Placement de tâches dynamique et flexible sur processeur multicoeur asymétrique en fonctionnalités

Alexandre Aminot

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Alexandre Aminot

Thèse dirigée par **Henri-Pierre Charles**

préparée au sein **CEA List**
et de **Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Placement de tâches dynamique et flexible sur processeur multi-coeur asymétrique en fonctionnalités

Thèse soutenue publiquement le **01/10/2015**,
devant le jury composé de :

Mr, Quema, Vivien
Professeur, Grenoble INP / ENSIMAG, Président
Mr, Gogniat, Guy
Professeur, Université de Bretagne-Sud - UEB, Rapporteur
Mr, Pegatoquet, Alain
Maître de Conférences, Université de Nice Sophia Antipolis, Rapporteur
Mr, Cohen, Albert
Directeur de recherche, INRIA-ENS, Examinateur
Mr, Dupont de Dinechin, Benoit
Directeur de la technologie, Kalray, Examinateur
Mme, Heydemann, Karine
Maître de Conférences, Université Pierre et Marie Curie , Examinatrice
Mr, Charles, Henri-Pierre
Directeur recherche, CEA LIST , Directeur de thèse
Mr, Lhuillier, Yves
Ingénieur Chercheur, CEA LIST, Encadrant CEA
Mr, Castagnetti, Andrea
Ingénieur Chercheur, CEA LIST, Encadrant CEA, Invité

Table des matières

Table des figures

Liste des tableaux

Remerciements

Pour commencer, je souhaiterais remercier les rapporteurs, pour le temps qu'ils ont accordé à la lecture de cette thèse et à l'élaboration de leur rapport très détaillés. Je remercie Guy Gogniat et Alain Pegatoquet d'avoir accepté cette charge et de l'intérêt qu'ils ont porté à mes travaux. Je remercie de plus Alain, pour avoir relevé toutes les typographies et fautes dans le document, je lui en suis très reconnaissant.

Je souhaiterai également remercier Viven Quema le président du jury de thèse et le jury de thèse, composé de Karine Heydemann, Albert Cohen et Benoit Dupont de Dinechin. Merci pour vos retours sur mes travaux, cela m'est très important.

Je remercie le CEA de m'avoir accueillie dans sa structure pour permettre de faire mes preuves en tant que chercheur. Merci à Raphael David, directeur du LCE, qui a souvent été à mon écoute.

Un grand merci à mes encadrants Yves Lhuillier, Andrea Castagnetti, Alain Chataigner et mon directeur de thèse Henri-Pierre Charles.

Yves, merci pour la liberté (non voulu ?) que tu m'as laissé. Les discussions informelles avec toi ont toujours été très productives. Merci d'avoir été toujours présent lorsque j'en avais besoin (et que je réussissais à te capturer dans les couloirs :)). # pommesfrites # pommessautées

Andrea, plus qu'un encadrant, tu es devenu un ami. Nos célèbres *despé+chips guacamole* me manquent ! Pouvoir partager cette thèse avec toi fut très enrichissant. Tes remarques et critiques ont toujours été très pertinentes. Merci de m'avoir beaucoup challengé pendant la deuxième année, tu as réussi à valoriser encore plus mes travaux. Merci d'avoir passé avec moi ces derniers moments pour la rédaction du mémoire. Tes retours ont amélioré la qualité du manuscrit d'une façon immensurable ! Maintenant merci d'être mon ami et de partager l'escalade, la course à pied, les aventures en montagnes et les soirées.

Alain, merci de m'avoir accompagné la première partie de la thèse, tes compétences techniques et ta précision m'inspireront toujours.

Henri-Pierre, merci m'avoir accompagné tout au long de ces 3 ans. Merci de m'avoir supporté dans tous les projets et idées que j'ai essayé de lancer. Je te remercie de m'avoir accueilli à Grenoble pour finaliser la rédaction. Sans cette période qui m'a permis d'être productif et d'échanger avec toi, la fin de rédaction aurait été très compliquée.

Merci à Omar Hammami pour m'avoir fait confiance pendant deux ans pour l'encadrement des TPs de l'ES201 à l'ENSTA. Merci à Patrick Garda pour m'avoir fait confiance lors de l'encadrement des TPs d'architecture système à l'université Paris 6.

De plus, je souhaite remercier aussi tous ceux qui ont contribué au bon cheminement de cette thèse : Merci à Fernando Endo pour nos échanges et la mise en place des outils de simulation collaboratifs intra laboratoires. Merci à Matthieu Jan, Jean-Thomas Acquaviva et Loic

Cudennec pour nos discussions et vos retours sur mes travaux de thèse. (Merci aussi à Loic pour l'inspiration musicale.)

Un énorme merci à Annie Strabonie, la maman du département DACLE (qui donne même des coups de pied aux fesses...). La magicienne de l'administration, merci pour m'avoir aidé pendant 3 ans. Nous finissons ensemble notre contrat au CEA, cela me fait plaisir de partager ça avec toi. Le futur va nous être très nouveau et incertain, c'est excitant !

Merci à toutes les personnes du DACLE avec qui j'ai discuté et échangé au cours de ces 3 ans.

Un merci spécial pour Suresh Pajaniradja, alias "le mec qui à tout fait dans la vie". On a partagé des super moments autour de la photographie, la montagne et le karaté ! Tu es toujours là pour apporter tes compétences et donner de très bons conseils. Merci.

Merci à l'équipe Mehdi Darouich, Stephane Chevobbe, Erwan Piriou, Alexandre Chariot et Benjamin Chomarat. Tous ces repas passés avec vous étaient toujours plus n'importe quoi les uns que les autres, j'en garde de très bons souvenirs !

Merci à Charly Bechara pour tous nos échanges sur la thèse, l'innovation et l'entrepreneuriat ;).

Merci aussi à Alexandre Guerre, Alexandre Carbon, Tanguy Sassolas, Nicolas Ventroux et Stephane Louise. Merci à Rachid Dafali, les discussions avec toi au début de thèse m'ont beaucoup motivé. Merci à l'équipe du relais course à pied du laboratoire : Chiara Sandionigi, Lilia Zaourar, Safae Dahmani, Thomas Peyret, Benoit Tin, Karim Ben Chehida et Caaliph Andriamisaina.

Un énorme merci et clin d'oeil à tous les doctorants avec qui j'ai partagé ces 3 ans : Safae Dahmani, la princesse du LCE, Aziz Dziri, avec qui on se comprend tellement que c'est toujours agréable de partager nos histoires de vie, et Karl-Edouard Berger, le célèbre président de l'association des doctorants du CEA IDF ;). Merci aussi aux anciens doctorants Julien Peeters et Mathieu Texier, qui m'ont transmis connaissances et astuces qui m'ont bien aidé dans ma thèse.

Un merci à tous les acteurs du bureau 2040. Merci à Vincent, Boukary O., Julien Collet, qui ont été mes derniers co-bureau. Bon courage à Vincent et Julien pour finir vos thèses. Vu l'humour que vous apportez dans le bureau, je suis sur que vos thèse finiront avec le sourire :).

Un merci aussi (aux stagiaires) : Marvin Faix, avec le célèbre mini-panier de basket, qui a permis de nous défouler et tenir la distance. Et à David Cohen, avec qui j'ai continué de partager un projet personnel de développement hors du laboratoire.

Merci aux autres connaissances des laboratoires connexes, tel que Sergiu Carpov, Oana Stan, Cyril Faure, Pascal Albéric et Khanh Do Xuan avec qui on a bien rigolé.

Une petite dédicace à la dame de l'accueil de Nano Innov, qui utilisait une voix sensuelle juste pour annoncer qu'une voiture avait oublié d'éteindre ses feux. Son bonjour joyeux rendait la journée tout de suite agréable.

Merci au labo du LIALP de m'avoir accueilli pendant un mois à Grenoble pour finaliser la rédaction. Avec un merci spécial pour l'équipe de grimpe : Caroline Queva , Mathieu Barbe, Maxime Louvel et Rémi Druilhe.

Enfin, la thèse a été réussi en partie grâce à la vie extérieure au laboratoire qui m'a permis de décompresser, de m'inspirer et de tenir ces 3 ans. Merci aux personnes qui ont partagé ma vie à coté de cette thèse et qui m'ont supporté dans ce que j'ai fait.

En particulier, je souhaiterai remercier mes parents, ma soeur et mon frère (et à *Wifi* leur dalmatien qui est toujours joyeuse !). Je me rappellerai toujours que quand j'étais jeune, je voulais faire entomologiste, mais vous m'aviez dit qu'il fallait être docteur et que ça allait être

dur (surtout vu les notes que j'avais à l'école...). Du coup j'ai naturellement poussé vers ma deuxième passion de l'époque, l'informatique. Et maintenant voilà je que j'obtiens mon doctorat en informatique... comme quoi j'aurais peut-être pu peut-être faire entomologiste finalement... Dans tous les cas, ça reste souvent de la gestion de *bug*. En tout cas, merci pour votre soutien moral tout au long de mes études. Un grand merci d'avoir subvenu à mes besoins lors de l'école d'ingénieur pour me permettre de me concentrer sur mes études et non sur comment payer mon loyer. Votre amour pour mon frère, ma soeur et moi-même m'inspira toujours. Merci pour tout ce que vous nous donnez.

Je souhaiterai remercier aussi toute la *la brigade du kiff* (Jeremy Bergeret, Aurelien Chevalarias, Lucas Oliveri, David Peyrassou et Alexis Sevestre) avec qui j'ai passé des moments magiques et qui j'espère, va continuer le plus longtemps possible. Merci d'avoir poussé à faire des Week-ends et vacances ensemble. Merci au temps passer à organiser tous ça. Tous ces moments m'ont permis de finir ma thèse sans burnout.

Merci à Maxime François, Adrien Taffanel, deux amis très chers qui ont chacun trouvé leur passion et avec qui les discussions peuvent toujours être sans fin. Vous m'inspirez les gars ! Merci.

Merci à Virginie Salaün, tu as toujours été là quand j'en ai eu besoin. Merci énormément pour tout le temps que tu m'as donné, pour ton écoute et pour tes conseils.

Merci à la team de sportifs Grenoblois Julien Guepet, Bastien Deshayes et Clément Léger, qui m'ont hébergés plus d'une fois à Grenoble (un peu trop à l'improviste des fois, désolé) et avec qui j'ai pu décompresser en montagne et dans les bars. Merci aussi à Felipe Chies pour tous les moments passés ensemble.

Merci à Lucie M. de m'avoir supporté pendant une bonne partie de la thèse. Ton attitude très reposante calmait souvent mon énergie un peu trop débordante.

Merci à mes semi-coloc Émilie et Martin, avec qui j'ai pu partager toutes mes histoires de thèse, d'entrepreneuriat et de soirées. Je suis tellement reconnaissant de votre soutien.

Thanks to Begüm E. Thanks you for all your support and the time you gave me. You really are an unique woman. I wish you the best for your thesis. Your are so harsh with yourself that I am sure your thesis will be a very high quality thesis.

Merci à Andreï klochko. On s'est rencontré dans le RER en rentrant du boulot... tu lisais une bible sur les réseaux informatiques et maintenant on se retrouve à chaque évènement autour de la création d'entreprise. Je me rappellerai toujours de cette semaine à monter notre projet autour de la réalité virtuelle, c'était exceptionnel. Je suis sûr que nos chemins continuerons à se croiser ! Ton intelligence et ton dynamisme me sera toujours très inspirant. Merci.

Merci à Jean-Lucien Mazeau. Nos discussions après le boulot et nos échanges philosophiques ont toujours été très instructifs et inspirant. Il ne faut surtout pas qu'on se perde de vue.

Merci à toute la team WWO : Luca Incarbone, Vincent Leboeuf, Alain Girault.

Un merci spécial à Luca avec qui j'ai pu partagé pleins de moments qui me seront toujours gravés : Le trail de vulcain, la roche forio, le voyage à Torino avec la folle grande voie *athletico* de la sacra de san michele et le Rocciamelone. Je me rappellerai toujours de la soirée *châtaigne* dans ton studio dans le 15ème .. entrée : châtaignes grillées, plat principal : pâtes aux châtaignes, sauce à la châtaigne, dessert : marron glacé ! Multo Bene !

Pour conclure ces remerciements, je remercie aussi tous ceux qui j'ai oublié et que j'ai eu l'occasion de rencontrer pendant cette thèse :). En tout cas, j'espère tous vous recroiser dans la vie.

Je finirai aussi par remercier toutes les personnes qui m'ont inspiré pour faire cette thèse. Merci à tous les chercheurs qui sont passionnés, qui font avancer la science et qui, grâce à leurs avancées, m'ont motivé à faire un doctorat.

