aidé durant le temps où il était au laboratoire Tima.

Je tiens à remercier l'ensemble des personnes de Kalray pour le grand degré de liberté qu'ils m'ont laissé et l'excellente ambiance qu'il y avait au travail. Cette bonne ambiance m'a fortement aidé à aller au bout de ma thèse. Je tiens à remercier entre autres Patrice Gerin à la fois pour ses blagues et ses explications sur le fonctionnement du Kalray processeur et du simulateur de Kalray. Je remercie Nicolas Brunie pour le temps qu'il a passé à m'expliquer les calculs à virgule flottante et notre collaboration sur ce sujet.

Merci à l'ensemble des gens de Tima avec lesquels j'ai pu discuter et échanger. En particulier Marcos Cunha qui a terminé sa thèse juste avant moi et avec lequel nous avons partagé nos expériences respectives de fin de thèse. Et je souhaite toute la réussite possible à Omayma Matoussi qui a pris la suite de mes travaux.

Pour finir, je remercie l'ensemble de ma famille qui m'a encouragé, en particulier mes parents qui ont eu la gentillesse de relire attentivement mon manuscrit à la recherche des fautes d'orthographe, mes grand-parents et ma tante Nicole qui auraient vraiment aimé assister à ma soutenance mais qui n'ont pas pu, et mon oncle Pierre et son épouse Michelle. Je remercie vivement aussi mes cousins Amélie et Charles, et mon colocataire François, pour la bonne ambiance qu'il y avait la veille et le matin de ma soutenance m'aidant à gérer le stress.

Table des matières

Chapitre 1

Introduction

Le transistor bipolaire a été inventé en décembre 1947 par les américains John Bardeen, William Shockley et Walter Brattain dans les laboratoires Bell suivi en 1948 par l'invention du transistor à jonction. L'utilisation du silicium et la mise en place des technologies MESA et PLANAR à la fin des années 1950, début des années 1960, marquent le début des circuits intégrés et le déclin des tubes électroniques. La longueur de grille était de l'ordre de 55 μm dans ses premières versions, elle est maintenant de 14 nm pour les technologies de pointe. La conjecture de Moore, prédisant le doublement du nombre de transistors sur une même surface tous les 2 ans, a permis de passer de quelques transistors par circuit imprimé à plus d'1 milliard par circuit intégré.

La mise en place d'un flot de conception se basant notamment sur un ensemble de briques de base, sur une description du circuit au niveau dit *register transfert level* (RTL), sur une phase de synthèse, de placement/routage et de validation, a permis de gérer l'augmentation du nombre de transistors et d'implémenter en matériel des composants d'une complexité toujours croissante.

Depuis 2004, la course à l'augmentation de la fréquence de l'horloge des processeurs a pourtant atteint sa limite. La cause vient de la hausse significative de la consommation et des problèmes de dissipation thermique qu'elle engendre. Cependant l'industrie du semiconducteur fait de son mieux pour que la conjecture de Moore continue d'être vraie. Ce nombre croissant de transistors peut être utilisé notamment pour augmenter le nombre de cœurs par puce et intégrer des réseaux sur la puce pour interconnecter ces cœurs, les bus partagés ne passant pas à l'échelle. La mise à jour de 2011 des prévisions de l'ITRS (*International Technology Roadmap for Semiconductors*) annonce 447 unités de calcul (Processing Engine (PE)) par puce en 2015 et environ 6 000 PE en 2026. Pour intégrer de manière fonctionnelle l'ensemble de ces unités sur une même puce, les architectures many-cœurs ont fait leur apparition. Elles utilisent les réseaux sur puce pour relier entre eux la grande quantité de cœurs qu'elles possèdent. Les sociétés à la pointe dans ce domaine sont la société Kalray avec ses processeurs Andey et Bostan (figure 1.1), Tilera avec le Tile-Gx (figure 1.2) et le Tile-Mx ou encore Intel avec ses Xeon Phi : Knights Corner, Knights Landing et Knights Hill.

La conception de circuits de cette taille est un véritable défi architectural et technologique. Avoir du code faisant fonctionner l'intégralité de ces circuits au moment de leur commercialisation est encore plus difficile. Ainsi, le développement du logiciel doit commencer le plus tôt possible, bien avant de disposer physiquement du circuit, et avant même d'en figer la totalité de la spécification. Ce développement en avance de phase peut alors per-

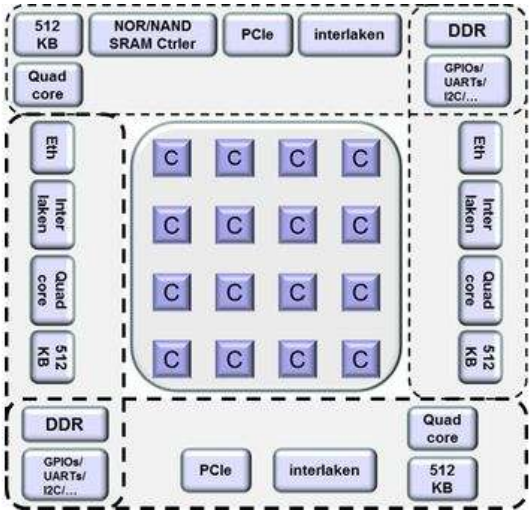


FIGURE 1.1 – Architecture du processeur Bostan de Kalray

mettre de guider la prise de décisions à l'aide d'une exploration architecturale. Pour cela, il faut disposer d'un modèle abstrait du circuit dont la vitesse d'exécution est raisonnable et qui puisse aisément être modifié. Un tel modèle est appelé « plateforme virtuelle ».

Différents niveaux de simulation existent. Une simulation du circuit au niveau RTL peut être effectuée. Cette simulation est utile pour tester des fonctions élémentaires du circuit, elle est par contre beaucoup trop lente pour simuler une application finale complexe. Pour permettre une simulation de l'application et du circuit en un temps raisonnable pour le développeur de l'application, le niveau d'abstraction de la description du circuit est adapté au besoin. Pour une précision au cycle près de l'ensemble du circuit, le niveau de description dit *cycle accurate* (CA) existe. La précision peut être diminuée en simplifiant les transferts entre les composants, les différents signaux étant regroupés sous la forme d'une transaction, c'est le modèle de simulation dit *Transaction Level Model* (TLM). L'ensemble des techniques de simulation permettant un haut niveau d'abstraction est regroupé sous l'appellation *System Level*. Le modèle de simulation TLM est ainsi une technologie possible pour faire du *System Level*. Ces niveaux fournissent un compromis entre rapidité de simulation et précision, à la fois temporelle et fonctionnelle.

L'utilisation d'une plateforme virtuelle permet de faire une exploration architecturale rapide, de définir un modèle de référence pour la conception matérielle de plus bas niveau, mais également de co-développer le logiciel et le matériel. La référence industrielle actuelle en termes de modélisation des systèmes électroniques numériques en vue de leur simulation est la bibliothèque C++ SystemC.

Indépendamment du niveau d'abstraction de la plateforme virtuelle, la technologie utilisée pour simuler le(s) cœur(s) (appelé(s) architecture(s) *cible(s)*) la plus couramment utilisée est l'interprétation ISS (*Instruction Set Simulation*). Le principe consiste à lire dans le modèle de la mémoire (ou des caches) les instructions du programme à simuler une par une, à les décoder puis à reproduire leur comportement sur les différents éléments constitutifs du circuit. Ce type de simulateur est malheureusement beaucoup trop lent pour simuler l'ensemble des cœurs des processeurs many-cœurs en un temps raisonnable pour les développeurs de logiciel, l'augmentation du nombre de cœurs ne faisant qu'empirer la situation. En effet, augmenter le nombre de cœurs permet de traiter des données de taille supérieure

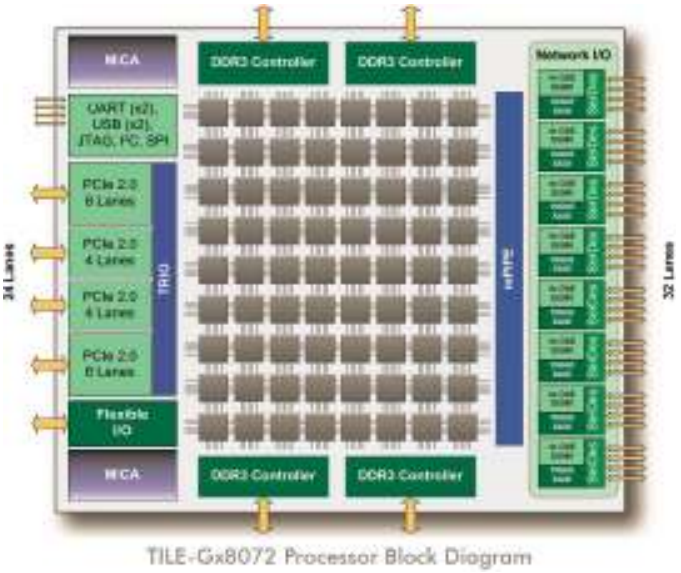


FIGURE 1.2 – Architecture du Tile-Gx8072 de Tilera

ou d'exécuter plus de tâches en parallèle. Il y a donc généralement une augmentation de la quantité de code à simuler proportionnelle au nombre de cœurs. Accélérer la vitesse de simulation nécessite de remplacer l'approche à base d'interprétation par une technique plus rapide. Les différentes candidates sont la simulation compilée, la traduction binaire statique/dynamique et la simulation native. Comme la précision temporelle de la simulation n'est pas une contrainte dans les premières phases du développement logiciel, la simulation native est le candidat naturel pour réduire au maximum le temps de simulation. L'objectif de la simulation native est de compiler puis d'exécuter le maximum de la pile logicielle directement sur la machine *hôte*, tout en continuant d'accéder aux modèles transactionnels des composants matériels.

Le manuscrit est structuré de la manière suivante. Dans le chapitre 2, les principales techniques de simulation d'un cœur et les spécificités architecturales des systèmes many-cœurs sont présentées, suivies par certains problèmes actuels de la simulation native dans le cadre de la simulation de systèmes many-cœurs.

Nous présentons ensuite dans le chapitre 3 l'état de l'art dans le domaine de la simulation native y compris pour les unités de calcul à virgule flottante et les travaux existants relatif au découplage temporel.

Dans le chapitre 4, nous proposons une stratégie de simulation native pour répondre au problème de simulation fonctionnelle correcte des caratéristiques propres au cœur cible. De plus le problème consistant à simuler correctement une unité de calcul à virgule flottante sera traité dans le cas particulier de la simulation compilée. La simulation compilée permet d'avoir un accès assez direct au code binaire cible ce qui est nécessaire ici.

Le chapitre 5 mettra en avant l'implémentation de notre solution sur laquelle nous effectuons diverses expérimentations dont nous analysons les résultats.

Le chapitre 6 abordera les problèmes de performance inhérents à la simulation native lors du passage à l'échelle du nombre de cœurs. Les expérimentations sont directement inclues avec nos propositions pour illustrer leurs intérêts et leurs faiblesses.

En conclusion, un bilan de cette thèse et des perspectives de recherche seront proposés.

Chapitre 2

Problématique

La technologie de la micro-électronique permet d'intégrer toujours plus d'éléments sur une même puce rendant de plus en plus complexe son architecture. La simulation au niveau transactionnel est devenue un élément essentiel pour la conception de telles puces, complémentaire des technologies de simulation. Différentes techniques de simulation des cœurs existent avec leurs avantages et leurs inconvénients respectifs : Instruction Set Simulator (ISS), *simulation compilée*, *simulation binaire statique*, *simulation binaire dynamique*, *simulation native*.

Nous allons présenter le contexte général de la simulation, suivi d'une présentation rapide des techniques citées précédemment. Les architectures many-cœurs seront ensuite détaillées, y compris les spécificités architecturales variées, comme les événements entre cœurs ou les unités flottantes. Nous montrerons finalement les problématiques qui se dégageront de ces présentations.

2.1 Présentation générale de la simulation de puces

Simuler une puce consiste à construire un modèle abstrait représentant au mieux le comportement interne de la puce, à exécuter ce modèle sous la forme d'une application et à voir comment il se comporte. Un tel modèle, transformé en application logicielle, est appelé *plateforme de simulation* ou *plateforme virtuelle*. La puce que l'on souhaite simuler est appelée la *machine cible* ou la *plateforme cible* (parfois aussi appelée *plateforme physique* ou *plateforme réelle*).

Pour construire une plateforme virtuelle, on reprend globalement la structure de la puce. Les différents composants matériels (mémoire, *Direct Memory Access (DMA)*, cœurs, accélérateurs, contrôleurs, ...) se retrouvent sous forme de composants logiciels. Ces composants peuvent communiquer entre eux pour reproduire leur liaison physique sur la puce. Une fois construite, la plateforme de simulation est exécutée telle une application sur la machine dite *hôte*. La figure 2.1 reprend l'ensemble de ces notions.

La référence industrielle actuelle en termes de modélisation des systèmes électroniques numériques en vue de leur simulation est la bibliothèque C++ *SystemC*. Différents niveaux d'abstraction de la plateforme virtuelle peuvent être utilisés en fonction du niveau de détails désirés : cycle (*Cycle Accurate (CA)*), transaction (*Transaction Level Modeling (TLM)*), système. Ces niveaux fournissent un compromis entre rapidité de simulation et précision. SystemC est un système de simulation par événements discrets. Son ordonnanceur est séquentiel. L'un des intérêts majeur de SystemC est de séparer la partie comportement interne des com-

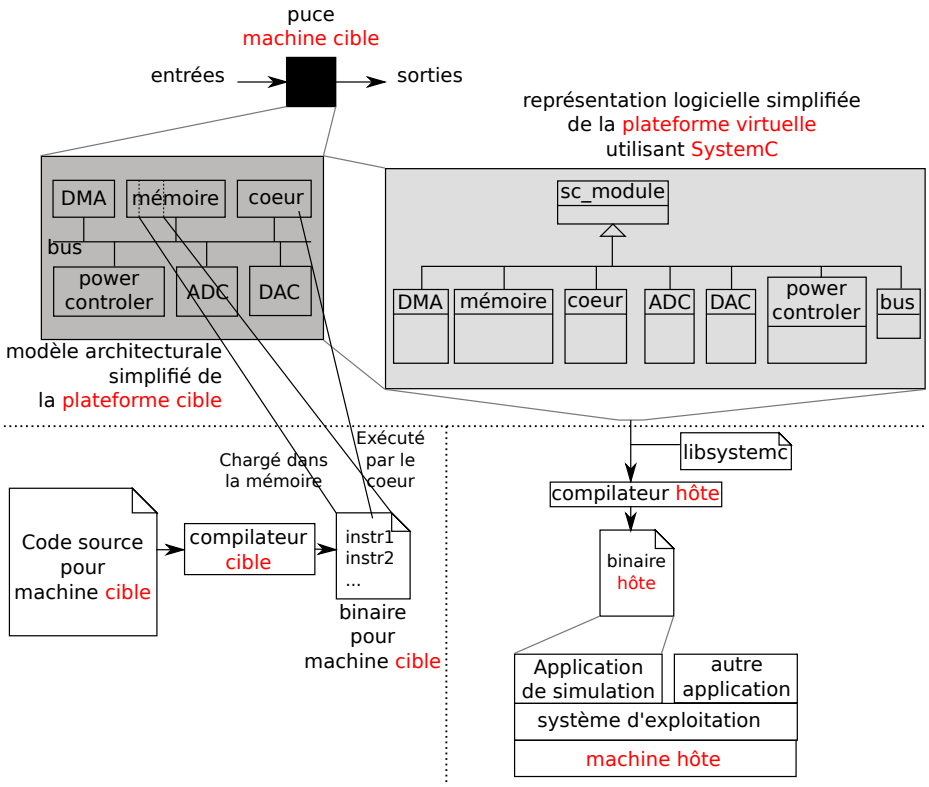


FIGURE 2.1 – Principes généraux de la simulation d'une puce

posants et la partie communication. Cela augmente la portabilité des composants et seules leurs connexions sont alors nécessaires.

Pour des raisons de compromis entre précision et vitesse de la simulation, nous nous placerons au niveau TLM. La norme OSCI TLM 2.0 introduit entre autres 2 styles de modélisation : *loosely-timed* et *approximately-timed*. Le coding style *approximately-timed* demande que la précision des informations temporelles des transactions soit au niveau des différentes phases de la transaction matérielle. Ce qui est nouveau et qui nous intéressera davantage est le coding style *loosely-timed* qui autorise les différentes tâches (*threads*) à s'exécuter en avance sur le temps global de la plateforme. On désigne par *découplage temporel* ce relâchement des contraintes temporelles. L'intérêt principal du *découplage temporel* est de réduire le nombre de changements de contexte et donc d'accélérer la simulation. Chaque thread gère son propre temps local. Il peut soit incrémenter son temps local et continuer son exécution, soit se synchroniser avec la plateforme (faire un `wait`(δ_{temps_local})). Pour garantir une certaine synchronisation entre les threads, la notion de *quantum* de temps est utilisée. Un thread ne peut pas avoir une avance temporelle sur le reste de la plateforme de plus d'un *quantum*. Le coding style conseille de forcer la synchronisation de tous les threads à des périodes régulières, le *Global Quantum* : tous les $10\mu s$ par exemple.

2.2 Techniques de simulation d'un cœur

Pour la plateforme de simulation, nous nous plaçons au niveau TLM. Différentes techniques de simulation des cœurs existent, apportant des compromis différents entre la vitesse de simulation et la précision de celle-ci.

2.2.1 Notions préliminaires

Le fonctionnement type d'un cœur est de lire des instructions dans la mémoire (en passant éventuellement par des caches) et de les exécuter. En fonction du type de cœur utilisé sur la puce, la granularité de lecture des instructions peut changer. Pour les architectures classiques (Complex Instruction Set Computer (CISC), Reduced Instruction Set Computer (RISC)), l'entrée du pipeline prend les instructions une à une. Sur les architectures de type *Very Long Instruction Word (VLIW)*, plusieurs instructions peuvent être lues par le premier étage du pipeline. On appelle un *paquet d'instructions* (*bundle*) un tel regroupement d'instructions. Les paquets d'instructions sont définis lors de la compilation du binaire à exécuter.

Le binaire peut également être découpé selon d'autres critères pour les besoins de la simulation. On appelle *bloc de base* (Basic Block (BB)) une suite continue d'instructions pour lesquelles :

— si le cœur exécute la première instruction/paquet d'instructions du BB, le cœur exécutera les autres instructions/paquets d'instructions du BB.
— si le cœur exécute une des instructions du BB, il a forcément exécuté les instructions précédentes du BB juste avant.
— cette suite d'instructions est la plus grande possible respectant les 2 conditions précédentes.

Les sauts d'instructions et les instructions cibles de ces sauts marquent les limites des BB.

Une autre définition d'un BB est utilisée dans le cas de la simulation binaire dynamique (Dynamic Binary Translation (DBT)). Seul le point 1 et la maximisation de la taille du BB sont utilisés. Avec cette définition, l'intersection de 2 BB peut être non nulle contrairement à la définition précédente.

2.2.2 Les méthodes par interprétation d'instruction

Instruction Set Simulator - ISS La plupart des techniques de simulation se basent sur l'interprétation des instructions du binaire cible. Pour interpréter une instruction, il y a 3 phases principales :

— la lecture
— le décodage
— l'exécution du comportement sur le modèle du cœur et de la plateforme. Cet exécution peut être découpée en plusieurs phases (une par étage du pipeline par exemple) selon le niveau de précision que l'on désire.

L'ISS exécute ces 3 étapes de façon dynamique. Il simule une instruction/paquet d'instructions à la fois comme montré dans la figure 2.2.

Simulation compilée L'idée de la *simulation compilée* a été introduite dans les années 1990 par Mills *et al.* [MAF91] puis repris dans les années 2000 par Reshadi [RMD03] et Zhu [ZG02]. La lecture et le décodage sont fait de façon statique. Pour chaque instruction/paquet d'instructions, du code haut-niveau (*e.g. C*) correspondant à son comportement est généré puis compilé pour la machine hôte. Pour optimiser la simulation, les limites des blocs de base (BB) du binaire cible peuvent être déterminées statiquement. Le code haut-niveau généré est alors regroupé par BB cible. Lors de la simulation, un *run-time* déterminera quel BB doit être exécuté comme expliqué sur la figure 2.3.

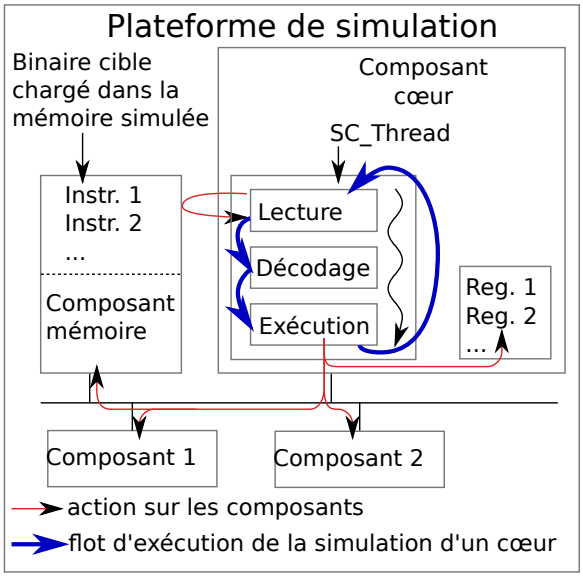


FIGURE 2.2 – Principe de la simulation ISS

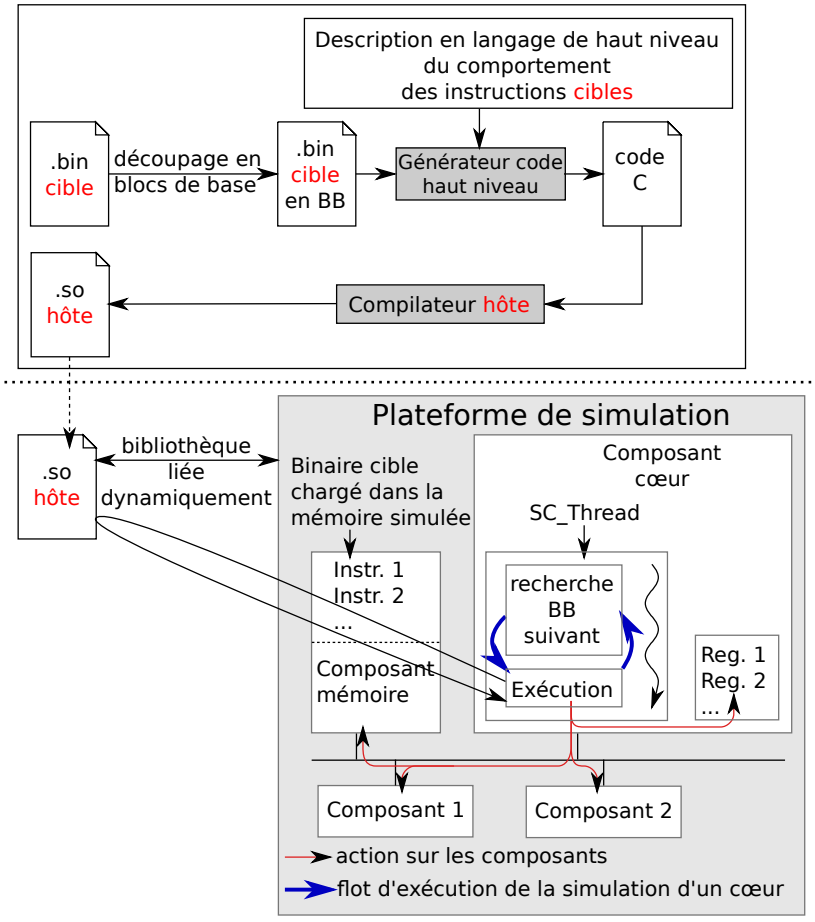


FIGURE 2.3 – Principe de la simulation compilée

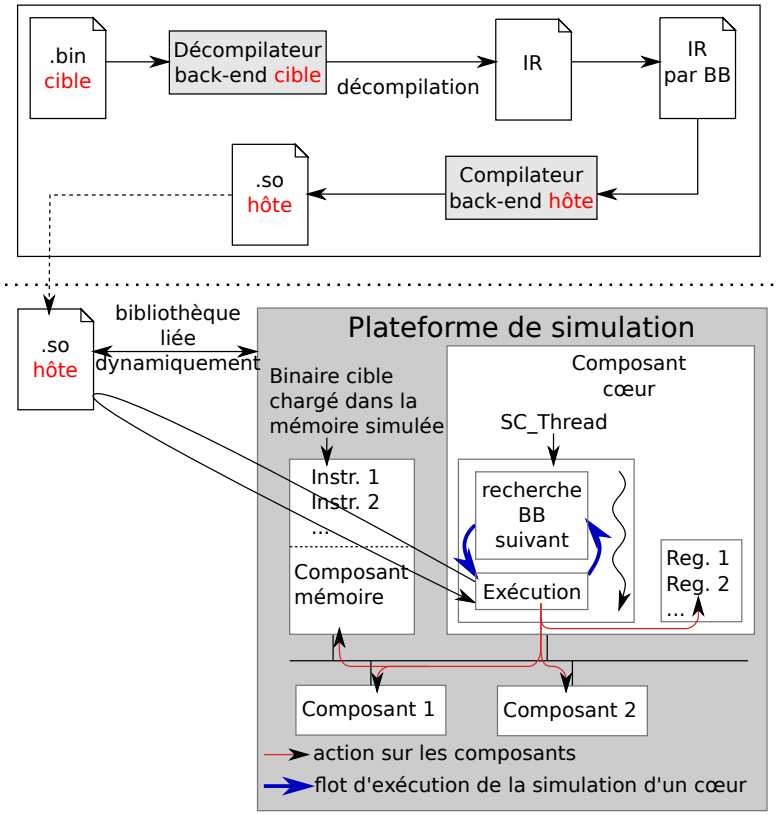


FIGURE 2.4 – Principe de la Static Binary Translation (SBT)

Simulation binaire statique - SBT La *SBT* (*Static Binary Translation*) lit le binaire et le décode aussi de façon statique. Mais la décompilation du binaire cible s'arrête au niveau du langage intermédiaire du compilateur comme visible dans la figure 2.4. Ce code en langage intermédiaire est ensuite recompilé pour l'hôte. S'arrêter au langage intermédiaire permet d'obtenir un code hôte plus optimisé qu'en simulation compilée. Par contre cette technique est dépendante du compilateur utilisé : le *back-end* du compilateur doit être disponible pour le cœur hôte et le cœur cible.

Les problèmes principaux de ces techniques dites *statiques* sont leur difficulté à gérer les sauts indirects, les bibliothèques dynamiques, les accès virtuels à la mémoire et leur incapacité à gérer du code auto-modifiant. De plus le temps de décompilation puis recompilation est assez coûteux.

Simulation binaire dynamique - DBT La *DBT* (*Dynamic Binary Translation*) réalise la décompilation/recompilation des BB au cours de la simulation comme montré dans la figure 2.5. Le code auto-modifiant et les sauts indirects peuvent ainsi être gérés. Un cache est utilisé pour stocker les traductions des BB et ainsi amortir leur coût de traduction.

L'ISS est la plus précise de ces méthodes, pouvant être précise au cycle près. Les 3 autres sont moins précises mais permettent d'obtenir une accélération de la simulation d'un à plusieurs ordres de grandeur par rapport à un ISS.

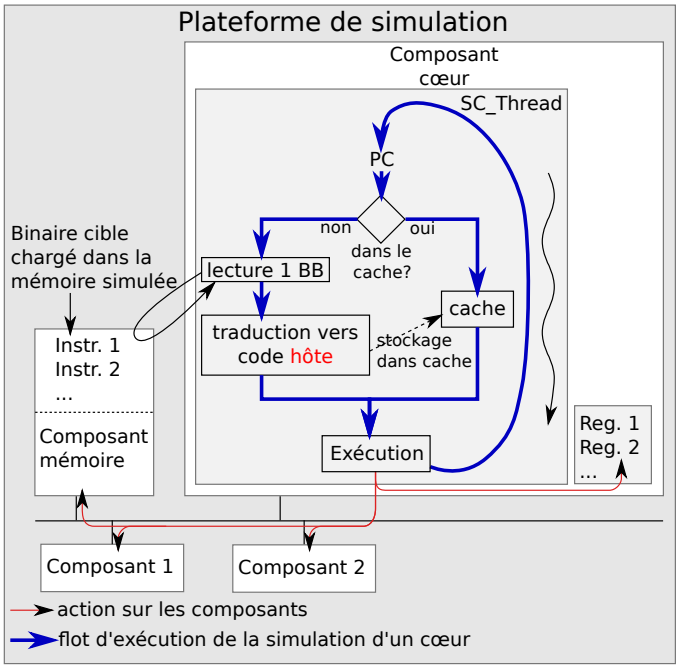


FIGURE 2.5 – Principe de la DBT

2.2.3 La simulation native

La *simulation native* (aussi appelée *source level simulation*) permet d'obtenir une accélération de la simulation jusqu'à 2, voire parfois 3 ordres de grandeur par rapport à un ISS. L'idée est de compiler directement le code source cible pour la machine hôte évitant ainsi les phases de lecture, décodage du code binaire cible.

Pour pouvoir compiler directement le code source il faut définir une interface de programmation (*API*) sur laquelle le code puisse s'appuyer. La façon de gérer le comportement de l'API peut varier. On peut utiliser un ISS comme dans [GKL+07] pour exécuter le code se trouvant sous cette API. Cette solution nécessite de mettre en place un ISS en plus de la simulation native, mais également l'interface entre les deux modes de simulation. On peut aussi écrire le comportement de cette API exprès pour la machine hôte comme dans [GSC+07]. Cette API est alors appelée *Hardware Abstraction Layer (HAL)*.

Compiler directement le code source implique que l'on perd l'ensemble des informations du binaire cible. Des instructions spéciales sont alors utilisées pour permettre de retrouver une notion temporelle lors de l'exécution du code binaire hôte obtenu à partir du code source cible. Ces instructions sont appelées des *annotations*. Elles peuvent être directement insérées dans le code source [CPVM10], ou insérées dans la représentation intermédiaire (*IR*) du compilateur [WH09, BGP09].

En simulation native, 2 espaces d'adressage coexistent : l'espace d'adressage de la plateforme cible (e.g. accès aux composants mappés en mémoire) et celle de la plateforme hôte. La virtualisation résout de façon simple et élégante ce problème comme expliqué dans [SHP12]. Les mécanismes de virtualisation assistée par le matériel (HAV) sont maintenant disponibles dans la plupart des processeurs hautes performances récents. Le HAV a été introduit par les constructeurs pour permettre le support efficace à la virtualisation, voire le support tout court pour les processeurs de type x86 qui possèdent un jeu d'instruction non virtualisable [RI00]. Un nouveau mode (au sens *user/kernel*) dit *guest* est disponible. Ce mode

possède son propre espace d'adressage, les *adresses physiques guest*, et une étape de traduction matérielle supplémentaire permet la traduction des *adresses physiques guest* en *adresses physiques hôtes*. Les mécanismes de HAV sont accessibles par l'entremise d'un hyperviseur (ou Virtual Machine Monitor (VMM)), et les auteurs de [SHP12] présentent une implantation de la simulation native basée sur l'hyperviseur du noyau Linux appelé Kernel Virtual Machine (KVM). La traduction étant gérée en amont, les composants SystemC et le code natif peuvent utiliser les adresses de la plateforme cible de manière totalement transparente. KVM permet de créer et de gérer des hyperviseurs (VMM) qui eux-mêmes permettent de créer une ou plusieurs machines virtuelles (Virtual Machine (VM)).

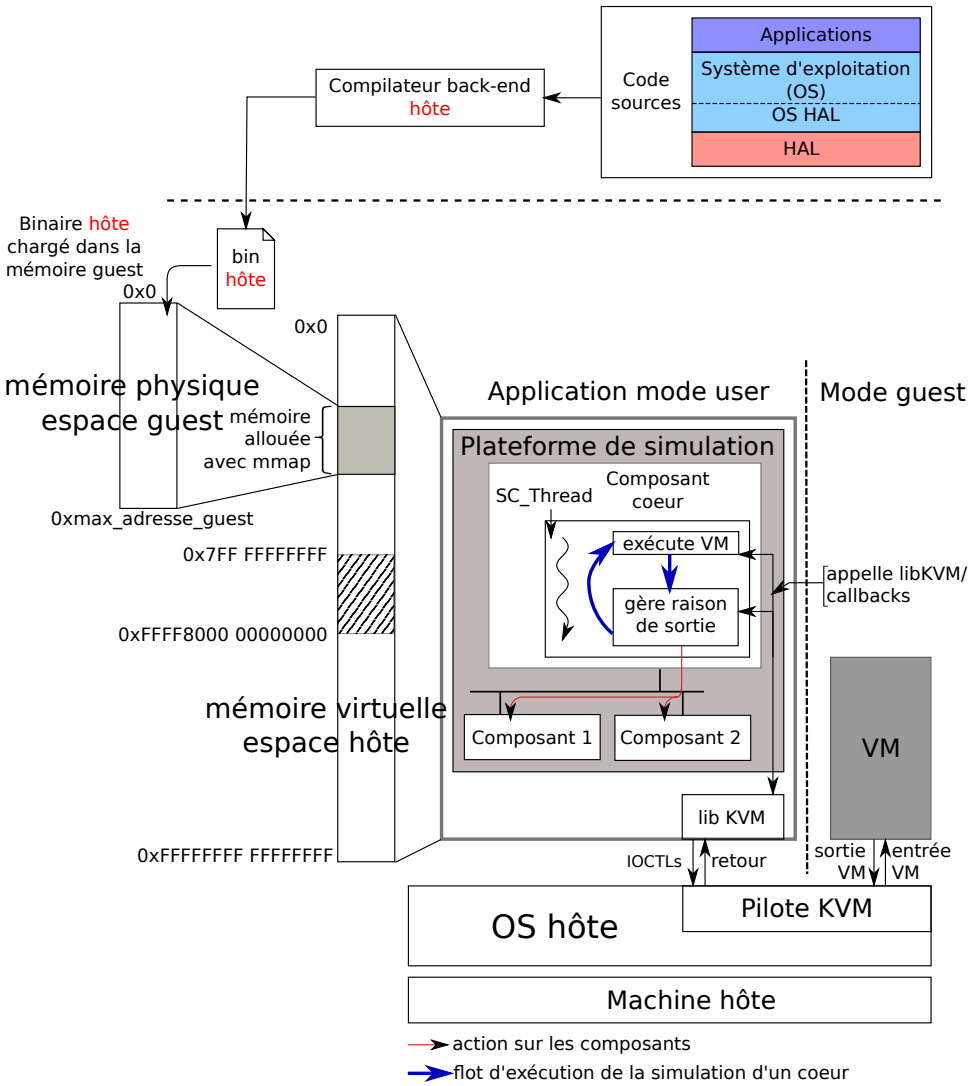


FIGURE 2.6 – Principe de la simulation native basée sur le HAV

La figure 2.6 illustre cette solution basée sur KVM. Une bibliothèque (*libKVM*) permet de faciliter la configuration de l'hyperviseur et des VM, mais aussi les interactions entre le mode user et le mode guest. Le composant servant à simuler un cœur est étendu d'une interface avec KVM lui permettant de contrôler la VM qui lui est associée. De plus une zone mémoire allouée dans la mémoire virtuelle de l'application de simulation en mode user est fournie à une VMM. Du point de vue de l'application simulée, l'ensemble des couches logi-

cielles (HAL, système d'exploitation, applications) est compilé dans un binaire hôte. Cette compilation insère des annotations temporelles pour les besoins de la simulation. D'un autre côté, la plateforme de simulation décrite en SystemC est compilée sans aucune connaissance du binaire à simuler. Le binaire hôte est chargé dans l'espace mémoire associé à la VMM. La VM exécutera le code qui lui est fourni dans l'espace mémoire du mode *guest*.

La simulation d'un cœur consiste à exécuter la VM qui lui est associée, puis à gérer la raison pour laquelle la VM a rendu la main au composant cœur. Lors de l'exécution du code cible par une VM, des annotations sont rencontrées. Ces annotations permettent de faire avancer en local le temps simulé par la VM. Quand la VM rend la main au composant cœur qui lui est associé, la somme des annotations rencontrées lors de l'exécution de la VM va former T_{sim} : le temps total passé à simuler le composant cœur. Le temps simulé du composant cœur est alors avancé du temps collecté : wait(T_{sim}) en SystemC. Nous désignons par *synchonisation* ce méchanisme. Chaque cœur est simulé par grand laps de temps pour diminuer le nombre d'aller retour entre le mode user et le mode guest. La simulation se fait donc sous le paradigme appelé *découplage temporel*.