Enfin, je finirai par citer une phrase qui m'inspire dans beaucoup de moments et qui s'ap-

plique très bien à la thèse : *Dream but don't sleep.*

Résumé

Pour répondre aux besoins de plus en plus variés des applications (performance et efficacité énergétique), nous nous intéressons dans cette thèse aux architectures émergentes de type multi-cœur asymétrique en fonctionnalités (FAMP). Ces architectures sont caractérisées par une mise en œuvre non-uniforme des extensions matérielles dans les cœurs (ex. unitée de calculs à virgule flottante (FPU)). Les avantages en surface sont apparents, mais qu'en est-il de l'impact au niveau logiciel, énergétique et performance ?

Pour répondre à ces questions, la thèse explore les profils d'utilisation des extensions dans des applications de l'état de l'art et compare différentes méthodes existantes. Pour optimiser le placement de tâches et ainsi augmenter l'efficacité, la thèse propose une solution dynamique au niveau ordonnanceur, appelée *ordonnanceur relaxé*.

Les extensions matérielles sont intéressantes car elles permettent des *accélérations difficilement atteignables par la parallélisation* sur un multi-cœur. Néanmoins, *leurs utilisations par les applications sont faibles* et *leur coût en termes de surface et consommation énergétique sont importants*. En se basant sur ces observations, les points suivants ont été développés :

- **Nous présentons une étude approfondie sur l'utilisation de l'extension vectorielle et FPU dans des applications de l'état de l'art**.
- **Nous comparons plusieurs solutions de gestion des extensions à différents niveaux de granularité temporelle d'action pour comprendre les limites de ces solutions et ainsi définir à quel niveau il faut agir.**
- **Nous proposons une solution pour estimer en ligne la dégradation de performance à exécuter une tâche sur un cœur sans extension.** Afin de permettre la mise à l'échelle des multi-cœurs, le système d'exploitation doit avoir de la flexibilité dans le placement de tâches. Placer une tâche sur un cœur sans extension peut avoir d'importantes conséquences en énergie et en performance. Or à ce jour, il n'existe pas de solution pour estimer cette potentielle dégradation.
- **Nous proposons un *ordonnanceur relaxé*, basé sur notre modèle d'estimation de dégradation, qui place les tâches sur un ensemble de cœurs hétérogènes de manière efficace. Nous étudions la flexibilité gagnée ainsi que les conséquences aux niveaux performance et énergie.** Les solutions existantes proposent des méthodes pour placer les tâches sur un ensemble de cœurs hétérogènes, or, celles-ci n'étudient pas le compromis entre qualité de service et gain en consommation pour les architectures FAMP.

Nos expériences sur simulateur ont montré que l'ordonnanceur peut atteindre une flexibilité de placement significative avec une dégradation en performance de moins de 2%. Comparé à un multi-cœur symétrique, notre solution permet un gain énergétique au niveau cœur de 11 % en moyenne. Ces résultats sont très encourageants et contribuent au développement d'une plateforme complète FAMP. Cette thèse a fait l'objet d'un dépôt de brevet, de trois communications scientifiques internationales (plus une en soumission), et contribue à deux projets européens. De plus, le laboratoire met en place la structure pour réaliser du transfert industriel en proposant une brique technologique qui se base sur les travaux de cette thèse.

Abstract

To meet the increasingly heterogeneous needs of the applications (in terms of performance and energy efficiency), this thesis focuses on the emerging functionally asymmetric multi-core processor (FAMP) architectures. These architectures are characterized by non-uniform implementation of hardware extensions in the cores (ex. Floating Point Unit (FPU)). The area savings are apparent, but what about the impact in software, energy and performance ?

To answer these questions, the thesis investigates the nature of the use of extensions in state-of-the-art's applications and compares various existing methods. To optimize the tasks mapping and increase the efficiency, the thesis proposes a dynamic solution at scheduler level, called relaxed scheduler.

Hardware extensions are valuable because they speed up part of code where the parallelization on multi-core isn't efficient enough. However, the hardware extensions are under-exploited by applications and their cost in terms of area and power consumption are important. Based on these observations, the following contributions have been proposed :

- **We present a detailed study on the use of vector and FPU extensions in state-of-the-art's applications**.
- **We compare multiple extension management solutions at different levels of temporal granularity of action, to understand the limitations of these solutions and thus define at which level we must act**.
- **We propose a solution for estimating online performance degradation to run a task on a core without a given extension**. To allow the scalability of multi-core, the operating system must have flexibility in the placement of tasks. Placing a task on a core with no extension can have important consequences for energy and performance. But to date, there is no way to estimate this potential degradation.
- **We simulated a scheduler, called *relaxed scheduler*, based on our degradation estimation model, which maps the tasks on a set of heterogeneous cores effectively. We study the flexibility gained and the performance and energy implications**. Existing solutions propose methods to map tasks on a heterogeneous set of cores, but they do not study the trade-off between quality of service and energy consumption gain for FAMP architectures.

Our experiments with simulators have shown that the scheduler can achieve a significantly higher mapping flexibility with a performance degradation of less than 2 %. Compared to a symmetrical multi-core, our solution enables an average energy gain at core level of 11 %. These results are very encouraging and contribute to the development of a comprehensive FAMP platform. This thesis has been the subject of a patent application, three international scientific communications (plus one submission), and contributes to two active European projects. Plus, the laboratory is setting up the structure for industrial transfers by offering a technology based on the outputs of this thesis.

Acronymes

API interface de programmation 'Application Programming Interface' . . . 36
ASIC circuit intégré propre à une application 'Application-Specific Integrated Circuit' 35
CB cœur basique coeur sans extension . . . 26
CE cœur étendu coeur avec au moins une extension . . . 7
CBEA Cell Broadband Engine Architecture . . . 36
CFS ordonnanceur équitable 'Completely Fair Scheduler' . . . 109
CIL 'Common Intermediate Language' . . . 45
CPI cycles par instruction . . . 35
CPU unité centrale de traitement 'Central Processing Unit' . . . 35
DRAM mémoire vive dynamique 'Dynamic Random Access Memory' . . . 36
DSP processeur de signal numérique 'Digital Signal Processor' . . . 23
DVFS ajustement dynamique de la tension et de la fréquence 'Dynamic Voltage and Frequency Scaling' . . . 24
EMS système de gestion d'énergie 'Energy Management System' . . . 8
FAMP multi-cœur asymétrique en fonctionnalités 'Functionally Asymmetric Multi-core Processor' . . . 17
FP virgule flottante 'Floating Point' . . . 76
FPGA circuit logique programmable 'Field-Programmable Gate Array' . . . 35
FPU unitée de calculs à virgule flottante 'Floating Point Unit' . . . 17
GPGPU 'General-Purpose processing on Graphics Processing Units' . . . 36
GPU processeur graphique 'Graphics Processing Unit' . . . 23
HPC calcul à haute performance 'High Performance Computing' . . . 23
ILP parallélisme niveau instruction 'Instruction-Level Parallelism' . . . 35
IoT objets connectés 'Internet of Things' . . . 23
ISA jeu d'instructions 'Instruction Set Architecture' . . . 28
JIT juste-à-temps 'Just-In-Time' . . . 52
MLP parallélisme niveau mémoire 'Memory-Level Parallelism' . . . 35
MPSoC système sur puce multi-processeur 'MultiProcessor System-on-Chip' . . . 23
PCA analyse en composantes principales . . . 95
PAMP multi-cœur asymétrique en performance 'Performance Asymmetric Multi-core Processor' . . . 24
PIE 'Performance Impact Estimation' . . . 35
ROI régions d'intérêt 'Region Of Interest' . . . 57
SE système d'exploitation . . . 26
SIMD accélérateur vectoriel 'Single Instruction on Multiple Data' . . . 24
SMP multi-cœur symétrique 'Symmetric MultiProcessing' . . . 24
VLIW aux instructions très longues 'Very Long Instruction Word' . . . 36

Chapitre 1

Introduction

Aujourd'hui, les systèmes d'informations ont plusieurs modes de fonctionnement afin de s'adapter à l'activité de l'utilisateur. Dans le cadre des smartphones, pour certaines applications, la qualité de service et la performance sont prioritaires (ex. jeux, multimédia). Ensuite, lorsque le téléphone est faiblement utilisé, l'efficacité énergétique est prioritaire (ex. vérification de mail, attente d'appel). Cette tendance apparaît aussi dans les objets connectés 'Internet of Things' (IoT), où le besoin en calcul est important lorsque des données sont à traiter et à envoyer sur internet. Puis, lorsqu'il n'y a pas d'activité, le système doit juste être en attente et consommer le moins possible. Les serveurs deviennent aussi très sensibles à l'énergie consommée, à tel point que les serveurs de calcul à haute performance (HPC) sont maintenant aussi classés par "opérations par watt" dans le green500 [Lis]. Les besoins en ressources deviennent donc de plus en plus hétérogènes. La puissance nominale de calcul est toujours très élevée, mais ce besoin varie dans le temps. Pour que le matériel puisse répondre à ces besoins dynamiques de puissance de calculs et à ces contraintes énergétiques, celui-ci doit pouvoir s'adapter dynamiquement.
Dans ce contexte, la thèse étudie l'architecture émergente multi-cœur asymétrique en fonctionnalités (FAMP) (détaillée ci-dessous) et propose une solution de placement de tâches dynamique et flexible afin de répondre aux besoins hétérogènes des applications. La suite de ce chapitre reprend les contraintes techniques et détaille les travaux réalisés durant cette thèse.

Afin de pouvoir répondre aux besoins croissants de puissance de calculs, la technologie a évolué pour permettre de graver de plus en plus de transistors sur une même surface de silicium. L'évolution est présentée figure 1.1. Cette évolution impacte également la fréquence d'horloge et la consommation énergétique. Les prédictions montrent aussi une stabilisation du nombre de cœurs. De plus, la densité des transistors est maintenant si importante qu'ils ne peuvent pas tous être utilisés en même temps, pour des raisons de dissipation thermique. Ainsi, ces contraintes tendent à favoriser des plateformes hétérogènes, où seulement les parties du circuit répondant aux besoins de l'application sont activées.

En parallèle de l'évolution de la technologie liée à la gravure, les architectures des processeurs ont évolué. Pour gagner en puissance, les processeurs exploitent le parallélisme d'instructions, le parallélisme des fils d'exécution, de plus, ils se sont aussi spécialisés. Par exemple, le processeur commun s'est décliné en processeurs spécialisés comme le processeur de type vectoriel, processeur de signal numérique (DSP), processeur graphique (GPU), processeur cryptographique, processeur reconfigurable et processeur neuronal.

Pour répondre aux besoins hétérogènes et dynamiques des applications, plusieurs processeurs spécialisés sont assemblés dans un système sur puce multi-processeur (MPSoC). Cette configuration peut être très efficace et performante mais elle pose encore des défis au niveau

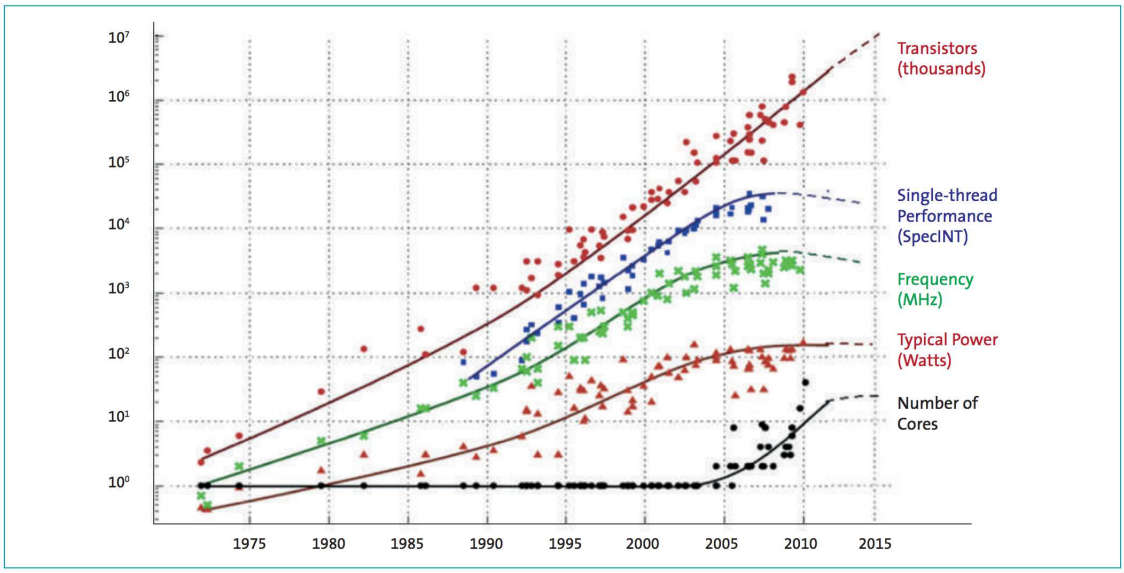


FIGURE 1.1 – Evolution du nombre de transistors dans les processeurs et conséquences.

logiciel. Pour faciliter la programmation, l'architecture multi-cœur symétrique (SMP) correspond à une duplication de cœurs similaires. Ces architectures sont en constante évolution et le nombre de cœurs dans les processeurs a fortement augmenté ces dernières années. Comme par exemple avec l'architecture Kalray, qui proposent des processeurs contenant 256 cœurs [Kal15]. De plus, pour augmenter l'efficacité énergétique, des techniques matérielles ont été mises en place telles que l'ajustement dynamique de la tension et de la fréquence (DVFS) ou l'allumage des cœurs sur demande. Néanmoins, le paradigme de parallélisme sur multi-cœur a des limites, puisque certaines applications sont difficilement parallélisables. Ainsi pour accélérer ces applications, les cœurs intègrent de plus en plus d'extensions matérielles, telles qu'une FPU, une unité accélérateur vectoriel 'Single Instruction on Multiple Data' (SIMD) ou encore une unité cryptographique.