Une fenêtre maximale d'avancement est définie pour garantir la simulation du reste des composants. Mais contrairement au quantum défini dans le *coding style loosely-timed* de SystemC TLM 2.0, il n'y a pas de *Global Quantum*. Il y a seulement une fenêtre d'avancement maximale de la simulation d'un cœur. Celui-ci peut par contre se synchroniser quand il le souhaite avec la plateforme de simulation sans devoir au préalable consommer tout son quantum.

La technique de simulation native retenue est la simulation utilisant la virtualisation. Elle possède de meilleures caractéristiques en terme de vitesse de simulation et résout simplement le problème du double espace d'adressage. Dans ce rapport de thèse, nous ne nous intéresserons pas à la précision temporelle de la simulation native mais seulement à des problèmes fonctionnels. Pour permettre une progression de la simulation, une évolution du temps est nécessaire. Nous utilisons des annotations directement ajoutées au préambule et à la terminaison des fonctions du code cible par une option du compilateur : `-finstrument-functions`. Le temps contenu dans ces annotations est arbitraire.

2.3 Architectures des systèmes many-cœurs

Les systèmes many-cœurs relativement généralistes (par opposition aux processeurs graphiques - Graphics Processing Unit (GPU)) et éventuellement personnalisables commencent à arriver sur le marché. Nous allons décrire leur caractéristiques principales. Le premier point intéressant est l'architecture des cœurs.

Deux catégories entrent en concurrence. La première utilise des cœurs spécialement conçus pour les many-cœurs. Le but est de minimiser la taille et la consommation des cœurs permettant ainsi d'en maximiser le nombre sur une puce. C'est par exemple le cas des cœurs utilisés dans le PEZY-I, le PEZY-SC de la société PEZY, le processeur Andey de Kalray, ou le premier many-cœurs de Tilera, le TilePro. La deuxième repose sur un écosystème déjà existant pour faciliter le passage aux architectures many-cœurs. Le Tile-Mx de Tilera utilise ainsi des cœurs ARMv8. Les processeurs Knights Landing d'Intel possèdent des cœurs basés sur l'architecture Silvermont.

Pour augmenter encore le parallélisme sans augmenter la taille du cœur, la plupart des solutions utilisent un jeu d'instructions avec un parallélisme explicite. Ainsi, Knight Corner

(+50 cœurs avec du multi-threading 4 voies) se base sur un cœur x86 auquel a été ajouté une large unité *Single Instruction Multiple Data (SIMD)*. D'autres exemples utilisent des processeurs VLIW spécifiques comme le GX-72 de Tilera (72 cœurs) et la puce Andey de Kalray (256 cœurs). Le fait que ces cœurs soient spécifiques implique que le jeu d'instructions comprend des instructions inhabituelles, optimisées pour les marchés ciblés.

Pour interconnecter les cœurs, des réseaux spécifiques sont intégrés dans chaque puce. Chaque cœur peut être directement relié à ce réseau. C'est le cas de Knights Corner qui connecte ses cœurs à travers un bus circulaire. Les cœurs du TilePro de Tilera sont tous connectés entre eux à l'aide d'un réseau sur puce - Network on Chip (NoC).

Les cœurs peuvent également être regroupés en sous-système (appelé grappe ou tuile) autour d'une interconnexion commune. Le sous-système est ensuite connecté à une interconnexion globale. C'est par exemple le cas du TileMx dans lequel une tuile est composée de 4 cœurs qui partagent des ressources matérielles spécifiques comme un tampon pour le passage de messages, les tuiles sont reliées entre elles par 5 NoC. Concernant les puces de Kalray, le sous-système est appelé grappe et contient 16 cœurs utilisés pour les calculs (appelés *Processing Engine (PE)*), assistés par un cœur dédié au contrôle (appelé *Ressource Manager (RM)*). Les cœurs de la grappe partagent une mémoire locale et des périphériques. L'interconnexion globale se base sur 2 NoC.

Finalement un dernier point architectural diffère entre les systèmes many-cœurs : la cohérence des caches. Le Tile-Gx de Tilera utilise une architecture avec cohérence de cache permettant d'avoir une solution moins perturbante concernant les modèles de programmation de l'ensemble du système. Le passage à l'échelle de ce type de solutions est toujours une question ouverte et peut expliquer le petit nombre de processeurs inclus dans le système - moins d'une centaine. L'autre solution est d'avoir un système sans cohérence de cache, réduisant ainsi la complexité du matériel et reportant les problématiques de passage à l'échelle à la partie logicielle. La puce Andey possède ainsi 256 cœurs sans cohérence de cache. Cette solution change davantage les modèles de programmation et les outils de développement qui doivent prendre en compte cette absence de cohérence.

Un modèle abstrait d'une architecture many-cœur peut être représenté par la figure 2.7.

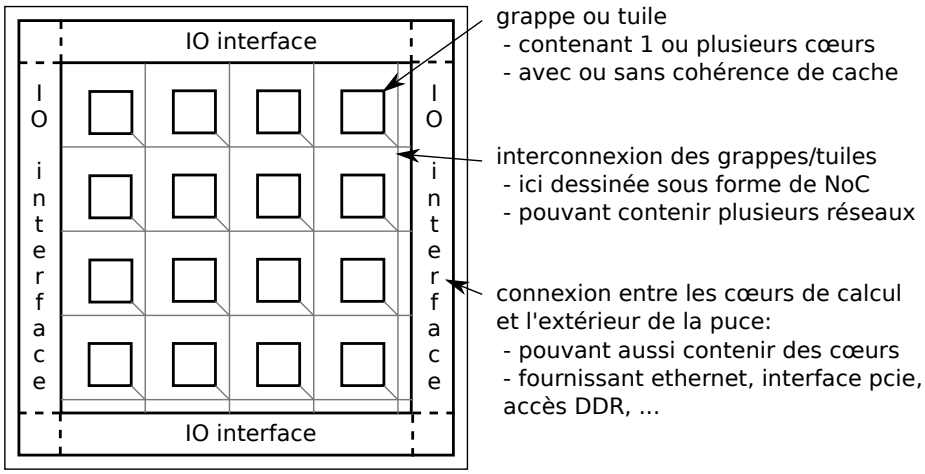


FIGURE 2.7 – Modèle abstrait d'une architecture many-cœurs

2.4 Introduction sur le calcul à virgule flottante

Un élément important pour caractériser la puissance d'un cœur est l'unité de calcul à virgule flottante (*FPU*)). Il s'agit d'un élément crucial pour beaucoup de domaines d'applications : le domaine du *High Performance Computing* (*HPC*), de la finance (*e.g.* les méthodes de Monte Carlo), l'exploration des sous-sols pour le gaz et le pétrole (*e.g.* algorithme de migration inversée dans le temps), le graphisme 3D, et toutes les simulations des phénomènes physiques. Même les systèmes embarqués, qui utilisaient jusque-là du calcul à point-fixe [BR05] pour des raisons de coûts unitaires du produit, des raisons énergétiques et/ou pour faciliter les preuves d'algorithmes, se sont mis à adopter le calcul flottant. En effet, les capacités d'intégration actuelles ont grandement réduit la différence de coût entre les unités de calcul à virgule fixe et les unités de calcul à virgule flottante. Ce facteur n'est plus un des critères essentiels pour le choix entre les 2 types de calcul [FS04]. Le calcul à virgule flottante permet d'avoir plus de précisions à travers un large intervalle dynamique que le calcul à virgule fixe. De plus les nombres à virgule flottante sont globalement plus faciles d'utilisation. Pour illustrer ce changement dans le domaine de l'embarqué, on notera que les processeurs ARM, qui représentent une grande majorité des processeurs utilisés dans ce domaine, contiennent une FPU, appelée *Vector Floating Point Architecture* (*VFP*) depuis les ARMv5TE. On retrouve ainsi le calcul flottant dans le domaine du contrôle comme l'automotive (*ABS*, suspension active), ou les systèmes de contrôles industriels (contrôle de mouvement).

Les nombres à virgule flottante (*FP*) sont un moyen d'approximer les nombres réels avec une certaine précision (fixée en fonction de la représentation du nombre) à travers un large intervalle dynamique. Un nombre FP x est défini par 2 entiers M et e, et $s \in \{-1, 1\}$ tel que : $x = (-1)^s \times M \times 2^e$. M est appelé la fraction (aussi appelé le significande ou la mantisse), e l'exposant et s est le bit de signe. La séparation de l'exposant et du significande garantit la précision uniforme quel que soit l'intervalle utilisé. Les nombres FP suivent une répartition logarithmique à travers les différents intervalles comme montré dans la figure 2.8.

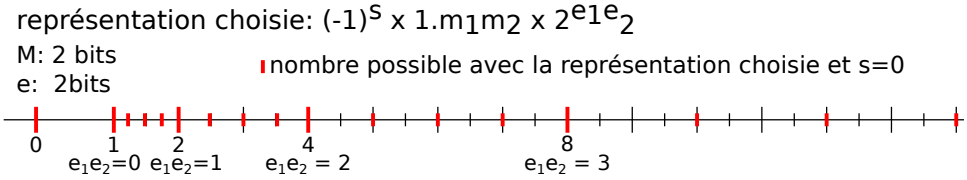


FIGURE 2.8 – Répartition logarithmique des nombres à virgule flottante

Tous les nombres réels ne peuvent pas être représentés sous forme de nombre FP. De plus le résultat d'une opération entre 2 nombres FP n'est pas forcément un nombre FP. Il a été nécessaire d'introduire l'opération d'arrondi pour transformer un nombre non représentable en virgule flottante en un nombre représentable. La valeur de l'erreur d'arrondi montre la distance entre le nombre réel d'origine et sa représentation en virgule flottante. Par exemple, $a = 1.0001 \times 2^0$ et $b = 1.0001 \times 2^{-2}$ sont 2 nombres à virgule flottante avec 5 bits de fraction. L'addition des 2 nombres donne un nombre non représentable dans le format d'entrée : $r = a + b = 1.010101 \times 2^0$. r doit alors être arrondi. Notons $\circ$ l'opération d'arrondi, r' le résultat après arrondi et ϵ l'erreur d'arrondi. Nous choisissons d'arrondir au plus près :

$$\begin{aligned} r' &= \circ(a+b) = 1.0101 \times 2^0 \\ \epsilon &= |r' - r| = 1.0 \times 2^{-6} \end{aligned}$$

2.4.1 La norme *IEEE 754*

Avant 1985, chaque constructeur avait son propre format pour représenter les FP avec différentes tailles de significandes et d'exposants, ses propres règles d'arrondi, et ses propres valeurs spéciales. Parfois le même constructeur maintenait des systèmes différents pour des produits différents. En 1985, la norme *IEEE 754* est publiée. Elle définit une spécification du format FP maintenant acceptée par tous. Elle facilite le portage de code à travers différentes implémentations : sous certaines conditions (rester dans un domaine valide, exécuter la même séquence d'opérations identiques), 2 implémentations différentes de la norme doivent retourner un résultat identique.

La norme spécifie des formats (table 2.1), des types d'arrondi (table 2.2), les exceptions qui doivent être levées et quand, le comportement de certains opérateurs. Dans la norme, l'exposant est toujours stocké sous la forme d'un nombre positif. Pour recentrer l'exposant autour de 0, un *biais* b est appliqué à l'exposant $e_{stocké}$ de tel sorte que l'exposant réel du nombre FP soit $e = e_{stocké} - b$. La valeur du biais dépend du format.

Un autre élément important pour respecter la norme concerne la façon dont les calculs doivent être effectués. Chaque opération doit être réalisée en précision infini, l'arrondi vers le format final n'étant effectué qu'à la fin.

Nom	type C	précision (bits)	significande (bits)	exponent (bits)	biais
binary32/simple précision	float	24	23	8	-127
binary64/double précision	double	53	52	11	-1023

TABLE 2.1 – Principaux formats de la norme IEEE 754

Nom	Description
au plus près	nombre FP le plus proche, avec un significande pair si à égale distance de 2 nombres FP
vers $+\infty$	nombre FP le plus petit r tel que $r \geq x$
vers $-\infty$	nombre FP le plus grand r tel que $r \leq x$
vers 0	vers $+\infty$ si $x \leq 0$, sinon vers $-\infty$

avec r le nombre FP après arrondi
et x le nombre FP après calcul en précision absolue.

TABLE 2.2 – modes d'arrondi de la norme IEEE 754

2.4.2 Nombre normal et nombre dénormalisé

La norme introduit la distinction entre 2 catégories de nombres : les nombres FP normaux et les nombres FP dénormalisés. Les FP normaux représentent la grande majorité des nombres FP. Ils sont évalués à $(-1)^s \times 1.M \times 2^{e+biais}$. Le nombre étant normalisé, son bit de poids fort vaut toujours 1 et donc il n'est pas présent dans l'encodage. La précision de l'encodage est égale à la taille du significande M plus 1 pour le bit implicite (toujours évalué à 1).

Pour garder certaines propriétés mathématiques autour de 0 et augmenter le nombre de valeurs représentables, on utilise les nombres dit dénormalisés. Un nombre FP dont l'encodage de l'exposant est à sa valeur minimale est interprété comme étant dénormalisé. Son exposant est alors évalué à $emin = 1 + biais$ et le nombre dénormalisé à $(-1)^s \times 0.M \times 2^{emin}$.

2.4.3 Emulation logicielle des opérations FP

Pour simuler les opérations à virgule flottante de manière à garantir la précision (avoir en sortie le même résultat au bit près et les mêmes exceptions levées), une émulation logicielle utilisant du calcul entier est utilisée. La figure 2.9 illustre ce mécanisme. Les nombres sont tout d'abord stockés sous une représentation avec des entiers dans laquelle le signe, la mantisse et l'exposant sont facilement accessibles et pour laquelle le nombre de bits est suffisant pour pouvoir faire les calculs en précision absolue. Les calculs sont ensuite faits sur chaque partie séparément à l'aide d'opérations entières. Le résultat obtenu est alors arrondi en fonction du type d'arrondi choisi et si besoin certaines exceptions sont levées.

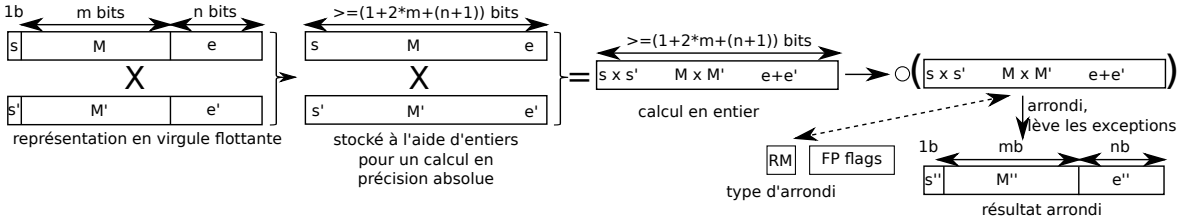


FIGURE 2.9 – Emulation logicielle d'une multiplication en virgule flottante à l'aide d'entiers

2.5 Problématiques liées aux changements architecturaux et aux nouveaux besoins de simulation

2.5.1 Simulation fonctionnelle précise de caractéristiques architecturales importantes et non triviales

Le passage aux many-cœurs d'une part, le domaine de l'embarqué d'autre part, mais plus globalement l'évolution des architectures amènent les cœurs à posséder de nouvelles caractéristiques architecturales. Ces caractéristiques peuvent être micro-architecturales (*e.g.* nombre d'étages du *pipeline*, nouveau prédicteur de branchement). Cela va alors impacter le temps d'exécution du code cible. D'autres caractéristiques sont macro-architecturales.

Nous illustrons cette évolution à l'aide des 3 exemples suivants.

— De même que les systèmes d'exploitation pour système multi-processeurs symétriques (*symmetric multiprocessor - SMP*) proposent des services de communication entre les tâches et entre les processus à l'aide d'interruptions et/ou de boîtes aux lettres, le matériel peut aussi fournir des moyens de communication inter-cœurs. Ainsi les cœurs Kalray possèdent des ressources appelées *événements*. L'envoi d'un événement d'un cœur à un autre est réalisé avec une latence très faible.

— Les *timers* peuvent être directement intégrés au sous-système cœur au lieu d'être un composant à part entière. Le composant cœur qui est simplement une VM doit alors prendre en compte ce nouvel élément et cela de façon suffisamment précise.

— Des instructions propres au cœur simulé : optimisation pour un type d'algorithme précis ou commande particulière (lancer un événement) peuvent être introduites.

Or la virtualisation nous permet d'avoir une machine virtuelle avec des caractéristiques copiant la machine hôte. Cela ne nous permet pas de simuler ces nouvelles caractéristiques.

2.5.2 Simulation rapide et précise d'une FPU

La FPU est une caractéristique importante des cœurs et son comportement a un impact fonctionnel sur les applications exécutées par la puce. Si des différences existent entre la sortie d'une opération à virgule flottante en simulation et la sortie de l'opération sur la machine cible, des différences de comportements entre une application tournant en simulation et la même application tournant sur la vraie puce peuvent être visibles. Les erreurs de calcul en virgule flottante ont tendance à se propager et à s'accentuer. Cela peut aller d'erreurs d'arrondi tolérables dans des applications non critiques à une divergence non attendue dans un flux de contrôle pour un code parallèle. Ces différences peuvent entraîner des bugs difficilement visibles et éventuellement dangereux, en particulier dans le domaine de l'embarqué avec des puces many-cœurs où le parallélisme est exploité au maximum.

Pour illustrer cette divergence possible, nous utilisons le même code C présenté dans le listing 2.1 compilé et exécuté sur deux machines différentes. La première utilise un processeur X86_64, souvent utilisé comme processeur hôte. La deuxième utilise un cœur K1, utilisé comme machine cible. Les résultats présentés à la suite du code montrent une divergence de comportement. Pour le X86_64, le compilateur a généré une multiplication et 2 soustractions pour réaliser l'accumulation (line 6). L'arrondi après la multiplication enlève de la précision et conduit à un résultat valant 0 (line 12). Pour le K1, le compilateur génère un *Fused Multiply and Sustract (FMS)* et ensuite une soustraction séparée pour l'accumulation. L'opération FMS réalise une multiplication et une soustraction avec seulement un arrondi final : $\circ(a \times b - c)$. La précision est gardée à l'intérieur de l'instruction FMS et une valeur différente de 0 est accumulée (line 16). Les expressions contractées sont autorisées par le langage C, mais désactivables par l'utilisateur via le pragma `FP_CONTRACT`. Or le compilateur gcc ignore ce pragma. Pour ne pas violer la norme, les contractions ne sont autorisées que si l'option `-funsafe-math-optimizations` de gcc est activée. Or le compilateur du K1 (`k1-gcc`) se permet de faire cette optimisation sans la présence de cette option. Il n'est pas le seul à faire cela, le compilateur d'Intel (*icc*) le fait aussi. L'option `-fp-model` mettant `fast` à 1 (équivalent de `-funsafe-math-optimizations` pour gcc) est activée par défaut pour *icc*.

```
1  void main(void) {
2    volatile float a[2] = {0x1.000002p10,0x1.000002p10};
3    volatile float acc = 0.0f;
4
5    for (int i = 0 ; i < 100000 ; ++i)
6      acc += -(0x1.000004p20f - a[0] * a[1]);
7    printf("resultats des iterations = %e\n", acc);
8  }
9
10 % gcc -O2 example.c -o test_x86
11 % ./test_x86
12 % resultats des iterations = 0.000000e+00
13
14 % k1-gcc -O2 ./example.c -o test_k1
15 % k1-cluster -- test_k1
16 % resultats des iterations = 1.490116e-03
```

Listing 2.1 – Boucle d'accumulation basique et son exécution

Une technique de simulation n'ayant pas notion du code binaire cible (comme la simulation native) pour générer le comportement de l'application n'a aucune chance de générer le bon comportement de la FPU. Mais même des techniques comme la simulation compilée doivent faire attention pour simuler le bon comportement sans pour autant passer beaucoup de temps de simulation à garantir cette équivalence.

La technique par émulation logicielle des calculs en virgule flottante à l'aide d'entiers garantit de simuler correctement la FPU. Elle requiert cependant beaucoup de temps de calcul hôte : de nombreux calculs entiers sont effectués pour une opération à virgule flottante, des types plus larges doivent être utilisés, des tests sur les entrées et sur la sortie sont réalisés, le type d'arrondi est testé puis appliqué. Pour des simulations dites "lentes" comme l'ISS, l'ensemble de ces calculs est noyé dans le volume des opérations à effectuer. Mais pour des simulations plus rapides telles que la simulation compilée, la SBT et la DBT, le temps passé à émuler le calcul à virgule flottante devient vite non négligeable pour des applications utilisant massivement le calcul à virgule flottante. Concernant la simulation native, il n'y a pas d'interprétation des instructions cibles et donc le seul moyen est d'espérer que la FPU hôte soit la même que la FPU cible.

A priori on peut penser se reposer sur l'implémentation d'une FPU conforme à la norme IEEE 754 sur la machine hôte et sur la machine cible pour garantir l'équivalence. Malheureusement ce n'est pas toujours exact : des zones de la norme sont laissées au choix de l'architecte.

Tout d'abord nous pouvons considérer le cas des *Not a Number (NaN)*. Il s'agit d'une valeur spéciale de l'encodage FP. Un NaN est retourné quand des opérandes invalides sont fournis à une opération FP (*e.g.* $\infty - \infty$ ou $0/0$). Seuls les bits de l'exposant, le bit de signe et le premier bit du significande sont spécifiés par la norme. Souvent les autres bits contiennent des informations sur la raison ayant conduit à avoir un NaN. Mais leur contenu est laissé au choix du designer.

Un autre élément de la norme pouvant varier entre les implémentations est la *tininess*. Un résultat est dit *tiny* s'il se trouve strictement entre -2^{emin} et 2^{emin}. L'exception appelée *underflow* est soulevée si un résultat est *tiny* et s'il est inexact (le résultat du calcul en précision absolu n'est pas le même que le résultat arrondi). A cause de raisons historiques, la norme autorise 2 manières pour détecter la *tininess*. La section 7.5 de la norme IEEE 754-rev08 indique : "the implementer shall chose how tininess is detected", la personne implémentant la norme doit choisir comment la tininess est détectée. Les 2 détections possibles sont : avant ou après l'opération d'arrondi.

Le calcul à virgule flottante a évolué depuis 1985 : de nouveaux formats et de nouveaux opérateurs sont apparus. Pour prendre en compte ces évolutions, une révision majeure de la norme a été publiée en 2008 : IEEE 754-rev08. Les nouveaux formats binaires sont présentés dans le tableau 2.3. Des formats de nombres à virgule flottante décimaux sont aussi introduits. Une des contributions importantes de la révision de la norme est l'opération *Fused Multiply and Add (FMA)* qui réalise une multiplication et une addition avec seulement un arrondi final : $\circ(a \times b + c)$. Grâce à son unique arrondi final, cet opérateur possède des propriétés mathématiques très utiles pour les algorithmes numériques. Par exemple l'implémentation de la racine carrée ou de la division à l'aide de la méthode de Newton nécessite un FMA. Cet opérateur est apparu dans les années 90 dans les architectures POWER de IBM et a depuis été adopté par beaucoup de plateformes (*e.g.* ARM Neon). Cependant les cœurs x86 ne peuvent réaliser cette opération que depuis Haswell (commercialisé à partir de 2013).

De plus se reposer uniquement sur la norme ne permet absolument pas de gérer de nouvelles instructions de calcul à virgule flottante (optimisées pour un domaine particulier

Nom	type C	précision (bits)	significande (bits)	exponent (bits)	biais
binary16/demi précision		11	10	5	-15
binary128		113	112	15	-1023

TABLE 2.3 – Nouveaux formats binaires introduits par la norme IEEE 754-rev08

par exemple). Néanmoins il est important de pouvoir simuler rapidement les architectures contenant des FPU tout en garantissant la précision des calculs. Les techniques de simulation présentées précédemment (autre que la simulation native) ont directement accès au binaire cible et peuvent donc essayer de garantir cette précision. La méthode pour simuler les instructions à virgule flottante doit être améliorée pour permettre à ces techniques de simulation dites "rapides" de conserver leur rapidité même en simulant une application riche en calcul à virgule flottante.

2.5.3 Augmentation du degré de parallélisme de la machine cible

Les architectures multi-cœurs introduisent du parallélisme dans la plateforme cible. Le niveau de parallélisme augmente encore quand on passe à des architectures many-cœurs. Or ce vrai parallélisme est difficilement reproduisible en simulation. En effet la simulation est faite par événements discrets. La nécessité de garantir la causalité entre les événements amène l'algorithme d'ordonnancement à être séquentiel. Des techniques de simulation parallèle par événements discrets [Fuj90] existent mais ne sont pas très répandues de par leur complexité ou leur faible degré de parallélisme.

L'ordonnanceur de SystemC est séquentiel. Les cœurs sont donc simulés les uns après les autres. En simulation native basée sur le HAV, le modèle du découplage temporel est utilisé : un cœur est exécuté pendant un grand temps simulé. Il se retrouve donc en avance sur le reste de la plateforme. Les mécanismes de synchronisation entre les threads/processus du code cible peuvent entrainer des temps de simulation plus grands que nécessaire à cause du *découplage temporel* introduit dans la simulation. Cela peut avoir des conséquences négatives sur les mécanismes de synchronisation entre les threads/processus du code cible.

La figure 2.10 (respectivement figure 2.11) montre comment on peut perdre du temps de simulation lors d'une prise de verrou (respectivement lors d'une barrière avec attente active). Le cœur0 prend le verrou en premier (1) puis il relâche la main (2) car il a consommé tout son quantum de temps. Le cœur1 est alors sélectionné (3). Il tente aussi d'obtenir le verrou (4) mais ne réussit pas à l'obtenir car le cœur0 le tient. Il va donc consommer tout son quantum de temps à essayer de prendre le verrou avant de relâcher la main (5). Le cœur0 est de nouveau sélectionné (6) et relâche le verrou (7) avant le temps de simulation atteint par le cœur1. Le cœur1 a simulé trop loin dans la simulation. En effet, il aurait dû prendre le verrou à (7). La différence de temps entre (5) et (7) correspond à une erreur temporelle de simulation mais aussi à du temps perdu lors de la simulation car on simule inutilement un cœur. La figure 2.11 montre de la même manière comment on peut perdre du temps de simulation lors d'une barrière avec attente active.

Dans les 2 cas, la VM simulant un cœur va continuer à s'exécuter jusqu'à la fin de son *quantum*. Or l'événement pouvant débloquer la situation ne peut provenir que d'un autre cœur. Cette progression trop en amont d'un cœur entraîne une perte de temps de simulation et aussi une perte en précision du temps simulé. Rappelons que cette thèse ne porte pas sur la précision temporelle de la simulation, seul l'aspect fonctionnel nous intéresse. Or ce

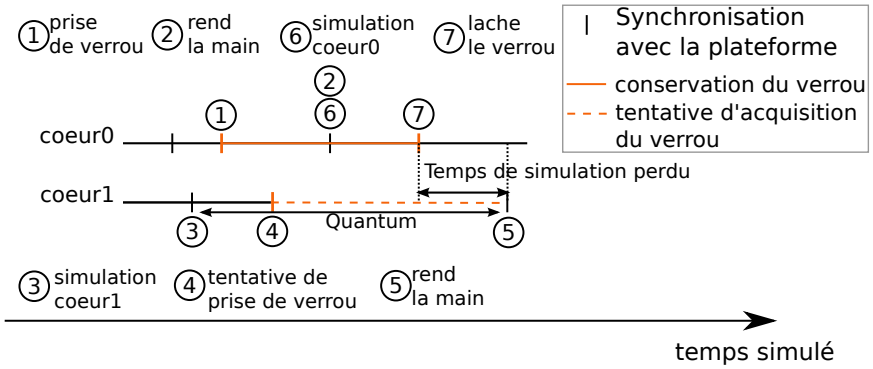


FIGURE 2.10 – Perte de temps de simulation en présence de concurrence pour l'acquisition d'un verrou

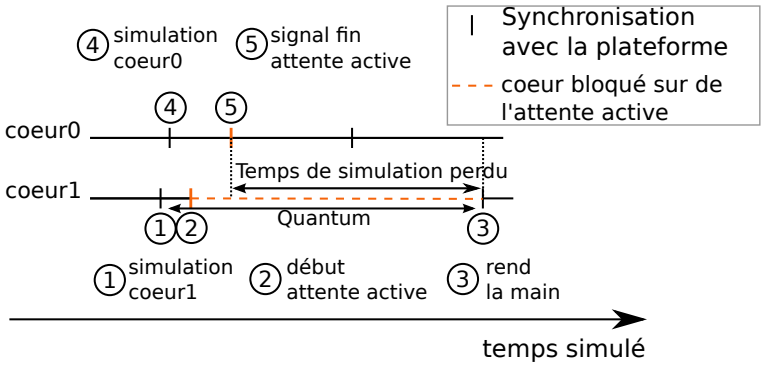


FIGURE 2.11 – Perte de temps de simulation en présence d'attente active

phénomène est amplifié avec l'augmentation du nombre de cœurs et l'enchevêtrement des verrous et des attentes actives.

Pour illustrer ce problème de passage à l'échelle de la simulation native, nous utilisons quelques applications issues du benchmark SPLASH-2, présenté par [WOT+95]. La plateforme de simulation utilisée représente une grappe de calcul de la puce Andey de Kalray. Le nombre de tâches dans les applications simulées est adapté au nombre de cœurs (PE) instanciés qui varient de 1 à 16. Pour comparer uniquement les performances de simulation sans tenir compte du surcoût dû à l'instanciation et à la destruction des composants, le temps d'exécution est mesuré entre l'étape `end_of_elaboration` de la plateforme SystemC, qui indique le début de la simulation, et la fin de `sc_start`, qui en indique l'achèvement. Les simulations sont réalisées sur un Intel® Core™ i7-3820 CPU à 3.60 GHz avec la version 3.3.4 de Linux. Les simulations sont réalisées avec un processeur hôte à 8 cœurs, Intel ® Xeon® CPU W3550 fonctionnant à 3.07 GHz avec Linux 3.10.0. La figure 2.12 représente l'accélération d'une simulation native basée sur KVM par rapport à une simulation utilisant des ISS. En fonction de l'application utilisée, notre simulation native ne passe pas à l'échelle de la même manière. On distingue principalement deux cas : les applications avec une accélération presque constante et celles avec une accélération qui diminue quand le nombre de PE augmente.

La première catégorie contient *FFT*, *LU* et *water-spatial*. Les tâches de ces applications ne se synchronisent presque jamais. Elles n'utilisent que quelques barrières et verrous, comme montré dans [WOT+95]. La deuxième catégorie contient *radix*, *ocean*, *water-nsquared* et *ray-*

trace. Beaucoup plus de synchronisations sont utilisées. Il y a, par exemple, des milliers de prises de verrous dans *ocean*, *water-nsquared* et *raytrace*, plus quelques centaines de barrières (utilisant de l'attente active) dans *ocean*. Dans *radix*, il n'y a pas de prise de verrou mais de l'attente active sur une condition est réalisée.

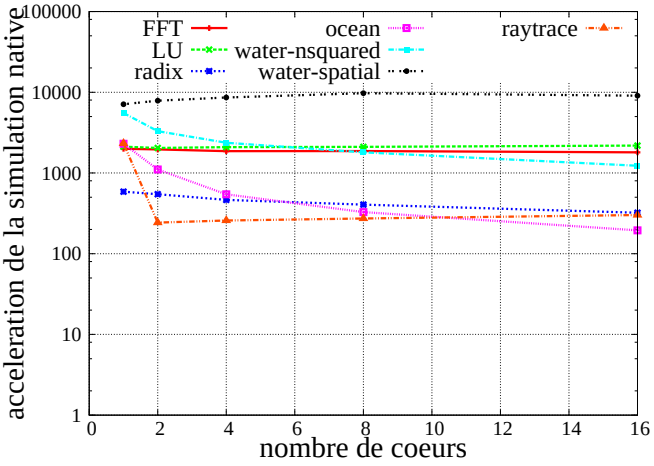


FIGURE 2.12 – Accélération de la simulation native par rapport à l'ISS

Ainsi la simulation native avec la structure présentée précédemment passe facilement à l'échelle dans le cas d'applications ayant des threads qui n'interagissent pas entre eux. Par contre les performances de simulation chutent drastiquement dès que l'on commence à synchroniser les threads, à ajouter des sections critiques tout en augmentant le nombre de cœurs à simuler.

Un autre exemple montrant ce problème de passage à l'échelle avec une autre technique de simulation native se trouve dans la thèse de Gerin [Ger09], chapitre 7. L'application MJPEG est simulée sur une plateforme dont le nombre de cœurs varie de 1 à 8. Les performances du simulateur natif sont comparées à celles d'un simulateur CABA (*cycle accurate, bit accurate*). Le tableau 2.4 donne l'accélération de leur simulation native par rapport au simulateur CABA qu'ils ont utilisés (ces chiffres sont tirés du rapport de thèse de Gerin). Là encore, quand le nombre de cœurs simulés augmente, l'accélération fournie par la simulation native diminue.

nombre de cœurs simulés	1	2	3	4	6	8
Accélération	181	78	60	62	58	58

TABLE 2.4 – Accélération de la simulation native par rapport à une plateforme CABA pour une application MJPEG

2.6 Conclusion

Les many-cœurs sont les futurs processeurs. La présence de spécificités architecturales propres aux cœurs cibles et non présentes sur l'hôte va devenir de plus en plus fréquente. Dans le même temps le parallélisme disponible en matériel et le parallélisme utilisé dans les

applications vont augmenter. Il est nécessaire de disposer d'une simulation fonctionnellement précise et rapide pour prototyper les futures puces, faire de l'exploration architecturale et développer les applications allant sur ces many-cœurs.

Or nous avons vu que la simulation native basée sur la virtualisation utilise des machines virtuelles qui ont les mêmes caractéristiques que la machine hôte. Cela ne permet pas de simuler certaines spécificités architecturales propres à la machine cible. Notamment, la FPU hôte n'est pas simulable précisément en simulation native et la technique actuelle d'émulation logicielle à l'aide d'entiers n'est pas assez rapide pour les techniques dites "rapides" telles que la simulation compilée, la SBT et la DBT. De plus les performances du passage à l'échelle de la simulation native sont directement dépendantes de l'application s'exécutant sur la plateforme de simulation.

Dans le cadre de cette thèse, nous nous posons les questions suivantes :

— Comment simuler efficacement en simulation native basée sur le HAV de nouvelles fonctionnalités propres à la machine cible ?

— Comment simuler efficacement et précisément des calculs en virgule flottante, en se limitant dans un premier temps à la simulation compilée ?

— Comment permettre à la simulation native d'avoir de bonnes performances lors du passage à l'échelle sans dépendre de l'application s'exécutant sur la plateforme de simulation ?

Chapitre 3

Évolution de la simulation : l'efficacité et la précision

Les premières formes de simulation native sont apparues dans les années 2000 [GLMS02] avec l'idée d'exécuter nativement l'application cible à l'intérieur de la plateforme virtuelle tel un logiciel *bare metal*. Depuis de nombreuses améliorations ont eu lieu.

Nous verrons tout d'abord la progression de la simulation native jusqu'au stade actuel, la technique de simulation native basée sur le HAV. De nombreuses recherches ont eu lieu pour compenser la perte de précision dûe à l'augmentation du niveau d'abstraction. Mais la simulation native, et plus particulièrement la simulation native basée sur le HAV, apporte un autre problème peu ou pas abordé qui est la simulation performante et fonctionnellement correcte de caractéristiques spécifiques aux cœurs cibles. La simulation rapide et numériquement exacte de la FPU fait partie de ces caractéristiques spécifiques non abordées même à un niveau plus modeste, tel que la simulation compilée. Finalement la simulation native introduit des problèmes connus pour les personnes utilisant le découplage temporel comme modèle temporel de simulation : des erreurs temporelles et de causalité.

3.1 Petite histoire de la simulation native

L'idée d'exécuter nativement l'application cible à l'intérieur de la plateforme virtuelle est apparue dans les années 2000 avec [GLMS02]. Pour permettre de simuler des applications plus compliquées comme des applications multi-tâches, ou le partage logiciel de ressources matérielles, un modèle haut niveau d'un système d'exploitation est encapsulé à l'intérieur d'un composant TLM [YGG03], fournissant ainsi une interface haut niveau à l'application cible comme représenté par la figure 3.1. Mais la quantité de code cible pouvant être simulée avec ce système reste limitée : l'ensemble des couches logicielles de niveau égal ou plus bas que le système d'exploitation ne sont prises en compte que par un modèle abstrait loin de l'implémentation finale.