Les extensions matérielles intégrées aux cœurs ont l'avantage de pouvoir accélérer certaines parties de code souvent difficilement accélérables sur un multi-cœur. Par exemple, le calcul d'une racine carrée sur une donnée à virgule flottante peut être réalisé mille fois plus rapidement avec une extension FPU qu'avec une librairie d'émulation sur un cœur pour données entières. Par contre, ces extensions consomment de la surface et de l'énergie, souvent à cause de leurs bancs de registres dédiés. Par exemple l'extension NEON&VFP de ARM occupe environ 30% de la surface d'un cœur cortex A9. De plus, ces extensions sont spécialisées pour seulement un type de calcul, donc elle ne sont utilisées que sporadiquement. Dans le cadre des architectures SMP, tous les cœurs doivent contenir ces extensions pour respecter la symétrie du jeu d'instructions. Cette contrainte réduit l'efficacité en énergie et en surface du multi-cœur.

Pour améliorer l'efficacité des architectures SMP, un compromis entre les architectures MPSoC et SMP est proposé avec les architectures de type multi-cœur asymétrique en performance (PAMP) et multi-cœur asymétrique en fonctionnalités (FAMP). L'architecture PAMP contient des cœurs performants et gourmands en énergie, et des cœurs faible consommation avec le même jeu d'instructions. **L'architecture FAMP contient la même base de cœurs,**

mais certains n'ont pas toutes les extensions matérielles. Contrairement au PAMP, l'architecture FAMP est une architecture émergente qui n'est pas encore dans les processeurs disponibles en production. Ces architectures sont présentés figure 1.2.

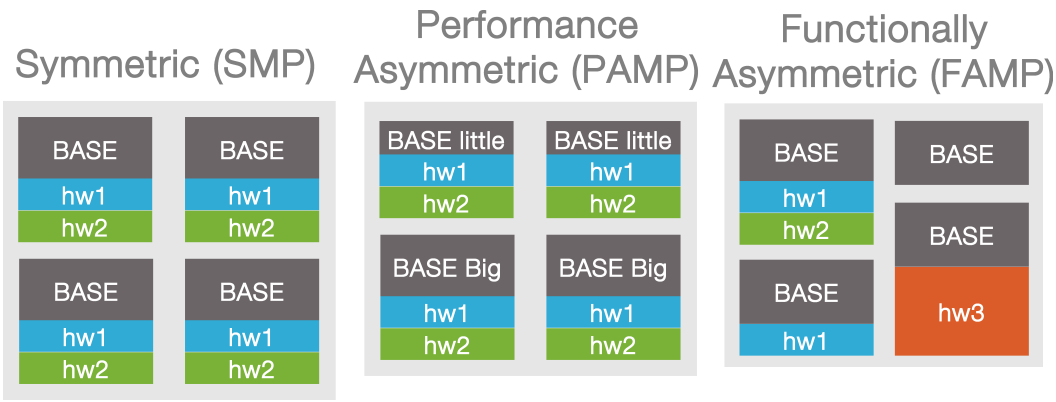


FIGURE 1.2 – Architectures multi-cœur symétrique (SMP), multi-cœur asymétrique en performance (PAMP) et multi-cœur asymétrique en fonctionnalités (FAMP). `Base` est une portion que tous les cœurs contient (par exemple le calcul sur données entières), `hw1, hw2, hw3` sont des extensions matérielles (ex. SIMD et FPU). `Base little` est une version faible consommation de `Base`, `Base Big` est une version gourmande en énergie mais très performante de `Base`.

L'architecture FAMP est intéressante car elle réduit les coûts des extensions matérielles qui sont généralement dupliquées dans chaque cœur. Dans la figure, ces extensions sont représentées par `hw1, hw2`. En distribuant les extensions seulement sur certains cœurs, la surface et la consommation énergétique peuvent être optimisées. En effet, en exécutant sur un cœur basique (sans extension) une application qui n'utilise pas les extensions, les autres cœurs avec extensions peuvent être éteints (ou utilisés par d'autres applications). Pour reprendre l'exemple du smartphone cité précédemment, les cœurs avec des extensions seraient activés seulement lors de l'exécution des applications multimédia. Cette architecture permet également une spécialisation plus poussée des extensions, car celle-ci apparait un nombre de fois plus limité.

Néanmoins, l'asymétrie du jeu d'instructions de l'architecture FAMP pose des défis au niveau de son utilisation. Logiquement, une instruction spécialisée pour une extension ne peut être exécutée que sur un processeur avec cette extension. Cette contrainte doit être prise en compte pour exploiter la plateforme de façon efficace.

Dans cette thèse, nous nous intéressons à l'architecture FAMP car nous pensons que cette plateforme peut permettre la mise à l'échelle des processeurs multi-cœurs et ainsi de répondre aux besoins dynamiques et hétérogènes des applications. A ce jour, l'architecture FAMP est une architecture émergente qui n'existe pas en production. **L'objectif de la thèse est de contribuer à démontrer le potentiel de l'architecture FAMP, en étudiant la variabilité d'utilisation des extensions et l'intérêt à avoir une solution de placement de tâches dynamique et flexible.** Les problématiques scientifiques liées à ces deux points sont décrites dans la section suivante.

Problématiques scientifiques

Cette section détaille les problématiques scientifiques liées à la variabilité d'utilisation des extensions et l'intérêt à avoir une solution de placement de tâches dynamique et flexible.

La première problématique est l'étude des profils d'utilisation des extensions. Ce sujet est peu étudié dans la littérature et il n'y a d'ailleurs aucune étude qui analyse en détails la variation d'utilisation d'une extension dans le temps. Or, le profil d'utilisation des extensions peut permettre des optimisations par rapport au placement des tâches. Par exemple, lors de phases variables (forte utilisation puis faible utilisation), le système d'exploitation (SE) pourrait déplacer les tâches et utiliser les cœurs avec ou sans extension seulement pendant certaines phases.

Les questions traitées sont donc : *quels sont les profils d'utilisation de l'extension dans les applications de l'état de l'art ?* Ensuite, en s'appuyant sur ces profils d'utilisation : *à quelle granularité temporelle pouvons-nous agir pour optimiser au mieux l'utilisation des extensions ?*

La deuxième problématique est la contrainte du jeu d'instructions par rapport au placement des tâches. Nativement, lorsqu'une application contient des instructions liées à une extension dans une certaine section, cette section doit être exécutée sur un cœur contenant l'extension. Or cette contrainte limite fortement le placement de tâche vis-à-vis du SE. Cette contrainte doit être prise en compte pour répondre aux besoins hétérogènes des applications.

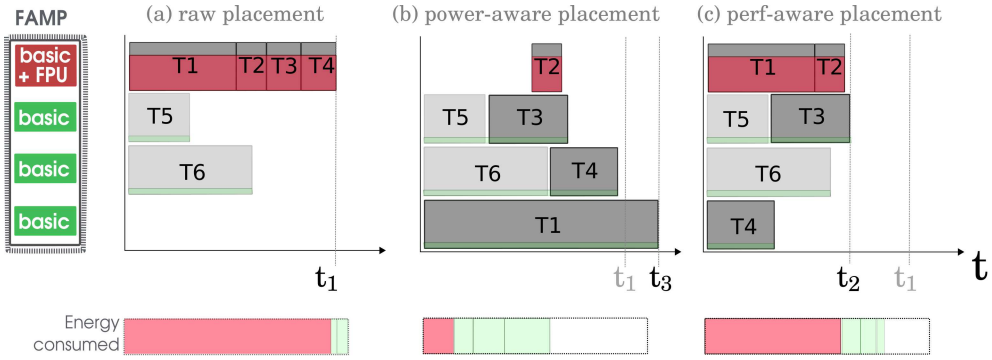
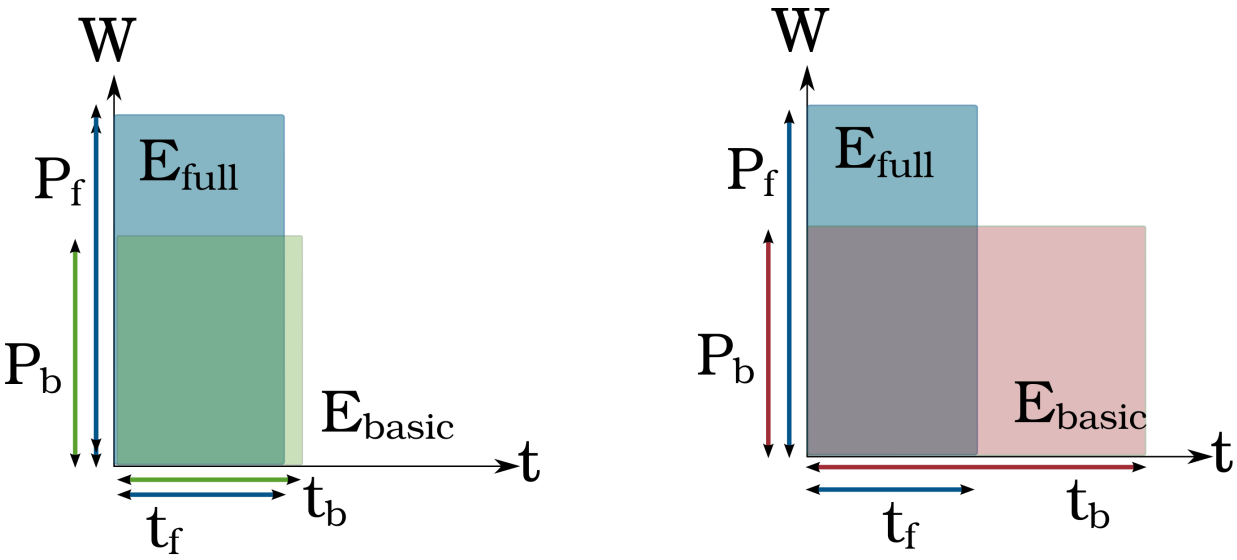


FIGURE 1.3 – Le système d'exploitation a besoin de flexibilité dans le placement de tâches pour permettre plusieurs points de fonctionnement

Un SE a besoin de flexibilité dans le placement de tâche pour optimiser l'exécution globale. Par exemple, pour 6 tâches données, indépendantes entre elles, dont les tâches T1 à T4 utilisent une extension, différents exemples de placement sont présentés en Fig. 1.3. Le placement (a) utilise les coeurs basiques (CB) seulement pour les tâches qui au départ n'utilisent pas l'extension. Dans le placement (c), les tâches T3 et T4 sont placés sur des cœur basique (CB) pour gagner en performance globale (même si l'exécution de T3 et T4 est plus longue). Dans le placement (b), les tâches sont presque toutes sur des CB (sauf T2 qui a trop d'accélération grâce à l'extension). Le temps d'exécution global est alors plus long mais la somme des énergies consommées est plus faible. Ainsi, la flexibilité permet une meilleure exploitation de tous les cœurs et la possibilité d'avoir plusieurs points de fonctionnement (placement (b) efficacité énergétique, placement (c) performance). Dans le cadre d'une architecture FAMP, les solutions contraintes par le jeu d'instructions empêchent cette flexibilité.

Pour permettre plus de flexibilité, il est nécessaire de pouvoir placer une tâche sur n'importe quel type de cœur. Dans l'état de l'art il existe des solutions techniques [GNVL14, NDR+11, DVT12, GNL13] permettant d'avoir du code disponible pour plusieurs type de cœurs.



(a) $E_{basic} < E_{full}$ (b) $E_{full} < E_{basic}$

FIGURE 1.4 – Différence de consommation énergique pour une tâche s'exécutant sur un cœur avec extension ou un cœur sans extension. (a) la tâche consomme moins d'énergie sur un cœur basique (CB). (b) la tâche consomme moins d'énergie sur un cœur étendu (CE).

Néanmoins, n'importe quelle section de code ne doit pas être placée sur un cœur sans extension. Les coûts liés au fait de ne pas utiliser l'extension peuvent être importants et engendrer une perte d'efficacité énergétique de l'architecture. En effet, le fait de placer une tâche qui est fortement accélérée par un CE sur un CB peut dégrader tellement la vitesse d'exécution que la réduction de la puissance énergétique est contrebalancée par le surcoût en temps. Ce principe est expliqué figure 1.4. L'énergie étant l'intégrale, l'exécution sur un CB peut être moins efficace en performance et en énergie. Sur la figure, l'exécution de l'application sur un CE donne un temps t_f avec une puissance moyenne P_f, donnant ainsi une consommation d'énergie E_{full}. Ensuite, il y a deux cas pour l'exécution de l'application sur un CB. Le schéma (a) donne un temps t_b qui est un peu plus long, avec une puissance moyenne P_b, donnant ainsi une consommation E_{basic} inférieur à E_{full}. Dans le deuxième cas (b), le temps t_b est bien plus long que t_f, la consommation E_{basic} est supérieure à E_{full}. On peut faire l'hypothèse que les puissances moyennes P_f et P_b varient peu au cours du temps, car leurs valeurs sont principalement liées à la dissipation statique (et non par la puissance dynamique des instructions). Par contre le rapport $\frac{t_b}{t_f}$ va varier en fonction de l'utilisation de l'extension.

La question traitée dans la thèse est *comment déterminer si le placement de la future section de code peut être effectuée sur un CB ?* Nous notons que dans cette thèse nous ne nous intéressons pas aux heuristiques d'optimisation de l'ordonnancement dans le cadre multitâches, mais uniquement au placement *physique* des tâches sur les cœurs. La problématique est de montrer dans un premier tant qu'il est possible d'avoir la flexibilité de placer les tâches sur n'importe quel cœur, et de mesurer cette flexibilité en fonction de la dégradation locale causée aux tâches. Étant donné que l'utilisation de l'extension varie dans le temps, il est nécessaire d'avoir une

solution qui puisse être exécutée en ligne. Ainsi l'ordonnanceur peut prendre des décisions de placement en connaissant le ralentissement engendré par l'exécution d'une tâche sur un CB. Au final, ce type d'ordonnanceur n'est donc plus contraint par le jeu d'instructions (ISA) mais seulement par un seuil de perte de performance maximale. Nous l'avons appelé, l'*ordonnanceur relaxé*.

Pour résumé, **l'objectif de la thèse est de savoir si l'utilisation variable des extensions permet des optimisations dynamiques et si un *ordonnanceur relaxé* est efficace sur une architecture FAMP**.

Approche

Pour savoir si l'utilisation variable des extensions permet des optimisations dynamiques et si un ordonnanceur relaxé est efficace sur une architecture FAMP, l'approche a été dans un premier temps de récupérer des informations sur l'usage des extensions, de voir les limites des solutions actuelles, puis de proposer et évaluer une nouvelle solution : l'*ordonnanceur relaxé*.

La récupération d'informations sur l'usage des extensions est importante pour comprendre la nature et les types de comportement. Ces derniers vont impacter l'efficacité des solutions logicielles et matérielles. Pour avoir un aperçu large, des applications de l'état de l'art de différents domaines comme l'embarqué, le calcul de vision et le multimédia ont été utilisées. Ces études ont été réalisées sur des processeurs ARM et x86 pour des extensions FPU et SIMD.
Ensuite, pour comprendre les limites des solutions actuelles et avoir des informations sur la granularité d'action pour gérer ces extensions, une étude a été réalisée comparant trois méthodes (au niveau application, ordonnanceur et instructions).

Dans l'état de l'art, il n'y a pas de solution qui permette à l'ordonnanceur de prendre la décision de placer une tâche sur un CB, ni d'information sur l'efficacité d'une solution à ordonnanceur relaxé. Pour permettre à l'ordonnanceur de prendre la décision, nous nous sommes intéressés à la manière d'estimer en ligne l'intérêt à utiliser une extension. L'approche a été d'étudier la solution naïve (qui utilise seulement le pourcentage). Notre étude sur l'extension FPU de ARM a montré qu'elle n'était pas assez précise, donc nous avons proposé une solution plus fine.

Pour étudier l'*ordonnanceur relaxé*, une simulation a été réalisée pour estimer le gain en flexibilité et l'impact sur les performances et la consommation en énergie. À ce jour, la plateforme FAMP n'existe pas en production. Pour valider ces hypothèses et ces propositions, une plateforme de simulation a été mise en place en se basant sur des cœurs ARM Cortex A9 et leurs extensions NEON&VFP. Ces modèles de processeurs ont été choisis car ils sont largement utilisés dans l'embarqué et bientôt dans d'autres domaines comme les μserveurs (ex. le projet européen euro-server [hp]). Une partie des résultats a aussi été obtenue sur des processeurs *AMD Opteron* afin d'étudier l'utilisation des extensions vectorielles *SSE* et *AVX*.

Les contributions scientifiques et résultats sont présentés dans la section suivante.

Contributions

Les principales contributions de cette thèse sont :

- l'analyse des profils d'utilisation des extensions FPU et SIMD pendant l'exécution d'applications de l'état de l'art, provenant de 5 suites d'applications de l'embarqué, d'algorithmes de vision et d'applications de calcul intensif.
- l'étude de la granularité d'action pour gérer les extensions (niveau application, ordonnanceur et instructions). Trois méthodes de l'état de l'art ont été analysées et les résultats montrent que la granularité du SE semble prometteuse. Néanmoins la contrainte du jeu d'instructions est limitante pour le niveau de granularité SE.

Pour placer efficacement les tâches, l'*ordonnanceur relaxé* prend en compte l'intérêt à utiliser l'extension et ne regarde pas uniquement l'apparition des extensions. Pour étudier cet *ordonnanceur relaxé*, les travaux réalisés dans cette thèse ont été :

- la réalisation d'une solution fine pour estimer l'intérêt à utiliser ou non une extension (en montrant que la solution utilisant seulement le pourcentage n'est pas assez précise pour prendre des décisions). La solution simple possède une erreur de 68 % en moyenne et notre solution présente une erreur de 14 %.
- la mise en place d'un environnement de simulation d'une architecture FAMP et d'un *ordonnanceur relaxé* pour estimer la performance et la consommation énergétique d'une telle plateforme.
- la mesure de la flexibilité d'un *ordonnanceur relaxé* et l'impact de celui-ci sur la performance et la consommation énergétique.

Les résultats ont montré l'intérêt d'avoir un ordonnanceur relaxé dynamique pour bien gérer certains cas d'utilisation de l'extension. Les résultats clés sont que grâce à un *ordonnanceur relaxé*, la flexibilité de placement peut atteindre 50 à 80% pour une perte de performance entre 2 et 12%. Cette flexibilité permet d'améliorer les gains d'énergie d'un rapport de deux comparés à un ordonnanceur contraint par l'ISA. Ces résultats montrent clairement l'intérêt de l'architecture FAMP comme candidat adapté aux besoins hétérogènes des applications, et sont très encourageants pour le développement d'une plateforme complète FAMP.

Ces travaux contribuent aux projets Européen *BENEFIC* (Best ENergy EFficiency solutions for heterogeneous multI-core Communicating systems) [BEN] et *Things2Do* (THIN but Great Silicon 2 Design Objects) [THI].

BENEFIC est un projet de recherche et développement, constitué de 19 organisations incluant industriels, PMEs et universités, provenant de France, des Pays-Bas et du Portugal. Le projet est labellisé dans la structure *CATRENE*, du groupe *EUREKA*. Les travaux de la thèse contribue à la recherche de solutions efficaces en énergie dans le cadre d'architectures multi-cœurs.

Le projet *Things2do* est un projet pour la conception et le développement de l'écosystème pour la technologie FD-SOI. Il est constitué de 45 organisations européennes et il est labellisé *Eniac Join Undertaking*. Dans le cadre de ce projet, une puce utilisant la technologie FD-SOI va être réalisé en se basant en partie sur l'architecture FAMP et la solution proposée dans cette thèse.

La thèse a fait l'objet de trois publications [ALCC14, ALC+15a, ALC+15b] (plus une en soumission) et d'un dépôt de brevet [ALC+15c]. De plus, un transfert technologique est en perspective avec la réalisation d'une brique technologique dans le laboratoire.

Ces travaux sont détaillés dans la suite de ce document et le plan est présenté dans la prochaine section.

Plan du manuscrit

Le chapitre 1 introduit la thèse. Le contexte est posé tout en décrivant les motivations, les objectifs et les contributions de la thèse.

Le chapitre 2 décrit l'état de l'art et place les travaux de la thèse par rapport à l'existant. Les thèmes abordés sont l'hétérogénéité dans les multi-cœurs, la gestion des extensions matérielles et la gestion de l'asymétrie dans une architecture FAMP.

Le chapitre 3 caractérise l'utilisation des extensions dans les applications de l'état de l'art.

Le chapitre 4 détaille trois méthodes à plusieurs niveaux de granularité pour gérer l'extension FPU. Elles sont comparées et les limites de ces solutions sont discutées.

Le chapitre 5 montre que la méthode naïve pour estimer l'accélération d'une extension (en fonction du pourcentage d'utilisation) n'est pas précise. Une nouvelle méthode en ligne pour estimer cette accélération y est proposée.

Le chapitre 6 présente un ordonnanceur qui utilise l'estimateur d'accélération afin d'optimiser le placement des tâches.

Enfin, le chapitre 7 conclut et ouvre les perspectives.

Chapitre 2

État de l'art

Sommaire

2.1 Introduction

Cette thèse explore des hypothèses pour améliorer la mise à l'échelle et l'efficacité des multi-cœurs et des many-cœurs. Ce problème peut-être résolu à plusieurs niveaux, depuis la conception du processeur jusqu'à son utilisation : au niveau semi-conducteur, architecture matérielle, logiciel de système d'exploitation jusqu'à la façon dont sont programmées et organisées les applications. De plus, il s'agit d'un problème multidimensionnel avec des objectifs antagonistes. L'optimisation d'une dimension telle que la simplicité de programmation, la flexibilité, la performance, l'efficacité énergétique ou la surface réduite, va impacter les autres dimensions. Néanmoins, certains objectifs comme la performance ou l'efficacité énergétique varient avec le temps. En s'appuyant sur cette dynamique, des solutions pourraient être plus efficaces dans plus de dimensions.

Ainsi, dans cette thèse nous nous intéressons principalement à l'architecture des processeurs et à la façon de gérer les tâches sur ces architectures.

A ce niveau d'action, un des grands concepts pour permettre un espace de calcul et d'efficacité en énergie plus large est d'ajouter de l'hétérogénéité dans l'architecture multi-cœur. Le papier *The Future of Microprocessors* [BC11] prévoit que l'évolution des microprocesseurs sera l'hétérogénéité.

Dans les multi-cœurs et multiprocesseurs, différents types d'hétérogénéité existent avec chacun leurs avantages et leurs désavantages. Ils sont décrits dans la première section.

Dans ce spectre d'hétérogénéité, nous nous intéressons principalement à l'hétérogénéité des extensions matérielles à travers les cœurs. Ces plateformes s'appellent des architectures multi-cœur asymétrique en fonctionnalités (FAMP). Ces extensions sont intéressantes car elles permettent de fortes accélérations mais elles sont coûteuses en énergie et en surface. Pour les utiliser efficacement, plusieurs méthodes existent. Ces méthodes sont présentées dans la deuxième section.

Enfin, l'hétérogénéité des extensions dans une plateforme multi-cœur permet de réduire la surface et potentiellement l'énergie, mais pour cela, il faut résoudre des défis de gestion de la non-uniformité du jeu d'instructions. La troisième section présente les méthodes actuelles pour gérer cette non-uniformité.

2.2 Hétérogénéité dans les multi-cœurs

L'hétérogénéité dans les multi-cœurs et multiprocesseurs est visible principalement par leur interface de programmation. Plus l'hétérogénéité est forte dans le jeu d'instructions, la mémoire, la représentation des données et plus il est nécessaire de rajouter des couches pour gérer l'hétérogénéité. Ces couches ne sont pas toujours transparentes pour l'utilisateur. Elles peuvent impliquer de la complexité dans la programmation et la gestion de la plateforme. De plus, l'exposition de cette hétérogénéité au programmeur peut impliquer une meilleure utilisation mais elle rend aussi le travail beaucoup plus fastidieux. Et dans certains cas, le fait de laisser la main à l'utilisateur ou à des outils hors ligne ne permet pas d'utiliser les connaissances du contexte en ligne, ce qui n'est pas optimal. Il y a un compromis entre hétérogénéité et homogénéité à trouver.