Pour palier à ce problème, une solution proposée [KGW+07] est de mettre en place un système hybride en réintroduisant l'usage d'un ISS pour l'ensemble des fonctions considérées comme critiques (code assembleur, code ayant des effets de bords sur le système, bibliothèques héritées, etc.). Mais cette solution nécessite un découpage du code simulé entre simulation native et simulation par ISS. Cela peut être fait au moment de la compilation [MRRJ05] ou dynamiquement au cours de la simulation [GKK+08]. Si la proportion de code critique n'est pas négligeable, alors l'utilisation de l'ISS ralentit significativement la

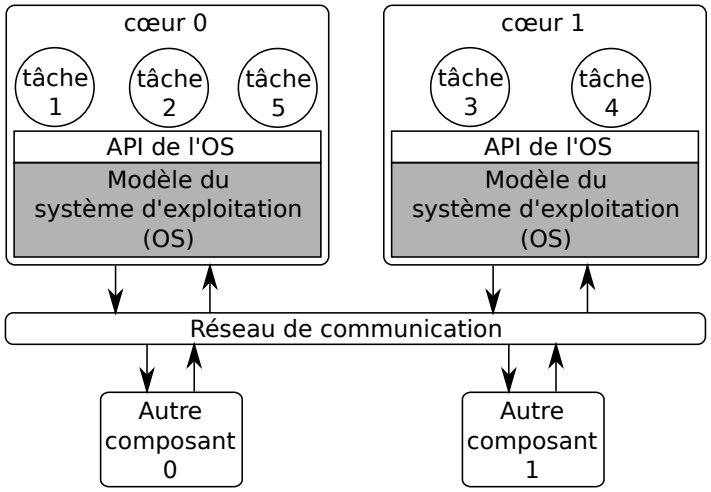


FIGURE 3.1 – Encapsulation des tâches logicielles à l'intérieur d'un composant matériel possédant un modèle du système d'exploitation

simulation. De plus, un problème fondamental et globalement mal résolu vient de la difficulté à maintenir la cohérence entre les ISS et les modèles de processeurs natifs.

La définition d'une fine couche d'abtraction du matériel (Hardware Abstraction Layer (HAL)) [GSC+07] permet de maximiser la quantité de code cible simulé nativement tout en se passant d'un ISS. Ainsi les développeurs peuvent tester la quasi-totalité de leur code à l'exception de quelques fonctions très bas niveau, celles du HAL.

On observe donc d'un côté une augmentation du niveau d'abstraction de la simulation et d'un autre côté une diminution de la quantité de code cible devant être abstrait (ou simulé de manière non native).

Problème du double espace d'adressage Le problème de la cohabitation de 2 espaces d'adressage (l'espace d'adressage de la plateforme cible et celui de la machine hôte) a été soulevé par Gerin [GGP08] et Posadas [PV09]. En effet, le code exécuté nativement tourne dans l'espace d'adressage hôte alors qu'il a été écrit pour touner dans l'espace mémoire cible. Il est possible de profiter du fait qu'une exception est levée lorsqu'un processus du système d'exploitation hôte tente d'accéder à une adresse virtuelle se trouvant dans une page invalide. C'est l'idée présentée par Posadas [PV09], qui propose d'invalider dans le système hôte l'ensemble des pages mémoires correspondant aux adresses cibles auxquelles le processeur doit accéder pour lire et écrire dans les composants matériels de la plateforme. Lors d'un accès à l'une des adresses cibles, une exception est levée, puis transmise au simulateur qui génère la traduction d'adresse nécessaire. Cette technique apporte une accélération plutôt faible par rapport à ce que l'on peut espérer de la simulation native.

Gerin [GGP08] propose une technique pour unifier la représentation mémoire. Chaque composant exporte un ensemble de symboles permettant, au moment de l'édition de lien dynamique ayant lieu lors de l'élaboration de la plateforme, de réaliser un remappage des adresses. Ceci consiste à changer les adresses cibles des composants pour qu'elles correspondent aux adresses hôtes allouées aux composants dans le simulateur. Cette solution oblige à modifier les composants et à ajouter une phase de liaison spécifique, ce qui la rend par conséquent incompatible avec les approches de modélisation préexistantes. Ces deux propositions ne sont cependant pas viables lorsque les espaces d'adressage cible et hôte se

recouvrent.

Pour répondre efficacement à ce problème, l'utilisation en simulation native de l'assistance matérielle à la virtualisation a été introduite par [SHP12] et développée dans [Ham13] pour les machines cibles mono-processeur de type ARM. L'application cible est exécutée en mode *guest* tandis que la plateforme de simulation est exécutée en mode *user*. De même qu'une table des pages est maintenue par le système d'exploitation pour assurer la traduction des adresses virtuelles utilisateurs vers les adresses physiques, une autre table des pages (*shadow page table*) est maintenue par l'hyperviseur pour garantir la traduction des adresses physiques guest vers les adresses physiques hôtes. Concernant le support matériel pour accélérer ces traductions, le système d'exploitation en mode user peut utiliser une TLB (*Translation Lookaside Buffer*) ou une MMU (*Memory Management Unit*). L'équivalent matériel pour accélérer la traduction réalisée par l'hyperviseur est l'*extended page table* pour Intel et la *nested page table* pour AMD.

3.2 Simulation des composants liés au cœur

Concevoir un simulateur possédant une grande précision temporelle est difficile, voire impossible [GKO+00]. Cependant même un simulateur ayant des métriques imprécises en valeur absolue peut prévoir correctement des tendances comme le temps de simulation ou la consommation. Dans ce contexte, des techniques de simulation de plus en plus rapides sont utilisées au détriment de leur précision absolue. Néanmoins ces simulations doivent rester fonctionnellement correctes et avec une précision suffisante. L'exécution native du code cible n'a pas de lien avec son exécution sur la machine cible. De plus la simulation native basée sur le HAV ne peut fournir que les caractéristiques fonctionnelles du cœur hôte et non pas les caractéristiques spécifiques des nouvelles architectures cibles.

3.2.1 La simulation native et la recherche de la précision

Les annotations La simulation temporelle du cœur cible est reproduite à l'aide d'annotations. Les annotations sont des instructions spécifiques non fonctionnelles rajoutées généralement de manière automatique dans le code à simuler. Cela permet de rendre compte du temps d'exécution d'une application en fonction des contraintes architecturales de la plateforme cible. Mais cela ne permet nullement de simuler des caractéristiques macro-architecturales spécifiques présentes à l'intérieur du composant cœur. Différentes solutions existent pour insérer des annotations.

La gestion des annotations temporelles peut être faite de 3 manières différentes. Tout d'abord, elle peut transformer le binaire cible final en du code haut niveau (*e.g.* du C) qui reproduit le comportement temporel de l'application comme présenté dans [vM96, BKL+00] et illustré par la figure 3.2a. Cela permet de tenir compte des optimisations du compilateur cible mais des modifications peuvent être apportées lors de la compilation du code haut niveau par le compilateur hôte.

Toujours en se basant sur le binaire cible mais en utilisant également les informations de débogage, les annotations peuvent être insérées au niveau du code source comme proposé par [WSH08, HAG08, LLT10] et illustré par la figure 3.2b. Cette solution est complexe à mettre en place. De plus des modifications d'écriture du code source peuvent être nécessaires pour insérer correctement les annotations.

Finalement les annotations peuvent être insérées au niveau de la représentation intermédiaire du compilateur natif. Cela permet d'obtenir une équivalence entre le graphe d'exécu-

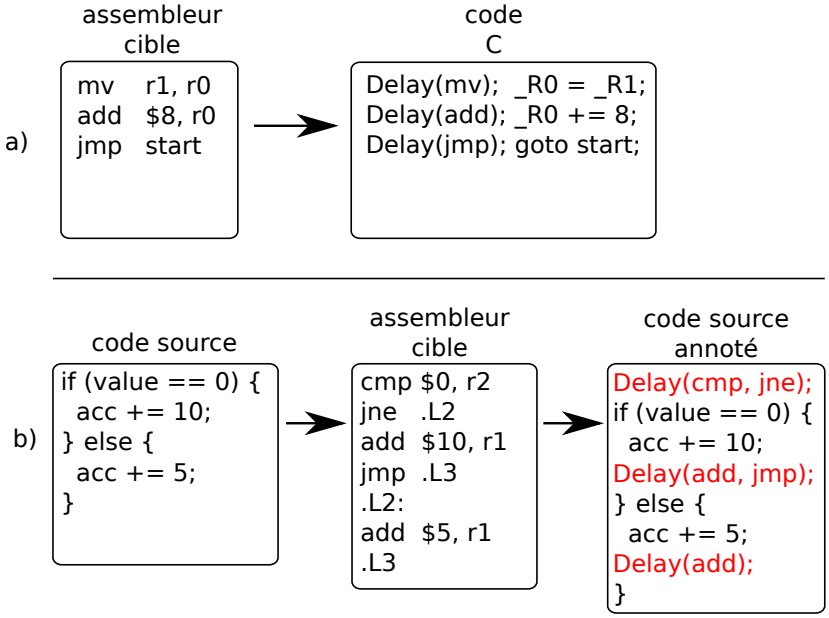


FIGURE 3.2 – Différentes techniques d'annotations : génération de code haut niveau et insertion des annotations dans le code haut niveau

tion des blocs de base cibles et hôtes. Cette technique a été démontrée avec l'IR de GIMPLE par Wang [WH09] et avec celle de LLVM par Bouchhima [BGP09]. Même si le travail que nous décrivons dans la suite se concentre sur la simulation fonctionnelle, son extension en un simulateur permettant la mesure de performance du code simulé est une perspective intéressante.

Simulation temporelle de composants Il existe des composants liés aux cœurs n'ayant pas un impact fonctionnel direct mais uniquement un impact temporel sur le temps d'exécution d'une application. Il s'agit de l'ensemble de la hiérarchie mémoire, en particulier les caches (or problème de cohérence de cache).

Pour accélérer la simulation d'un cache d'instruction, Castillo *et al.* [CPVM10] proposent de ne plus simuler précisément les caches comme dans les ISS. Au lieu de cela, une application est représentée par l'ensemble de ses lignes d'instructions chargeables dans le cache comme représenté dans la figure 3.3. Chaque représentation d'une ligne est stockée dans un tableau dans l'ordre des adresses des instructions. Chaque case du tableau indique si la ligne est présente ou pas dans le cache et dans quel groupe du cache elle se trouve. Une passe d'annotations est ajoutée pour indiquer quelles sont les lignes d'instructions utilisées par bloc de base. A la fin de l'exécution de chaque bloc de base, un test est réalisé pour savoir si les lignes d'instructions sont présentes ou non dans le cache et réaliser les remplacements prévus par le protocole. L'intérêt principal de cette méthode est qu'elle permet de savoir en temps constant si la lecture d'une instruction génère un *HIT* ou un *MISS*. Mais pour associer un bloc de base en assembleur à son équivalent en langage C, des labels sont insérés dans le code C à l'aide d'instruction `asm volatile`. Cela empêche certaines optimisations du compilateur : les déplacements d'instructions assembleurs hors des bornes décrites par ces labels ne sont plus possibles.

Díaz *et al.* [DPV10] proposent une méthode pour simuler les caches de données en simulation native. Ils reprennent la méthode d'annotation proposée par Castillo [CPVM10]

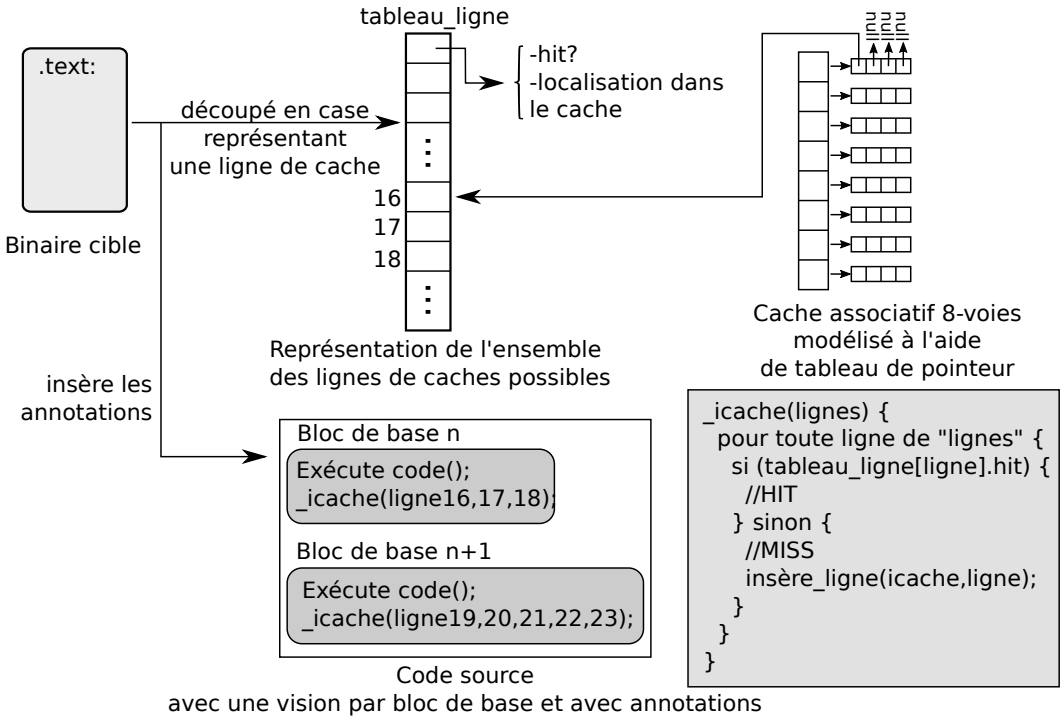


FIGURE 3.3 – Simulation rapide d'un cache d'instruction

et la complètent pour ajouter des annotations sur les lectures et les écritures de données. La localité spatiale leur permet de considérer les accès mémoires en simulation native très semblables à ceux de la machine cible. Pour renforcer cette hypothèse, le même *front-end* du compilateur est utilisé et la génération de lien a lieu dans le même ordre. De plus ils modifient les structures de données du code compilé pour la machine hôte dans le but de garantir que les structures de données aient la même taille en simulation native et sur la machine cible. L'idée d'un tableau de bits réprésentant de façon concise l'ensemble de la mémoire est de nouveau utilisée [PDV11] pour savoir rapidement et en temps constant si un *HIT* a été généré par le cache de données.

Pour simuler les accès mémoires de façon plus précise, les allocations mémoires du tas et de la pile sont simulées [LMGS13a, WH13]. Il est aussi proposé de récupérer l'adresse d'un pointeur en ajoutant une variable simulant ses assignements, cette variable est ensuite utilisée dans les annotations d'accès mémoire. Différentes politiques d'annotations du code haut niveau sont décrites en fonction des syntaxes de contrôle utilisées. Cette simulation donne des résultats très précis mais limite l'accélération fournie par la simulation native.

Pour accélérer la simulation des caches, une réduction du nombre d'annotations générant des accès aux caches [LMGS13c] est proposée en retirant les accès redondants ou les accès pour lesquels on peut déterminer statiquement qu'ils généreront un *HIT*.

Un autre composant du cœur ayant un impact temporel et non fonctionnel est le prédicteur de branchement. Faravelon [FFP15] propose une simulation d'un prédicteur pour de la DBT. Des études statiques ont également eu lieu, notament pour estimer un pire temps d'exécution [CP00].

Pour compenser la perte d'information dûe au niveau d'abstraction de la simulation native, et donc la perte de précision temporelle, de nombreux travaux ont été réalisés pour

améliorer la précision temporelle du code cible simulé nativement. Néanmoins uniquement les aspects temporels sont pris en compte. Il n'y a pas eu de travaux pour permettre dans le cadre de la simulation native basée sur le HAV, de prendre en compte des caractéristiques matérielles liées aux cœurs cibles, non présentes sur le cœur hôte, et n'ayant pas d'impact temporel mais uniquement fonctionnel.

3.2.2 Unité de calcul à virgule flottante

Les unités de calcul à virgule flottante (FPU) sont maintenant largement utilisées même dans les processeurs embarqués. Néanmoins une simulation rapide et précise numériquement ne semble pas avoir été recherchée jusqu'à présent. Selon nous, cet état de fait vient principalement des 3 raisons suivantes.

— Tout d'abord la majorité des programmeurs considèrent le calcul à virgule flottante comme étant similaire (ou en tout cas très proche) du calcul avec des réels sans tenir compte des concepts de base [Gol91].
— Ensuite les quelques développeurs ayant conscience des difficultés du calcul flottant, soit évitent de l'utiliser car il est difficile de prouver la correction des calculs [Mon08, BF07], soit se reposent sur la norme IEEE 754. Cependant il y a plusieurs révisions de la norme et un processeur peut ne supporter qu'un sous-ensemble de la norme ou introduire de nouveaux opérateurs.
— Finalement l'émulation logicielle des calculs à virgule flottante basée sur des entiers résout le problème de la correction de la simulation des calculs FP car elle garantit l'équivalence des opérations au détriment de la vitesse de simulation.

Nous définissons la simulation d'un opérateur de calcul flottant comme étant *numériquement équivalent* à l'opérateur implémenté sur la machine cible si pour toutes les valeurs d'entrée de l'opérateur, la sortie en simulation est bit à bit identique à celle de la machine cible et les mêmes informations (*e.g.* flags) sont remontées si besoin. Prenons comme exemple la multiplication de deux nombres à virgule flottante. Soit $\otimes$ le symbole représentant l'opérateur de la multiplication de 2 nombres à virgule flottante simple précision et retournant un nombre en simple précision. Soit l'opération suivante : $1.00111_b \times 2^{10000000_b} \otimes 1.00001_b \times 2^{00000100_b}$ (les 0 finaux du significande ne sont pas notés). Si la machine cible retourne $1.0100000111_b \times 2^{10000100_b}$ et ne soulève aucun flag, la simulation devra faire de même.

De nombreux travaux ont eu lieu pour améliorer l'implémentation des FPU, vérifier leur implémentation, définir de nouveaux opérateurs, prouver la correction d'un calcul flottant. Mais aucun à notre connaissance n'a essayé d'améliorer les performances de simulation tout en garantissant le même comportement numérique, *i.e.* les opérations simulées sont numériquement équivalentes aux vraies opérations. De manière plus surprenante, les simulateurs de GPU tels que le mode émulation de CUDA [CDDP10] ou des approches *ad-hoc* telles que [GBR15] utilisent directement la FPU hôte pour simuler les instructions de calcul flottant cible sans prendre aucune précaution.

En parcourant le spectre des travaux ayant lieu sur la FPU, une amélioration de la consommation de la FPU est proposée dans [JA15] à travers une implémentation de celle-ci se basant sur des portes logiques représentant des opérations réversibles [Lan61]. L'intérêt des portes réversibles est qu'elles sont sensées peu ou pas chauffer car il n'y a pas de perte d'information. Mais ces études reposent sur les qubits et les opérations quantiques qui ne sont actuellement toujours pas fonctionnelles de façon certaines. En continuant sur l'amélioration des caractéristiques de la FPU, une analyse de la consommation énergétique et de

l'occupation spatiale d'une FPU est réalisée dans [SD15] en comparant une implémentation avec une technologie en 45nm et en 15nm. L'étude montre que l'implémentation en 15nm d'une FPU prend environ 1/3 de la surface utilisée en 45nm et la consommation énergétique est réduite d'un facteur 3 à 4.

Un autre aspect lié à la FPU est la vérification de son implémentation. Pour prouver qu'elle est correcte, une méthode de vérification formelle proposée par [JWPB05] permet de vérifier que chaque instruction de calcul flottant s'exécute correctement. Mais cette méthode ne permet de vérifier le fonctionnement de la FPU que dans une configuration très particulière du cœur, son pipeline ne devant contenir qu'une seule instruction. Pour étendre cela à n'importe quelle configuration du pipeline, un test d'équivalence est rajouté par [KPA+14]. Tout d'abord une instruction, prouvée juste si exécutée seule, est exécutée seule puis elle est exécutée au milieu d'une suite d'instructions. Les résultats des 2 suites d'opérations sont comparés pour vérifier que l'instruction donne le résultat correct malgré la possibilité de vider le pipeline (*flush*) ou de récupérer les sorties du pipeline pour une instruction déjà en cours de traitement dans le pipeline (*by-passing des arguments*).

Finalement des travaux portent sur la proposition et l'utilisation de nouveaux opérateurs. Le problème de l'erreur du double arrondi est résolu dans [BM08] grâce à l'utilisation de l'arrondi vers le nombre à virgule flottante le plus proche avec un significande impair. Malheureusement cet arrondi ne fait pas partie de la norme IEEE 754 et n'est généralement pas implémenté en matériel. Les opérations de FMA et FMS introduites dans la norme IEEE 754-rev08 sont utilisées dans [SS12] pour implémenter une transformée de Fourier rapide (*FFT*).

3.3 Découplage temporel de la simulation native

Avec une simulation basée sur la technique des ISS, les composants cœurs sont simulés cycle par cycle. Or le coût des changements de contexte (entre composants simulés) est largement supérieur à l'exécution d'une instruction en simulation native. C'est pourquoi la granularité d'exécution du composant cœur en simulation native ne peut pas être aussi fine qu'en ISS. Le composant cœur simule donc un ensemble conséquent de blocs d'instructions à chaque fois qu'il est élu. Son temps simulé avance significativement par rapport aux autres composants et un certain nombre d'accès mémoire peut être réalisé lors d'une exécution. La simulation native est donc utilisée dans un contexte dit de *découplage temporel*.

Quelle que soit la technique de simulation native, un problème de découplage entre le temps de la plateforme SystemC et celui des processeurs natifs apparaît. Le découplage temporel est connu pour introduire des erreurs temporelles et de causalité dans les interactions entre les composants. Des mécanismes spéciaux doivent être mis en place pour garantir une simulation correcte et maintenir de bonnes performances de simulation. Ce problème se retrouve par exemple dans les communications basées sur des FIFO [HCG+13].

Pour limiter l'effet du découplage temporel sur les interruptions, un mécanisme de partage du temps simulé par quantum de temps est mis en place [MEJ+12]. Néanmoins les fonctions simulées en natif sont exécutées entièrement avant que le temps simulé ne soit consommé, ce qui ne permet pas de simuler les événements à l'instant précis de leur arrivée. Le quantum est donc une borne inférieure, la borne supérieure étant dictée par la fin de l'exécution de la fonction. Cela peut introduire des problèmes fonctionnels si la fonction interagit avec ces événements, et en conséquence un certain nombre de fonctions ne peuvent plus être exécutées en natif (mais simulées avec un ISS), ce qui peut ralentir significativement la vitesse de simulation.

Pour compenser les erreurs temporelles introduites par des accès concurrents à des ressources partagées, une méthode analytique [LMGS13b] est proposée pour corriger dynamiquement le temps simulé des initiateurs (e.g. un cœur). Elle se base sur le taux d'utilisation d'une ressource et la priorité d'accès de l'initiateur pour ajuster le délai nécessaire à l'initiateur pour accéder à cette ressource. Le temps simulé peut ainsi être déduit mais l'ordre correct d'accès n'est pas garanti pour un accès concurrent à une variable partagée ou à une FIFO.

Différents travaux ont eu lieu concernant la correction fonctionnelle de l'exécution d'une application dans un contexte de découplage temporel. Leur but est de s'assurer que les accès aux variables partagées en simulation se fassent dans le même ordre que sur la plateforme physique. Pour cela, une synchronisation avec la plateforme de simulation à chaque accès à une variable partagée est réalisée [WFWT09, WWFT11]. Pour minimiser le nombre de synchronisations, la synchronisation peut se faire uniquement sur le début et la fin des sections critiques [YZH+13]. Cependant la détection des prises de verrous et leur relachement se fait sur du code binaire x86. Cela implique qu'il faut tester l'ensemble des instructions exécutées. De plus la méthode n'est pas compatible avec des protocoles de synchronisation non basés sur des verrous.

En se basant sur ces points de synchronisation et en prédisant le temps de simulation pour accéder au point suivant, différentes stratégies d'ordonnancement sont proposées [WFWT13]. Mais cela est fait dans le contexte de la simulation compilée et ne peut pas être directement transposé à la simulation native.

Orientation vers un simulateur parallèle

Le découplage temporel peut impacter la parallélisation de la simulation, ou inversement la parallélisation peut renforcer les problèmes de découplage temporel.

Les techniques de simulation par événements discrets, utilisées par l'ordonnanceur de SystemC, sont difficilement parallélisables [Fuj90]. Néanmoins différentes tentatives ont eu lieu. La première idée très conservative est de paralléliser les tâches à l'intérieur d'un même *delta-cyle* [CCZ06, PCC+09, SLPH10]. Cela a été réalisé pour SystemC mais aussi pour d'autres systèmes de SLDL (*System-Level Description language*) tel SpecC [DCHG11]. Toujours dans les méthodes conservatives, une méthode appelée *Out-of-Order Parallel Discret Event Simulation* [CHD12, CD13] a été développée pour trouver les dépendances entre les tâches lors de la compilation et permettre de simuler en parallèle sur quelques cycles. Mais le niveau de parallélisme reste encore bas.

Une autre solution est d'introduire la notion de durée à l'intérieur de la syntaxe de systemC [Moy13]. Plusieurs tâches peuvent ainsi être simulées en parallèle mais uniquement dans le contexte du *coding style* dénommé *loosely timed* qui autorise plus de souplesse dans la gestion du temps (seul le temps du début et de la fin d'une transaction comptent).

De nombreuses tentatives pour paralléliser la simulation entre les composants cœurs existent en s'affranchissant partiellement de l'ordonnanceur de SystemC. Parallel Embra [Lan08] propose une simulation fonctionnelle et parallèle en instanciant un simulateur par groupe de cœurs cibles. Aucune synchronisation n'est faite entre les cœurs. La cohérence du système repose sur le fait que tous les cœurs accèdent à la même mémoire et donc voient les événements dans le même ordre. Ce système est clairement non déterministe.

Graphite [MKK+10] est un autre simulateur parallèle qui vise un très grand nombre de cœurs cibles et qui peut notament être distribué sur plusieurs machines hôtes. Pour garantir un bon passage à l'échelle, les synchronisations entre les cœurs sont minimisées, pouvant

conduire à des erreurs de causalité. Pour minimiser la désynchronisation entre les unités de simulation, chaque grappe simulée choisit régulièrement une autre grappe et se synchronise avec elle. Ce mécanisme a l'avantage de pouvoir être complètement distribué et d'éviter une synchronisation globale assez lourde. Mais le niveau de simulation se place au dessus de l'abstraction d'un système d'exploitation.

Dans la même optique, InvadeSIM [RSHT15] utilise un ensemble d'*activités* qu'il associe à un thread. Ces activités (qui correspondent à des tâches cibles) sont créées et associées dynamiquement. Le code source est directement compilé pour le système hôte sans aucune annotation. Le temps simulé est estimé à l'aide d'un mécanisme appelé *time-warping* [RHT12] qui prend comme paramètre d'entrée le nombre d'instructions hôtes utilisées et qui est constitué d'un ensemble de formules modélisant un processeur.

Une autre optique est d'utiliser un émulateur déjà existant. COREMU [WLC+11] utilise Qemu comme élément de base à la parallélisation. Pour chaque processeur simulé, une instance de Qemu est générée. L'ordonnanceur des différents processeurs virtuels prend notament en compte les détenteurs d'un verrou ou les vaines tentatives d'en obtenir un pour optimiser la sélection des processeurs virtuels.

Dans le cas particulier des architectures many-cœurs, des réseaux sur puces (NoC) sont utilisés pour communiquer entre les grappes (ou tuiles). Ce partitionnement naturel peut être utilisé pour paralléliser la simulation [MMGP10, WSL+14] : chaque grappe est simulée par une instance différente, chaque instance communiquant avec les autres par une abstraction des NoC. Les informations sur les temps de communication des NoC permettent de déduire des informations importantes pour minimiser la désynchronisation des grappes tout en garantissant un bon degré de parallélisme.

3.4 Conclusion

Nous avons vu que la simulation native a progressé pour passer d'une tâche haut niveau compilée pour la machine hôte et encapsulée à l'intérieur du composant cœur à la simulation de presque toute la pile logicielle, HAL mis à part. Ce code s'exécute à l'intérieur d'une VM qui est associée au composant cœur. De nombreux travaux ont amélioré la précision temporelle de la simulation native en insérant dans le code source ou dans la représentation intermédiaire (IR) du compilateur des annotations. Ces annotations servent par exemple à caractériser le temps d'exécution d'un bloc de base, à reproduire les accès mémoires aux instructions ou aux données. Elles permettent la simulation des composants liés aux cœurs ayant un impact temporel mais non fonctionnel sur l'exécution d'une application. Il n'y a pas eu de recherche pour simuler en simulation native basée sur le HAV les composants liés aux cœurs et ayant des caractéristiques fonctionnelles non présentes dans le cœur hôte.

La FPU fait partie des éléments fonctionnels pouvant varier entre le cœur cible et le cœur hôte. Des recherches pour améliorer les performances en terme de consommation énergétique ou de surface d'une FPU ont été faites. D'autres recherches mettent en avant des techniques de vérification de l'implémentation de la FPU. D'autres encore proposent de nouveaux opérateurs ou de nouveaux usages de ceux-ci. Mais la question de savoir comment réaliser une simulation numériquement équivalente de la FPU cible tout en garantissant des vitesses de simulation performantes n'a pas été étudiée.

Finalement la simulation native pose le problème du découplage temporel de la simulation. Ce problème est principalement visible lors d'accès à des ressources partagées. Les techniques de simulation utilisées pour travailler sur le découplage temporel n'utilisent pas

la simulation native basée sur le HAV. De plus les problèmes de perte de temps simulé dûs au découplage temporel ne sont pas mis en avant.

Nous allons voir comment, en simulation native basée sur le HAV, tenir compte des composants liés aux cœurs et ayant des caractéristiques fonctionnelles non présentes dans le cœur hôte. Nous verrons aussi un moyen efficace de faire une simulation numériquement équivalente de la FPU cible dans le cadre restreint de la simulation compilée. Finalement le problème du découplage temporel en simulation native basée sur le HAV est traité et des solutions sont proposées.

Chapitre 4

Simulation fonctionnelle

Les recherches concernant les techniques de simulation sont concentrées sur l'amélioration de la vitesse de simulation. Une augmentation du niveau d'abstraction a permis l'apparition de nouvelles formes de simulation : la simulation compilée, la SBT, la DBT et la simulation native. Mais plus le niveau d'abstraction augmente, plus le manque de précision temporelle se fait ressentir. De nombreuses recherches ont donc eu lieu pour améliorer la précision de ces techniques, mais relativement peu pour supporter des caractéristiques fonctionnelles nouvelles. Le problème qui nous intéresse se concentre sur l'aspect fonctionnel de la simulation. La simulation native basée sur le HAV utilise très fortement les caractéristiques du processeur hôte. Or les puces cibles essaient de se démarquer en intégrant des caractéristiques de plus en plus éloignées de celles que l'on peut trouver dans un processeur hôte pour répondre, par exemple, à des besoins particuliers de certains marchés.

C'est dans ce contexte que nous allons voir comment peuvent être simulées des caractéristiques architecturales nouvelles en simulation native basée sur le HAV. De plus, comme vu dans le chapitre 2, la simulation native des calculs en virgule flottante ne peut pas garantir le même comportement que celui de la FPU cible. Plus largement, une simulation fonctionnellement exacte d'une FPU consomme beaucoup de temps de simulation pour les techniques de simulation dites rapides telles que la simulation compilée ou la DBT. Nous présenterons la simulation compilée utilisée pour illustrer notre proposition, puis nous verrons comment tirer profit de la FPU hôte tout en garantissant la simulation numériquement équivalente de la FPU cible.

4.1 Simulation fonctionnelle correcte de fonctionnalités propres à la machine cible

Nous allons voir comment gérer des caractéristiques architecturales spécifiques à une machine cible avec la simulation native basée sur le HAV. Le premier élément est la gestion d'une machine cible multi-cœurs. C'est une caractéristique plutôt généraliste mais elle n'est cependant pas trivialement résolvable en simulation native basée sur le HAV. Nous présenterons ensuite une extension de la technique présentée par [Ham13] pour supporter de façon efficace l'ensemble des caractéristiques spécifiques à une machine cible.

4.1.1 Rappel du fonctionnement de la simulation native basée sur le HAV

Initialisation de la plateforme de simulation

Comme entrevu dans le chapitre 2, l'idée de Shen [SHP12] est d'utiliser le support de la virtualisation disponible dans les processeurs généralistes récents (x86, PowerPC, SPARC, et ARM dernièrement) pour résoudre le problème du double espace d'adressage. Concrètement, ils proposent d'encapsuler une bibliothèque utilisateur de KVM dans le composant SystemC qui tient lieu de processeur, comme illustré dans la figure 2.6 présentée dans la section 2.2.3. Cette bibliothèque KVM instancie un moniteur de machine virtuelle (VMM) qui gère la machine virtuelle (VM) qui lui est associée.

Le principe général est le suivant : l'ensemble des couches logicielles (HAL, système d'exploitation, applications) est compilé dans un exécutable hôte (x86 en l'occurrence). Cette compilation insère des annotations pour les besoins de la simulation. Indépendamment, la plateforme de simulation décrite en SystemC est compilée sans aucune connaissance du binaire à simuler.

Lors du début de la simulation, un espace mémoire est alloué dans l'espace utilisateur et est passé à KVM à l'aide d'un appel système dédié. Cet espace mémoire, dans lequel est chargé le binaire exécutable, correspond à la mémoire visible par l'application simulée dans l'espace d'adressage cible (*guest*), c.-à-d. la mémoire de notre puce cible.

Exécution de la plateforme de simulation

Simuler un processeur consiste à exécuter la VM qui lui est associée grâce à des appels systèmes `ioctl(KVM_RUN)`. Cette VM va lire les instructions hôtes stockées en mémoire et les exécuter nativement. Cette exécution est non préemptive du point de vue de l'application utilisateur ayant lancé la VM. Lorsqu'une VM est en cours d'exécution, les annotations de temps contenues dans le code sont stockées jusqu'à ce que leur somme atteigne une certaine limite (le quantum de temps). La VM rend alors la main à l'application utilisateur en consommant d'un coup les annotations accumulées, cela permet aux autres composants de la plateforme virtuelle d'être simulés.

Après avoir exécuté une partie du code natif, la VM rendra la main pour une raison R (tampon d'annotation plein, défaut de page, accès mémoire autre que les adresses mémoires allouées précédemment, débug pour communiquer avec le stub GDB, etc.). Soit le pilote KVM peut gérer lui-même R (défaut de page par exemple), soit il ne peut pas et retourne en mode *user*. Le simulateur doit alors gérer R. La simulation du processeur du point de vue SystemC se résume en une boucle : exécution de la VM, consommation du temps simulé, gestion de R, et ce jusqu'à la fin de la simulation. Nous appellerons dans la suite de ce manuscrit une *trappe* l'action de quitter le mode *guest*.

Mécanisme d'échange entre l'application en mode user et l'application en mode guest

Un point important de la simulation native basée sur le HAV est le mécanisme de communication entre le code exécuté sur la VM et la plateforme de simulation. Ce mécanisme se base sur la capacité de KVM à propager les requêtes qu'il ne sait pas gérer jusqu'à l'application fonctionnant en mode user. Il y a 2 types d'accès I/O : les accès dit *memory mappped I/O* et les accès sur des ports. Les premiers correspondent à des accès mémoires classiques. Si un accès mémoire est demandé à une adresse non spécifiée comme faisant partie des plages d'adresse guest, alors KVM rendra la main à l'application en mode user. Les accès sur des

ports correspondent à des accès mémoires sur un système d'adresse complètement orthogonal au précédent. Cela est généralement utilisé en x86 dans le code de pilotes pour accéder à des périphériques. Les instructions spécifiques au x86 pour les accès sur des ports sont : `in[b|w|l|q]` et `out[b|w|l|q]`. Les processeurs embarqués n'ont pas de système d'accès à des ports mais font seulement des accès mémoires. Les risques d'interférence avec le code simulé est donc nul si nous nous limitons à des accès par des ports pour transmettre des requêtes à la plateforme en mode user. Ce mécanisme est appelé *trappe_io*. Si jamais un système basé sur des ports existe sur le cœur cible, il faudra alors réserver certains ports pour la simulation native.

Dans la figure 4.1, nous représentons uniquement la gestion de sortie à cause d'un accès sur un port. L'adresse en mémoire guest de la variable passée en paramètre à l'instruction `in/out` est transformée en une adresse en mémoire virtuelle utilisateur utilisable par l'application en mode user pour récupérer des données ou en transmettre à l'application guest. Ce mécanisme de communication entre l'application exécutée en mode guest et la plateforme s'exécutant en mode user permet d'échanger le contenu de variables ou de structures complètes (moyennant l'hypothèse que la représentation mémoire de la structure soit la même pour l'application user et guest). Par exemple, le mécanisme d'annotation utilise les *trappe_io* pour indiquer le temps d'exécution de l'application guest. Néanmoins le passage du mode guest au mode user est coûteux et prend plusieurs milliers de cycles.