La première partie examine l'existant des plateformes hétérogènes pour lesquelles le jeu d'instructions n'est pas impacté. Ces plateformes sont appelées *Single-ISA*. La deuxième partie examine les plateformes qui possèdent des jeux d'instructions différents. Ces plateformes sont appelées *Multi-ISA*. Enfin le sujet de la thèse est positionné par rapport à ces deux types d'hétérogénéité.

2.2.1 Single-ISA : architecture asymétrique en performance

Les architectures multi-cœur asymétrique en performance (PAMP) sont hétérogènes bien que leur jeu d'instructions reste le même quel que soit le cœur. L'hétérogénéité n'est pas due à différentes fonctionnalités, mais plutôt aux différentes implémentations de ces mêmes fonctionnalités. En effet, par exemple, il n'est pas nécessaire de toujours utiliser un cœur avec un fort parallélisme d'instructions ou une haute fréquence d'horloge si la plupart des instructions sont en attente de données en mémoire. Lorsqu'un processeur contient plusieurs cœurs avec différents niveaux de rendement, le système d'exploitation peut ainsi déplacer les tâches sur les cœurs en fonction des phases de calculs ou des phases de mémoires de chaque tâche. Le papier [DTD03] décrit bien l'apparition des phases dans les applications et l'intérêt d'avoir plusieurs types de cœurs. Ces phases peuvent être exploitées notamment avec du DVFS, mais pour permettre encore plus de gains, il est intéressant d'agir sur les caractéristiques du pipeline du cœur comme l'ordre d'exécution ou le nombre de voies. Les premiers travaux à montrer l'intérêt de ce type de plateforme sont ceux de Kumar *et al.* [KJT04,KZT05,KTR+04,KFJ+03,KTJ06]. Les avantages de ce type de plateforme ont aussi été montrés avec des applications de l'état de l'art pour plateformes mobiles et serveurs [GS13]. Dans le papier [GRSW04], les auteurs comparent quatre techniques permettant d'optimiser les coûts énergétiques : DVFS, cœurs asymétriques, cœurs de différentes tailles et spéculation de contrôle. Ils concluent que la combinaison des cœurs asymétriques et du DVFS représente l'approche la plus prometteuse pour avoir un multiprocesseur qui peut obtenir à la fois une excellente efficacité en terme de latence et d'excellentes performances de débit.

Dans cette section, les architectures sont d'abord présentées puis les modèles de programmation associés sont décrits.

Architectures Single-ISA

Pour permettre plusieurs points de fonctionnement performance/énergie et garder le même jeu d'instructions, les architectures intègrent des cœurs similaires en fonctionnalité mais différents en terme d'implémentation.

Ces différences sont par exemple en terme de technologie. Le modèle Tegra 3 de Nvidia [Var11] contient quatre cœurs normaux et un cœur *faible consommation*. Le cœur dit à *faible consommation* a la même architecture que les quatre autres, mais il est implémenté avec une technologie moins gourmande en énergie. Comme le cœur est implémenté pour de la faible consommation d'énergie, sa fréquence maximale de fonctionnement est basse (seulement 500 Mhz, contre 1,5 à 3 Ghz pour les autres). Dans l'architecture, chaque cœur peut être éteint ou allumé séparément, donc il est possible d'allumer seulement le cœur *faible consommation* lorsqu'il n'y a que des tâches avec un faible niveau de qualité de service à exécuter, puis d'activer les cœurs normaux au fur et à mesure des besoins des applications.

Une autre possibilité est de modifier la structure interne du pipeline du cœur. Comme c'est le cas par exemple, dans le modèle de processeur big-LITTLE de ARM [ARM11], représenté Figure 2.1. Ce modèle contient des "petits" cœurs de type Cortex-A7, et des "gros" cœurs de type Cortex-A15. Le cortex-A7 est dit "petit" car il se contente d'exécuter les instructions dans l'ordre, soutenu par un pipeline 2-voies non-symétrique contenant 8 à 10 étages. Au contraire, le Cortex-A15 exécute dans le désordre, soutenu par un pipeline 3-voies contenant 15 à 24 étages. Le ratio moyen de performance entre les deux cœurs est de deux et le ratio de consommation d'énergie entre le cortex A7 par rapport au cortex A15 est de trois et demi. Nvidia a aussi intégré ce modèle dans les plateformes Tegra 4 et Tegra X1. A titre d'exemple, dans le Tegra 4, il y a un "petit" cœur cortex A9 et quatre cœurs cortex A15. Dans le processeur général du Tegra X1, on dénombre quatre "gros" cœurs ARM A57 et quatre "petits" cœurs ARM A53.

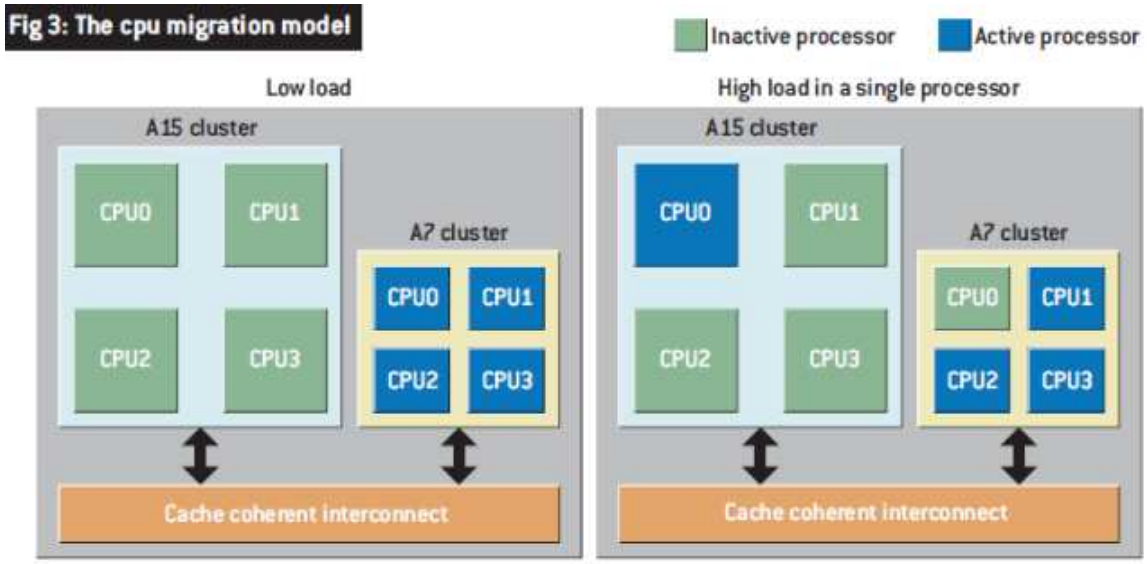


FIGURE 2.1 – Architecture big.LITTLE de ARM. Des "petits" cœurs et des "gros" cœurs permettent d'optimiser la performance ou l'énergie en fonction des besoins des applications (source : ARM)

Ces architectures ont le même jeu d'instructions. Néanmoins pour les exploiter efficacement il est nécessaire de bien placer les tâches. Les solutions de l'état de l'art sont détaillées dans la section suivante.

Gestion de l'hétérogénéité Single-ISA

Les architectures PAMP sont hétérogènes seulement en performance. Elles permettent ainsi d'exécuter n'importe quelle application sur n'importe quel cœur. Néanmoins, pour éviter des ralentissements par rapport aux besoins des applications, il faut que les tâches soient placées sur les cœurs les plus adaptés et que le partage des ressources soit juste. Actuellement ce concept n'est pas géré nativement par les systèmes d'exploitation. Ces derniers font l'hypothèse de n'avoir que des cœurs identiques.

Pour gérer et optimiser dynamiquement le placement de tâches, des solutions proposent de surveiller les applications et d'étudier leurs comportements. En effet, le ratio d'accélération entre un "gros" et un "petit" cœur n'est pas le même pour toutes les applications. Par exemple dans les travaux [SIZI11], le ratio pour un ensemble d'applications lorsqu'elles sont exécutées sur un Intel *core2duo* (gros cœur) plutôt qu'un Intel Atom varie entre 2.5x et 1.01x.

Par conséquent, il est nécessaire de pouvoir connaître le ratio ou la performance sur les autres cœurs et d'en informer l'ordonnanceur. Les solutions [SKR07,SSAJ+09,FSSP09,SSFP11] proposent de réaliser des analyses hors-ligne. Par exemple, la solution [SSAJ+09] utilise les erreurs de cache pour construire une signature de l'application. Ensuite, la signature est utilisée au moment du placement de tâche pour sélectionner le meilleur cœur. Dans le papier [LBKH07] d'Intel, les auteurs proposent que le système évalue le programme sur un "gros" cœur avant de le déplacer éventuellement sur des "petits" cœurs. Cette solution permet d'exploiter au maximum le cœur performant et d'activer les cœurs moins performants seulement dans le cas de sous-utilisation du premier.

Le ratio de performance entre les différents types de cœurs varie d'une application à l'autre, mais elle varie aussi à l'intérieur d'une même application. Ce ratio d'accélération est variable en fonction des phases de l'application [DTD03]. Des solutions plus dynamiques proposent donc d'analyser en ligne l'application et d'agir en conséquence des informations récupérées. Les premiers papiers sur le sujet [KFJ+03,BC06] utilisent l'échantillonnage des cycles par instruction (CPI). Dans le papier [KRH10], les auteurs identifient d'autres métriques qui caractérisent les potentiels bénéfices à placer une application sur un gros cœur plutôt qu'un petit cœur, comme par exemple les différents types de "stall" du cœur (interne et externe).

Le papier [VCJE+12] propose le modèle 'Performance Impact Estimation' (PIE) basé sur les métriques CPI, parallélisme niveau mémoire 'Memory-Level Parallelism' (MLP) et parallélisme niveau instruction 'Instruction-Level Parallelism' (ILP) . Grâce à ces informations, la performance de l'exécution peut être estimée sur un autre type de cœur (sans y exécuter l'application). Cela permet d'exploiter à fin grain la variation d'utilisation. Pour réduire les coûts d'estimation, [SIZI11] propose une analyse matérielle sur chaque cœur. Le papier [PLDM13] propose de prédire ces phases pour optimiser le placement. Le papier [LBK+10] détaille ce qu'il est nécessaire de changer dans un système d'exploitation pour s'adapter au déplacement de tâches et à l'asymétrie.

En conclusion, pour ces architectures, la complexité de programmation est faible car le jeu d'instructions est homogène. Par contre, le ratio surface/énergie/performance n'est pas optimal. Certaines parties de code sont faiblement accélérées par le parallélisme. De plus, certaines fonctions ne peuvent pas être implémentées de façon optimale en utilisant seulement des instructions native du CB (ex. *add*, *sub* et *mul*). Alors, lorsqu'un code utilise de façon intensive une fonctionnalité, l'implémenter en matériel dans le cœur peut permettre un vrai atout au niveau surface/énergie/performance. Le fait d'ajouter de l'hétérogénéité dans l'architecture peut permettre de répondre efficacement à ces besoins.

2.2.2 Multi-ISA

Cette section détaille les plateformes qui possèdent des jeux d'instructions différents entre les cœurs et/ou les processeurs. La spécialisation des cœurs a influencé la façon dont ils sont programmés. Les différences sont principalement au niveau des jeux d'instructions, de la gestion de l'espace mémoire et de la représentation des données. Cette spécialisation leur permet d'être très efficaces pour certains types de calculs. Ainsi, en assemblant ces différents processeurs dans la même plateforme, cela permet d'avoir plusieurs points de fonctionnement optimisés. Par contre, l'ensemble très hétérogène rend la programmation plus difficile.

Dans cette section, les architectures sont présentées puis les solutions pour gérer l'hétérogénéité sont décrites.

Architectures Multi-ISA

Les architectures multi-ISA sont très efficaces dans des domaines spécifiques. En plus d'architectures généralistes unité centrale de traitement (CPU), d'autres types de processeurs de calcul sont intégrées dans le système : GPU, DSP, circuit logique programmable 'Field-Programmable Gate Array' (FPGA) embarqués ou circuit intégré propre à une application 'Application-Specific Integrated Circuit' (ASIC).