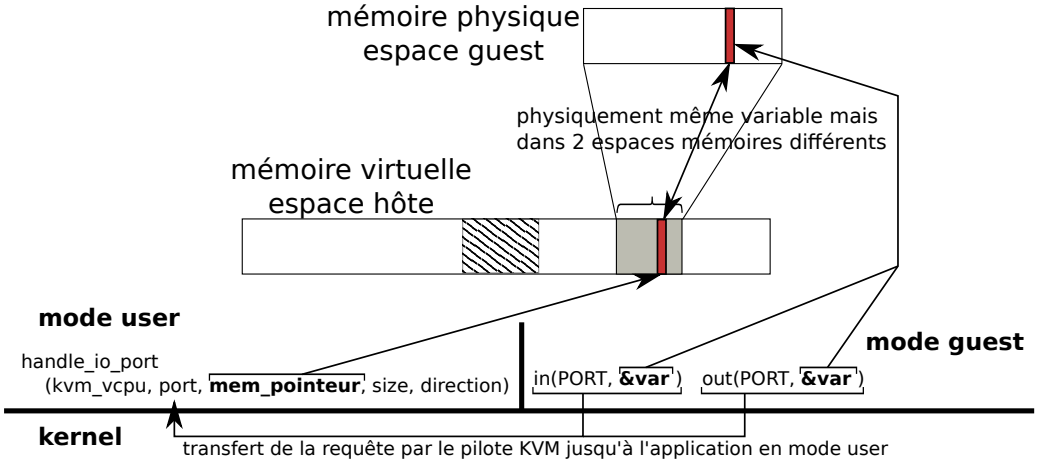


FIGURE 4.1 – Mécanisme de trappe permettant d'échanger des informations entre l'application s'exécutant dans une VM et la plateforme en mode user

4.1.2 Simulation de plusieurs cœurs

Nous allons voir maintenant comment gérer plusieurs cœurs en simulation native basée sur le HAV.

D'un cœur unique à une multitude de cœurs

Le changement d'un cœur à plusieurs cœurs simulés soulève différents problèmes liés au HAV. Le premier problème est lié au support du HAV par le système d'exploitation hôte. Le nombre de VM pouvant être instanciées à l'intérieur d'une VMM est ainsi limité à 254. Comme une VM est créée par cœur cible simulé, cette limitation peut devenir un problème pour la simulation d'un système many-cœurs. Cette limitation provient d'optimisation dans

l'implémentation de KVM. Elle peut être contournée en modifiant sa valeur dans le driver KVM de Linux et en recompilant ce driver. Cela permet d'instancier dans la même VMM l'ensemble des VM.

Une deuxième solution est de créer des groupes de cœurs, chaque groupe de cœurs étant associé à une VMM. Cette solution permet plus facilement de gérer l'augmentation du nombre de cœurs dans les puces. De plus, les many-cœurs actuels basés sur des tuiles ou des grappes de calcul présentent naturellement ce partitionnement. La solution peut aussi convenir aux many-cœurs non basés sur les tuiles/grappes en plaçant un seul cœur par VMM.

Les many-cœurs possèdent bien plus de cœurs que la machine hôte. Par conséquence plus d'une VM existe par processeur hôte durant l'ensemble de la simulation. Mais dans le cadre de KVM, il est conseillé de ne pas utiliser plus de 10 VM par processeur physique hôte pour éviter de sur-charger le système.

L'implémentation séquentielle de la simulation par événements discrets est telle qu'une seule VM à la fois est en cours d'exécution par plateforme SystemC, même si toutes les VM sont instanciées et en vie durant toute la simulation.

Boot natif

La simulation native basée sur le HAV introduit un problème lié au démarrage des cœurs cibles. En effet, les VM virtualisent les cœurs hôtes avec leur propre état. Pour une plateforme hôte basée sur des processeurs x86, KVM reproduit l'état des cœurs *uninitialized, init received, halted, sipi received, runnable*. Les transitions d'état sont aussi reproduites, en particulier pour le protocole de démarrage. Mais nous voulons simuler le protocole de démarrage de la machine cible, non pas celle de l'hôte.

Dans une approche séquentielle de la simulation, les machines virtuelles sont ordonnancées par la plateforme TLM. Si une VM n'est pas dans l'état *runnable* quand elle est exécutée, KVM attend que l'état de la VM passe à *runnable* sans rendre le contrôle au code s'exécutant en mode utilisateur. Puisque l'état interne de la VM ne changera jamais si le protocole de démarrage n'est pas exécuté, la simulation se retrouve dans une situation de blocage.

La façon par défaut de configurer une VMM est d'activer l'ensemble des options reproduisant la plateforme hôte normale avec en particulier ses contrôleurs nommés *Advanced Programmable Interrupt Controller (APIC)*. Malheureusement, des problèmes de superposition d'espaces d'adressage peuvent apparaître à cause de ces composants. Par exemple les registres des APIC locaux sont situés dans une zone mémoire de 4 KB adressable par le processeur physique avec une adresse par défaut commencant à `0xFEE00000`. Même si cette adresse peut être changée dynamiquement si besoin, au moment du boot KVM donne à l'APIC local *guest* l'adresse mémoire par défaut du composant sur la plateforme hôte. Si certains composants de la plateforme cible utilisent déjà cet espace mémoire, ou pire si la plateforme cible utilise tout l'espace mémoire disponible, des conflits entre les composants hôtes et cibles ont lieu. Donc le problème de superposition des espaces mémoires peut aussi être présent avec le HAV dans ces conditions.

Pour éviter cette situation, nous devons retirer le composant APIC de la VMM. KVM retourne alors dans un état où par défaut l'ensemble des VM est dans l'état *runnable*. Cela règle le problème de boot : il n'y a plus nécessité à respecter le protocole de démarrage multi-cœurs du x86. Mais cela introduit des problèmes concernant la propagation des interruptions jusqu'aux VM. Certaines modifications de la plateforme de simulation décrites dans 4.1.3 nous permettent de résoudre simplement ce problème.

4.1.3 Composants atypiques en simulation native

Les processeurs many-cœurs ont souvent des spécificités architecturales inhabituelles qui ne sont donc pas présentes sur l'architecture du processeur hôte. Certaines spécificités dépendent de caractéristiques micro-architecturales et affectent seulement le temps d'exécution des applications (*e.g.* la profondeur du pipeline ou la taille des caches) mais certaines autres sont fonctionnelles (*e.g.* la communication entre cœurs à l'aide d'événements matériels). Les détails micro-architecturaux peuvent être partiellement reproduits en utilisant des techniques d'annotations plus ou moins précises [LLT10, WH12]. Nous proposons ici une méthode pour gérer en simulation native les caractéristiques fonctionnelles non présentes sur la machine hôte.

Une méthode générale pour gérer ces spécificités architecturales est d'utiliser un modèle TLM à l'intérieur de la plateforme virtuelle décrivant ces spécificités. Un appel à ce modèle est réalisé à travers une *trappe_io*. Nous supposons que la spécificité architecturale est utilisée dans le code cible à travers des primitives du HAL, ce qui est cohérent avec la définition d'une couche d'abstraction matérielle. Le développeur software n'a donc aucune idée de la manière dont la caractéristique est implémentée (sur le vrai matériel et sur le simulateur).

La *trappe_io* force la VM à rendre la main et la ressource est simulée en utilisant le modèle TLM dans la plateforme virtuelle avant de retourner à l'exécution de la VM. La méthode est simple et facile à mettre en place. Son principal problème est qu'elle peut ralentir significativement la simulation à cause de trappes trop fréquentes.

Une trappe complète, une transition du mode guest au mode kernel puis finalement au mode user prend quelques milliers de cycles. Après cela il faut ajouter le temps nécessaire pour gérer la raison pour laquelle la VM a rendu la main et le temps de sélection du prochain composant à exécuter. Cela peut conduire à plusieurs dizaines de milliers de cycles hôtes perdus pour chaque trappe.

Pour minimiser le sur-coût des trappes, il est nécessaire de limiter les *trappe_io* à des usages très ponctuels et les remplacer si possible par une implémentation équivalente en mode guest. Nous détaillons notre solution qui consiste en une extension du modèle du processeur pour gérer les spécificités architecturales. Cette extension est faite à la fois dans la plateforme TLM en mode user et dans le HAL en mode guest. La Figure 4.2 illustre l'extension du modèle TLM alors que la Figure 4.3 illustre l'extension du HAL.

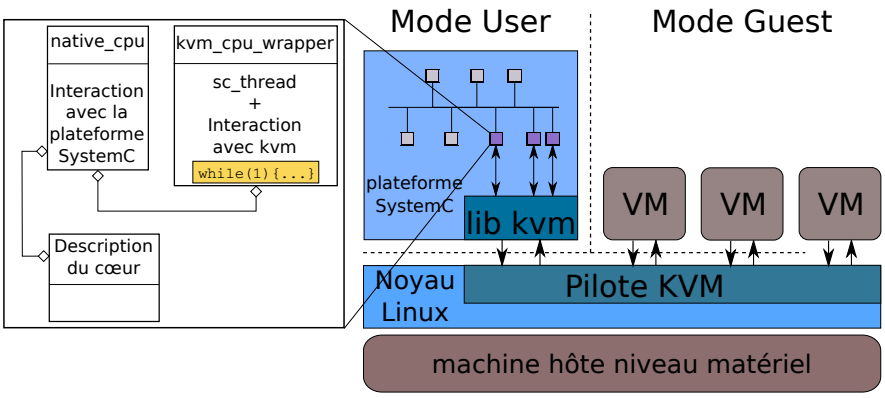


FIGURE 4.2 – Simulation de plusieurs processeurs spécifiques

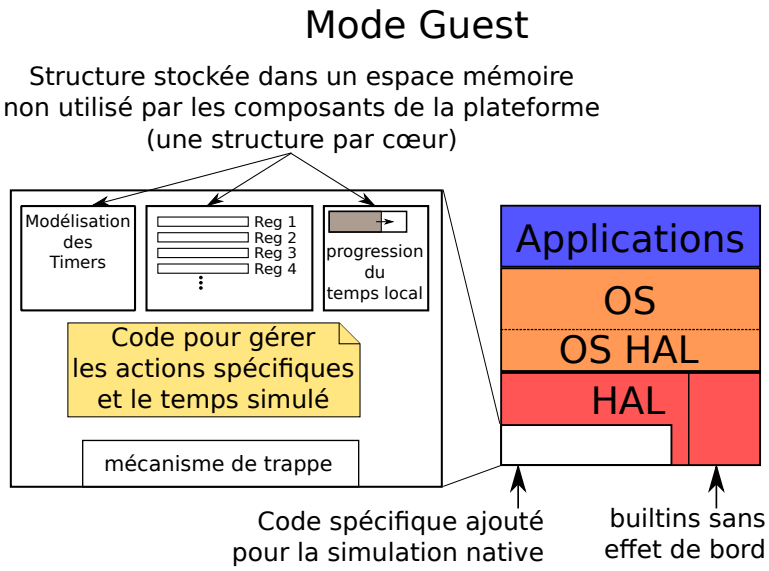


FIGURE 4.3 – Extension du HAL pour supporter des caractéristiques architecturales spécifiques

Le modèle du processeur contient :

— un wrapper KVM responsable de la communication avec KVM (*kvm_cpu_wrapper*),
— un wrapper natif responsable des interactions avec les autres composants de la plateforme (*native_cpu*),
— une description du cœur comprennant le temps local du cœur, ses registres et son état (power off, running, idle, *etc*).

L'état du processeur peut être accédé en mode guest à travers l'extension du HAL. L'inclusion des ressources locales au processeur à l'intérieur de la VM évite d'utiliser le mécanisme de trappe. Une région mémoire est ajoutée en mode guest pour stocker les registres du cœur et les autres informations importantes comme illustré par la Figure 4.4. Cette région mémoire ne correspond à aucune vraie zone mémoire du processeur cible. Elle est réservée pour la simulation native : les applications, le système d'exploitation simulé, et même la chaîne d'outillage (plus précisément l'éditeur de lien) ne doivent pas avoir connaissance de l'existence de cette zone mémoire. Les points d'accès à cette zone mémoire sont cachés dans le HAL et mis à jour par la partie TLM de la plateforme virtuelle avant chaque exécution d'une VM.

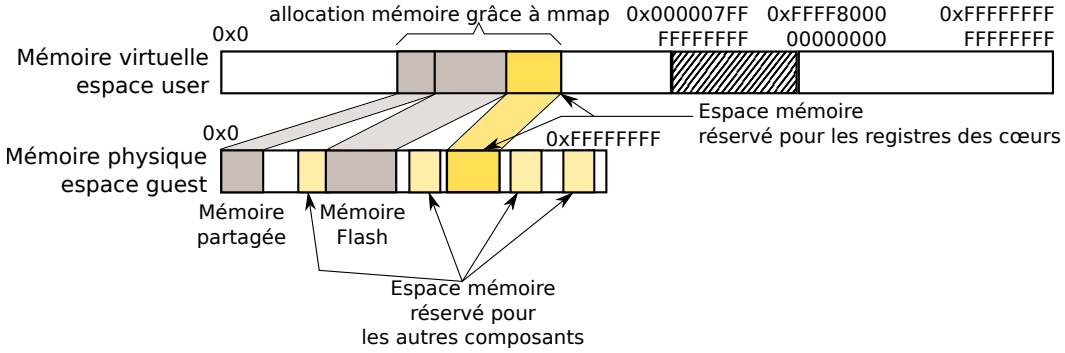


FIGURE 4.4 – Distribution de la mémoire entre le mode user et guest

Cependant le problème de découplage temporel introduit par la simulation native réduit l'ensemble des caractéristiques qui peuvent être gérées en mode guest. Pour respecter la causalité, un composant ne doit pas avoir d'action sur un autre composant se situant dans son passé ou dans son futur. Par construction le temps simulé des processeurs virtuels est toujours supérieur ou égal au temps global de la plateforme virtuelle. De manière similaire, chaque cœur simulé peut avoir un temps simulé différent de celui des autres cœurs. Par conséquence les cœurs virtuels ne peuvent pas agir n'importe quand sur les composants de la plateforme de simulation. Ils doivent d'abord se synchroniser avec la plateforme pour garantir que leur temps simulé est le même que celui des composants avec lesquels ils veulent communiquer. Le principe de causalité est ainsi respecté.

Cette méthode est maintenant illustrée à travers 3 exemples : les instructions de calcul avancé, les communications entre cœurs par événements matériels et la gestion des *timers*. Nous expliquerons ensuite comment gérer simplement les mécanismes d'interruption en se basant sur les modifications architecturales de la plateforme de simulation expliquées précédemment.

Instructions de calcul avancé

Les cœurs peuvent avoir des instructions optimisées pour le traitement intensif de données. Ces instructions sont généralement trop complexes et spécifiques pour être générées par le compilateur et doivent être insérées manuellement par le développeur. Si du code en assembleur cible est écrit (en utilisant par exemple la déclaration C `asm`), alors il n'y a pas d'autre solution que de réécrire la fonction d'une manière différente pour que le compilateur hôte puisse générer les instructions hôtes correctes. Certains compilateurs fournissent des primitives appelées *builtins* ou *intrinsèques* pour permettre l'utilisation de ces instructions dans un langage haut niveau : c'est l'approche que nous supposons.

Pour la simulation native, toutes les builtins spécifiques au cœur cible sont implémentées comme des fonctions du HAL. Les instructions de calcul avancé n'ont pas d'effet de bord sur la plateforme (*i.e.* elles agissent seulement sur l'état du cœur sur lequel elles sont exécutées). Donc aucune synchronisation avec le modèle TLM n'est nécessaire. Une émulation du comportement de ces instructions peut donc être réalisée en mode guest pour éviter de trapper.

Communications matérielles inter-processeurs

Les systèmes d'exploitations (OS) fournissent des services pour les communications entre threads ou entre processus en utilisant des interruptions et/ou des *mailboxes* pour les OS SMP (symétrique multi-processeurs). Le matériel peut de même fournir des ressources pour des communications à faible latence entre processeurs appelées *événements*. Evidemment, un événement envoyé par un cœur peut avoir des effets de bords sur le cœur destinataire et doit respecter la causalité. Il est nécessaire de se synchroniser avec le modèle TLM avant d'envoyer l'événement pour garantir la cohérence temporelle. Cette synchronisation force le cœur à se trouver au temps global simulé de SystemC quand il envoie l'événement.

La réception de l'événement est réalisée par le cœur destinataire à travers un test sur ses registres d'événement, en utilisant un appel d'une fonction du HAL. De nouveau une synchronisation est nécessaire avant de lire ses registres.

Gestion des timers

Les processeurs modernes peuvent embarquer un *timer* interne à chaque cœur. L'écoulement du temps a besoin d'être modélisé précisément pour lever les interruptions au bon moment. Nous proposons de modéliser la progression du timer en mode guest, comme illustré par la Figure 4.3, dans laquelle les registres des timers sont visibles. Ainsi, quand le temps local des processeurs simulés progresse (*i.e.* une annotation temporelle est rencontrée), les registres des timers sont directement mis à jour, sans aucune trappe. Comme les interruptions levées par les timers n'ont qu'un effet local sur le cœur simulé, utiliser une trappe ne semble pas nécessaire. Cependant une *trappe_io* est quand même générée pour rendre la main à la plateforme virtuelle quand un timer atteint un état dans lequel il doit en lever une. La plateforme virtuelle peut alors changer l'état du processeur simulé pour reproduire le mécanisme cible d'interruption.

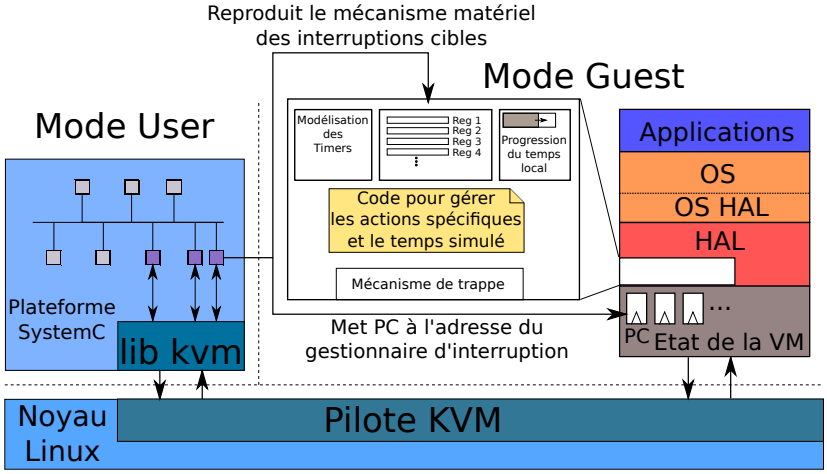
Gestion des interruptions

Le retrait des composants matériels APIC, comme expliqué dans la Section 4.1.2 lève le problème de la propagation des interruptions au cœur simulé. Notre proposition précédente permet d'accéder aux registres des cœurs cibles en mode user à travers la plateforme TLM et en mode guest à travers le HAL. Comme la plateforme TLM et le HAL ont la même vue de l'état du cœur cible, cela résout le problème de la propagation de l'interruption. La plateforme virtuelle reproduit le mécanisme d'interruption matérielle de la plateforme cible en mettant à jour les registres du cœur cible ainsi que certains registres de la VM (comme le pointeur d'instruction de la VM placé à l'entrée du gestionnaire d'interruption) comme représenté dans la Figure 4.5a. Quand la VM est exécutée, elle exécute le gestionnaire d'interruption (Figure 4.5b) et peut lire à travers le HAL les registres cibles simulés pour savoir exactement ce qui est arrivé, tout comme cela aurait été fait sur la plateforme cible, au lieu des flags hôtes. Une *trappe_io* spéciale est utilisée pour le retour d'interruption. Finalement la plateforme restaure l'état comme il était avant l'interruption pour reproduire un retour d'interruption (Figure 4.5c).

Comme la VM n'est pas préemptible à partir du mode user, les interruptions générales venant des périphériques sont gérées uniquement quand le cœur est synchronisé avec la plateforme.

4.2 Notions préliminaires pour le calcul à virgule flottante

En simulation native, les instructions FP générées lors de la compilation du code source vers du code binaire hôte correspondent aux comportements hôtes sans garantie qu'il y ait une relation avec les instructions FP cibles. Par conséquent nous n'utilisons pas la simulation native pour essayer de simuler précisément le comportement d'une FPU cible. Nous décidons d'utiliser la simulation compilée car elle présente les avantages d'être relativement facile à mettre en place et d'être une technique se basant sur le code binaire cible. Nous allons tout d'abord détailler la simulation compilée utilisée. Après cela des précisions sur la norme IEEE 754 sont fournies. Finalement notre proposition concernant la gestion des instructions FP est expliquée.



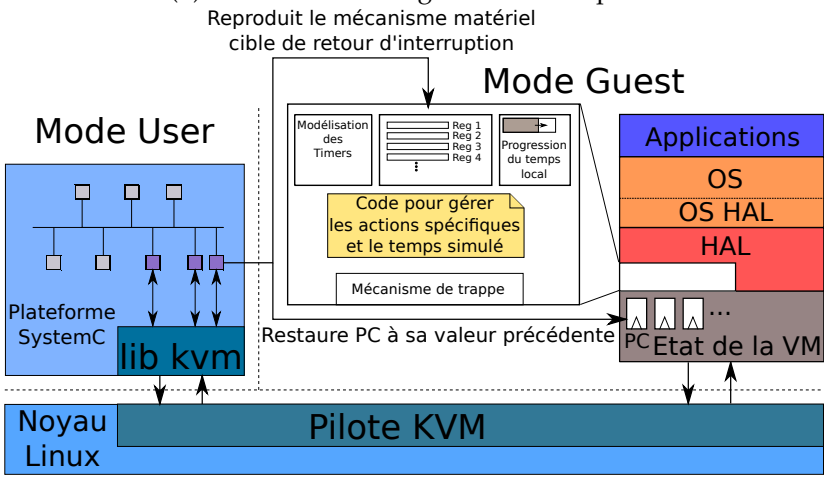
(a) Positionne les registres pour gérer l'interruption

```
inline HAL_sauvegarde_reg() {
    sauvegarde_reg_hote();
    sauvegarde_reg_cible();
}

inline HAL_retour_interruption() {
    restaure_reg_cible();
    restaure_reg_hote();
    io_trappe(RET_FROM_INT);
}
```

```
gestionnaire_interruption() {
    HAL_sauvegarde_reg();
    gère_raison_interrupt(check_reg_flags());
    HAL_retour_interruption();
}
```

(b) Code assembleur gérant l'interruption



(c) Restaure dans l'état précédent l'interruption

FIGURE 4.5 – Mécanisme de gestion d'une interruption

4.2.1 Présentation de la simulation compilée

La simulation compilée se divise en 3 parties :
— une phase de traduction qui prend en entrée un binaire cible et une description du comportement des instructions et génère du code haut niveau (du C dans notre cas),
— la compilation du code généré en une bibliothèque dynamique à l'aide d'un compilateur hôte (*gcc* dans notre cas),
— une phase d'exécution de l'application sur la plateforme de simulation.

La phase de traduction est d'abord présentée. Elle s'appuie sur une bibliothèque comportementale décrivant les instructions cibles. La phase de traduction est séparée en deux parties : une phase de désassemblage et construction du graphe d'exécution (*CFG - Control Flow Graph*) puis une phase de génération du code haut niveau représentant le comportement des instructions. La phase de compilation n'est pas détaillée car il s'agit simplement d'utiliser le compilateur hôte. L'environnement d'exécution de la simulation est finalement décrit.

La bibliothèque comportementale

La bibliothèque comportementale doit fournir 3 fonctionnalités. Tout d'abord elle doit permettre de décoder une instruction binaire, de la stocker sous une représentation de son choix, et d'indiquer les limites des paquets d'instructions (*bundles*). Ensuite elle doit être capable de fournir des informations compréhensibles à partir de sa représentation interne. Cela permet de savoir quand l'on rencontre une instruction de saut ou une boucle matérielle, et de connaître les paramètres d'une instruction assembleur (numéro de registre, valeur d'un immédiat). Il peut y avoir plusieurs API possibles. Une API simple est de retourner la représentation assembleur. Finalement elle doit être capable de générer le comportement d'une instruction.

Construction du CFG

Nous supposons que le binaire d'entrée est au format elf par simplicité d'écriture mais l'ensemble des étapes suivantes fonctionne pour d'autres formats tant que la contrainte de disposer statiquement de l'ensemble du code à simuler reste vrai. La phase de traduction se divise en plusieurs étapes illustrées par le listing 4.1.

La première consiste à récupérer dans le binaire l'ensemble des parties contenant du code à exécuter (*e.g.* section .text). A partir de là, on peut reconstruire la suite d'instructions du code binaire et former les paquets d'instructions (*bundles*) dans le cas d'un cœur VLIW (ligne 7 à 11).

Pour faciliter la construction du CFG, nous utilisons la table des symboles pour connaître l'adresse de début de chaque fonction. La liste des paquets d'instructions obtenue précédemment est parcourue pour trouver l'ensemble des ruptures de séquences possibles (ligne 13). Une rupture de séquences est un paquet d'instructions contenant une instruction de saut ou une destination d'instruction de saut. Le paquet d'instructions suivant une instruction de saut et les paquets d'instructions de destinations sont marqués comme point d'entrée d'un bloc de base (BB). Les sauts peuvent être divisés en 2 grands types différents : les sauts à une adresse connue (relative ou absolue) et les sauts indirects dont la valeur de destination va dépendre de l'exécution et être stockée dynamiquement dans un registre R_d. Pour la première catégorie, l'ensemble des adresses de destinations est facilement calculable par lecture de l'instruction de saut.

```
extrait_info
  (
    entrée : elf_name;
    sorties : liste_bundle, liste_point_d'entrée
  )
{
  elf = parse_elf(elf_name);
  pour chaque section exécutable de (elf) {
    bundles = recupere_les_bundles_des_sections();
    ajoute_en_queue(liste_bundle, bundles);
  }
  recupere_symboles_utiles(elf, liste_bundle);
  trouve_points_entrées(elf, liste_bundle, liste_point_d'entrée);
}

simulation_compilée
  (
    entrée: elf_name;
    sortie: génération code C
  )
{
  liste_bundle, liste_point_d'entrée;
  extrait_info(elf_name, liste_bundle, liste_point_d'entrée);
  tableau_bb = construit_les_basics_blocs
          (liste_point_d'entrée, liste_bundle);
  construit_graphe_des_transitions(tableau_bb);
  timing_info = calcul_temps_exécution_bb(tableau_bb);
  génere_code_x86(tableau_bb, timing_info);
}
```

Listing 4.1 – Génération du code C en simulation compilée

Dans le cas des sauts indirects registres, le calcul des adresses de destination est plus compliqué. Il faudrait pouvoir connaître l'ensemble des valeurs possibles du registre R_d. Deux éléments viennent nous aider. Tout d'abord l'ensemble des adresses de destinations possibles est stocké dans des tables de saut (*jump tables*). Il nous faut donc trouver ces tables pour connaître l'ensemble des adresses destinations possibles. De plus une table de sauts est stockée de façon continue. La fin de la table peut être déduite lorsque l'on commence à lire une adresse de destination non valide. Si le saut indirect n'est pas un *appel* indirect registre (mais un *saut* indirect registre) et s'il a été généré par le compilateur (et non manuellement dans du code assembleur), l'adresse de destination du saut est alors contenue dans la plage d'adresse de la fonction cible contenant le saut indirect.

De plus dans notre cadre applicatif, le cœur K1 du processeur MPPA®-256 de Kalray, la suite d'instructions amenant à un saut indirect est assez caractéristique. Cela nous permet de trouver assez facilement l'adresse de début de la table de saut. L'exemple présenté dans le listing 4.2 est un exemple type de la construction d'un saut indirect.

```
1 make $r2 = 202212;;
2 lw.add.x4 $r1 = $r1[$r2];;
3 igoto $r1;;
```

Listing 4.2 – Code assembleur type d'un saut indirect

Lorsque l'on trouve un saut indirect, le registre contenant l'adresse de destination du saut R_d est mémorisé. Ici, il s'agit de $r1$ (ligne 3). Les paquets d'instructions sont alors parcourus en sens inverse pour trouver la dernière affectation de R_d. R_d peut prendre directement un immédiat ce qui nous informe sur l'adresse de destination du saut. Plus généralement R_d reçoit la valeur d'une case d'un tableau, le tableau étant pointé par un registre nommé R_t. Dans notre exemple, $r1$ reçoit l'adresse contenue dans la case d'index $r1$ du tableau pointé par $r2$ (avec des cases de 4 bytes) (ligne 2). R_t est le registre contenant l'adresse de la table de sauts. On parcourt donc les instructions en sens inverse jusqu'à trouver la dernière affectation de R_t. Cela nous donne l'adresse de départ de la table de sauts. Dans notre exemple, $r2$ reçoit l'adresse d'une table de saut commençant en 0x315e4 (202212 en décimal, ligne 1).

Cependant il peut y avoir des cas où l'on n'arrive pas à trouver la table. En effet les affectations de R_d et R_t peuvent se trouver très loin avant ou après dans le code avec une série de sauts conduisant le cœur à l'exécution du saut indirect. En pratique ces cas sont très rares. De plus une redondance dans l'utilisation d'une table de sauts peut avoir lieu. Les points d'entrée des BB contenus dans une table de sauts peuvent avoir déjà été rencontrés comme point d'entrée à cause d'autres instructions de sauts.

Une fois que l'on a trouvé l'ensemble des points d'entrée des BB, les BB peuvent être formés (ligne 24 du listing 4.1). Le CFG est alors construit dans le but d'améliorer, durant la simulation, le temps de recherche du prochain BB à exécuter (ligne 26). Dans le but de connaître les temps de simulation, chaque BB est annoté d'un temps d'exécution en fonction de ses prédécesseurs (ligne 27). Finalement le code simulant le comportement des BB est généré (ligne 28).

Génération du comportement des BB

Pour générer le comportement d'un BB, il faut tout d'abord savoir générer le comportement d'une instruction. Pour cela nous utilisons la bibliothèque comportementale fournie en entrée. Le pipeline est découpé en trois grandes parties : *fetch*, *execute* et *commit*. Pour chacune de ces étapes, la bibliothèque contient l'ensemble des comportements possibles. Lors de la génération du code, nous générons les étapes nécessaires pour chaque instruction.

La deuxième étape est de savoir générer le comportement d'un paquet d'instructions. Comme les instructions parcourent les étages du pipeline en parallèle, nous appelons successivement les fonctions simulant le *fetch* pour l'ensemble des instructions du paquet d'instructions, puis les fonctions *execute* puis les fonctions *commit*. La simulation se comporte comme si les résultats du paquet d'instructions étaient disponibles immédiatement après l'exécution du paquet d'instructions : lors de l'exécution du paquet d'instructions suivant, les résultats sont disponibles et valides. Néanmoins les résultats peuvent ne pas être directement disponibles dans le vrai pipeline car plusieurs cycles sont nécessaires pour traiter intégralement le paquet d'instructions. Sur le vrai matériel, le cœur K1 détecte les dépendances entre 2 paquets d'instructions et attend que le résultat soit disponible avant de conti-

nuer l'exécution. Les problèmes de write-after-read et read-after-write n'existent donc pas dans le cas du K1. Ces problèmes de dépendance affectent uniquement le temps simulé. La bibliothèque comportementale prend en compte les dépendances lors de la génération du temps simulé d'un BB.

Le code généré pour un BB est regroupé en une fonction : `fct_bb_0xStartAddr_du_BB`. Pour faciliter l'exécution de la simulation, lors de la génération, nous générons une série de structures permettant de retrouver plus facilement les informations. Ainsi un BB est décrit de la façon suivante :

— un identifiant,
— une adresse cible de départ (l'adresse du premier paquet d'instructions du BB),
— une adresse cible de fin (l'adresse du dernier paquet d'instructions du BB),
— une structure d'annotations (contenant les informations temporelles, énergétiques, etc. utiles du BB),
— un tableau de successeurs,
— un pointeur de fonction vers la fonction simulant ce BB.

L'ensemble des descriptions de BB est regroupé dans un tableau servant de base à la simulation. L'identifiant d'un BB correspond à son indice dans ce tableau.

L'ensemble de ces fonctions est compilé à l'aide du processeur hôte. Le gain de vitesse de simulation fourni par la simulation compilée vient de la traduction des instructions qui est effectuée une seule fois et de la capacité du compilateur hôte à fournir du code binaire optimisé.

Environnement d'exécution de la simulation

Lorsque l'ordonnanceur SystemC élit un cœur pour être simulé, celui-ci s'exécute pendant un certain laps de temps de simulation (car le découplage temporel est utilisé). La simulation d'un cœur est effectuée par la fonction `sim_step` présentée dans le listing 4.3. Elle se caractérise par l'exécution d'une boucle qui simule un BB, met à jour les annotations, puis cherche le prochain BB à exécuter. La sortie de boucle arrive lorsque le cœur a utilisé tout son quantum, lorsque le cœur est passé en mode pause (*idle*) ou arrêté, ou lorsqu'une autre raison force une synchronisation du cœur avec la plateforme.

```
1  int sim_step(struct core_t *core) {
2    Rétablit_état_du_coeur(core);
3    /* Main loop */
4    while (!synchro_requise(core)) {
5      int nb_boucle = core->bb_info_table[bb_id].bb_fct(core);
6      mise_à_jour_annotation(core, bb_id, nb_boucle);
7      bb_id = recherche_bb_suivant(core, bb_id, K1_NPC);
8    }
9    Sauvegarde_état_du_coeur(core);
10   return cyclesAConsommer;
11 }
```

Listing 4.3 – Boucle basique de simulation

Deux cas particuliers sont notamment traités durant la simulation : les boucles matérielles et le CFG non complet (à cause des sauts indirects registres). Pour optimiser le trai-

tement de ces boucles, du code est rajouté spécialement pour optimiser leur simulation. Lorsqu'une telle boucle est détectée et que celle-ci ne s'étend que sur un BB (ce qui est très fréquemment le cas), la fonction simulant le BB contient elle-même une boucle qui permet de simuler la boucle matérielle sans avoir besoin de revenir à la boucle principale. Le nombre de tours de boucle est ensuite nécessaire pour mettre à jour correctement à partir des annotations la consommation de temps, d'énergie, le nombre d'instructions, etc.

Lors de la recherche du prochain BB, le tableau des successeurs est d'abord parcouru. Le tableau des successeurs contient des structures regroupant l'identifiant d'un BB et son adresse de départ en mémoire cible. Si le successeur n'est pas trouvé, la liste complète des BB est parcourue de façon dichotomique en testant l'adresse de départ de chaque BB par rapport à la prochaine valeur du pointeur d'instruction. Une erreur est levée si l'on ne trouve pas de BB commençant par cette adresse. Cela peut arriver si la table de sauts d'un saut indirect n'a pas été trouvée lors de l'étude statique des sauts.

Lors du passage d'un cœur à un autre, l'identifiant du prochain BB à exécuter est sauvegardé/restauré et les annotations collectées sont retournées pour permettre la progression de la simulation, et éventuellement stockées dans un fichier de trace pour permettre un post-traitement.

4.2.2 Compléments sur la norme IEEE 754

La norme IEEE 754 a déjà été introduite dans le chapitre 2. Nous allons maintenant présenter quelques notions supplémentaires concernant cette norme : les formats étendus, les exceptions et les valeurs spéciales de la norme.

Les formats étendus

Dans la section 2.4.1, nous avons présenté les formats simple et double précision pour un encodage binaire dont un rappel de l'encodage est donné dans la figure 4.6. Il existe aussi des formats étendus et des formats extensibles. Les formats étendus se caractérisent par une précision et des valeurs d'exposants supérieures ou égales à celles du type qu'ils étendent. Les formats extensibles fournissent la possibilité à l'utilisateur de choisir dynamiquement d'étendre ces valeurs.