Les architectures hétérogènes les plus répandues sont de type CPU + GPU ou CPU + GPU + DSP et CPU + FPGA. Ces architectures sont mises en œuvre dans les set-top box, dans les

smartphones, dans des systèmes embarqués, mais aussi dans des serveurs ou des supercalculateurs. Des exemples d'implémentation sont l'Omap de Texas Instrument (CPU + GPU + DSP) ou l'architecture MPSoC Viper [DJR01] de Philips, contenant un CPU contrôleur, un CPU aux instructions très longues 'Very Long Instruction Word' (VLIW) pour le multimédia, plus des processeurs spécialisés pour le décodage vidéo.

Une autre architecture, qui présente moins d'hétérogénéité, mais qui permet d'exploiter le parallélisme à différents niveaux, est l'architecture Cell Broadband Engine (CBEA) [Gsc07]. Elle contient un cœur principal et 8 cœurs spécifiques. Le cœur principal est un cœur faible-consommation (dans l'ordre, deux voies) et les cœurs spécifiques sont des processeurs vectoriels (128 registres de 128 bits). Cette plateforme a par exemple été utilisée dans le premier supercalculateur petaflops "roadrunner" et dans des consoles de jeux vidéo comme la Playstation 3.

Ces plateformes hétérogènes permettent des performances non atteignables avec un CPU simple. Leurs spécialisations sont conçues pour répondre aux besoins de points de fonctionnement très performants, mais l'hétérogénéité du jeu d'instructions et d'accès à la mémoire rend leur programmation complexe et parfois leur efficacité sous-optimale.

De plus avec l'apparition de concept comme 'General-Purpose processing on Graphics Processing Units' (GPGPU), où le GPU est exploité pour des applications au départ conçues pour le CPU, l'écart entre les deux type de processeurs devient encore plus contraignant. Un solution est d'avoir un espace mémoire unifié et partagé. Par exemple, les architectures Intel Sandy Bridge [Intb], AMD Kaveri [AMD] et NVIDIA Tegra k1 [Nvi] partagent différentes ressources telles que les derniers niveaux de caches, les contrôleurs mémoires et les accès à la mémoire vive dynamique (DRAM) située hors de la puce.

Nous notons aussi qu'il y a des plateformes hétérogènes ayant des processeurs de catégorie similaire mais d'architectures différentes pour permettre d'exploiter les spécificités de chaque architecture et d'avoir une compatibilité avec différents binaires. Par exemple, le papier [DVT12] s'intéresse à l'exploitation d'un système contenant un processeur ARM et un processeur Intel x86.

Dans tous les cas, pour gérer et exploiter ces différentes hétérogénéités le système d'exploitation, l'hyperviseur ou le programme applicatif doivent être adaptés. Des solutions pour améliorer ces points sont présentées dans la section suivante.

Gestion de l'hétérogénéité multi-ISA

Contrairement aux architectures *single-ISA*, qui requièrent principalement des modèles de programmation pour optimiser le placement de tâche, dans le cas du *multi-ISA*, des modèles sont aussi nécessaires pour gérer les différents ISA, la mémoire non uniforme et les représentations des données.

Pour gérer les processeurs spécifiques, il y a souvent des langages associées. Par exemple pour programmer les GPU, on utilise les langages tels que OpenGL [SSKLK13] ou NVIDIA CUDA [Nvi08]. Pour l'architecture CBEA, il y a le langage IBM Cell SDK [GLS99] ou d'autres interface de programmation (API) et de parallélisation comme OpenMP [OOS+08] et MPI [GLS99]. Pour l'architecture FPGA, on utilise les langages décrivant le niveau circuit comme le VHDL [CB99] ou le Verilog [Ver], et des langages plus haut niveau comme SystemC [LMSG02]. La plupart de ces langages permettent une programmation efficace de ces architectures, mais sont uniquement utilisables pour leurs processeurs dédiés.

Pour permettre, plus d'interopérabilité, il y a des langages multi-architectures. Les langages à directives tels que HMPP [DBB07], OpenACC [Sta], StarPU [AN09] et OpenMP [CJVDP08, WCC+07] ajoutent de l'information autour de certaines sections de code (en C/C++ par exemple). Ces informations permettent au compilateur, adapté pour comprendre ces directives, de générer du code pour certains cœurs spécialisés. La plupart de ces langages supportent ou intègrent les langages spécialisés cités ci-dessus. L'assemblage d'API et de langages comme OpenCL [SGS10] permet de générer un programme hôte qui contrôle l'exécution et des cœurs d'application distribuables sur différents types de processeurs. Des librairies comme le C++ AMP [GM12] sont incluses dans directX11 de Microsoft pour exprimer le parallélisme et ainsi générer du code pour GPU et CPU.

Pour plus de détails, un état de l'art exhaustif est présenté dans le papier [BDH+10] qui présente l'histoire et les solutions existantes dans le calcul hétérogènes et en particulier le matériel, les outils et les techniques et algorithmes pour les architecture de type CBEA, GPU et FPGA.

2.2.3 Positionnement

L'architecture ciblée dans la thèse est l'architecture multi-cœur asymétrique en fonctionnalités (FAMP). Elle se place entre les architectures Single-ISA et Multi-ISA. Dans la suite, nous positionnons les architectures Single-ISA et Multi-ISA par rapport à l'architecture FAMP.

Positionnement Single ISA

Par rapport aux travaux de la thèse, les propriétés hétérogènes des *Single-ISA* peuvent être aussi implémentées en plus de l'asymétrie en fonctionnalités. Certaines fonctionnalités internes d'un cœur ne sont pas obligatoires si une extension peut aussi les satisfaire ou inversement. Par exemple, lorsqu'un code est régulier et bien parallélisable, celui-ci va être exploitable par un accélérateur vectoriel (SIMD), donc il n'est pas nécessaire d'avoir un cœur qui sache exécuter plusieurs instructions en parallèle dans le désordre. Dans ce cas là, une base de cœur dans l'ordre est suffisante. À l'inverse, un code très irrégulier ne va pas pouvoir exploiter l'accélérateur vectoriel, mais bien être accéléré par un cœur capable d'exécuter des instructions dans le désordre (par rapport à un cœur qui ne gère qu'un flux d'exécution que dans l'ordre).

De plus, les solutions utilisées pour optimiser le placement de tâches peuvent être une source d'inspiration pour le cas des architectures asymétriques en fonctionnalité, car de la même façon, nous essayons d'utiliser au mieux les différentes phases d'exécution. Par contre, ces solutions ne sont pas nativement applicables dans notre cas car elles ne considèrent pas les mêmes métriques, d'où les travaux réalisés dans cette thèse.

Positionnement Multi-ISA

Les architectures Multi-ISA amènent de la complexité dans la programmation à cause des différents jeux d'instructions et modes de gestion de la mémoire. D'autres complications émergent au niveau de la conception et du débogage des applications. Le fait d'utiliser différents espaces mémoires, des chaînes de compilations et des outils qui rajoutent des couches entre l'application et le langage machine rendent tous les processus de programmation, de correction d'erreur et de fiabilité plus complexes.

Le papier [BSM01] résume les problèmes par rapport à l'hétérogénéité. La plupart des problèmes évoqués ne sont pas encore résolus. La vision proposée par ce papier pour que l'hé-

	SMP	SOC	PAMP	FAMP
la taille	-	+	++	+++
la Performance	-	+++	+	++
la dissipation énergétique	-	+++	+	++
la complexité	+++	–	+++	+
l'adaptabilité	+	-	++	+
le fait d'être spécialisé ou généralisé	–	+++	+	++
la facilité de trouver les erreurs	+++	–	+++	++

TABLE 2.1 – Comparaison d'architectures multi-cœurs sur des paramètres clés pour l'embarqué. 1 (mauvais) - 5 (bien). Architecture multi-cœur symétrique (SMP) Multi-ISA SoC (CPU+GPU) ; architecture multi-cœur asymétrique en performance (PAMP) ; architecture multi-cœur asymétrique en fonctionnalités (FAMP)

térogénéité soit efficace est de créer un environnement de calcul hétérogène automatiquement, grâce à un langage indépendant de la machine avec des directives utilisateur : (1) pour permettre la compilation d'une application donnée en un code efficace pour n'importe quelle machine de l'environnement hétérogène (2) pour aider à la décomposition de l'application en tâche et sous-tâche, (3) pour faciliter la compréhension des besoins de calculs de chaque tâche.

Considérant ces trois points, dans la thèse, nous nous intéressons aux architectures FAMP, qui ont une faible hétérogénéité et qui réduit le problème (1). Néanmoins, il est encore nécessaire de gérer l'asymétrie du jeu d'instructions. Nous ne nous intéressons pas au problème (2), nous prenons les applications comme elles ont été développées et agissons au niveau d'un découpage déjà mis en place (quantum de l'ordonnanceur par exemple). Pour le problème (3), nous proposons une solution en ligne qui permet d'estimer le gain que l'on peut atteindre en utilisant une extension. Cela permet à l'ordonnanceur de comprendre les besoins, d'acquérir de la flexibilité de placement de tâche et ainsi agir en fonction de ces objectifs (efficacité énergétique, performance, etc).

Pour résumer les avantages et désavantages de chaque architecture, le Tableau 2.1 les compare sur différents critères. L'architecture SMP est la plus utilisée car elle est facile à programmer. Par contre elle n'est pas optimisée matériellement pour la haute performance et l'efficacité énergétique. Le MPSoC est optimisé pour l'efficacité et la haute performance au détriment du coût de programmation et du débogage. Le PAMP permet un très bon compromis entre efficacité énergétique et programmation, mais la symétrie du jeu d'instructions limite la haute performance. Le FAMP valorise plus le coté efficacité et performance en acceptant de perdre un peu de simplicité de programmation. Néanmoins il y a seulement une faible hétérogénéité au niveau du jeu d'instruction, ce qui rend cette hétérogénéité potentiellement gérable au niveau système d'exploitation de façon transparente pour les utilisateurs.

Pour conclure, afin de réduire les désavantages majeurs de l'hétérogénéité, la thèse fait le choix de garder l'homogénéité au niveau mémoire, et d'avoir seulement une hétérogénéité au niveau des fonctionnalités spéciales des cœurs. D'un point de vue architectural, l'avantage est que les architectes de processeurs (comme ARM) proposent souvent le modèle du même cœur avec ou sans les extensions. Ainsi, d'un point de vue programmation, une partie des outils pour gérer les cœurs sont déjà disponibles pour une architecture de type FAMP.

Cette hétérogénéité en fonctionnalité se traduit par l'ajout d'extensions matérielles dans

les cœurs. Mais avant de s'intéresser seulement aux architectures multi-cœurs avec une hétérogénéité en fonctionnalité, la prochaine section expose les solutions globales de gestion de ces extensions.

2.3 Gestion des extensions matérielles

Les extensions matérielles des cœurs sont un des centres d'intérêt de cette thèse. Les extensions permettent souvent une forte accélération de certaines fonctions qui ne sont pas (ou peu) parallélisables sur un multi-cœur. Par exemple, la Figure 2.2 présente les différentes extensions disponibles pour un cœur Xtensa. Les extensions peuvent être très spécifiques et accélérer jusqu'à deux cent cinquante fois certaines fonctions. Ce qui ne serait pas possible juste en parallélisant la fonction sur un multi-cœur. Ces atouts impliquent que leur intégration est souvent cruciale dans un mono-cœur et encore maintenant dans un multi-cœur.

Par contre, une extension matérielle est coûteuse en surface et en énergie (lorsque l'accélération ne balance pas l'augmentation de puissance utilisée par l'extension). Un exemple de plan de puce d'un Cortex A9 est visible Figure 2.3. D'après les mesures (réalisées avec l'outil *ImageJ* [SRE12]) sur l'image, les extensions NEON+VFP (qui partagent les mêmes registres) prennent 36 % d'un cœur.

Il est donc nécessaire d'optimiser leur implémentation et leur utilisation en fonction des besoins. Nous présentons dans la suite plusieurs principes pour réduire les coûts de ces extensions.

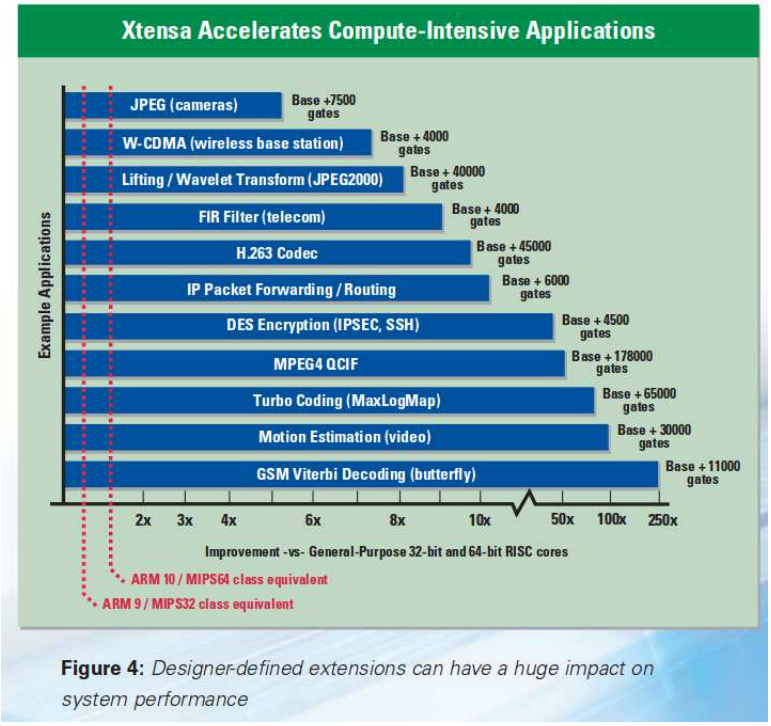
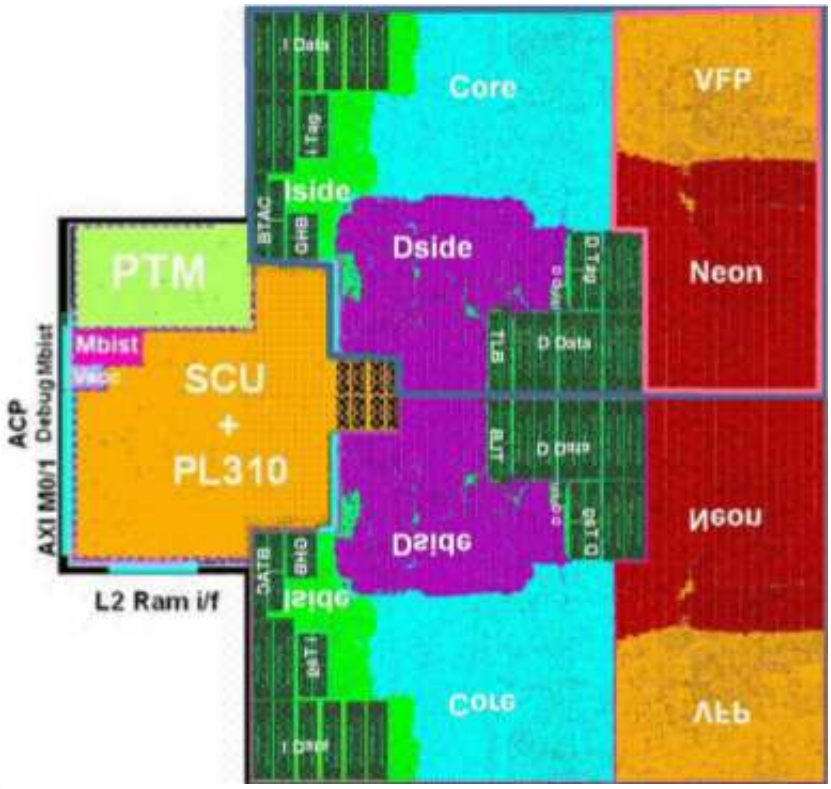


FIGURE 2.2 – Le processeur Xtensa 7 et ces extensions matérielles. (source : Tensilica)



CORTEX A9 FLOORPLAN

Floorplan from Osprey – 1.9W TDP 2GHz hardmacro version
of the dual core ARM cortex A9 in the TSMC 40G process (total size of only 6.7 mm2)

		Area	% of chip	% of one core
1	Chip	741826	100.00%	
2	One core w/ SCU,PTM	300283	40.48%	
3	One NEON + VFP	109255	29.46%	36.38%
4	Core part	64268	17.33%	21.40%
5	NEON	61900	16.69%	20.61%
6	VFP	45393	12.24%	15.12%

Area Information handled with ImageJ

FIGURE 2.3 – Plan d'une puce Cortex A9 dual-core de Osprey.

2.3.1 Allumage dynamique des extensions

La premier concept est d'éteindre/allumer dynamiquement l'extension, seulement quand l'application qui est exécutée en a besoin. Pour cela, le circuit doit avoir plusieurs domaines d'alimentation, dont un spécialement pour l'extension. Ainsi, il est possible de couper le domaine d'alimentation spécialisé à l'extension lorsque celle-ci n'est pas utilisée. Cette solution s'appelle le *power-gating*. L'architecture i.MX51 de Freescale implémente cette spécificité et son fonctionnement est décrit dans l'article [Xu10].

Il existe, en outre, le concept de *clock-gating*, où l'objectif est de gérer plusieurs domaines d'horloge. Ainsi les différents domaines d'horloge des extensions peuvent être coupés si nécessaire afin de réduire la consommation dynamique d'énergie. Cependant le *clock-gating* ne permet pas de réduire l'énergie statique consommée par les registres de l'extension. Ces registres peuvent être nombreux, par exemple l'extension NEON de ARM contient 16 registres de 128bits (pour comparaison le cœur Cortex A9 contient 16 registres de 32 bits).

L'avantage du *power-gating* est d'être transparent à la programmation, car sa gestion peut être implémentée en matériel. Le désavantage est que cette solution ne réduit pas la surface requise par les extensions. Au contraire, elle rajoute de la surface due au contrôleur et à la séparation des domaines d'alimentation.

2.3.2 Différentes implémentations des extensions

Pour réduire la surface et la consommation énergétique, le papier [RAKK13] propose d'implémenter la même extension mais avec plus ou moins de performance. Par exemple un cœur contient une extension "simple" avec une voie de calcul, et un autre cœur contient une extension "puissante" avec trois voies de calcul. L'idée est similaire aux architectures *Single-ISA* présentées précédemment. L'objectif est de garder le même ISA pour tous les cœurs mais d'avoir des performances d'exécution plus ou moins grandes. En fonction de l'intensité d'utilisation de l'extension (et du parallélisme d'instructions possible) l'application s'exécute sur le cœur avec une extension plus performante seulement quand cela est nécessaire. Néanmoins, cette solution reste coûteuse en énergie, même avec les extensions "simples", car il est toujours nécessaire d'avoir un minimum de registres spécialisés pour permettre les calculs. Et le nombre de registres ne peut pas être changé en fonction de la taille car cela à un impact sur l'allocation de registres durant la phase de compilation.

2.3.3 Partage des extensions

Dans un contexte multi-cœur, une autre solution consiste à partager des extensions entre les cœurs. Par exemple l'architecture Bulldozer AMD [BBSG11, RKK14] partage une FPU entre deux cœurs. Sa structure est présentée en Figure 2.4. Cette architecture permet de réduire par deux la surface utilisée par les extensions tout en gardant un jeu d'instructions symétrique à travers tous les cœurs. Par contre, cette solution demande de modifier la structure des cœurs et de rajouter du contrôle autour de la FPU pour permettre la gestion du partage entre les deux cœurs. De plus cette solution nécessite une gestion au niveau système d'exploitation afin d'éviter d'envoyer deux applications qui utilisent de façon intense la FPU sur le même ensemble de cœurs, évitant ainsi de saturer la FPU partagée.

2.3.4 Distribution non-uniforme des extensions

Dans un contexte multi-cœur, une autre solution consiste en la distribution non-uniforme des extensions à travers les cœurs. Ce concept permet de créer une architecture multi-cœur asymétrique en fonctionnalités (FAMP), présentée en Figure 2.5. Dans cette Figure, quatre

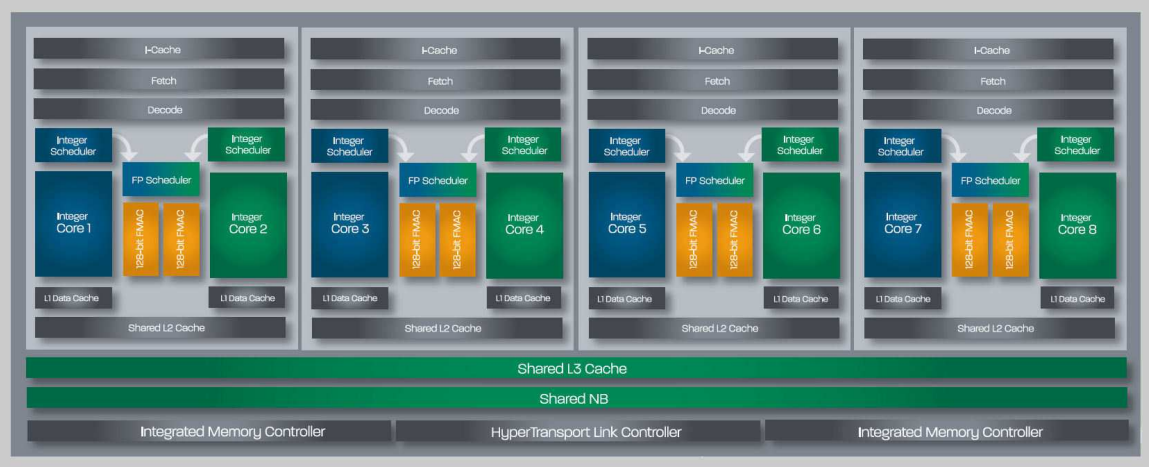


FIGURE 2.4 – Architecture bulldozer. L'extension FPU est partagée entre deux cœurs entiers (source : AMD)

cœurs ont la même structure "BASE" et certains cœurs ont des extensions "hw1,hw2,hw3" en plus.

Dans la littérature, ces plateformes sont appelées *Overlapping-ISA* [LBK+10] ou *shared-ISA* [GNL13]. Les plateformes FAMP ne sont actuellement que des plateformes de recherche : Par exemple, Intel propose la plateforme QuickIA [CSH+12], avec la possibilité d'avoir une asymétrie au niveau de l'extension vectorielle (SSE & AVX) et/ou de l'extension AES Crypto. Les autres plateformes utilisent des processeurs configurables tels que des processeurs Microblaze [GNL13] sur FPGA ou Tensillica Xtensa [SP09].

Ces architectures permettent de diminuer la surface globale et la consommation d'énergie. En effet, en exécutant sur un CB (sans extension) une application qui n'utilise pas les extensions, les autres cœurs avec extensions peuvent être éteints (ou utilisés par d'autres applications). Cette architecture permet également une spécialisation plus poussée des extensions, car celles-ci apparaissent un nombre de fois plus limité. Cette architecture ouvre de nouvelles perspectives.

En contrepartie, certains cœurs ont un jeu d'instructions plus large. Cette asymétrie doit être prise en compte lors du placement de tâches sur les cœurs.

2.3.5 Positionnement

Dans le cadre de cette thèse, nous nous intéressons aux plateformes dans lesquelles les extensions sont distribuées de façon non-uniforme entre les cœurs (architectures FAMP). Ce concept nous semble un bon compromis entre l'hétérogénéité matérielle avec tous les avantages que celle-ci apporte, tout en conservant une difficulté modérée dans la programmation de la plateforme. Néanmoins, ce concept architectural est assez nouveau (premières publications en 2009-2010) et il reste encore à démontrer la viabilité et le potentiel de ce type de plateforme. Le gain en surface est intuitif, mais il n'y a que peu d'information sur l'efficacité énergétique et les coûts associés à la gestion d'une telle asymétrie dans le jeu d'instructions.

Les différentes solutions traitant déjà la gestion de l'asymétrie sont détaillées dans la prochaine section.

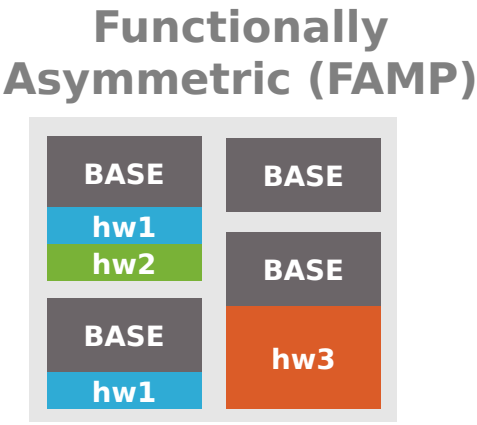


FIGURE 2.5 – Architecture multi-cœur asymétrique en fonctionnalités (FAMP). Les extensions ne sont pas distribuées sur tous les cœurs, mais une base est commune à chaque cœur.

2.4 Gestion de l'asymétrie pour les architectures FAMP

La gestion de l'asymétrie peut se faire à différents niveaux, matériel, système d'exploitation, application et manuellement au moment du développement. Un point transversal est la manière dont l'application va être encodée avec ces ISA. Dans l'article [RKBH11], les auteurs analysent les défis logiciels pour gérer les architectures asymétriques en fonctionnalités. Ils présentent trois manières de s'interfacer avec l'hétérogénéité : l'*approche restrictive*, l'*approche hybride*, et l'*approche unifiée*. Dans l'état de l'art, il y a aussi une quatrième manière de gérer l'hétérogénéité : l'*approche virtualisée*.