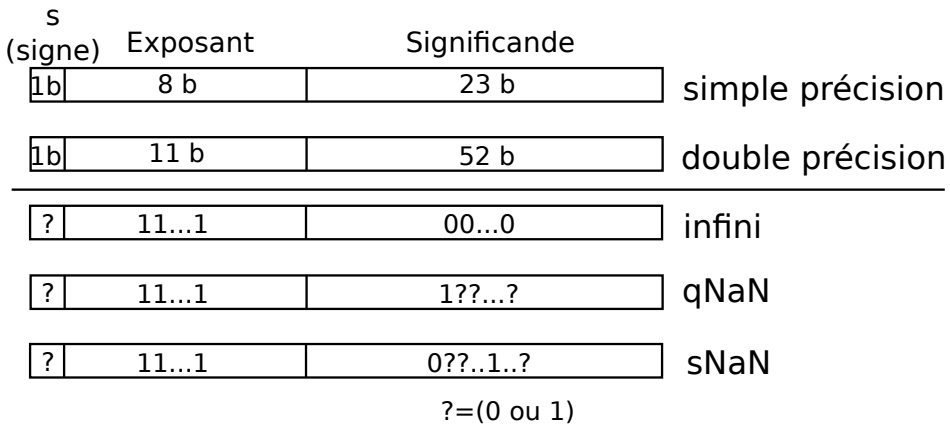


FIGURE 4.6 – Encodage des principaux formats FP et des valeurs spéciales de la norme

Exceptions

La norme définit 5 exceptions différentes. Le moment où les exceptions sont levées et leur comportement sont en partie définis par la norme et en partie par le langage dans lequel l'application est codée. Ces exceptions sont :

- Opération invalide : les opérandes ne sont pas valides pour l'opération à réaliser.
- Division par zéro : diviser un nombre fini non nul par zéro, ou log(0.0).
- Dépassement de capacité par le haut (*overflow*) : la valeur la plus grande du format de destination est inférieure à la valeur du résultat après un arrondi théorique avec la valeur de l'exposant non bornée.
- Dépassement de capacité par le bas (*underflow*) : un résultat est dit *tiny* s'il se trouve dans l'intervalle $]-2^{emin}; 2^{emin}[$. Cette détection peut être faite soit avant, soit après l'arrondi selon le choix du concepteur de l'opérateur. Pour que l'exception *dépassement de capacité par le bas* soit levée, il faut de plus que le résultat ne soit pas exact. L'exception *inexacte* est alors aussi levée.
- Inexacte : les résultats avant et après arrondi ne sont pas les mêmes.

Valeurs spéciales

Il existe 2 valeurs spéciales décrites dans la norme : $+/-\infty$ et *Not a Number (NaN)*. $+/-\infty$ est implémentée à l'aide de l'exposant maximal du format et un significande à 0. Les NaN sont divisés en 2 catégories : qNan (quiet NaN) et sNaN (signaling NaN). Les sNaN doivent lever l'exception *invalid operation* s'ils sont utilisés dans une opération sauf pour certaines opérations de conversion, alors que les qNaN ne lèvent pas cette exception. Leur encodage est montré dans la figure 4.6.

4.3 Simulation efficace d'une unité de calcul à virgule flottante

Nous allons maintenant voir comment améliorer la gestion des calculs à virgule flottante en utilisant la FPU hôte ou certaines fonctions des bibliothèques mathématiques hôtes.

Nous faisons la supposition que la machine hôte et la machine cible ont la même endianness et que leur FPU est conforme au standard IEEE 754. Malgré ces points communs, des différences peuvent exister entre les 2 implémentations. Nous proposons de séparer le support des instructions FPU cible à l'aide de la FPU hôte en 3 catégories comme illustré par la figure 4.7 :

(a) la machine cible et la machine hôte implémentent les mêmes instructions de la norme IEEE 754,

(b) la machine cible respecte la révision 2008 de la norme mais pas l'hôte,

(c) les instructions cibles ne sont pas spécifiées par la norme.

Nous allons maintenant détailler comment gérer chacune de ces catégories.

4.3.1 Association directe des instructions cibles et hôtes

Tout d'abord, les formats des instructions FP cibles doivent être supportés par la plateforme hôte sinon des problèmes de précision et d'arrondi apparaîtront dans les calculs. Une plateforme cible peut avoir défini des formats étendus conformes à la norme IEEE 754. Or ces formats peuvent ne pas être supportés par la machine hôte. Dans ce cas, la FPU hôte

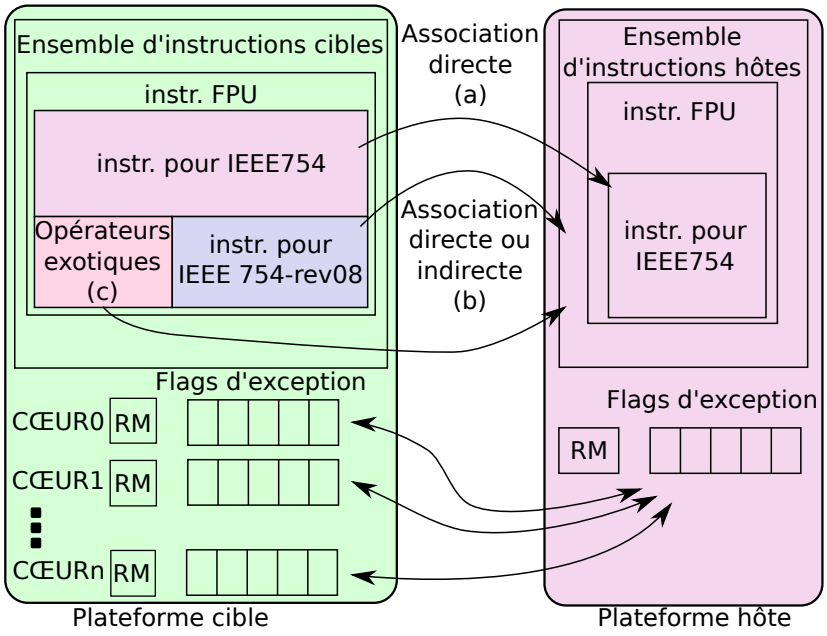


FIGURE 4.7 – Association des instructions FP, mode d'arrondi et flags d'exception de la cible sur l'hôte

ne peut pas être utilisée pour accélérer la simulation des calculs utilisant ce format étendu. Cependant comme les formats couramment utilisés sont la simple et la double précision, nous limitons notre étude à ces 2 formats.

Essayer de simuler un format cible avec un format hôte plus précis peut générer des erreurs dites de *double arrondi* quand un arrondi vers la précision du format cible est réalisé à partir du format de l'hôte [MDMM13]. Nous illustrons ce problème à l'aide de l'exemple suivant : $\circ_{p=3}(\circ_{p=6}(1.000100 + 1.000000 \times 2^{-7}))$, où $\circ_p$ représente l'opérateur d'arrondi au plus proche (avec arrondi à la mantisse paire s'il y a 2 nombres dans le format final à égale distance du nombre d'origine) vers un significande de $(p+1)$-bits :

$$\begin{aligned} \circ_{p=3}(1.000100 + 1.000000 \times 2^{-7}) &= 1.001(attendu) \\ \circ_{p=6}(1.000100 + 1.000000 \times 2^{-7}) &= 1.000100 \\ \circ_{p=3}(1.000100) &= 1.000 \neq 1.001(attendu) \end{aligned}$$

L'addition en précision infinie donne 1.0001001 mais le premier arrondi retire le bit de poids faible. Le deuxième arrondi est réalisé sur 1.000100 qui est à égale distance de 1.001 et de 1.000. Dans ce cas l'arrondi au plus près avec arrondi au chiffre pair force la mantisse à être paire, c'est-à-dire 000 dans notre cas. Le double arrondi de 1.0001001 donne 1.000 comme résultat au lieu de 1.001.

Pour les instructions ayant un format équivalent, il existe un sous-ensemble des instructions FP cible qui peuvent être simulées directement par des instructions hôtes. $fp32$ indique le format simple précision (codé avec 32 bits) et $fp64$ indique le format double précision (codé avec 64 bits). Si une multiplication $fp64 \times fp64 \rightarrow fp64$ existe dans les 2 architectures, l'étage *execute* du pipeline pour cette instruction cible peut être émulé à l'aide d'une instruction hôte.

Comme certaines parties de la norme laissent des choix architecturaux aux designers, l'association directe des instructions cibles vers celles de l'hôte ne garantit pas une complète

simulation numériquement équivalente (*e.g.* la représentation des NaN). Mais l'équivalence fonctionnelle des calculs peut encore être garantie en associant les bits de contrôle cibles et les flags d'exceptions aux bits de status hôte. Par exemple, la sélection du mode d'arrondi et les flags d'exceptions du K1 se trouvent dans le registre *Compute Status* (*CS*) alors qu'ils se trouvent dans le registre *MXCSR* pour le X86_64 ayant le composant *Streaming SIMD Extension (SSE)*. Concernant les exceptions, le comportement du dépassement de capacité par le bas (*underflow*) n'est pas complètement équivalent si la machine hôte et cible ne possèdent pas la même tininess. Par exemple, le X86_64 détecte le dépassement de capacité par le bas après l'arrondi alors que le K1 le détecte avant l'arrondi.

Une approche naïve va maintenir à jour les registres cibles et hôtes de status/contrôle de la FPU pour chaque instruction FP simulée, générant beaucoup de transferts entre les registres hôtes et les registres cibles simulés. Par exemple, l'instruction *stmxcsr* va écrire le registre *MXCSR* dans la mémoire et l'instruction *ldmxcsr* va écrire dans le registre *MXCSR* depuis une adresse mémoire. Mais exécuter ces instructions plusieurs fois par instruction de calcul flottant (pour mettre à jour le mode d'arrondi hôte, pour obtenir les exceptions levées lors de l'exécution de l'instruction) est coûteux. Par conséquent nous utilisons une politique de mise-à-jour paresseuse : une mise-à-jour est effectuée si et seulement si un accès aux registres cibles contenant le mode d'arrondi ou les flags d'exceptions est réalisé.

Les environnements de simulation sont généralement séquentiels donc différents cœurs cibles vont être simulés consécutivement sur le même cœur hôte. Ainsi différents modes d'arrondi et différents ensembles de flags d'exceptions (1 mode d'arrondi par cœur cible et 1 ensemble de flags par cœur cible) sont associés aux mêmes registres hôtes comme montré dans la figure 4.7. Chaque fois que la simulation élit un nouveau cœur à simuler, les registres du processeur hôte doivent être mis à jour avec le mode d'arrondi et les exceptions du cœur cible élu.

Pour la simulation par interprétation d'instructions ISS dans laquelle la simulation est réalisée cycle par cycle, la mise à jour du mode d'arrondi et des flags d'exception à chaque élection d'un cœur cible est contre-productif par rapport à une mise à jour seulement pendant la simulation d'instructions de calcul flottant. En effet, la simulation est réalisée cycle par cycle. Il y a donc plusieurs changements de cœurs simulés par cycle.

À cause de cette dépendance sur les registres, la mise-à-jour de façon paresseuse est intéressante uniquement pour les techniques de simulation rapide dans lesquelles les cœurs cibles sont simulées pendant une tranche de temps suffisante (*i.e.* utilisant un quantum keeper dans le langage SystemC).

4.3.2 Association non-directe des instructions : les opérateurs définis dans la norme IEEE 754-rev08

La plateforme hôte peut avoir un support partiel ou une absence de support des opérateurs introduit par la norme IEEE 754-rev08, mais la machine cible peut les avoir implémentés. Par exemple, les cœurs K1 implémentent le FMA simple précision (présenté dans la section 2.5.2) alors que le X86_64 est conforme à la norme originale mais n'implémente l'opérateur FMA que depuis la génération Haswell. Par conséquent nous proposons 2 méthodes pour émuler ces instructions sur les anciens processeurs X86_64.

Décomposition basique

La décomposition basique du FMA est simple et rapide mais ne garantit pas l'équivalence stricte des calculs. Le FMA simple précision est séparé en une suite d'opérations

simples supportées par l'hôte : conversion des entrées simples précisions en double précisions, une multiplication double précision suivie par une addition double précision, et finalement une conversion double précision vers simple précision. L'implémentation contient seulement 4 opérations FP. La multiplication ne génère pas d'erreur d'arrondi. Pour garantir cette propriété, un significande de p-bit nécessite un format avec au moins un significande de $2p$-bit. Néanmoins, cette méthode est moins précise que l'opérateur de base car elle souffre du problème de double arrondi après l'addition.

Méthode par propagation d'erreur

Pour éviter le problème de double arrondi, nous proposons une méthode utilisant une addition exacte et un test sur l'erreur. Nous émulons le FMA comme précédemment, mais nous calculons aussi l'erreur dûe à l'émulation et nous l'utilisons pour calculer le résultat final. La sortie de cette implémentation est bit à bit égale à la spécification du FMA. L'algorithme détaillé est présenté dans le tableau 4.1.

TABLE 4.1 – Algorithme de propagation d'erreur pour le calcul d'un FMA

	a, b et c, des nombres $fp32$ Calcul de $\circ_{fp32}(a \times b + c)$
1.	$A = \circ_{fp64}(a)$, $B = \circ_{fp64}(b)$, $C = \circ_{fp64}(c)$
2.	$M = \circ_{fp64}(A \times B)$
3.	$S, T =$ **2Sum**(M, C)
4.	$r = \circ_{fp32}(S)$
5.	Si **test_arrondi**(r, S), alors **mise_à_jour**(r, T)

Comme les tailles des significandes et des exposants de $fp64$ sont plus grands que $fp32$, les conversions dans (1) sont exactes, il n'y a pas d'erreur d'arrondi. Comme la taille du significande de $fp64$ est plus de 2 fois celle du significande de $fp32$ et comme l'exposant de $fp64$ est suffisament grand pour contenir l'exposant des 2 produits, (2) est exact et $M = a \times b$. L'algorithme **2Sum** [Knu69, KLLM12] fournit la somme exacte de 2 nombres FP, M et C, sous la forme de 2 nombres FP, S et T, de même précision et ne se superposant pas : $S + T = M + C$ avec $S = \circ_{fp64}(M + C) = \circ_{fp64}(a \times b + c)$.

Le reste de l'algorithme détermine si r a été arrondi de la bonne manière, *i.e.* vérifier que : $r = \circ_{fp32}(a \times b + c)$, et corriger r si cela est nécessaire : si $r = \circ_{fp32}(S) = \circ_{fp32}(\circ_{fp64}(a \times b + c))$ est différent de $\circ_{fp32}(a \times b + c)$. Le listing 4.4 illustre les explications des tests effectués sur r. **test_arrondi** étudie $E = \|S - r\|$. Si $E = 2^{e_s - p}$, où p est la taille du significande final (simple précision dans ce cas) et e_s est l'exposant sans le biais de S, alors S est un point intermédiaire : un nombre à mi-distance entre 2 nombres $fp32$ comme montré ci-dessous (avec p=23).

			significande					
S=	x	...	x	1	0	...	0	$\times 2^{e_s}$
bit	52		$52 - p + 1 = 30$	$52 - p = 29$			0	

Dans un tel cas, une erreur de double arrondi peut avoir lieu si l'arrondi choisi est l'arrondi vers le nombre à virgule flottante le plus proche. Dans le cas où l'arrondi est vers le haut, vers le bas, ou vers 0, il n'y a pas d'erreur d'arrondi dans le calcul de r quelque soit la valeur de S.

```
bool test_arrondi(r, S) {
  return ((E = ||S − r||) == 2^(e_S − p));
}

void mise_à_jour(r, T) {
  if (T == 0) {
    /* déjà correctement arrondi */
    return;
  }
  if (S et T ont le meme signe) {
      mettre à jour r en s'éloignant de 0
  } else {
      mettre à jour r en se rapprochant de 0
  }
}
```

Listing 4.4 – Si **test_arrondi**(r, S, T), alors **mise_à_jour**(r, T)

Dans le cas où l'arrondi est au plus proche et S est un point intermédiaire, la valeur de T indique s'il y a eu une erreur d'arrondi et la correction qu'il faut apporter à r. Si $T = 0$, alors r a déjà été correctement arrondi. Si S et T ont le même signe, alors $r = \circ_{fp32}(S)$ doit être arrondi en s'éloignant de 0 (vers $+\infty$ si positif, vers $-\infty$ quand négatif), dans le cas inverse r doit être arrondi vers 0. La fonction **mise_à_jour** fournit cet arrondi correct.

La solution que nous proposons ne peut pas ête utilisée pour émuler un FMA double précision. En effet la plupart des machines hôtes n'implémentent pas un format avec 2 fois plus de précision que binary64. Même la précision étendue du x86_64 n'est pas suffisante (binary80). Or cela est nécessaire pour réaliser la multiplication sans perdre d'information. Pour réaliser l'émulation du FMA dans la précision d'origine, une solution basée sur l'arrondi au nombre impair existe [BM08]. Mais ce mode d'arrondi n'est pas inclu dans la norme IEEE 754 et il n'est pas supporté en matériel. Nous devons alors utiliser une émulation logicielle de l'arrondi au nombre impair ce qui ralentit significativement son exécution.

Preuve de l'absence d'erreur de double arrondi dans la calcul de r pour l'arrondi vers le haut, le bas et vers zéro

Nous allons montrer que $r = \circ_{fp32}(a * b + c)$ avec l'arrondi vers le haut quelque soit les valeurs de a, b et c sans avoir besoin d'utiliser **test_arrondi**(r, S, T) et **mise_à_jour**(r, T).

Il suffit de montrer que $r = \circ_{fp32}(S + T)$ car $S + T = M + C$, donc $\circ_{fp32}(S + T) = \circ_{fp32}(M + C) = \circ_{fp32}(a * b + c)$. Rappelons que par construction, $S = \circ_{fp64}(M + C)$ et $r = \circ_{fp32}(S)$.

Soit H et L deux suites de bits (donc nombre toujours positifs) permettant la décomposition du significante de S tel que :
$S = (-1)^s \times (1.H \times 2^{e_S} + 0.L \times 2^{e_S - t_H})$
avec $t_H = taille(H) = taille\_significande(fp32) = 23b$.
Pour simplifier, on se place dans le cas S positif ($s = 0$).

Si $L \neq 0$, $r = \circ_{fp32}(S) = 1.H \times 2^{e_S} + 1 \times 2^{e_S - t_H}$.
car l'arrondi est vers le haut.
Or S et T ne se superposent pas, $T < S$,
et la précision de S est supérieure à la précision $fp32$.
Donc T n'influence pas l'arrondi de $\circ_{fp32}(S + T)$.
Donc $r = \circ_{fp32}(S + T)$

Si $L = 0$, $r = \circ_{fp32}(S) = 1.H \times 2^{e_S}$.
si $T = 0$, $\circ_{fp32}(S + T) = 1.H \times 2^{e_S} = r$.
si $T < 0$, $\circ_{fp32}(S + T) = 1.H \times 2^{e_S} = r$
car S et T ne se superposent pas et l'arrondi est vers le haut.
si $T > 0$,

$$
\begin{aligned}
\circ_{fp32}(S) &= \circ_{fp32}(\circ_{fp64}(M + C)) \text{ car } S = \circ_{fp64}(M + C) \\
&= \circ_{fp32}(\circ_{fp64}(S + T)) \text{ car } S + T = M + C \\
&= \circ_{fp32}(S + (1 \times 2^{e_S - taille\_significande(fp64)}))) \\
&\quad \text{car } T > 0, \text{ l'arrondi est vers le haut} \\
&\quad \text{et S et T ne superposent pas.} \\
&= 1.H \times 2^{e_S} + 1 \times 2^{e_S - t_H} \text{ car arrondi vers le haut.} \\
&\neq 1.H \times 2^{e_S} \text{ donc erreur.}
\end{aligned}
$$

donc T ne peut pas être positif si $L = 0$.

Un résultat équivalent peut être montré avec $s = 1$ ainsi que pour les nombres sous-normaux avec $S = 0.H \times 2^{e_S} + 0.L \times 2^{e_S - t_H}$. Par un raisonnement similaire, on en déduit qu'il n'y a pas d'erreur de double arrondi si l'arrondi est vers le bas ou vers 0.

4.3.3 Association non-directe des instructions : les opérateurs exotiques

Les opérateurs exotiques sont des opérateurs qui ne sont pas définis dans la norme mais qui sont implémentés dans des instructions du processeur cible. Ils sont utilisés pour simplifier ou améliorer l'implémentation de certaines fonctions FP de base, ou optimiser le processeur pour un algorithme précis ou une classe d'algorithmes.

Les instructions FP du K1 comprennent des instructions servant à générer des nombres *racines* (*seed*) utilisés dans 3 primitives : la division, l'inverse, et l'inverse de la racine carrée comme détaillé dans la thèse de Brunie [Bru14]. Ces primitives logicielles sont implémentées dans la *libgcc* et dans la *libm* en utilisant la méthode de Newton-Raphson. Cette méthode nécessite une valeur initiale fournie par les instructions génératrices de nombres racines. Nous proposons de remplacer les fonctions de base qui utilisent un opérateur exotique par les fonctions hôtes équivalentes.

Ces fonctions font partie de bibliothèques bien connues. Leurs caractéristiques sont spécifiées soit par la norme IEEE 754 (*e.g.* la division), soit par la bibliothèque (*e.g.* l'inverse de la racine carrée). Par conséquent les applications utilisant ces bibliothèques ne doivent pas reposer sur une connaissance spécifique de l'architecture mais seulement sur les caractérisques spécifiées des bibliothèques et de la norme, qui sont les mêmes pour la machine cible et hôte.

Pour la SBT et la simulation compilée, il est assez simple de réaliser ces substitutions si les informations de débogage ont été inclues dans le binaire cible. Cela peut être fait statiquement en parcourant la table des symboles pour récupérer l'adresse des fonctions à remplacer dans le binaire cible et les remplacer lors de la phase de traduction par la fonction hôte associée. Pour la DBT, un test doit être réalisé pour chaque nouvelle traduction d'un bloc pour vérifier s'il correspond au début d'une des fonctions à remplacer. En simulation

native, le binaire peut être lié avec la libm hôte ou une libm personnalisée spécialement compilée pour la simulation (*e.g.* avec des annotations ajoutées à l'intérieur des fonctions).

4.4 Conclusion

La simulation native basée sur le HAV sépare nettement la simulation de la plateforme et la simulation des cœurs. De plus la dépendance du comportement de la simulation par rapport à la plateforme hôte est accrue. Nous avons proposé de faciliter le partage d'informations entre la plateforme virtuelle cible (simulée dans l'espace utilisateur) et la simulation des cœurs (espace guest) en ajoutant un espace mémoire d'échange en espace guest. Cet espace va contenir les registres des cœurs et des informations sur des composants étroitements couplés à un cœur tel un timer. Le but premier de cet espace est de minimiser le nombre d'aller-retour entre l'espace utilisateur et l'espace guest qui consomme un nombre de cycles non négligeable en utilisant le mécanisme de *trappe_io*. Cet espace mémoire facilite aussi la simulation avec le HAV de certains mécanismes matériels tels les interruptions. Comme une couche d'abstraction du matériel (HAL) est utilisée, cette extension mémoire est invisible pour le développeur.

Le deuxième point que nous développons est la simulation fonctionnellement exacte d'une FPU. La simulation native ne peut pas répondre à ce problème car elle n'a pas la notion des instructions FP cibles lors de la génération du code binaire hôte. Nous utilisons donc une technique de simulation compilée. Les instructions FP cibles sont regroupées en 3 domaines. Les instructions cibles qui correspondent à des opérations présentes sur le processeur cible et hôte, peuvent être directement simulées par les instructions hôtes équivalentes. Des algorithmes accélérants la simulation des instructions cibles n'ayant pas d'équivalent hôte sont souhaitables pour continuer à garantir de bonnes performances de simulation. Deux algorithmes pour l'instruction FMA simple précision sont proposés. Finalement nous proposons de remplacer des fonctions mathématiques cibles par leur équivalent hôte dans le cas où l'implémentation cible utilise des instructions exotiques. La bibliothèque mathématique garantit normalement le même comportement sur le cœur cible et hôte.

Chapitre 5

Expériences pour la simulation fonctionnelle

Nous avons vu dans le chapitre précédent une méthode pour gérer efficacement les caractéristiques architecturales particulières des processeurs cibles en simulation native basée sur le HAV. Nous avons aussi présenté la simulation compilée et une simulation optimisée de la FPU cible. Dans ce chapitre, différentes expériences sont réalisées pour voir l'effet de nos propositions sur la simulation.

Tout d'abord, une évaluation de la simulation native proposée est effectuée. Elle comprend notamment les politiques de gestion des caractéristiques architecturales spécifiques. L'impact de notre simulation de la FPU sur la simulation d'une application est ensuite présenté accompagné de compléments d'explications sur notre implémentation de la simulation compilée.

5.1 Plateforme cible et benchmarks

La plateforme cible modélisée pour les expériences est le MPPA®-256 de Kalray. Comme vu précédemment, il est composé de 256 cœurs VLIW spécifiques dédiés aux traitements des données (Processing Engine (PE)) et de 32 cœurs servant à la gestion de la plate-forme (Ressource Manager (RM)). Les cœurs sont regroupés par grappe de calcul de 16 PE et de 1 Ressource Manager (RM). D'autres grappes gérant les entrées/sorties sont constituées de 4 RM. L'ensemble de ces grappes est interconnecté par 2 NoC. Le MPPA®-256 est composé de 16 grappes de calcul et de 4 grappes gérant les entrées/sorties.

Les grappes de calcul possèdent 2 Mo de mémoire partagée, sans cohérence de cache. Pour les expériences, la mémoire partagée est augmentée à 2 Go par grappe afin que les *benchmarks* parallèles usuels puissent être mis en œuvre sans modification. Cette modification n'a pas d'impact sur les comparaisons entre la simulation native et le simulateur de référence (un ISS).

Les différents simulateurs utilisent les mêmes modèles pour les composants de la plateforme, seul le composant modélisant les cœurs est différent. Il s'agit soit d'un ISS, soit d'un wrapper de la simulation native, soit d'un wrapper de la simulation compilée.

Les applications utilisées proviennent de la suite de tests SPLASH-2, présentée par Woo et al. [WOT+95]. Les applications ayant un temps de simulation beaucoup trop long en ISS ne sont pas conservées. Le nombre de tâches dans les applications simulées est identique au nombre de PE instanciés dans une grappe.

5.2 Evaluation de la simulation native basée sur le HAV dans le cas d'un système many-cœurs

L'évaluation de notre technique de simulation native est faite dans 3 configurations différentes. Nous allons d'abord observer le comportement de la simulation native par rapport à la politique de trappe vue dans le chapitre 4.1 uniquement au niveau d'une grappe de calcul. Une comparaison avec l'exécution native est ensuite réalisée et finalement une simulation sur tout le MPPA® est effectuée.

Comme vu dans la problématique, la simulation native basée sur le HAV pose des problèmes de passage à l'échelle. Cela sera abordé théoriquement et expérimentalement dans le chapitre 6.

Pour comparer uniquement les performances de simulation sans tenir compte du surcoût dû à l'instanciation et à la destruction des composants, le temps d'exécution est mesuré entre l'étape `end_of_elaboration` de la plateforme SystemC, qui indique le début de la simulation, et la fin de `sc_start`, qui en indique l'achèvement. Les simulations mono-grappes sont réalisées sur un Intel® Core™ i7-3820 CPU à 3.60 GHz avec la version 3.3.4 de Linux. Les simulations multi-grappes sont réalisées sur une machine ayant 24 cœurs, Intel® Xeon® CPU à 2.67 GHz, avec la version 3.2.51 de Linux.

5.2.1 Politique de trappe

Cette expérience compare la simulation des mêmes applications utilisant 2 approches différentes pour la simulation native. Dans le premier cas, *trappe toujours*, des trappes sont toujours générées pour simuler des caractéristiques présentes seulement sur le processeur cible et donc simulées dans la plateforme en mode utilisateur. Dans le deuxième cas, *trappes améliorées*, des trappes sont générées uniquement si la caractéristique ne peut pas être simulée en mode *guest* comme expliqué dans la partie 4.1.3.

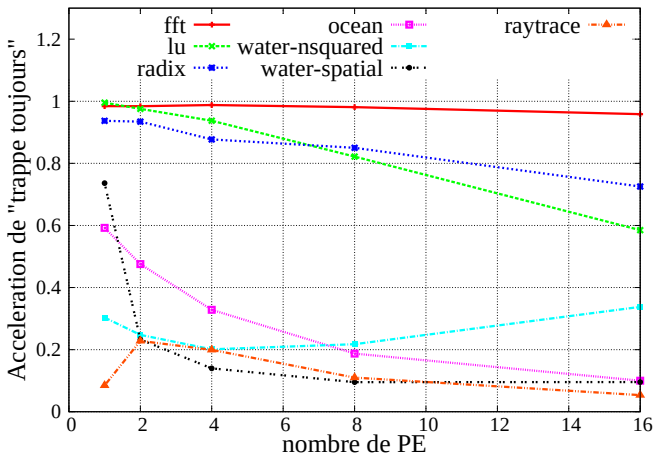


FIGURE 5.1 – Accélération de la simulation avec la politique *trappe toujours* par rapport à la politique *trappes améliorées*

Les caractéristiques pour lesquelles les trappes sont ou ne sont pas générées sont dans notre cas utilisées par le code du système d'exploitation cible. Il s'agit principalement des

timers et de builtins comme *count trailing zero* ou *get/set register*. Les résultats sont montrés dans la figure 5.1.

La diminution du nombre de trappes donne toujours de meilleurs résultats. De plus, on peut constater que dans la plupart des cas, la politique *trappe toujours* passe moins bien à l'échelle que les *trappes améliorées*. Dans l'ensemble des autres expériences en simulation native, seule la politique *trappes améliorées* est utilisée.

5.2.2 Comparaison avec l'exécution native

Nous comparons maintenant la simulation native par rapport à l'exécution native pour voir le ralentissement de l'application dû à la simulation. L'exécution native correspond à l'exécution de l'application au-dessus du système d'exploitation hôte, l'application ayant été directement compilée pour la machine et le système d'exploitation hôte. Cela nous renseigne sur la vitesse maximale d'exécution des applications. En effet, il n'y a pas de simulation de la plateforme et l'application dispose d'un vrai parallélisme pour les tâches des applications.

La figure 5.2 montre l'accélération de l'exécution native par rapport à la simulation native. Nous observons sans surprise que l'exécution native est plus rapide que la simulation native. Dans le pire cas, il y a 2 ordres de grandeur entre l'exécution et la simulation native (pire cas qui aurait tendance à augmenter si nous augmentions encore le nombre de PE simulés). Un point positif est la limitation de l'accélération à environ 20 pour *FFT*, *LU* et *ocean*, mais aussi pour les autres applications quand le nombre de PE reste faible. Quand le nombre de PE (et donc de tâches) augmente, une accélération presque linéaire est visible (particulièrement pour *raytrace*) car l'exécution native dispose d'un vrai parallélisme contrairement à la simulation native. Cela montre que notre technique de simulation permet une bonne vitesse de simulation tout en modélisant des caractéristiques architecturales spécifiques.

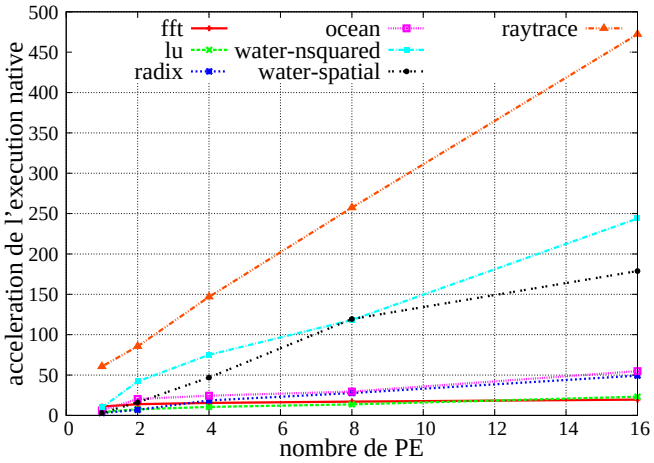


FIGURE 5.2 – Accélération de l'éxécution native par rapport à la simulation native sur une grappe

Il n'est pas possible de comparer notre technique à d'autres techniques de simulation native car la quasi intégralité du code cible est simulée dans notre cas, y compris le système d'exploitation au-dessus du HAL, or les autres travaux n'ont pas fonctionnellement cette capacité. Puisque ce travail étend celui de Hamayun [SHP12] à des systèmes many-cœurs

ayant des caractéristiques architecturales spécifiques, il n'est pas non plus possible de s'y comparer.

5.2.3 Tests multi-grappes

Pour les expériences multi-grappes, le MPPA®-256 est entièrement simulé. La comparaison des différentes techniques de simulation (*trappe toujours* et *trappes améliorées*) n'est pas présentée car la différence de temps de simulation est complètement masquée par le temps de transfert des binaires entre les grappes.

Une grappe d'entrées/sorties copie le binaire de l'application sur les 16 grappes de calcul à travers le NoC. Chaque grappe de calcul va exécuter le même code puis indiquer l'achèvement de sa tâche à la grappe d'entrées/sorties. Le nombre de PE par grappe de calcul varie de 1 à 16, le nombre total de PE va donc varier de 16 à 256. 1 RM est toujours utilisé pour chaque grappe de calcul. Chaque grappe est simulée dans un processus hôte différent et instancie sa propre VMM. La communication entre les grappes est modélisée grâce à la même technique dans les 2 plateformes (plateforme basée sur l'ISS ou basée sur la simulation native). Nous nous reposons sur des travaux déjà existants pour garantir le passage à l'échelle du nombre de grappes et le parallélisme de la simulation entre les grappes [MMGP10, WSL+14]. Les grappes sont toutes simulées en parallèle sur les 24 cœurs de la machine hôte. Les résultats de ces expériences sont montrés dans les figures 5.3 et 5.4.

raytrace n'est pas utilisé car un fichier d'entrée est nécessaire. Cela augmente grandement les transferts de données à travers le NoC. Le temps de simulation est alors constitué presque exclusivement de transferts NoC et l'impact de la simulation native ne serait plus visible. Cela est déjà presque le cas pour les autres applications.

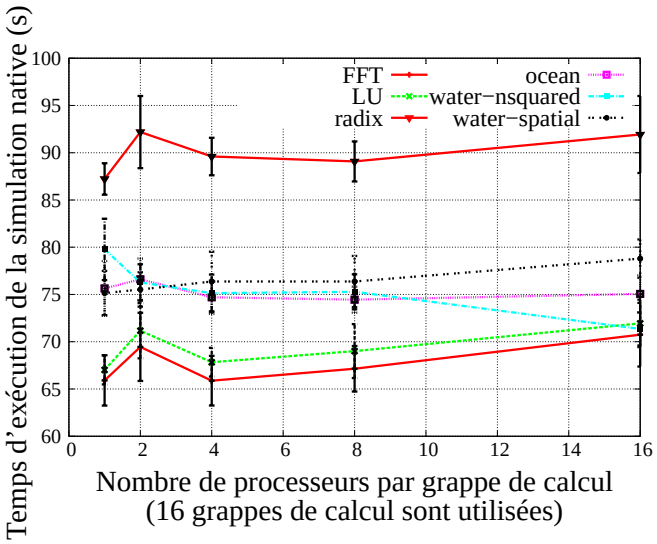


FIGURE 5.3 – Temps d'exécution de la simulation native avec la plateforme MPPA®-256

Une grande quantité de temps est perdue lors du transfert des binaires de la grappe d'entrées/sorties jusque dans chaque grappe de calcul ce qui ralentit beaucoup la simulation native. Pour information, le temps de simulation en natif des applications dans une grappe de calcul est d'environ 1 seconde sur la machine hôte utilisée. Mais le processus de copie du binaire utilise le composant *Direct Memory Access (DMA)* qui se charge d'envoyer le binaire à travers le NoC. Le composant DMA est le même pour les 2 plateformes et est

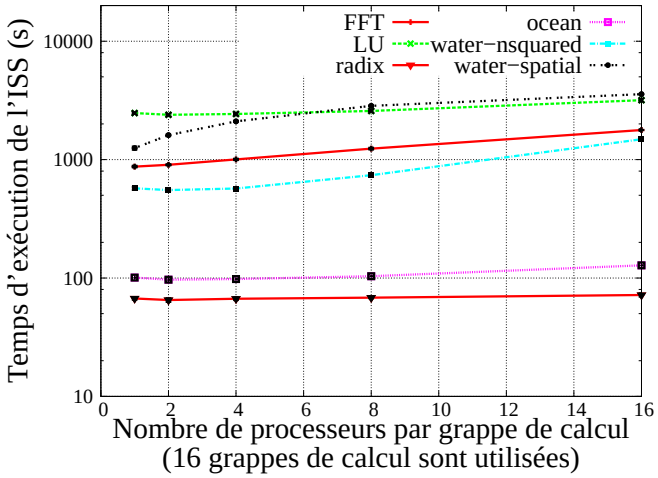


FIGURE 5.4 – Temps d'exécution de la simulation par ISS avec la plateforme MPPA®-256

simulé très précisément, c.-à-d. au cycle prêt quand des échanges de données doivent être simulés. De plus la taille des binaires natifs est supérieure à celle du binaire cible comme montré dans le tableau 5.1. Le temps de transfert est donc plus grand en natif.

	FFT	LU	ocean	radix	water-nsquared	water-spatial
taille des binaires cibles	4,0 Mo	4,0 Mo	4,2 Mo	4,3 Mo	4,2 Mo	4,2 Mo
taille des binaires natifs	5,3 Mo	5,3 Mo	5,4 Mo	5,6 Mo	5,4 Mo	5,4 Mo

TABLE 5.1 – Comparaison de la taille des binaires (applications compilées pour 16 PE)

Ainsi quand on regarde l'accélération de la simulation montrée par la figure 5.5, on constate que *radix* est légèrement plus lent en simulation native qu'en ISS (accélération d'environ 0,75) et *ocean* a une accélération très faible (environ 1,4). Le temps de simulation du binaire en ISS n'est pas assez grand pour compenser le temps supplémentaire nécessaire en natif pour transférer le binaire.

Le temps d'exécution des applications sur une grappe est négligeable par rapport au temps de transfert du binaire. De plus la variation du temps de simulation native des applications (*FFT*, *LU*, *water-nsquared* et *water-spatial*) n'est pas très élevée par rapport au temps global de la simulation native (environ 5 secondes de variation). Finalement la variation du temps de simulation native des applications est très faible par rapport aux variations de la simulation ISS (environ 5 secondes contre quelques centaines à quelques milliers de secondes). Ainsi l'amélioration de l'accélération quand le nombre de PE augmente ne dépend presque que de l'augmentation du temps de simulation ISS des applications. La simulation native est ainsi plus performante quand le nombre de PE augmente à l'échelle du MPPA®-256.

Cela montre qu'une technique de simulation rapide n'a d'intérêt que lorsque tous les composants sont au même niveau de vitesse de simulation. Les gains obtenus deviennent

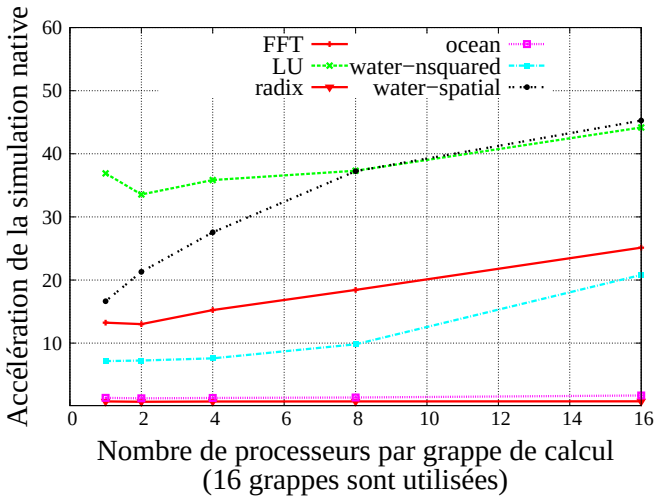


FIGURE 5.5 – Accélération de la simulation native par rapport à l'ISS pour l'ensemble du MPPA®-256

négligeables à partir du moment où un composant (ici le DMA) est simulé précisément à vitesse plus lente que le reste de la simulation.

5.3 Expériences liées à la simulation des calculs à virgule flottante en simulation compilée

La simulation compilée nous sert de support pour notre proposition concernant la simulation fonctionnellement précise d'une FPU cible. Tout d'abord quelques détails d'implémentation vont être donnés pour pouvoir mieux comprendre les expériences réalisées. Les expériences seront ensuite présentées.

5.3.1 Détails d'implémentation de la simulation compilée

La figure 5.6 rappelle le fonctionnement de la simulation compilée. Les propositions faites dans la section 4.3 se traduisent par des modifications de la simulation compilée mises en gras et rouge sur le schéma. Nous allons tout d'abord voir les modifications de la bibliothèque de gestion des calculs flottants. Le remplacement des fonctions cibles par leurs fonctions équivalentes hôtes est détaillé ensuite.

Association direct des instructions FP hôtes et cibles

Pour appliquer une projection directe des instructions cibles sur des instructions hôtes, la génération de code C n'a pas été modifiée. Les modifications ont eu lieu dans la bibliothèque `libFP` permettant de gérer les calculs à virgule flottante. Pour illustrer nos propos, nous utilisons l'exemple d'une addition double précision comme présenté dans le listing 5.1 ligne 1. Le code généré pour simuler son comportement est représenté à la suite de l'instruction. La partie qui nous intéresse pour la simulation des calculs à virgule flottante correspond à la fonction se trouvant ligne 8. Il s'agit de la fonction reproduisant l'étage *execute* du pipeline. La figure 5.7 illustre la suite des appels nécessaires pour reproduire le comportement de

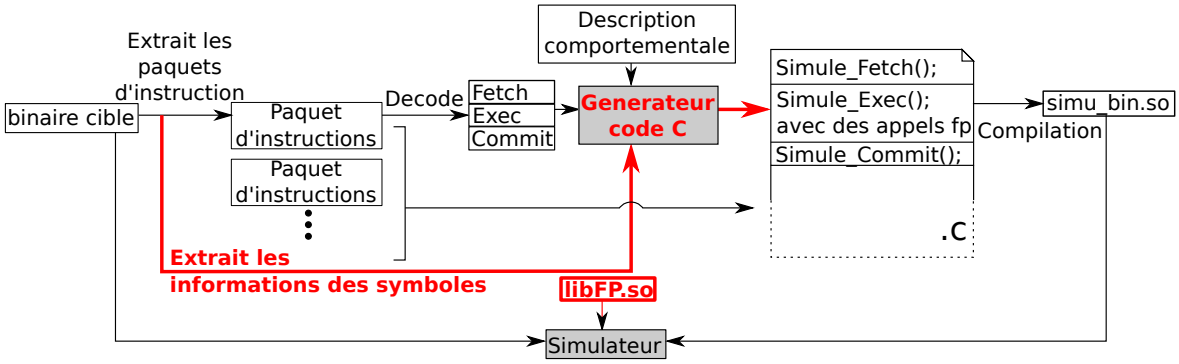


FIGURE 5.6 – Schéma de la simulation compilée avec gestion des calculs flottants

cet étage *execute*. Nous allons parcourir l'ensemble de ces appels jusqu'à l'implémentation finale pour montrer la différence de gestion des calculs flottants.

En fonction de la technique utilisée (ISS ou simulation compilée), certaines fonctions sont expansées en place (inlining) pour des raisons de performance ou de génie logiciel. Cela a un impact sur notre capacité à mesurer le temps passé dans certains groupes de fonctions comme expliqué dans le protocole de mesure section 5.3.2.

```
1  faddd $r0r1 = $r2r3, $r0r1;;
2
3  InstrOperand_ operands13[Operands__MAXOPERANDS];
4  const OperandDecoded opDecoded13[Operands__MAXOPERANDS] =
5      {172,173,172,0,0,0,0,0,};
6  fetch_k1dp_FDMA_registerW_registerQ_registerP
7      (core, operands13, opDecoded13, 1);
8  execute_k1dp_FADDD_registerM_registerQ_registerP
9      (core, operands13, opDecoded13, 1);
10 commit_k1dp_FDMAWD_registerM_registerQ_registerP
11     (core, operands13, opDecoded13, 1);
```

Listing 5.1 – Instruction K1 et code C généré pour simuler cette instruction en simulation compilée

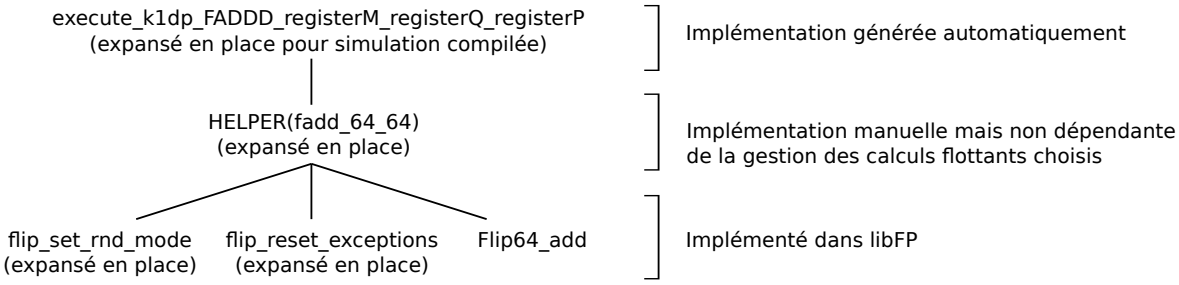


FIGURE 5.7 – Arbre d'appels pour émuler l'étage *execute* d'une addition double précision

```
1  static inline void execute_k1dp_FADDD_registerM_registerQ_registerP
2    (core_t *core, InstrOperand_ *this,
3      const OperandDecoded *decoded, Processor processor)
4  {
5    LECTURE DES ENTRÉES STOCKÉES DANS: argument1, argument2, RM
6    result1 = HELPER(fadd_64_64)(RM,argument1, argument2);
7    LECTURE/ECRITURE DES EXCEPTIONS inexact,invalid,overflow DANS CS
8    HELPER(writeOperand)(core, this, 0,
9        Int128_toUInt64(Int128_zx(result1, 64))
10   );
11 }
```

Listing 5.2 – Comportement de l'étage execute pour une addition double précision

La première fonction (fonction *execute*) est générée automatiquement à partir d'une description comportementale des instructions. Une version simplifiée de cette fonction est présentée dans le listing 5.2. Les éléments importants sont les fonctions de conversion d'une représentation à une autre (en grande partie cachées dans le pseudo-code) et l'appel à un *helper* pour faire la "vraie" addition (ici `fadd_64_64` ligne 6). Un helper est une fonction qui n'est pas générée automatiquement à partir d'une description architecturale et qui est donc définie manuellement. La gestion du mode d'arrondi et des exceptions est contrôlée par des variables globales dans la bibliothèque spécialisée dans la gestion des calculs à virgule flottante. Cet helper se contente principalement de mettre à la bonne valeur ces variables globales et à rediriger l'appel vers la fonction faisant réellement le calcul.

Finalement le listing 5.3 présente l'implémentation dans la bibliothèque de gestion des calculs flottants de la fonction de calcul utilisée par l'exemple. Il s'agit ici de l'émulation logicielle basée sur les entiers. La gestion des exceptions devant être faite manuellement, plusieurs tests sont effectués pour gérer ces cas. Des sous-fonctions sont ensuite appelées pour effectuer le cœur des calculs. Cela reflète la grande quantité d'instructions hôtes à exécuter pour effectuer une instruction de calcul FP cible.

D'un autre coté, le listing 5.4 présente la même fonction mais avec l'association de l'instruction cible sur la FPU hôte. On remarque la simplicité de la fonction `Flip64_add` constituée d'une simple addition, le matériel se charge d'effectuer tout ce qui était précédemment implémenté en logiciel. Les paramètres de la fonction sont stockés dans les registres des cœurs simulés et sont donc récupérés en tant qu'entiers même si, sémantiquement, ils représentent un nombre FP. Pour permettre au compilateur de générer la bonne instruction assembleur, une conversion directe de la représentation entière à la représentation flottante est réalisée grâce à l'utilisation d'une `union`.

Nous venons de voir que nos modifications ne changent pas le flux de gestion de la simulation compilée mais restent locales à la bibliothèque d'émulation des calculs. Les avantages de l'association direct par rapport à la gestion logicielle par émulation du calcul à virgule flottante repose en grande partie sur la simplification de la fonction effectuant le calcul mais aussi sur la possibilité de ne plus devoir récupérer les flags grâce à la politique de *mise à jour paresseuse*.

```
/*
 * emulation à partir d'entier
 */
typedef uint64_t flip64;
flip64 Flip64_add(flip64 a, flip64 b) {
  flag aSign, bSign;
  aSign = Flip64_sign( a );
  bSign = Flip64_sign( b );

  if (Flip64_isInf(a) && Flip64_isInf(b) &&
      (Flip64_sign(a) ^ Flip64_sign(b))) {
    Flip_raise(FLIP_INVALID);
  }
  if (Flip64_issNaN(a) || Flip64_issNaN(b))
    Flip_raise(FLIP_INVALID);
  if (Flip64_isNaN(a)) return Flip64_quiet_from_sNaN(a);
  if (Flip64_isNaN(b)) return Flip64_quiet_from_sNaN(b);

  if ( aSign == bSign ) {
    return Flip64_addFloatSigs( a, b, aSign );
  } else {
    return Flip64_subFloatSigs(a, b, aSign);
  }
}
```

Listing 5.3 – Implémentation de l'addition en double précision à l'aide d'une émulation logicielle

```
/*
 * Calcul à l'aide d'une instruction
 * de calcul flottant hote assembleur.
 */
flip64 Flip64_add(flip64 a, flip64 b) {
  union {double f; flip64 u;} fa, fb, res;
  fa.u = a; fb.u = b;
  res.f = fa.f + fb.f;
  return res.u;
}
```

Listing 5.4 – Implémentation de l'addition en double précision en utilisant la FPU hôte

Remplacement des fonctions FP cibles

Le remplacement des fonctions FP cibles par leurs équivalents hôtes se fait lors de la génération du code haut-niveau. Une fonction est générée pour simuler le BB cible contenant le point d'entrée de la fonction à remplacer. Elle contient normalement un ensemble de fonctions *fetch*, *execute* et *commit* simulant les instructions contenues dans ce BB. Au lieu de cela, nous générons à l'intérieur de cette fonction un appel à la fonction hôte équivalente. Il faut néanmoins garantir le respect de l'API du cœur simulé. Ainsi la fonction hôte équivalente est entourée de code garantissant la récupération des arguments, la gestion du code de retour et la gestion du pointeur d'instruction comme illustré par le listing 5.5 pour la fonction `sqrt`. La simulation compilée peut ainsi continuer à s'exécuter correctement.

```
1  int bb_fct_sqrtf(core_t *core)
2  {
3    /* get args */
4    union { float f; uint32_t u;} r0;
5    r0.u = K1_GRF_R0;
6    /* execute function */
7    union { float f; uint32_t u;} res;
8    res.f = sqrtf(r0.f);
9    /* set result in R0 */
10   K1_GRF_R0 = res.u;
11   /* set PC and NEXT_PC */
12   K1_SRF_PC = K1_NPC;
13   K1_NPC = K1_SRF_RA;
14   return 1;
15 }
```

Listing 5.5 – Fonction encapsulant l'appel à la fonction hôte sqrt

5.3.2 Résultats expérimentaux

Comme vu précédemment, la simulation des instructions est décomposée en 3 parties : *fetch*, *execute* et *commit*. Pour les instructions FP, l'exécution de l'étage *execute* est découpée en 2 parties comme vu dans le listing 5.2 : d'une part les appels aux fonctions de calcul présentes dans la bibliothèque *libFP* à partir d'un *helper* et d'autre part les différentes lectures, écritures et tests pour obtenir les arguments, le mode d'arrondi et les exceptions à partir du registre CS. Les résultats sont présentés dans le tableau 5.2 pour la simulation utilisant l'ISS fourni par Kalray.

Concernant la méthode de mesure, *Valgrind* [NS07] fournit pour une application hôte une estimation du nombre de cycles hôtes passés à exécuter chacune de ses fonctions ainsi que le nombre total de cycles pour exécuter l'application. L'application hôte dans notre cas est le simulateur et nous pouvons déduire de ces mesures le pourcentage de temps passé dans les différentes fonctions exécutées par l'application de simulation.

Pour l'ISS, les fonctions ne sont pas expansées en place sauf les fonctions permettant de gérer les variables globales contrôlant l'émulation FP (*e.g* `flip_set_rnd_mode`). On obtient donc le temps passé dans les fonctions de calcul de *libFP*, T_{FP}, et le temps passé dans les

fonctions *execute* des instructions de calcul flottant, T_{exec}. Or $T_{FP} \subset T_{exec}$. Le reste du temps passé dans les fonctions *execute* correspond à la lecture des arguments, la lecture et les tests sur les exceptions et le mode d'arrondi. La ligne *différents tests FP* du tableau correspond à ce temps, *i.e.* : $T_{exec} - T_{FP}$. Les éléments mesurés sont en gras dans le tableau, les autres résultats en sont déduits. Nous utilisons un tableau pour représenter les résultats car les valeurs mesurées sont trop petites pour être représentables correctement sur un diagramme contrairement aux résultats avec la simulation compilée.

	émulation FP		optimisation FP avec mise à jour de MXCSR pour chaque instr. FP		benchmark
temps de simulation FP (libFP)	3.3 s	**1.8%**	0.2 s	**0.09%**	LU
différents tests pour le FP	1.3 s	**0.7%**	2.3 s	**1.2%**	
reste du temps de simulation	179.1 s	97.5%	186.6 s	98.71%	
total du temps de simulation	**183.7 s**	100%	**189.0 s**	100%	
temps de simulation FP (libFP)	7.0 s	**1.1%**	0.4 s	**0.07%**	FFT
différents tests pour le FP	1.9 s	**0.3%**	4.4 s	**0.7%**	
reste du temps de simulation	632.7 s	98.6%	630.3 s	99.23%	
total du temps de simulation	**632.6 s**	100%	**635.2 s**	100%	
temps de simulation FP (libFP)	0.9 s	**5.5%**	0.1 s	**0.5%**	radix
différents tests pour le FP	0.4 s	**2.4%**	0.7 s	**4.2%**	
reste du temps de simulation	15.6 s	92.1%	16.3 s	95.3%	
total du temps de simulation	**16.9 s**	100%	**17.1 s**	100%	
temps de simulation FP (libFP)	1.4 s	**3.8%**	0.1 s	**0.28%**	ocean
différents tests pour le FP	0.6 s	**1.6%**	1.0 s	**2.7%**	
reste du temps de simulation	35.9 s	94.6%	37.5 s	97.02%	
total du temps de simulation	**37.9 s**	100%	**38.7 s**	100%	
temps de simulation FP (libFP)	2.2 s	**0.8%**	0.3 s	**0.1%**	water-nsquared
différents tests pour le FP	0.8 s	**0.3%**	1.7 s	**0.6%**	
reste du temps de simulation	272.8 s	98.9%	282.1 s	99.3%	
total du temps de simulation	**275.8 s**	100%	**284.1 s**	100%	

TABLE 5.2 – Découpage du temps de simulation pour l'ISS (FP = floating point)

Les résultats en simulation compilée sont présentés dans la figure 5.8. Pour la simulation compilée, les fonctions *fetch*, *execute* et *commit* sont expansées en place. Par conséquent Valgrind ne peut plus mesurer le nombre de cycles nécessaires pour exécuter ces fonctions. Cependant les appels aux fonctions de calcul de *libFP* ne sont pas expansés en place, ce qui permet d'avoir encore cette mesure. Une autre mesure concerne le temps passé à simuler les fonctions cibles candidates à un remplacement : `sqrt`, `__divdf3` et `__divsf3`. Ces fonctions utilisent très peu d'instructions flottantes, donc leur temps passé dans *libFP* n'est pas visible, par contre les quelques instructions FP qu'elles utilisent sont principalement des instructions exotiques, d'où leur remplacement.

Analyse des résultats

Pour l'ensemble des applications déterministes, les résultats des calculs produits par la simulation utilisant l'approche proposée sont comparés à ceux de l'émulation logicielle et ils sont identiques. De plus, une suite de tests vérifiant la conformité de la *libFP* à la norme IEEE 754 est utilisée pour vérifier que notre approche donne des résultats exacts.

Le tableau 5.2 confirme que, pour l'ISS, le temps passé à effectuer de l'émulation logicielle des calculs flottants n'est pas significatif. Ce chiffre s'élève à environ 1% du temps

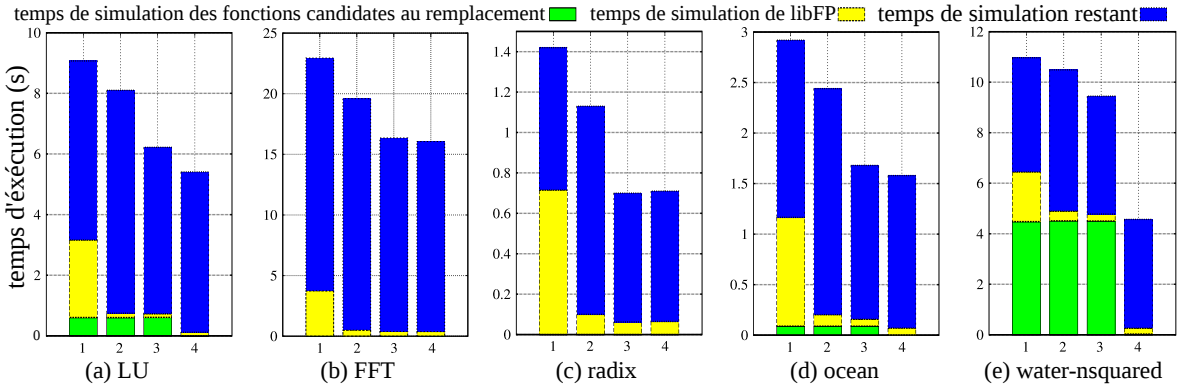


FIGURE 5.8 – Découpage du temps de simulation pour la simulation compilée (1 : émulation FP- 2 : FP hôte avec mise à jour de *MXCSR* pour chaque instruction FP - 3 : FP hôte avec mise à jour paresseuse - 4 : FP hôte avec mise à jour paresseuse + remplacement des fonctions)

total de simulation pour les applications sauf *radix* et *ocean* pour lesquelles il monte à 5,6% et 3,8%. La colonne 1 montre les résultats quand l'émulation logicielle à l'aide d'entiers est utilisée et la colonne 2 quand on utilise la solution avec la mise à jour des registres d'état/-contrôle seulement pour les instructions FP. L'exemple de mise à jour du mode d'arrondi du cœur hôte à partir du mode d'arrondi cible est donné dans le listing 5.6.

```
1  /* variable globale controlant l'arrondi courant */
2  int Flip_rnd_mode = DEFAULT_RND_MODE;
3
4  static inline void flip_set_rnd_mode(Flip_rnd_mode_t rnd) {
5    if (Flip_rnd_mode != rnd) {
6      Flip_rnd_mode = rnd;
7      uint32_t mxcsr = 0;
8      __asm__ volatile ("stmxcsr␣%0;\n" : "=m" (mxcsr));
9      mxcsr &= ~(0x3 << 13);
10     mxcsr |= (switch_field_k1_to_x86(rnd) & 0x3) << 13;
11     __asm__ volatile ("ldmxcsr␣%0;\n" : : "m" (mxcsr));
12   }
13 }
```

Listing 5.6 – Mise à jour du mode d'arrondi sur le cœur hôte

La séparation des 2 mesures : temps passé dans la bibliothèque *libFP* et temps passé dans les différents tests sur les registres d'états de la FPU, nous permet de voir l'impact de notre gestion des calculs flottants sur ces 2 aspects. Même si le pourcentage de temps passé dans *libFP* est réduit d'un facteur 8 à 20, le temps global de simulation n'est pas réduit. En effet la mise à jour des registres d'état/contrôle en utilisant les instructions x86_64 *ldmxcsr* et *stmxcsr* fait augmenter le temps passé dans les fonctions *execute*. Ces instructions nécessitent beaucoup de temps car les opérandes sont chargés depuis la mémoire ou en mémoire, alors qu'en même temps les optimisations du compilateur sont réduites à cause de l'usage permis par *gcc* d'assembleur intégré dans du code C comme illustré par le listing 5.6. Ainsi notre

proposition est moins performante que l'émulation logicielle des calculs flottants à cause de la mise à jour des registres d'état/contrôle. De plus la politique *mise à jour paresseuse*, qui dans le cas de l'ISS génère plus de mises à jour, est encore moins compétitive.

On observe que le temps restant (non passé dans l'exécution des fonctions *execute* des instructions FP - troisième ligne du tableau pour chaque application) augmente dans le cas de notre proposition. *Valgrind* simule chaque instruction de l'application sur un CPU synthétique. *Cachegrind* (outil utilisé avec Valgrind) simule les interactions du programme avec la hiérarchie mémoire de la machine. Nous supposons que les instructions *ldmxcsr* et *stmxcsr* ne sont pas simulées avec assez de précision par le couple *Valgrind/Cachegrind* et le nombre de cycles nécessaires à leur exécution est sous-évalué. En effet, *MXCSR* est un registre d'état et un accès à un tel registre génère des comportements particuliers. Des cycles de gel arrivent si le renommage d'un registre *MXCSR* se produit trop proche d'un précédent. Or cela arrive fréquemment dans les fonctions *execute* pour les calculs flottants.

Concernant la simulation compilée, on constate que pour les applications utilisées, le temps passé dans *libFP* se trouve entre 16% (pour *FFT*) et 50% (pour *radix*) du temps total de simulation quand l'émulation logicielle à base d'entiers des instructions FP est utilisée. L'optimisation des calculs FP permet de réduire significativement le temps passé dans cette bibliothèque. Mais les mises à jour fréquentes de *CS/MXCSR* réduisent le gain global en augmentant la section "*reste du temps de simulation*". Avec les optimisations des calculs FP, plus la politique *mise à jour paresseuse*, le temps passé dans la bibliothèque *libFP* chûte à 9% pour *radix* et entre 2% et 4% pour les autres applications. De plus la section "*reste du temps simulé*" est réduite par rapport à l'absence de la politique *mise à jour paresseuse*. Le temps total nécessaire pour simuler les fonctions candidates au remplacement atteint plus d'un tiers du temps total de simulation pour *water-nsquared*. Après le remplacement par des fonctions hôtes, le ratio descend à moins de 1%.

Les optimisations FP et le remplacement des fonctions permettent une accélération de 1,4 à 2,4 pour l'ensemble des applications simulées à l'aide de la simulation compilée. Cela montre l'intérêt de l'approche.

5.4 Conclusion

Notre proposition pour simuler des caractéristiques architecturales spécifiques à l'aide de la technique de simulation native basée sur le HAV a été implémentée et testée dans le cadre du MPPA®-256 de Kalray comme plateforme cible. Nos expériences montrent que la simulation native se comporte mieux avec notre proposition pour réduire le nombre de trappes que la méthode basique quand le nombre de cœurs simulés augmente. En l'absence de composants ralentissant l'ensemble de la simulation et en mettant de côté l'absence de parallélisation de la simulation au niveau d'une grappe de calcul, la vitesse de notre méthode de simulation se trouve seulement à un ordre de grandeur de la vitesse de l'exécution native de la même application.

Nous avons implémenté une méthode de simulation compilée pour laquelle il est facile de changer la gestion des instructions de calcul à virgule flottante. Nous y avons inséré notre proposition de gestion des calculs flottants cibles à l'aide des instructions FP hôtes ou des fonctions mathématiques hôtes. Les résultats montrent que notre méthode donne des résultats numériquement équivalents et permet une accélération de la simulation par rapport à une émulation logicielle basée sur les entiers de 1.4 à 2.4 pour les applications simulées.

Chapitre 6

Passage à l'échelle de la simulation native

Le passage à l'échelle est un des points critiques de la simulation native car le nombre de cœurs ne va cesser d'augmenter, en particulier dans le cas des systèmes many-cœurs. Or nous avons vu dans le chapitre 2 que la simulation native pose des problèmes de passage à l'échelle.

Nous rappelons l'architecture du processeur Kalray (déjà présentée dans le chapitre 5) car les expériences vont être intégrées au fur-et-à-mesure pour illustrer nos propos. La plateforme cible utilisée pour les expériences est le MPPA®-256 (Multi Purpose Processor Array) [DdDAB+13] de Kalray. Il est composé de 256 cœurs VLIW spécifiques dédiés aux calculs (PE) et 32 cœurs de management (RM). Les cœurs sont groupés en 16 grappes de calcul contenant chacun 16 PE et 1 RM, l'ensemble étant interconnecté par 2 NoC. Chaque grappe de calcul contient 2 MB de mémoire partagée, et il n'y a pas de support matériel pour la cohérence des caches.

La plateforme de simulation représente une grappe de calcul du MPPA®-256 avec une extension à 2 GB de la mémoire partagée pour pouvoir exécuter des applications de référence sans modifier leur code. Les applications retenues correspondent à un sous-ensemble des applications proposées dans SPLASH-2 [WOT+95]. Pour ces tests, les applications utilisent autant de threads qu'il y a de PE instanciés dans une grappe. Notre critère de sélection des applications/kernels est que le temps de simulation reste raisonnable pour une simulation avec l'ISS de Kalray. Cet ISS est utilisé comme point de référence. Nous avons aussi retiré *radix* car il contient des appels à `pthread_cond_wait` alors que l'implémentation de notre solution, décrite dans la section 6.2 ne supporte pas pour l'instant les *conditions* (au sens *pthread* du terme). Concernant les performances, nous mesurons uniquement le temps de la simulation sans comptabiliser le temps d'instanciation et de destruction de la plateforme. La simulation tourne sur une machine hôte ayant 8 cœurs, Intel® Xeon® CPU W3550 tournant à 3.07 GHz et avec Linux 3.10.0.

Nous verrons différentes stratégies de simulation appuyées par leurs résultats expérimentaux. Une première approche utilise une simulation séquentielle (utilisant donc un seul cœur hôte) à l'intérieur d'une plateforme TLM comme habituellement avec SystemC. Une deuxième approche parallélisera cette simulation même si de possibles problèmes de saturation de KVM peuvent avoir lieu si un processeur many-cœurs entier est simulé. Mais tout d'abord nous rappellons le problème de passage à l'échelle dû au découplage temporel.

6.1 Rappel du problème dû au découplage temporel

Deux niveaux de simulation peuvent être envisagés : la simulation au niveau d'une grappe de calcul et la simulation de l'ensemble des grappes. Pour la simulation de l'ensemble des grappes, nous nous reposons sur des travaux déjà existants [MMGP10, WSL+14] qui permettent de garantir le passage à l'échelle de la simulation quand le nombre de grappes croît. Dans ces travaux, chaque grappe est simulée par un simulateur autonome. La communication entre les grappes est faite grâce à une abstraction du composant NoC. C'est cette abstraction qui permet la parallélisation de la simulation et de maintenir un certain niveau de synchronisation entre les grappes simulées : *i.e.* il existe une fenêtre temporelle dans laquelle les grappes peuvent être simulées sans risque d'erreurs de causalité, cette fenêtre de simulation dépend de la latence du NoC et du temps simulé des différentes grappes.

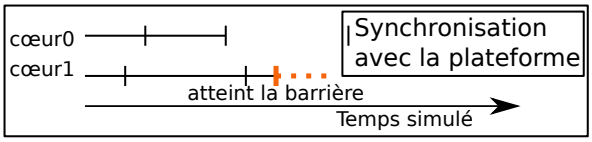
Les machines hôtes modernes sont maintenant multi-processeurs. Si elles contiennent n processors et en tenant compte que la limite recommandée par KVM est de 10 VM par processeur physique hôte, alors jusqu'à $10 \times n$ VM peuvent être exécutées en parallèle. Si la simulation d'une grappe est séquentielle alors jusqu'à $10 \times n$ grappes peuvent être simulées en même temps. Comme la performance globale est imposée par la grappe se simulant le plus lentement, notre objectif est d'éviter qu'une grappe ayant un grand nombre de cœurs à simuler ralentisse significativement l'ensemble de la simulation. Par conséquent nous allons nous concentrer sur le passage à l'échelle de la simulation à l'intérieur d'une grappe.

Comme les cœurs simulés s'exécutent par plage de temps ayant une granularité non négligeable, un découplage temporel des différents composants a lieu. Cela peut conduire à des erreurs de simulation. Et plus le nombre de cœurs augmente, plus la probabilité que ces problèmes arrivent, augmente. Une description de ces erreurs peut être trouvée dans [WFWT13].

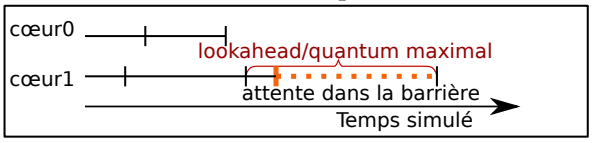
La figure 6.1 illustre un autre problème lié au découplage temporel : la simulation peut être fonctionnellement correcte mais être ralentie par des interactions entre 2 cœurs simulés. Ce problème a été identifié dans la communauté de la virtualisation [SSCZ13]. L'application cible simulée dans cet exemple est composée de 2 threads s'exécutant sur 2 cœurs différents. Tous les deux se synchronisent sur une barrière. Le *cœur1* atteint le premier la barrière mais son quantum de temps n'est pas entièrement consommé (Figure 6.1a). Il finit donc de le consommer avant de rendre la main à la plateforme de simulation (Figure 6.1b). Le *cœur0* est alors simulé. Il atteint la barrière permettant de libérer le *cœur1* (Figure 6.1c). Mais cela a lieu à un temps simulé inférieur au temps simulé actuel du *cœur1*. À cause du découplage temporel, le temps entre la libération de la barrière par le *cœur0* et la synchronisation du *cœur1* est perdu, que ce soit le temps simulé ou le temps de simulation, puisque le *cœur1* a simulé trop en avance durant sa période d'exécution. De manière identique, un accès à une variable partagée ou à un verrou peut générer ce type d'inconsistance mémoire et de perte de temps de simulation.

Résultats expérimentaux La première expérience compare la simulation séquentielle native (appelée *basic_seq_sim* présentée dans le chapitre 4) avec le simulateur ISS fourni par Kalray. Le tableau 6.1 présente les temps de simulation ISS pour les benchmarks comme point de référence pour toutes les autres expériences. La figure 6.2 présente l'accélération de *basic_seq_sim* comparée à l'ISS pour une simulation d'une seule grappe (l'échelle de l'axe des ordonnées est logarithmique).

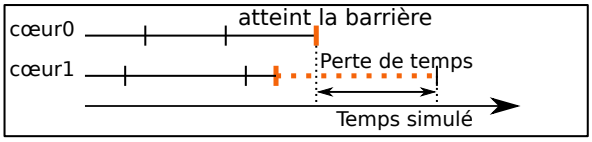
Dans le pire cas, notre approche fournit une accélération de 194 (*ocean*, 16 PE) et dans le meilleur cas une accélération de 9723 (*water-spatial*, 8 PE) par rapport à l'ISS. Mais en



(a) Barrière atteinte par le cœur1



(b) Le cœur1 est exécuté jusqu'à la fin de son quantum



(c) Erreur temporelle visible

FIGURE 6.1 – Erreur temporelle générée par du découplage temporel lors d'une synchronisation sur une barrière dans une plateforme multi-processeur

TABLE 6.1 – Temps de simulation de référence pour une simulation ISS mono-grappe

	1 PE	2 PE	4 PE	8 PE	16 PE
FFT	653 s	654 s	661 s	703 s	743 s
LU	3102 s	3017 s	3133 s	3283 s	3683 s
ocean	101 s	103 s	100 s	116 s	138 s
raytrace	7111 s	6718 s	6720 s	7242 s	8165 s
water-nsquared	680 s	685 s	716 s	814 s	911 s
water-spatial	1529 s	1968 s	2616 s	3756 s	4441 s

fonction de l'application à simuler, notre simulation ne passe pas à l'échelle de la même manière. Il y a globalement 2 catégories : les applications avec une accélération plutôt constante et celles avec une accélération décroissante en fonction du nombre de PE. Pour rappel, le nombre de threads instanciés correspond au nombre de PE disponibles. Dans la première catégorie il y a *FFT*, *LU* et *water-spatial*. Il y a en effet peu d'interactions entre les threads de l'application cible : seulement quelques barrières et verrous comme décrit dans [WOT+95].

Dans l'autre catégorie, *ocean*, *water-nsquared* et *raytrace*, la vitesse de simulation est toujours meilleure pour la simulation native mais l'accélération décroit quand le nombre de cœurs augmente. Il y a beaucoup plus d'interactions entre les threads des applications cibles. Par exemple, il y a des milliers de prises de verrous dans *ocean*, *water-spatial* et *raytrace*, et quelques centaines de barrières dans *ocean*.

6.2 Simulation séquentielle avec gestion des synchronisations

Wu *et al.* [WFWT13] proposent de synchroniser la simulation lors de chaque accès à des variables partagées. Cela a été testé dans le cadre de la simulation compilée, cependant il est beaucoup plus difficile de trouver les accès aux variables partagées en simulation native et de forcer une synchronisation au moment de ces accès. Dans le contexte de la norme

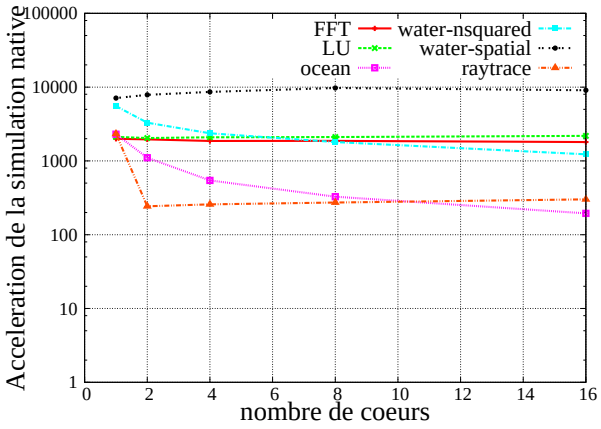


FIGURE 6.2 – Accélération de *basic_seq_sim* par rapport à une simulation ISS au niveau d'une grappe de calcul

POSIX, les accès à la mémoire partagée doivent être protégés par des *sémaphores*, des *mutex* ou un protocole particulier basé sur des *barrières* ou des *conditions* pour garantir le bon fonctionnement de l'application.

Dans le but d'éviter les pertes de temps de simulation dûes au découplage temporel, nous proposons de se synchroniser sur des primitives bas niveau telles que `lock`, `unlock`, `barrier_wait`, etc. ce qui permettra d'assurer la cohérence temporelle (du point de vue temps simulé) des accès à la mémoire partagée. L'idée est d'ajouter du code de contrôle spécifique dans les fonctions du HAL (typiquement une macro pour avoir le minimum d'impact sur le code cible) pour forcer la synchronisation lorsque cela est nécessaire. La liste 6.1 illustre les modifications à effectuer sur une version simplifiée de `lock`, `unlock` et `barrier_wait` avec l'insertion des macros `SYNC_LOCK`, `SYNC_UNLOCK`, `SYNC_BARRIER_SYNC`, `SYNC_BARRIER_RELEASE` et `SYNC_BARRIER_WAIT`. Pour gérer correctement les synchronisations entre les threads, l'environnement de simulation a besoin d'un ensemble d'informations sur l'avancement des synchronisations. Ces macros nous permettent d'obtenir ces renseignements.

Notre solution fait l'hypothèse qu'un thread est assigné à un cœur et qu'il y a seulement un thread par cœur, ce qui est généralement le cas avec les many-cœurs. Donc un thread et un cœur peuvent être associés. Nous détaillons notre politique de gestion seulement pour les verrous et les barrières, mais cela peut facilement être étendu aux conditions.

```
1  void __lock(__lock_t *lock) {
2    int get_lock = 0;
3    while (!get_lock) {
4      SYNC_LOCK(lock);
5      uint32t old = __builtin_cws(&(lock->lock), _LOCKED, _UNLOCKED);
6      if(old == _UNLOCKED){
7        get_lock = 1;
8      }
9    }
10 }
11
12
```

```
13 void __un_lock(__lock_t *lock) {
14   SYNC_UNLOCK(lock);
15   lock->lock = _UNLOCKED;
16 }
17
18 void __barrier_wait(__barrier_t *barrier) {
19   __lock(&barrier->guard);
20   SYNC_BARRIER_SYNC(barrier);
21
22   if(dernier_thread_a_atteindre_la_barriere(barrier)) {
23     SYNC_BARRIER_RELEASE(barrier);
24     remet_a_0_la_barriere(barrier);
25     __unlock(&barrier->guard);
26   } else {
27     enregistre_le_thread_dans_la_barriere(barrier);
28     __unlock(&barrier->guard);
29
30     SYNC_BARRIER_WAIT(barrier);
31     while(barriere_non_remise_a_0(barrier))
32       SYNC_BARRIER_WAIT(barrier);
33   }
34 }
```

Listing 6.1 – Exemple d'insertion de macro nécessaire à notre politique de synchronisation dans du code bas niveau

Gestion des barrières La gestion d'une barrière peut être divisée en deux machines à états - *Finite State Machine (FSM)*, une pour les threads qui sont en attente et une pour le dernier thread atteignant la barrière. Ce dernier thread signale à l'ensemble des threads attendant sur la barrière qu'ils peuvent la quitter. Ces FSM sont décrites dans la figure 6.3.

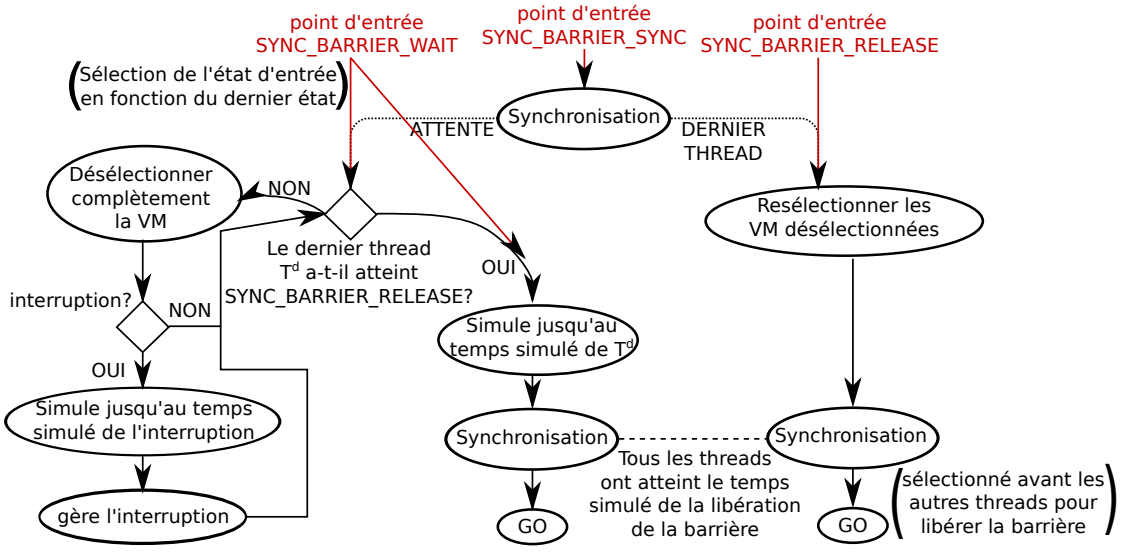


FIGURE 6.3 – Protocole de synchronisation pour une barrière

Dans le listing 6.1 ligne 20, la macro `SYNC_BARRIER_SYNC` est utilisée au début de la barrière pour forcer une synchronisation avec la plateforme de simulation. Cette synchronisation est nécessaire avant la sélection d'une des deux FSM pour garantir l'ordre d'arrivée des

threads dans la barrière. Une autre manière de garantir l'ordre d'arrivée réside dans la prise du verrou ligne 19. Si notre procédure de gestion des verrous est également appliquée, alors une synchronisation avec la plateforme aura déjà été effectuée lors de la prise du verrou. L'utilisation de la macro `SYNC_BARRIER_SYNC` n'est alors plus nécessaire.

Lorsque le code exécuté par une VM (associée à un cœur cible) doit attendre dans une barrière, l'état "Désélectionner complètement la tâche" de notre FSM est atteint. Pour désélectionner une VM, une *trappe_io* particulière est utilisée. Elle indique à l'ordonnanceur de la plateforme TLM de retirer la VM de la liste des composants ordonnançables. Cette action est différente d'une *trappe_io* servant à synchroniser la VM. Synchroniser une VM force seulement l'ordonnanceur à sélectionner le prochain composant éligible, i.e. le composant avec le temps simulé le plus faible. Une VM "complètement désélectionnée" n'est plus éligible même si son temps simulé est le plus faible. Ce mécanisme permet d'éviter qu'un cœur soit simulé plus loin que le temps simulé de la libération de la barrière. Cela empêche aussi une série inutile de trappes de synchronisation. Le dernier cœur (celui atteignant la barrière en dernier) rend de nouveau éligibles ces VM à travers une *trappe_io* et se synchronise pour permettre aux autres VM d'atteindre son temps simulé. Ce dernier thread est finalement sélectionné pour être simulé et ainsi relâcher la barrière.

Si une interruption visant un cœur "complètement désélectionné" est levée, le cœur est de nouveau sélectionné et simulé jusqu'au temps d'arrivée de l'interruption (qui est nécessairement avant que le dernier thread atteigne la barrière). L'interruption peut alors être levée et le gestionnaire d'interruption est simulé. Après le retour d'interruption, le cœur simulé peut soit continuer son exécution si la barrière a été relachée, soit être de nouveau rendu non éligible.

Le dernier thread T^d atteignant la barrière indique à l'ordonnanceur que l'ensemble des threads attendant sur la même barrière que lui peuvent de nouveau être sélectionnés. Pour cela une *trappe_io* particulière est encore utilisée. Comme il est le dernier à atteindre la barrière, il est aussi le thread avec le temps simulé le plus en avance. Une synchronisation avec la plateforme va permettre aux threads attendant sur la barrière de continuer leur attente jusqu'au temps simulé de la fin de la barrière. Le thread T^d est alors de nouveau simulé pour mettre à jour la barrière, permettant ainsi à l'ensemble des threads de quitter la barrière.

Le nombre de trappes additionnelles générées par le mécanisme de gestion de la barrière reste faible. Pour une barrière avec n threads, $n-1$ trappes ont lieu pour rendre non éligibles les cœurs, 1 trappe pour les rendre de nouveau éligibles et que le dernier thread se synchronise (la FSM contient deux états pour des raisons de lisibilité mais l'ensemble peut être fait en 1 trappe), et $n-1$ trappes pour que les threads se synchronisent tous au temps simulé correspondant à la libération de la barrière.

Gestion des verrous La gestion des verrous est également découpée en deux FSM : une pour la prise de verrou et l'autre pour le relâchement du verrou. La figure 6.4 représente ces FSM. Pour pouvoir prendre un verrou, un thread doit remplir trois conditions :

— le thread doit avoir atteint un temps simulé supérieur au temps simulé de la dernière libération du verrou,
— le verrou ne doit pas être déjà pris par un autre thread,
— le thread doit être sûr qu'un autre thread ne peut pas prendre le verrou dans son passé.

En fonction des conditions non remplies, le cœur simulé doit se synchroniser avec la plateforme, continuer à se simuler ou demander à ne plus être éligible.

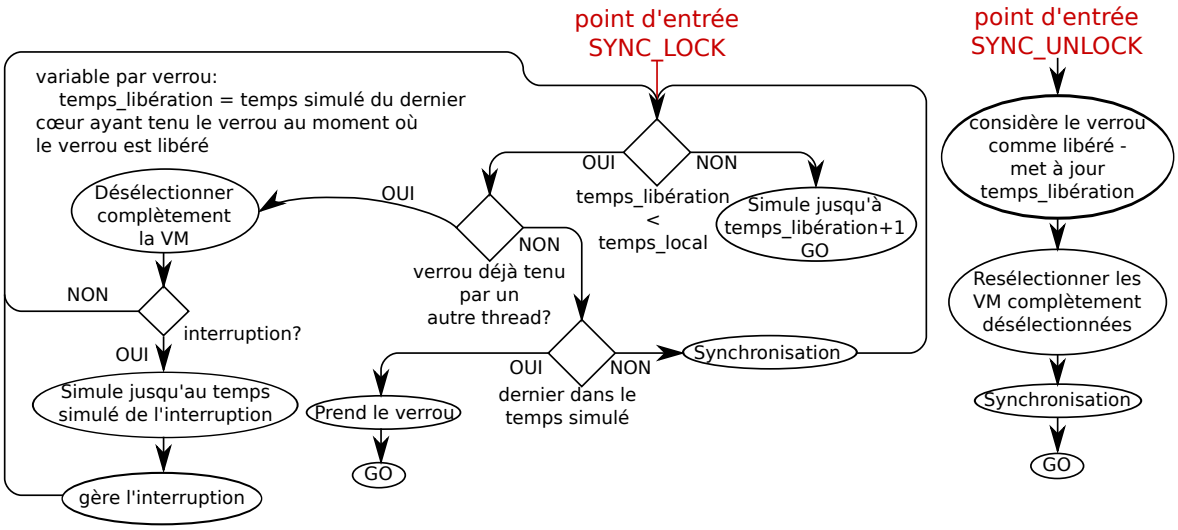


FIGURE 6.4 – Protocole de synchronisation pour un verrou

De même que pour la barrière, si une interruption arrive pour un thread ayant demandé à ne plus être éligible, ce thread est d'abord simulé jusqu'au temps de l'interruption puis l'interruption est réellement levée et simulée.

Dans le processus de relâchement du verrou, nous ne pouvons pas seulement rendre de nouveau éligibles les cœurs en attente. En effet, dès que le verrou sera relâché, le prochain thread à exécuter la ligne 5 du listing 6.1 prendra le verrou. Le temps simulé de ce thread n'est pas forcément valide par rapport au temps simulé du relâchement du verrou ce qui conduit à la superposition par rapport au temps simulé des prises de verrous comme illustré par la figure 6.5. La variable `temps_libération` est introduite pour forcer les cœurs à être simulés jusqu'au temps de relâchement du verrou. La synchronisation dans la FSM du relâchement du verrou permet aux autres threads d'atteindre ce temps simulé avant que le thread ne relâche réellement le verrou.

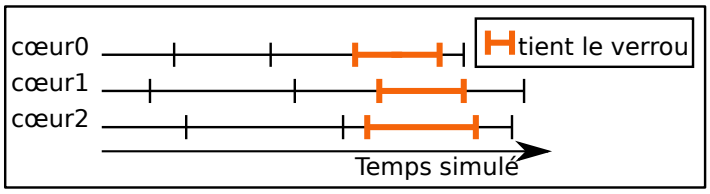


FIGURE 6.5 – Verrou tenu par plusieurs cœurs à des temps simulés identiques mais à des moments différents du point de vue du temps hôte

L'ajout de contrôle dans la simulation pour la gestion des verrous peut augmenter significativement le nombre de trappes. Soit n le nombre de threads essayant en même temps de prendre le même verrou. Un thread réussit à prendre le verrou avec une éventuelle synchronisation garantissant que le cœur a le temps simulé le plus faible. Les autres cœurs vont demander à être retirés de la liste des cœurs éligibles, cela nécessite $n - 1$ trappes. Soit i le nombre de threads essayant de prendre le verrou durant un tour. Quand le verrou est libéré, une *trappe_io* générée par le thread détenant le verrou permet de rendre de nou-

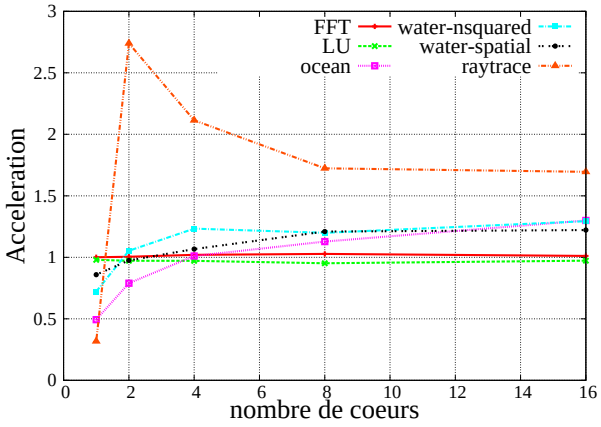


FIGURE 6.6 – Accélération de *sync_seq_sim* par rapport à *basic_seq_sim*

veau éligibles les threads précédemment "complètement désélectionnés". Ils sont simulés jusqu'au temps de relâchement du verrou et synchronisés avec la plateforme, cela génère i trappes. Le thread qui relâche le verrou est aussi synchronisé mais cela est inclu dans la *trappe_io* provoquant le réveil/l'éligibilité des threads en attente. Les threads qui ne prendront pas le verrou à ce tour seront de nouveau "complètement désélectionnés", soit $i-1$ trappes. Il y a donc $1+i+(i-1)=2i$ trappes durant un tour de prise de verrou. Au final, il y a $(n-1)+\sum_{i=1}^{n-1} 2i = (n-1)+n(n-1) = (n-1)(n+1) = O(n^2)$ trappes.

Si nous supposons que le système d'exploitation simulé utilise un ordre FIFO pour gérer les demandes de prise de verrous, comme c'est le cas pour nos expériences, on peut se limiter à rendre éligible un seul thread. Le nombre de trappes est ainsi réduit et devient linéaire en fonction de la concurrence d'accès au verrou : $(n-1)+\sum_{i=1}^{n-1}(1+1) = (n-1)+2(n-1) = 3(n-1) = O(n)$.

Résultats expérimentaux Nous comparons le temps de simulation de la simulation native avec notre protocole (appelé *sync_seq_sim*) et sans notre protocole, *basic_seq_sim*. La figure 6.6 présente les résultats. Encore une fois les résultats sont dépendants de l'application simulée.

Nous observons qu'il n'y a pas toujours un gain de temps simulé. En effet notre proposition augmente généralement le nombre de trappes. Le tableau 6.2 contient le nombre de trappes pour les différentes applications en fonction de la politique utilisée. Les différentes politiques sont :

1. pas de politique de gestion pour les verrous et les barrières,
2. la politique de gestion présentée précédemment mais avec seulement un cœur réveillé à chaque tour (*sync_seq_sim*),
3. la politique précédente mais avec tous les cœurs réveillés à chaque tour.

Nous observons que le nombre de trappes augmente significativement pour *ocean*, *raytrace* et *water-nsquared* entre (1) et (2). Ces 3 applications utilisent beaucoup de verrous. Mais entre (2) et (3), le nombre de trappes connait une forte augmentation uniquement pour *ocean*. La concurrence d'accès aux verrous est donc haute seulement dans ce cas. Cela vient des interactions de synchronisation entre les verrous et les barrières. En effet *ocean* est la

TABLE 6.2 – Nombre de trappes en simulation native avec et sans notre politique de synchronisation pour 16 PE

	Sans contrôle	Avec contrôle Réveille 1 seul thread	Avec contrôle Réveille tous les threads
FFT	4197	4623	5246
LU	31799	38938	40943
ocean	16350	97958	187008
raytrace	637007	4343536	4344000
water-nsquared	17106	82097	83204
water-spatial	8911	7421	8542

seule application faisant un grand nombre d'appels aux verrous et aux barrières. Même si le nombre de trappes augmente énormément pour *ocean*, *raytrace* et *water-nsquared*, notre proposition permet de sauver du temps de simulation pour ces applications quand on augmente le nombre de PE.

L'accélération de *water-spatial* augmente avec le nombre de PE car son nombre de trappes est réduit en utilisant notre politique. Les trappes arrivent principalement parce que (a) une VM a simulé trop en avance sur les autres (elle a consommé tout son quantum), appelons-les $trap_a$ ou (b) notre politique de gestion en a besoin, appelons les $trap_g$. *Water-spatial* se trouve dans le bon cas où le nombre de $trap_g$ ne dépasse pas la réduction du nombre de $trap_a$ Le nombre de $trap_a$ est de 6571 pour (1) contre 2096 pour (2) et (3). Seulement 3040 $trap_g$ sont ajoutées par (2) et 4161 $trap_g$ par (3).

LU est une des applications dont les threads ont peu d'interractions entre eux. Notre politique de gestion ne lui profite donc pas. Par contre son nombre de trappes augmente un peu, c'est pourquoi *LU* est un peu plus lente avec notre politique de gestion.

On observe un comportement étrange pour *raytrace* quand 2 PE sont instanciés. Avec *basic_seq_sim*, le temps de simulation de *raytrace* connait un saut significatif entre 1 PE et 2 PE, il passe d'environ 3 secondes de simulation à environ 27 secondes. Le pic d'accélération de *raytrace* avec 2 PE vient de là.

Malheureusement, une asymptote à l'accélération fournie par notre politique apparait autour de 4/8 PE. La raison principale à ce plateau est que le gain de temps fourni par notre politique est compensé par le temps perdu à cause de l'augmentation du nombre de trappes quand on augmente le parallélisme utilisé par l'application cible. Par conséquent une réduction du coût des trappes permettrait directement d'améliorer le passage à l'échelle de la simulation native basée sur le HAV en utilisant notre politique de synchronisation pour les verrous et les barrières.

Une autre manière de voir l'impact de notre approche est de comparer *sync_seq_sim* par rapport à l'ISS. Cela est représenté dans la figure 6.7. On observe que la diminution de l'accélération en fonction du nombre de PE est moins significative que dans la figure 6.2 mais est encore présente.

A défaut de pouvoir réduire le coût d'une trappe, nous essayons de réduire le nombre de trappes nécessaires à la gestion des barrières et des verrous. Pour cela, il faudrait déplacer le code de gestion des VM de l'ordonnanceur qui est actuellement dans le mode *user* jusqu'au mode *guest*. Mais cela force la simulation à devenir parallèle à l'intérieur d'une grappe.

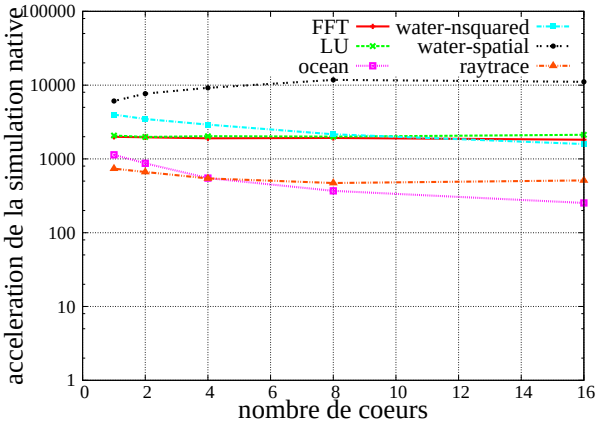


FIGURE 6.7 – Accélération de *sync_seq_sim* par rapport à l'ISS

6.3 Simulation parallèle

Nous allons maintenant étudier la simulation native basée sur le HAV en la parallélisant au niveau d'une grappe. Le nombre de VM exécutées simultanément sur la machine hôte peut alors saturer le système hôte dans le cas de la simulation complète d'un processeur many-cœurs. Néanmoins nous étudions cette solution pour voir si elle peut permettre de dépasser le problème de passage à l'échelle vu précédemment.

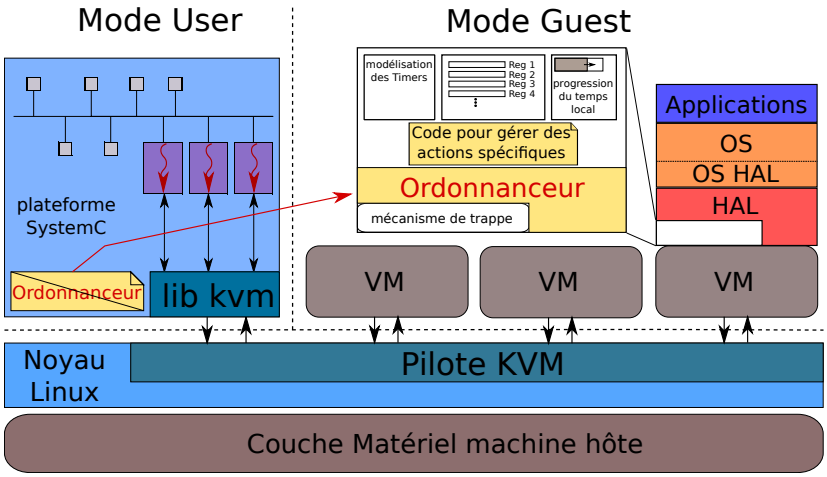


FIGURE 6.8 – Déplacement de l'ordonnanceur du mode user au mode guest dans le code HAL

Dans la simulation séquentielle utilisant notre politique de synchronisation, des trappes sont utilisées pour contrôler l'ordonnancement des cœurs cibles et ont été identifiées comme étant la source de la limitation du gain. L'objectif est de minimiser le nombre de trappes, nous décidons donc de déplacer l'ordonnanceur de la plateforme TLM en mode user au code exécuté en mode guest, plus précisément dans le HAL comme illustré par la figure 6.8. Il ne s'agit alors plus vraiment d'un ordonnanceur mais plutôt d'une gestion du temps simulé de chaque composant cœur.

Ce choix a plusieurs conséquences :
— les composants ayant besoin que le temps simulé avance pour simuler leurs comportements doivent être attachés à une VM de telle sorte que l'ordonnanceur puisse les autoriser à s'exécuter.
— la plateforme de simulation doit simuler les cœurs en parallèle car l'ordonnanceur est maintenant exécuté par le code s'exécutant sur les VM. Un thread est donc créé par cœur cible. Il attend sur une condition pthread en mode user tant que l'état du cœur cible associé est à *power_off*. Son passage à *running* lui permet de démarrer son exécution qui est alors gérée par l'ordonnanceur en mode guest.
— la capacité de KVM d'augmenter le nombre de VM et de VMM devient un possible problème. Pour simuler complètement un MPPA®-256 avec 256 PE et 32 RM, une machine hôte avec au moins 29 cœurs est nécessaire (10 VM par cœur hôte, $(256 + 32)/10 = 28,8$).

Nous verrons tout d'abord une première approche minimisant complètement le contrôle de l'ordonnanceur sur l'exécution des VM. Puis une deuxième approche augmentera le contrôle pour limiter les problèmes rencontrés.

6.3.1 Simulation parallèle avec une limite au décalage temporel possible

Le premier ordonnanceur implémenté garantit qu'un cœur ne prend pas trop d'avance du point de vue temps simulé par rapport aux autres cœurs simulés. Le composant ayant le temps simulé le plus faible donne le temps de référence de l'ensemble du système simulé. Une fenêtre temporelle, dans laquelle les composants peuvent être simulés, est définie. Elle se caractérise par sa taille, définissant l'avancement maximal d'un composant par rapport à un autre.

Pour avoir un point de comparaison, nous commençons par ne pas implémenter de gestion temporelle des verrous et des barrières, la partie fonctionnelle pour les variables partagées étant garantie matériellement par l'accès atomique des VM à la mémoire lors des instructions de *compare*&*swap* (ou instructions équivalentes). Cette implémentation est nommée *basic_par_sim*.

Un problème important de la simulation parallèle basée sur le HAV est qu'il n'y a pas de moyen de désélectionner rapidement une VM et de la réveiller plus tard. L'instruction x86 `hlt` qui permet normalement de placer un cœur dans l'état `HALT` (équivalent à un état idle) quand elle est exécutée dans le niveau de privilège 0, génère en mode guest une trappe qui est propagée jusqu'au mode user par KVM. C'est alors équivalent à notre *trappe_io* permettant de rendre une VM non éligible ou une synchronisation d'un composant cœur. Or le changement de place de l'ordonnanceur a pour objectif de minimiser le nombre de trappes, donc nous décidons de ne pas produire une trappe mais d'utiliser de l'attente active quand un cœur a simulé trop en avance.

Expérience La figure 6.9 représente l'accélération de *basic_par_sim* par rapport à l'ISS. La première observation concerne la capacité de passage à l'échelle qui est meilleure que l'approche séquentielle jusqu'à 8 PE. Pour 16 PE, nous observons une réduction considérable de l'accélération. La simulation parallèle est confrontée à un problème classique de contention de ressources. La machine hôte a seulement 8 cœurs alors que 16 cœurs sont simulés. L'ordonnanceur de KVM n'a pas connaissance de l'existence de notre ordonnanceur tournant en mode guest. Par conséquent, une VM, VM_a peut être exécutée par KVM alors qu'elle a atteint l'avancement maximal autorisé et donc exécute de l'attente active. Dans le même

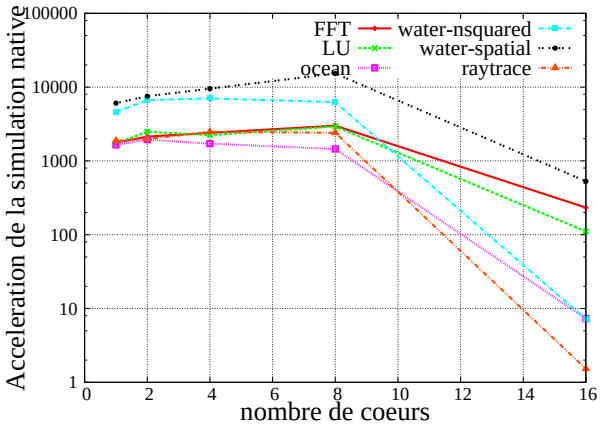


FIGURE 6.9 – Accélération de *basic_par_sim* par rapport à l'ISS de Kalray

temps, la VM la plus en retard, VM_l peut ne pas être exécutée par KVM car seulement 8 VM sur les 16 instanciées peuvent être exécutées simultanément.

Ce problème de contention est vérifié à l'aide d'une machine hôte ayant 16 cœurs hôtes. L'ISS et la simulation parallèle sont exécutées sur une machine ayant 2 socket Intel® Xeon® CPU E5520 tournant à 2.27 GHz avec Linux 3.10.0. La figure 6.10 présente l'accélération de *basic_par_sim* par rapport à l'ISS. La grande réduction de l'accélération n'est plus visible pour 16 PE.

Les autres propositions de simulation parallèle [WFWT13] rencontrent aussi ce problème de contention. Il est moins efficace d'utiliser notre ordonnanceur basique pour simuler plus de cœurs cibles qu'il n'y a de cœurs hôtes que d'utiliser l'approche séquentielle.

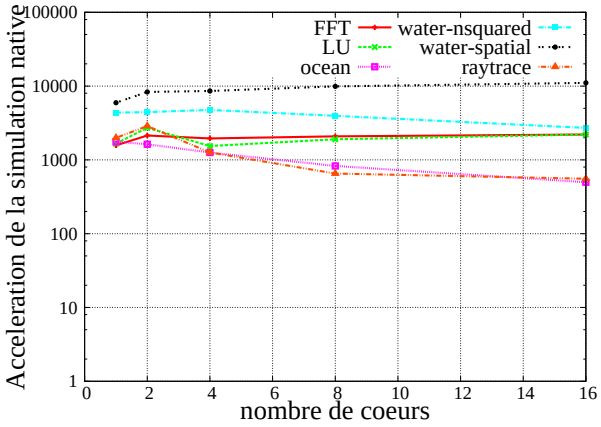


FIGURE 6.10 – Accélération de *basic_par_sim* par rapport à l'ISS de Kalray avec une machine hôte à 16 cœurs

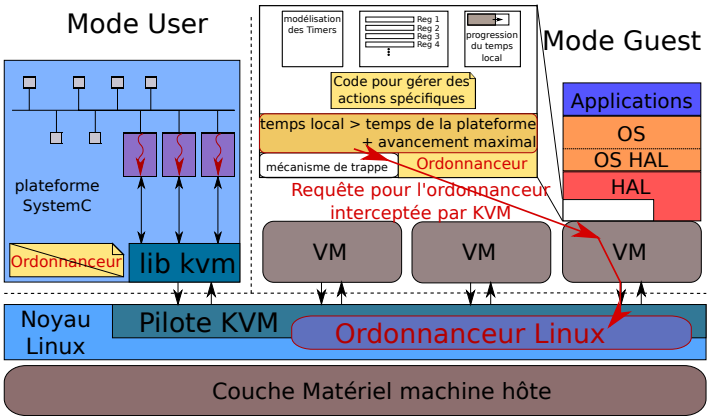


FIGURE 6.11 – Demande à l'ordonnanceur KVM de sélectionner une autre VM grâce au mécanisme de trappe

6.3.2 Gestion partielle des synchronisations

Pour améliorer la simulation parallèle, nous prenons inspiration de la paravirtualisation et modifions légèrement le driver KVM. Une trappe spéciale est ajoutée pour permettre de modifier l'ordonnancement des VM par KVM. Cette trappe réalise un changement du mode guest au mode kernel. KVM ne la propage pas ensuite jusqu'au mode user mais reconnait que cette trappe indique à l'ordonnanceur qu'il doit élire une autre VM. Quand une VM simule trop en avance par rapport aux autres, elle utilise cette trappe pour demander à KVM d'élire une autre VM à sa place au lieu d'exécuter de l'attente active comme illustré par la figure 6.11.

La figure 6.12 résume l'exécution d'une tâche associée à une VM à l'intérieur de l'ordonnanceur de Linux. Cette tâche y est gérée sans distinction des autres tâches même si certaines propositions ont été faites pour améliorer la gestion des VM [TQfB+10] en ajoutant une liste de priorités spécialement pour les VM. Quand une tâche exécutant une VM est créée, un quantum de temps hôte lui est associé et elle est insérée dans la liste *run_queue* du cœur sur lequel elle s'exécute. *run_queue* est divisée en 140 listes, chaque liste correspondant au niveau de priorité des tâches. La tâche (associée à la VM) est insérée à la bonne priorité à l'intérieur des listes des tâches ACTIVES. L'ordonnanceur de Linux va finalement la sélectionner pour être exécutée. Pendant son exécution, on va entrer en mode guest, exécuter la VM et à certains moments retourner en mode kernel. En fonction de la raison de sortie de la VM, différentes actions peuvent être faites : un *callback* en mode user ou gérer cette sortie directement en mode kernel. Si la raison de sortie correspond à notre demande d'élire une autre VM, le bit `TIF_NEED_RESCHED` du champs `flags` de la structure `thread_info` est passé à 1. L'appel à `need_resched` retournera alors *true*, ce qui conduit à un appel à `__schedule`. La tâche est déplacée de la liste des tâches ACTIVES à celle des tâches EXPIRED. Quand la liste ACTIVE est vide, les listes ACTIVE et EXPIRED sont échangées. La tâche exécutant la VM peut ensuite être sélectionnée de nouveau par l'ordonnanceur de Linux. Si la raison ayant entrainé la demande d'élire une autre VM est toujours valide (*e.g.* toujours dans l'état ATTENTE d'une barrière), une nouvelle requête pour être désélectionnée est lancée.

Comme le but de ces expériences n'est pas la précision temporelle, nous réduisons le contrôle sur les barrières et les verrous au minimum pour obtenir la borne supérieure des performances. Par exemple, un thread atteignant l'état ATTENTE d'une barrière va demander à KVM d'élire une autre VM.

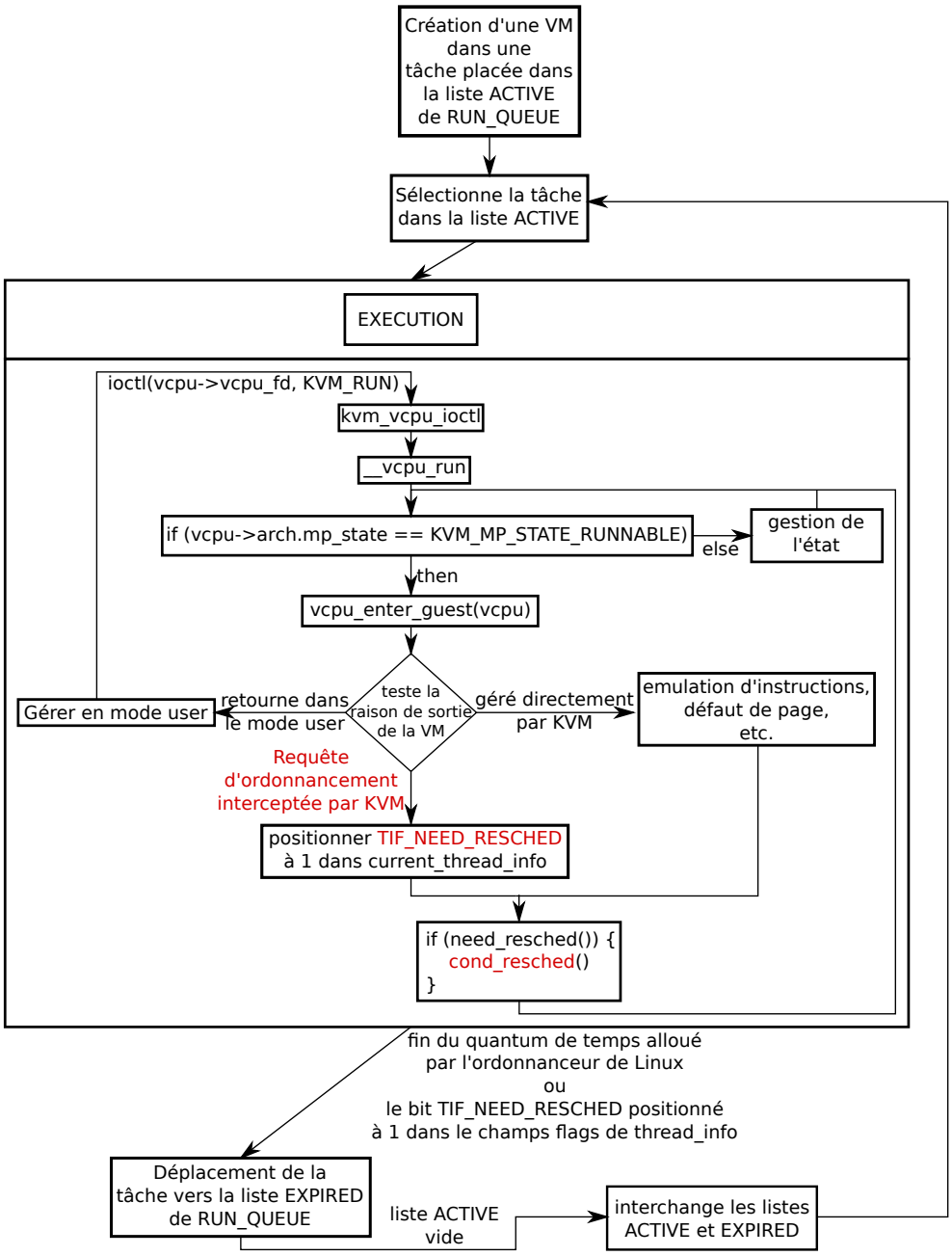


FIGURE 6.12 – Flot d'exécution de la tâche associée à une VM à l'intérieur de l'ordonnanceur de Linux

Experimentations La figure 6.13 présente l'accélération de la simulation parallèle utilisant des trappes pour contrôler l'ordonnancement (*advance_par_sim*) par rapport à l'ISS. Une réduction de l'accélération est encore présente avec 16 PE mais elle est bien moindre que précédemment.

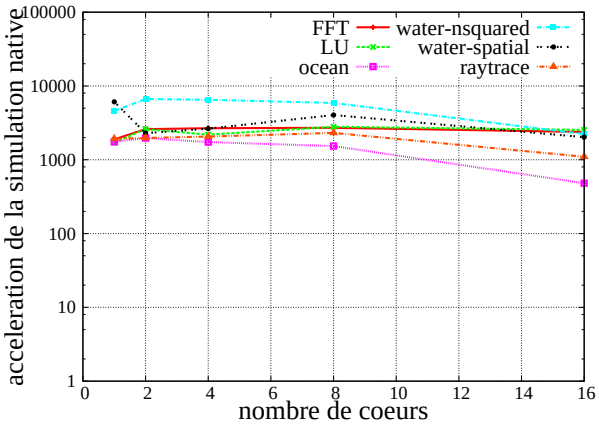


FIGURE 6.13 – Accélération de *advance_par_sim* par rapport à l'ISS de Kalray

La figure 6.14 compare *advance_par_sim* à *sync_seq_sim*. L'accélération pour certaines applications est plus grande que 1 pour seulement 1 PE simulé car la simulation parallèle, contrairement à la simulation séquentielle, n'a pas de trappe ajoutée pour la synchronisation. De plus il ne faut pas oublier que le cœur cible correspondant au RM est toujours simulé même s'il n'est pas comptabilisé dans le nombre de cœurs simulés. Il n'a en effet qu'un rôle de gestion et n'exécute pas les applications. Le problème de passage à l'échelle reste par contre toujours visible pour 16 PE.

De façon plutôt surprenante, car il contient seulement quelques barrières, *water-spatial* a des performances plus mauvaises avec la version parallèle. La raison semble être la suivante. Tous les threads n'atteignent pas la barrière au même moment. Certains threads, th_b, sont bloqués sur la barrière alors que d'autres, th_c, continuent de réaliser des calculs. Le problème arrive quand les threads th_s, $th_s \subset th_c$ sont bloqués par l'ordonnanceur (celui en mode guest) car ils sont trop en avance concernant le temps simulé. Les temps simulés des threads de th_b avancent lentement car ils génèrent des requêtes pour être désélectionnés, donc les threads dans th_s restent bloqués plus longtemps par l'ordonnanceur et de nombreuses requêtes pour être désélectionnés sont générées. Pour confirmer cette explication, l'application est simulée avec un avancement maximal infini pour empêcher l'ordonnanceur de bloquer un thread. Cette expérience est représentée par la ligne ayant pour légende *water-spatial infini* dans le graphique 6.14. Finalement le tableau 6.3 présente le nombre de requêtes pour élire une autre VM. *LU* et *water-spatial* illustrent le problème présenté avec une augmentation considérable du nombre de requêtes quand le nombre de PE augmente. Pour *raytrace*, le temps de simulation est significativement plus long que les autres applications ce qui explique pourquoi il y a tant de requêtes pour élire une autre VM.

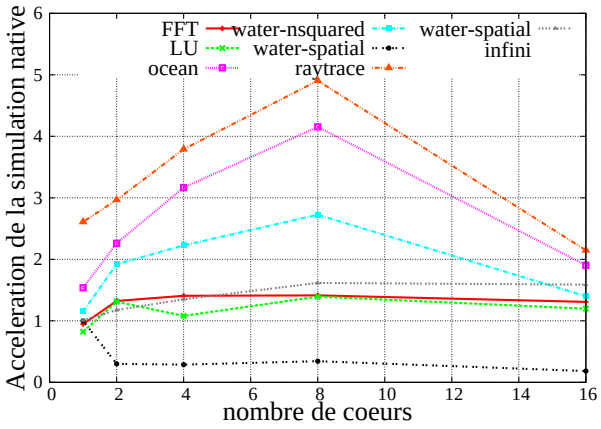


FIGURE 6.14 – Accélération de *advance_par_sim* par rapport à *sync_seq_sim*

TABLE 6.3 – Moyenne du nombre de requêtes pour être désélectionné sur 100 exécutions

	2 PE	4 PE	8 PE	16 PE
FFT	7427	95974	270888	536660
LU	113020	871156	2543076	4962326
ocean	3060	39067	230967	1341845
raytrace	908871	4361877	19877348	69568887
water-nsquared	10077	87428	458216	1614608
water-spatial	1517662	3314548	8493969	11684876

6.3.3 Vers une meilleure gestion des synchronisations

La simulation séquentielle génère trop de trappes pour gérer les synchronisations entre les threads. La simulation parallèle proposée souffre d'une mauvaise gestion lors de contention de ressources. En se basant sur les observations précédentes, une para-virtualisation plus avancée est nécessaire. L'ordonnanceur de nos VM devrait être intégré dans l'ordonnanceur du système d'exploitation hôte (Linux dans notre cas). Mais le principal problème de cette approche est de pouvoir maintenir ces changements à travers les évolutions du système d'exploitation hôte.

L'ensemble des éléments à ajouter sont les suivant :

— ajouter une liste UNSCHEDULED dans l'ordonnanceur de Linux.
— ajouter une trappe permettant de déplacer une VM de la liste ACTIVE à UNSCHEDULED.
— ajouter une trappe permettant de sélectionner les tâches UNSCHEDULED et capable de les replacer dans la liste ACTIVE.
— donner la connaissance de la progression du temps simulé des différents cœurs cibles à l'ordonnanceur de Linux lui permettant d'élire seulement les cœurs simulés les plus en retard dans le cas de contention de ressources.

Il existe déjà dans l'ordonnanceur Linux la possibilité de créer une liste pour les tâche attendant sur un événement. Il serait certainement possible d'utiliser directement ce mécanisme pour la liste UNSCHEDULED.

6.4 Conclusion

La granularité d'exécution des cœurs simulés force la simulation à utiliser la technique du découplage temporel. Il existe malheureusement un problème de passage à l'échelle dû à cette approche. Ce problème est renforcé pour la simulation native basée sur le HAV qui utilise des trappes pour communiquer entre la plateforme de simulation en mode user et le code cible s'exécutant à l'intérieur d'une VM en mode guest. Or ces trappes sont coûteuses.

Une première méthode qui vise à gérer la synchronisation entre les threads simulés est proposée. Le point de gestion choisi correspond aux variables partagées, plus particulièrement les procédures permettant de garantir la correction des accès mémoires à ces variables. Pour cela, du contrôle est incorporé dans les procédures bas niveau inclues dans le HAL pour gérer les fonctions de synchronisation (*e.g.* barrière, verrou). Cela améliore le passage à l'échelle. Mais le nombre de trappes pour contrôler l'ordre d'exécution des VM devient trop important et limite grandement son intérêt.

Partant de ce constat, nous avons conçu une seconde méthode dont le but est de réduire grandement le nombre de trappes de contrôle et place donc l'ordonnanceur dans le code en mode guest. La simulation des cœurs devient alors parallèle. Cette méthode apporte une solution partielle au problème de passage à l'échelle vu précédemment. Mais elle se heurte alors au problème de double niveau d'ordonnanceur : celui de Linux et le notre par dessus. On se retrouve alors avec des choix d'ordonnancement opposés entre les 2 ordonnanceurs pouvant conduire à une réduction très importante de la vitesse de simulation en cas de concurrence forte d'accès au cœur hôte par les VM. La solution pour cela serait de modifier l'ordonnanceur Linux pour qu'il n'ait pas seulement connaissance du temps hôte mais aussi connaissance du temps simulé pour sélectionner les tâches associées à une VM.

Chapitre 7

Conclusion

L'OBJECTIF global de cette thèse est de proposer une méthode permettant une simulation fonctionnelle la plus complète et rapide possible des cœurs d'une puce many-cœurs.

Dans l'introduction, nous avons vu que le nombre de transistors dans une puce a augmenté constamment. Même si la décennie à venir risque de voir la loi de Moore s'achever, du moins pour le CMOS, la complexité des systèmes actuels est telle que l'exploration architecturale ou le développement parallèle du design de la puce et du code applicatif est difficile. Pour cela, il est nécessaire de développer des plateformes virtuelles reproduisant le comportement de la machine cible. Avoir une bonne vitesse de simulation de ces plateformes est essentiel, notamment pour les systèmes many-cœurs et leur grand nombre de cœurs à simuler.

La simulation native est la technique de simulation des cœurs offrant la meilleure vitesse de simulation. L'adapter aux systèmes many-cœurs pose néanmoins certains problèmes. Utiliser d'autres techniques de simulation telle que la simulation compilée pose d'autres problèmes d'adaptation comme garantir la précision fonctionnelle de la FPU tout en maintenant les performances de simulation.

Dans la première partie de ce chapitre de conclusion, nous rappellerons les problématiques soulevées ainsi que les solutions apportées. Dans la partie suivante, nous définirons les perspectives de recherche mises en avant par les travaux présentés dans ce manuscrit.

7.1 Bilan

Les deux premières questions auxquelles nous avons apporté une réponse portent sur la correction fonctionnelle de la simulation. Nous avons montré qu'il était possible de simuler en simulation native des systèmes many-cœurs ayant des caractéristiques propres à la machine cible et non présentes sur le cœur hôte tout en garantissant des bonnes performances de simulation. Pour cela, l'accès à un espace mémoire supplémentaire est rendu possible en mode guest grâce à l'utilisation du support matériel à la virtualisation. Cet accès ne peut se faire qu'à travers le HAL et est donc caché des développeurs. Cet espace mémoire sert à stocker une description des cœurs cibles et de l'ensemble des composants inclus dans le composant cœur tel que des timers. Cela permet de simuler les particularités architecturales non présentes sur le cœur hôte tout en restant en mode guest, et donc d'éviter des changements de mode (mode guest vers mode kernel vers mode utilisateur) qui sont assez coûteux.

Un composant important des cœurs modernes, même dans le domaine des systèmes embarqués, est l'unité de calcul à virgule flottante (FPU). La FPU cible n'est pas forcément la même que la FPU hôte ce qui peut conduire à des différences de résultats dans les calculs si la FPU hôte est utilisée directement et sans précaution pour simuler la FPU cible. Nous avons limité notre étude à de la simulation compilée mais elle peut servir sans trop d'adaptation si l'on vise la traduction binaire statique. Nous avons proposé un découpage en 3 ensembles des instructions de calcul à virgule flottante cible.

L'ensemble des instructions implémentées à la fois sur le cœur cible et sur le cœur hôte forme le premier ensemble. Ces instructions cibles sont directement simulables par leur équivalent hôte moyennant quelques vérifications comme l'utilisation des mêmes formats sur les 2 architectures. Le deuxième ensemble est constitué des instructions conformes à la norme IEEE 754-rev08 mais non implémentées sur le cœur hôte. La révision 2008 de la norme introduit en particulier les opérations de *Fused Multiply and Add (FMA)*. Nous avons proposé une méthode d'émulation rapide du FMA simple précision en se basant sur des calculs en double précision et sur une détection d'une erreur de double arrondi permettant sa correction. Le dernier ensemble concerne les opérateurs exotiques. Ce sont les opérateurs définis spécialement pour le cœur cible mais sans être présents dans la norme IEEE 754 ou ses révisions. Nous proposons de remplacer les fonctions élémentaires qui utilisent ces opérateurs par leurs fonctions équivalentes hôtes si celles-ci sont définies dans des bibliothèques dont le comportement et la précision sont connues et respectées.

Nous garantissons ainsi qu'en simulation, les calculs en virgule flottante donnent les mêmes résultats numériques et soulèvent les mêmes signaux (*flags*) que sur la vraie puce tout en offrant une meilleure vitesse de simulation que la méthode basée sur l'émulation logicielle à base d'entiers. Les expériences ont montré qu'avec notre gestion des calculs à virgule flottante, une accélération de la simulation compilée de 1,4 à 2,4 par rapport à une émulation logicielle basée sur les entiers est atteinte pour les applications utilisées.

La dernière question est orientée performance de simulation. Nous avons constaté que la simulation native ne passait pas à l'échelle pour les applications cibles parallèles utilisant beaucoup de synchronisation entre les threads de l'application cible. Ce problème vient du découplage temporel inhérent à la simulation native qui amène la simulation à perdre du temps de simulation en simulant des instructions inutiles lors de procédures de synchronisation. Nous avons proposé une procédure de contrôle permettant de savoir où en est la procédure de synchronisation. L'obtention de ces informations permet de sélectionner la machine virtuelle à exécuter. Les expériences ont montré que le nombre de changements de mode devenait très élevé quand on augmente le nombre de cœurs simulés, réduisant rapidement le temps gagné par notre procédure.

Une deuxième approche était de déplacer l'ordonnanceur à l'intérieur des VM pour réduire le nombre de changements de mode, entraînant au passage la parallélisation de la simulation à l'intérieur d'une grappe. Cette méthode permet un meilleur passage à l'échelle mais elle se heurte à l'existence de deux niveaux d'ordonnanceur, celui de Linux et le nôtre, qui prennent parfois des décisions contraires. En cas de concurrence d'accès à la ressource cœur hôte, ce problème est très visible. Une transmission d'information de notre ordonnanceur à celui de Linux permet d'atténuer ce problème.

7.2 Perspectives

La première extension possible de ce travail est d'étendre le travail fait sur la FPU en simulation compilée à la simulation native. Pour cela, il faut établir une relation entre les blocs de base du binaire cible et ceux du binaire hôte, comme pour les annotations temporelles précises. Une fois cette relation obtenue, il faut pouvoir relier les instructions de calcul à virgule flottante utilisées dans le binaire natif aux instructions du binaire cible qu'ils représentent. Finalement une méthode permettant le remplacement des instructions de calcul à virgule flottante hôtes par une émulation garantissant l'équivalence de calcul doit être trouvée.

Une deuxième extension déjà abordée dans le chapitre 6 concerne l'ordonnanceur Linux. Il faudrait donner à l'ordonnanceur Linux l'accès au temps simulé des différentes VM. La combinaison de deux informations, le temps simulé par les VM et leur place dans les protocoles de synchronisation des threads cibles, permet de savoir quelle VM peut avancer dans la simulation d'un cœur cible sans générer d'erreur de causalité ou simuler inutilement des instructions. Il faudrait permettre à l'ordonnanceur du système d'exploitation hôte d'accéder à ces informations pour élire correctement les VM.

Nous avions fait l'hypothèse réaliste par rapport à notre cas d'étude qu'il n'y avait qu'un seul thread par cœur cible. Une extension de notre étude du passage à l'échelle de la simulation native et des problèmes de synchronisation qu'elle engendre, à une simulation autorisant plusieurs threads par cœur cible serait souhaitable pour traiter des cas plus généraux.

L'ensemble de nos expériences a été effectué avec une notion complétement arbitraire du temps. Une insertion d'annotations temporelles précises serait souhaitable, notamment pour mieux observer l'effet du découplage temporel sur l'accès aux ressources partagées. Néanmoins ce n'est pas une tâche triviale. Il serait donc intéressant de trouver un moyen facile de porter l'insertion d'annotations d'une architecture à une autre.

Conférences et Publications

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

Conférence national sans publication

1. G. Sarrazin, N. Fournel, P. Gerin, and F. Pétrot. Simulation native de systèmes many-cœurs pouvant avoir des caractéristiques architecturales non génériques. In *ComPAS*, 2014

Journal national

1. G. Sarrazin, N. Fournel, P. Gerin, and F. Pétrot. Simulation native basée sur le support matériel à la virtualisation cas des systèmes many-cœurs spécifiques. In *Technique et Science Informatiques (TSI)*, volume 34, pages 153–173, 2015

Conférence internationale

1. Guillaume Sarrazin, Nicolas Brunie, and Frédéric Pétrot. Virtual prototyping of floating point units. In *Proceedings of the 2016 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '16, pages 1:1–1:6, New York, NY, USA, 2016. ACM

Bibliographie

[BF07] S. Boldo and J.-C. Filliatre. Formal verification of floating-point programs. In *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pages 187–194, June 2007.

[BGP09] A. Bouchhima, P. Gerin, and F. Pétrot. Automatic instrumentation of embedded software for high level hardware/software co-simulation. In *Proceedings of the 14th Asia and South Pacific Design Automation Conference*, pages 546–551, 2009.

[BKL+00] Jwahar R. Bammi, Wido Kruijtzer, Luciano Lavagno, Edwin Harcourt, and Mihai T. Lazarescu. Software performance estimation strategies in a system-level design tool. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign*, CODES '00, pages 82–86, New York, NY, USA, 2000. ACM.

[BM08] S. Boldo and G. Melquiond. Emulation of a fma and correctly rounded sums : Proved algorithms using rounding to odd. *IEEE Transactions on Computers*, 57(4) :462–471, April 2008.

[BR05] P. Belanovic and M. Rupp. Automated floating-point to fixed-point conversion with the fixify environment. In *Rapid System Prototyping, 2005. (RSP 2005). The 16th IEEE International Workshop on*, pages 172–178, June 2005.

[Bru14] Nicolas Brunie. *Contributions to computer arithmetic and applications to embedded systems*. Theses, Ecole normale supérieure de lyon - ENS LYON, May 2014.

[CCZ06] B. Chopard, P. Combes, and J. Zory. A conservative approach to systemc parallelization. In VassilN. Alexandrov, GeertDick van Albada, PeterM.A. Sloot, and Jack Dongarra, editors, *Computational Science – ICCS 2006*, volume 3994 of *Lecture Notes in Computer Science*, pages 653–660. Springer Berlin Heidelberg, 2006.

[CD13] Weiwei Chen and Rainer Dömer. Optimized out-of-order parallel discrete event simulation using predictions. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 3–8, San Jose, CA, USA, 2013. EDA Consortium.

[CDDP10] Sylvain Collange, Marc Daumas, David Defour, and David Parello. Barra : A parallel functional simulator for gpgpu. In *2010 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, pages 351–360. IEEE, 2010.

[CHD12] Weiwei Chen, Xu Han, and Rainer Dömer. Out-of-order parallel simulation for esl design. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, pages 141–146, San Jose, CA, USA, 2012. EDA Consortium.

[CP00] Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2-3) :249–274, 2000.

[CPVM10] Juan Castillo, Hector Posadas, Eugenio Villar, and Marcos Martinez. Fast instruction cache modeling for approximate timed hw/sw co-simulation. In *Proceedings of the 20th Symposium on Great Lakes Symposium on VLSI*, GLSVLSI '10, pages 191–196, New York, NY, USA, 2010. ACM.

[DCHG11] R. Domer, Weiwei Chen, Xu Han, and A. Gerstlauer. Multi-core parallel simulation of system-level description languages. In *Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific*, pages 311–316, Jan 2011.

[DdDAB+13] Benoît Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benoît Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, and Thierry Strudel. A clustered manycore processor architecture for embedded and accelerated applications. In *IEEE High Performance Extreme Computing Conference*, pages 1–6. IEEE, 2013.

[DPV10] Luis Díaz, Hector Posadas, and Eugenio Villar. Obtaining memory address traces from native co-simulation for data cache modeling in systemc. In *XXV Conference on Design of Circuits and Integrated Systems (DCIS-2010)*, 2010.

[FFP15] A. Faravelon, N. Fournel, and F. Petrot. Fast and accurate branch predictor simulation. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2015*, pages 317–320, March 2015.

[FS04] Gene Frantz and Ray Simar. Comparing fixed-and floating-point dsps. *Texas Instruments, Dallas, TX, USA*, 2004.

[Fuj90] Richard M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10) :30–53, 1990.

[GBR15] Christoph Gerum, Oliver Bringmann, and Wolfgang Rosenstiel. Source level performance simulation of gpu cores. In *Design, Automation & Test in Europe Conference*, pages 217–222, March 2015.

[Ger09] P. Gerin. *Modèles de simulation pour la validation logicielle et l'exploration d'architectures des systèmes multiprocesseurs sur puce*. Thèse, Institut National Polytechnique de Grenoble - INPG, November 2009.

[GGP08] P. Gerin, X. Guérin, and F. Pétrot. Efficient implementation of native software simulation for mpsoc. In *Proceedings of the 11th Design, Automation and Test in Europe Conference*, pages 676–681, 2008.

[GKK+08] L. Gao, K. Karuri, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. Multiprocessor performance estimation using hybrid simulation. In *Proceedings of the 45th Design Automation Conference*, pages 325–330, 2008.

[GKL+07] Lei Gao, Stefan Kraemer, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. A fast and generic hybrid simulation approach using c virtual machine. In *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '07, pages 3–12, New York, NY, USA, 2007. ACM.

[GKO+00] Jeff Gibson, Robert Kunz, David Ofelt, Mark Horowitz, John Hennessy, and Mark Heinrich. Flash vs. (simulated) flash : Closing the simulation loop. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–58, 2000.

[GLMS02] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System design with SystemC*. Springer, 2002.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1) :5–48, March 1991.

[GSC+07] P. Gerin, Hao Shen, A. Chureau, A. Bouchhima, and A.A. Jerraya. Flexible and executable hardware/software interface modeling for multiprocessor soc design using systemc. In *Proceedings of the 12th Asia and South Pacific Design Automation Conference*, pages 390–395, 2007.

[HAG08] Yonghyun Hwang, Samar Abdi, and Daniel Gajski. Cycle-approximate retargetable performance estimation at the transaction level. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 3–8, New York, NY, USA, 2008. ACM.

[Ham13] Mian Muhammad Hamayun. *Native Simulation of Multi-Processor System-on-Chip using Hardware-Assisted Virtualization*. These, Université de Grenoble, October 2013. ISBN : 978-2-11-129179-9.

[HCG+13] Claude Helmstetter, Jérôme Cornet, Bruno Galilée, Matthieu Moy, and Pascal Vivet. Fast and accurate TLM simulations using temporal decoupling for fifo-based communications. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 1185–1188, San Jose, CA, USA, 2013. EDA Consortium.

[JA15] J. Jain and R. Agrawal. Design and development of efficient reversible floating point arithmetic unit. In *Communication Systems and Network Technologies (CSNT), 2015 Fifth International Conference on*, pages 811–815, April 2015.

[JWPB05] Christian Jacobi, Kai Weber, Viresh Paruthi, and Jason Baumgartner. Automatic formal verification of fused-multiply-add fpus. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*, DATE '05, pages 1298–1303, Washington, DC, USA, 2005. IEEE Computer Society.

[KGW+07] S. Kraemer, L. Gao, J. Weinstock, R. Leupers, G. Ascheid, and H. Meyr. Hysim : A fast simulation framework for embedded software development. In *Proceedings of the 5th International Conference on Hardware/Software Codesign and System Synthesis*, pages 75–80, 2007.

[KLLM12] Peter Kornerup, Vincent Lefèvre, Nicolas Louvet, and Jean-Michel Muller. On the computation of correctly rounded sums. *IEEE Transactions on Computers*, 61(3) :289–298, March 2012.

[Knu69] Donald E Knuth. Seminumerical algorithms. The art of computer programming, 1969.

[KPA+14] U. Krautz, V. Paruthi, A. Arunagiri, S. Kumar, S. Pujar, and T. Babinsky. Automatic verification of floating point units. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, pages 1–6, June 2014.

[Lan61] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3) :183–191, July 1961.

[Lan08] R Lantz. Fast functional simulation with parallel embra. In *Proceedings of the 4th Annual Workshop on Modeling, Benchmarking and Simulation*. Citeseer, 2008.

[LLT10] Kai-Li Lin, Chen-Kang Lo, and Ren-Song Tsay. Source-level timing annotation for fast and accurate tlm computation model generation. In *Proceedings of the 15th Asia and South Pacific Design Automation Conference*, pages 235–240. IEEE, 2010.

[LMGS13a] Kun Lu, D. Muller-Gritschneder, and U. Schlichtmann. Memory access reconstruction based on memory allocation mechanism for source-level simulation of embedded software. In *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, pages 729–734, Jan 2013.

[LMGS13b] Kun Lu, Daniel Muller-Gritschneder, and Ulf Schlichtmann. Analytical timing estimation for temporally decoupled tlms considering resource conflicts. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 1161–1166, March 2013.

[LMGS13c] Kun Lu, Daniel Muller-Gritschneder, and Ulf Schlichtmann. Fast cache simulation for host-compiled simulation of embedded software. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 637–642, March 2013.

[MAF91] Christopher Mills, Stanley C Ahalt, and Jim Fowler. Compiled instruction set simulation. *Software : Practice and Experience*, 21(8) :877–889, 1991.

[MDMM13] Érik Martin-Dorel, Guillaume Melquiond, and Jean-Michel Muller. Some issues related to double rounding. *BIT Numerical Mathematics*, 53(4) :897–924, 2013.

[MEJ+12] L.G. Murillo, J. Eusse, J. Jovic, S. Yakoushkin, R. Leupers, and G. Ascheid. Synchronization for hybrid mpsoc full-system simulation. In *Proceedings of the 49th Design Automation Conference*, pages 121–126, 2012.

[MKK+10] J.E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite : A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, Jan 2010.

[MMGP10] A. Mello, I. Maia, A. Greiner, and F. Pecheux. Parallel simulation of systemc tlm 2.0 compliant mpsoc on smp workstations. In *Proceedings of the 13th Design, Automation Test in Europe Conference*, pages 606–609, 2010.

[Mon08] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3) :12 :1–12 :41, May 2008.

[Moy13] Matthieu Moy. Parallel programming with systemc for loosely timed models : A non-intrusive approach. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 9–14, San Jose, CA, USA, 2013. EDA Consortium.

[MRRJ05] Anish Muttreja, Anand Raghunathan, Srivaths Ravi, and Niraj K. Jha. Hybrid simulation for embedded software energy estimation. In *Proceedings of the 42nd Design Automation Conference*, DAC '05, pages 23–26, 2005.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind : a framework for heavy-weight dynamic binary instrumentation. In *ACM Sigplan notices*, pages 89–100. ACM, 2007.

[PCC+09] Ezudheen P, Priya Chandran, Joy Chandra, Biju Puthur Simon, and Deepak Ravi. Parallelizing systemc kernel for fast hardware simulation on smp machines. In *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, PADS '09, pages 80–87, Washington, DC, USA, 2009. IEEE Computer Society.

[PDV11] Héctor Posadas, Luis Díaz, and Eugenio Villar. Fast data-cache modeling for native co-simulation. In *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, ASPDAC '11, pages 425–430, Piscataway, NJ, USA, 2011. IEEE Press.

[PV09] Héctor Posadas and Eugenio Villar. Automatic hw/sw interface modeling for scratch-pad and memory mapped hw components in native source-code co-simulation. In Achim Rettberg, MauroC. Zanella, Michael Amann, Michael Keckeisen, and FranzJ. Rammig, editors, *Analysis, Architectures and Modelling of Embedded Systems*, volume 310 of *IFIP Advances in Information and Communication Technology*, pages 12–23. Springer Berlin Heidelberg, 2009.

[RHT12] Sascha Roloff, Frank Hannig, and Jürgen Teich. Fast architecture evaluation of heterogeneous mpsocs by host-compiled simulation. In *Proceedings of the 15th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '12, pages 52–61, New York, NY, USA, 2012. ACM.

[RI00] John S Robin and Cynthia E Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium*, pages 129—-144. Usenix, 2000.

[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.

[RSHT15] S. Roloff, D. Schafhauser, F. Hannig, and J. Teich. Execution-driven parallel simulation of pgas applications on heterogeneous tiled architectures. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pages 1–6, June 2015.

[SBP16] Guillaume Sarrazin, Nicolas Brunie, and Frédéric Pétrot. Virtual prototyping of floating point units. In *Proceedings of the 2016 Workshop on Rapid Simulation and Performance Evaluation : Methods and Tools*, RAPIDO '16, pages 1 :1–1 :6, New York, NY, USA, 2016. ACM.

[SD15] S. Salehi and R.F. DeMara. Energy and area analysis of a floating-point unit in 15nm cmos process technology. In *SoutheastCon 2015*, pages 1–5, April 2015.

[SFGP14] G. Sarrazin, N. Fournel, P. Gerin, and F. Pétrot. Simulation native de systèmes many-cœurs pouvant avoir des caractéristiques architecturales non génériques. In *ComPAS*, 2014.

[SFGP15] G. Sarrazin, N. Fournel, P. Gerin, and F. Pétrot. Simulation native basée sur le support matériel à la virtualisation cas des systèmes many-cœurs spécifiques. In *Technique et Science Informatiques (TSI)*, volume 34, pages 153–173, 2015.

[SHP12] Hao Shen, M. Hamayun, and F. Pétrot. Native simulation of MPSoC using hardware-assisted virtualization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(7) :1074–1087, 2012.

[SLPH10] Christoph Schumacher, Rainer Leupers, Dietmar Petras, and Andreas Hoffmann. parsc : Synchronous parallel systemc simulation on multi-core host architectures. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, pages 241–246, New York, NY, USA, 2010. ACM.

[SS12] E.E. Swartzlander and H.H. Saleh. Fft implementation with fused floating-point operations. *Computers, IEEE Transactions on*, 61(2) :284–288, Feb 2012.

[SSCZ13] Xiang Song, Jicheng Shi, Haibo Chen, and Binyu Zang. Schedule processes, not vcpus. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, APSys '13, pages 1 :1–1 :7. ACM, 2013.

[TQfB+10] Ding Tao, Hao Qin-fen, Zhang Bing, Zhang Tie-gang, and Huai Li-ting. Scheduling policy optimization in kernel-based virtual machine. In *Computational Intelligence and Software Engineering (CiSE), 2010 International Conference on*, pages 1–4, Dec 2010.

[vM96] Vojin Živojnovic and Heinrich Meyr. Compiled hw/sw co-simulation. In *Proceedings of the 33rd Annual Design Automation Conference*, DAC '96, pages 690–695, New York, NY, USA, 1996. ACM.

[WFWT09] Meng-Huan Wu, Cheng-Yang Fu, Peng-Chih Wang, and Ren-Song Tsay. An effective synchronization approach for fast and accurate multi-core instruction-set simulation. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pages 197–204, New York, NY, USA, 2009. ACM.

[WFWT13] Meng-Huan Wu, Cheng-Yang Fu, Peng-Chih Wang, and Ren-Song Tsay. A distributed timing synchronization technique for parallel multi-core instruction-set simulation. *ACM Transactions on Embedded Computing Systems*, 12(1s) :54, 2013.

[WH09] Zhonglei Wang and Andreas Herkersdorf. An efficient approach for system-level timing simulation of compiler-optimized embedded software. In *Proceedings of the 46th DAC*, pages 220–225. ACM, 2009.

[WH12] Zhonglei Wang and Jörg Henkel. Hycos : hybrid compiled simulation of embedded software with target dependent code. In *Proceedings of CODES+ISSS*, pages 133–142. ACM, 2012.

[WH13] Zhonglei Wang and Jörg Henkel. Fast and accurate cache modeling in source-level simulation of embedded software. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 587–592, San Jose, CA, USA, 2013. EDA Consortium.

[WLC+11] Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. Coremu : A scalable and portable parallel full-system emulator. *SIGPLAN Not.*, 46(8) :213–222, February 2011.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs : Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 24–36, 1995.

[WSH08] Zhonglei Wang, Antonio Sanchez, and Andreas Herkersdorf. Scisim : A software performance estimation framework using source code instrumentation. In *Proceedings of the 7th International Workshop on Software and Performance*, WOSP '08, pages 33–42, New York, NY, USA, 2008. ACM.

[WSL+14] J.H. Weinstock, C. Schumacher, R. Leupers, G. Ascheid, and L. Tosoratto. Time-decoupled parallel systemc simulation. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–4, March 2014.

[WWFT11] Meng-Huan Wu, Peng-Chih Wang, Cheng-Yang Fu, and Ren-Song Tsay. A high-parallelism distributed scheduling mechanism for multi-core instruction-set simulation. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 339–344, New York, NY, USA, 2011. ACM.

[YGG03] Haobo Yu, Andreas Gerstlauer, and Daniel Gajski. RTOS scheduling in transaction level models. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 31–36. ACM, 2003.

[YZH+13] Fan-Wei Yu, Bo-Han Zeng, Yu-Hung Huang, Hsin-I Wu, Che-Rung Lee, and Ren-Song Tsay. A critical-section-level timing synchronization approach for deterministic multi-core instruction set simulations. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 643–648, San Jose, CA, USA, 2013. EDA Consortium.

[ZG02] Jianwen Zhu and Daniel D. Gajski. An ultra-fast instruction set simulator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(3) :363–373, June 2002.

Résumé Le nombre de transistors dans une puce augmente constamment en suivant la conjecture de Moore, qui dit que le nombre de transistor dans une puce double tous les 2 ans. On arrive donc aujourd'hui à des systèmes d'une telle complexité que l'exploration architecturale ou le développement, même parallèle, de la conception de la puce et du code applicatif prend trop de temps. Pour réduire ce temps, la solution généralement admise consiste à développer des plateformes virtuelles reproduisant le comportement de la puce cible. Avoir une haute vitesse de simulation est essentiel pour ces plateformes, notamment pour les systèmes many-cœurs à cause du grand nombre de cœurs à simuler. Nous nous focalisons donc dans cette thèse sur la simulation native, dont le principe est de compiler le code source directement pour l'architecture hôte, offrant ainsi un temps de simulation que l'on peut espérer optimal.

Mais un certain nombre de caractéristiques fonctionnelles spécifiques au cœur cible peuvent ne pas être présentes sur le cœur hôte. L'utilisation de l'assistance matérielle à la virtualisation (HAV) comme base pour la simulation native vient renforcer la dépendance de la simulation du cœur cible par rapport aux caractéristiques du cœur hôte. Nous proposons dans ce contexte un moyen de simuler les caractéristiques fonctionnelles spécifiques du cœur cible en simulation native basée sur le HAV.

Parmi les caractéristiques propres au cœur cible, l'unité de calcul à virgule flottante est un élément important, bien trop souvent négligé en simulation native conduisant certains calculs à donner des résultats différents entre le cœur cible et le cœur hôte. Nous nous restreignons au cas de la simulation compilée et nous proposons une méthodologie permettant de simuler correctement les opérations de calcul à virgule flottante.

Finalement la simulation native pose des problèmes de passage à l'échelle. Des problèmes de découplage temporel amènent à simuler inutilement certaines instructions lors de procédures de synchronisation entre des tâches s'exécutant sur les cœurs cibles, conduisant à une réduction de la vitesse de simulation. Nous proposons des solutions pour permettre un meilleur passage à l'échelle de la simulation native.

Abstract The number of transistors in one chip is increasing following Moore's conjecture which says that the number of transistors per chip doubles every two years. Current systems are so complex that chip design and specific software development for one chip take too much time even if software development is done in parallel with the design of the hardware architecture, often because of system integration issues. To help reducing this time, the general solution consists of using virtual platforms to reproduce the behavior of the target chip. The simulation speed of these platforms is a major issue, especially for many-core systems in which the number of programmable cores is really high. We focus in this thesis on native simulation. Its principle is to compile source code directly for the host architecture to allow very fast simulation, at the cost of requiring "equivalent" features on the target and host cores.

However, some target core specific features can be missing in the host core. Hardware Assisted Virtualization (HAV) is used to ease native simulation but it reinforces the dependency of the target chip simulation regarding the host core capabilities. In this context, we propose a solution to simulate the target core functional specific features with HAV based native simulation.

Among target core features, the floating point unit is an important element which is neglected in native simulation leading to potential functional differences between target and host computation results. We restrict our study to the compiled simulation technique and we propose a methodology ensuring to accurately simulate floating point computations while still keeping a good simulation speed.

Finally, native simulation has a scalability issue. Time decoupling problems generate unnecessary code simulation during synchronisation protocols between threads executed on the target cores, leading to an important decrease of simulation speed when the number of cores grows. We address this problem and propose solutions to allow a better scalability for native simulation.

ISBN : 978-2-11-129214-7