Ces quatre approches sont explicitées dans le Tableau 2.2 et sont décrites dans les sections suivantes. Pour chaque approche, les solutions les mettant en œuvre sont décrites, ainsi que leurs avantages et leurs inconvénients.

2.4.1 Approche unifiée

L'approche unifiée se base sur des applications qui peuvent contenir des instructions de toutes les extensions, sans tenir compte du fait que le cœur peut posséder ou non l'extension. Comme montré dans le Tableau 2.2, les instructions utilisables pour générer le binaire sont donc les ensembles $A, B, core$. Cela permet d'utiliser des applications qui au départ étaient compilées pour des multi-cœurs symétriques. L'avantage de cette approche est la possibilité d'exécution d'application sur les cœurs les plus puissants sans modification. En contrepartie, il est nécessaire de gérer les instructions qui ne sont pas exécutables sur les cœurs basiques.

Pour cela, la solution *faute-et-migration* (Fault&Migrate) [LBK+10, GKBS13] migre les tâches comme sur un multi-cœur symétrique (sans se préoccuper des instructions). Et lorsque des instructions spéciales sont prêtes à être exécutées sur un cœur qui ne possède pas l'extension adéquate, une interruption "instruction inconnue" est levée. Ainsi le système d'exploitation reprend la main et migre la tâche sur un cœur qui contient l'extension, pour que celle-ci continue son exécution. L'application est déplacée seulement pour un temps fini, ensuite elle retourne sur le CB. Cela permet de libérer le cœur avec l'extension pour les autres applications. Bien que cette solution offre des gains intéressants en termes de partage des ressources, elle peut souffrir

également de migrations excessives dues à un nombre élevé de fautes imprévisibles.

Pour éviter la migration après la faute, le papier [GNVL14] propose de transformer le binaire dynamiquement pour échanger l'instruction inconnue en une suite d'instructions basiques mettant en œuvre la même fonction.

D'une manière plus statique, les auteurs [SP09] proposent d'ajouter des annotations dans le code. Ces annotations forcent le compilateur à encoder certaines parties avec certains jeux d'instructions. Puis ces annotations sont utilisées en ligne, pour que le système d'exploitation sélectionne le cœur approprié pour la tâche. Le papier montre des gains en performance et énergie signifiants, par contre cette technique demande des efforts de développement supplémentaires.

2.4.2 Approche restrictive

Dans l'approche restrictive, les applications n'utilisent que les instructions partagées par tous les cœurs. Comme dessiné dans le Tableau 2.2, les instructions utilisables pour générer le binaire sont seulement celles de l'ensemble *core*. L'intérêt principal est de pouvoir placer les applications sur n'importe quel cœur sans aucune modification du code et sans générer de fautes. Par contre, pour tirer profit des extensions, il est nécessaire de retransformer les instructions basiques en instructions spécialisées. Pour cela, une solution est de méta-encoder les séquences d'instructions qui peuvent être transformées en une instruction spécialisée.

Le papier [GNL13] utilise de la réécriture de binaire dynamique et injecte les instructions avant l'exécution. Pour méta-encoder, la solution utilise les appels aux fonctions d'émulation. Ces fonctions sont dynamiquement remplacées par les instructions spécialisées.

Pour réduire les surcoûts en performance, *Liquid SIMD* [CHY+07] propose une implémentation matérielle pour la réécriture binaire de l'extension SIMD. Dans ce cas, la structure du CB avec l'extension doit être légèrement modifiée. Le méta-encodage est fait dans la séquence d'instructions. S'il y a un ensemble d'instructions qui se suivent avec certains motifs, le système détecte que l'ensemble est remplaçable par des instructions SIMD.

2.4.3 Approche hybride

Dans l'approche hybride, le binaire contient des versions de codes pour chaque type de cœurs. Le *multi-versioning* est souvent utilisé pour de l'instrumentation, du calcul adaptatif ou de l'optimisation dynamique [JLC11]. Dans le cadre du FAMP, cela permet d'avoir peu de surcoût en performance. En modifiant les outils de compilation ou en écrivant le code à la main, il est possible de générer plusieurs versions de code de la même fonction. Dans le papier [DVT12], les auteurs proposent d'avoir une représentation hybride en gardant une consistance des données en mémoire. Le programme est contenu dans un seul binaire avec deux sections *textes* et une seule section de *données*, donc avec des adresses virtuelles identiques. Avoir les mêmes adresses pour les données quelque soit la version réduit le coût de transformation pour la migration. Par contre, il est nécessaire de gérer le flux de l'exécution. En effet, il est difficile de commuter d'une version de code à une autre à n'importe quel moment car le processeur est dans un état, qui n'est pas obligatoirement stable. Pour résoudre ce problème, le papier [DVT12] propose d'utiliser de la transformation de code le temps d'arriver à un point de synchronisation du flux d'exécution.

Une deuxième limitation est la mise à l'échelle de cette solution avec un grand nombre d'extensions. En effet, cette solution est peu coûteuse en performance mais augmente considérablement l'espace mémoire utilisé par les binaires des applications. Si le nombre de cœurs différents est important, le nombre de versions augmentent et la taille des binaires également. Une dernière limitation de cette approche est que les optimisations réalisées par les compilateurs sont calculées hors-ligne, donc certaines optimisations qui peuvent tirer profit du contexte

en-ligne ne seront pas prises en compte.

2.4.4 Approche virtualisée

La virtualisation de l'ISA permet de représenter les applications sous une forme qui contient plus d'information que le sur-ensemble des instructions machines de la plateforme visée. Cela permet aussi d'avoir un code plus portable, et contrairement à la solution type multi-version, lorsque le nombre de cœurs différents est important, cela permet de ne pas occuper plus de place en mémoire. Par contre, cette solution amène une surcharge en performance car il est nécessaire, dans tous les cas, d'adapter une première fois le code au cœur sur lequel il s'exécute. Mais ce désavantage peut être transformé en avantage car cette adaptation de code dynamique permet des optimisations en fonction du contexte en-ligne.

La virtualisation est créée grâce à l'encodage de l'application dans une représentation intermédiaire comme LLVA [ALB+03] ou 'Common Intermediate Language' (CIL) du .Net Framework de Microsoft. Ensuite, pour exécuter le programme, la représentation intermédiaire est interprétée ou recompilée pour générer le langage machine adéquat au cœur.

Pour réduire les coûts d'adaptation lors de l'exécution, le papier [NDR+11] propose d'encoder seulement le jeu d'instructions spécialisées (dans leur cas SIMD) dans un langage intermédiaire utilisant CIL. De cette manière, les fonctionnalités partagées par tous les cœurs sont bien encodées avec les instructions machines, et les parties utilisant l'extension sont virtualisées. Des optimisations peuvent être réalisées en ligne en modifiant le flux d'instructions, par exemple en choisissant le déroulement de boucle (sur un processeur qui n'a pas de SIMD), ou en optimisant la taille des vecteurs traités par l'unité SIMD (4,8,16 données en parallèle), en fonction du cœur et de la taille des données.

2.4.5 Positionnement

Le Tableau 2.2 résume les avantages et les désavantages de chaque approche.

Dans l'article [RKBH11], les auteurs proposent d'utiliser plusieurs approches en même temps. Par exemple, pour le code du système d'exploitation, ils conseillent un mélange entre l'approche hybride et l'approche restrictive. En effet il est nécessaire d'éviter les fautes d'instructions inconnues avec certaines routines du système d'exploitation, car ces routines doivent potentiellement être exécutées par n'importe quel cœur. Pour les applications, ils proposent d'utiliser l'approche unifiée avec leur concept fautes-et-Migration (expliqué précédemment). Ils parlent d'utiliser l'émulation plutôt qu'une migration lors d'une faute, mais détaillent que la perte de performance est plus grande, d'une part par le fait qu'il n'y a pas l'extension, et d'autre part car la faute et le décodage de l'instruction (fautive) doivent être prises en compte pour chaque instruction émulée.

Dans cette thèse, les moyens pour permettre de placer n'importe quel code sur n'importe quel cœur ne sont pas explorés en détails. Par contre, la thèse explore de nouvelles façons d'ordonnancer les tâches en s'appuyant sur l'existence d'approches de type hybride, virtualisée et unifiée.

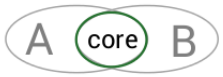
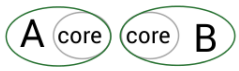
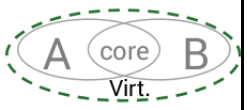
Unifiée	Restrictive	Hybride	Virtualisée
			
AVANTAGES			
• vue Single-ISA • faible cout de migration	• Facilement programmable • exécutable partout	• A $\bigcap$ B = $\emptyset$ est possible • code disponible à l'exécution	• A $\bigcap$ B = $\emptyset$ est possible • architecture agnostique • vue single-ISA • optimisation dynamique
DESAVANTAGES			
• A $\bigcap$ B > $\emptyset$ • Dépendant de l'apparition des instructions	• A $\bigcap$ B > $\emptyset$ • coût de départ pour exploiter les spécialités de A et B	• modification d'outils de compilation, binaire • Coût mémoire • ordonnancement haut fréquence complexe	• Cout de virtualisation

TABLE 2.2 – Résumé des approches pour gérer l'asymétrie de l'ISA

2.5 Conclusion

Dans ce chapitre, nous avons présenté les différents aspects liés à l'hétérogénéité dans les processeurs multi-cœurs. Cette hétérogénéité est nécessaire pour répondre aux objectifs d'efficacité énergétique et de performance. La difficulté de programmation rend l'utilisation de ces processeurs compliquée et pose aussi des problèmes de passage à l'échelle pour ces architectures.

Dans cette thèse, nous faisons l'hypothèse qu'une architecture asymétrique en fonctionnalité peut permettre la mise à l'échelle des multi-cœurs, de l'efficacité et de la performance sans augmenter considérablement la difficulté de programmation. Le potentiel de ces architectures vient du fait que les extensions permettent beaucoup de performance et que leurs utilisations varient entre les applications.

Pour résumer, d'après les études et les solutions existantes de cet état de l'art, il n'y a pas de caractérisation détaillée de l'utilisation des extensions. De plus, il n'y a pas de solution qui permette d'analyser en ligne l'intérêt à utiliser une extension, ni d'information sur les gains apportés par un ordonnanceur plus flexible dans le placement de tâches. En considérant ces points, nous présentons donc une étude sur l'utilisation de l'extension SIMD et FPU dans des applications de l'état de l'art. Puis nous comparons plusieurs solutions de gestion des extensions à différents niveaux de granularité pour comprendre les limites de ces solutions, et ainsi déterminer à quel niveau agir. Ensuite, nous proposons une solution pour estimer en ligne le coût de placement de tâches sur un CB plutôt que sur un CE. Et enfin, nous étudions la flexibilité gagnée lorsque l'ordonnanceur peut placer des tâches sur un CB, tout en respectant une qualité de service, ainsi que les conséquences au niveau performance et énergie.

Ainsi, la section suivante présente les profils d'utilisations de différentes applications de l'état de l'art, afin de mieux comprendre les gains potentiels que l'on peut obtenir en utilisant efficacement les extensions.

Chapitre 3

Usage des extensions dans les applications usuelles

Sommaire

3.1 Introduction

L'utilisation des extensions matérielles par l'application impacte directement l'efficacité énergétique et la performance du système. En présence de phases d'utilisation très concentrées ou de phases très peu intenses (un pourcentage faible d'instructions associées à l'extension), des procédés d'optimisation peuvent être mis en place. En fonction des comportements, la granularité d'action et les méthodes peuvent différer. Par exemple, s'il n'y a pas d'utilisation de l'extension pendant plusieurs milliers d'instructions, l'extension peut être éteinte pour économiser de l'énergie statique. A l'inverse si l'utilisation est très dispersée, il n'est pas judicieux d'éteindre et de rallumer l'extension car le surcoût en énergie et en performance va être supérieur aux gains.

De nos jours, l'utilisation des extensions dans les applications est très peu documentée. Or, la description de l'utilisation n'est pas si intuitive. Plusieurs facteurs ajoutent de la variabilité :

- la fonction intrinsèque de l'extension : une extension n'est utilisée que pour certains types de calculs ou types de données, elle n'est donc pas utilisée en permanence.
- les modifications effectuées par des outils intermédiaires entre la description du fonctionnement d'une application et son exécution : les outils tels que le compilateur et les librairies externes, mais aussi l'optimisation en ligne des machines virtuelles qui exploitent ces extensions. Par exemple, les compilateurs actuels sont programmés pour générer des applications qui s'exécutent sur des plateformes symétriques. Leurs objectifs sont donc d'exploiter au maximum les extensions. Ces compilateurs n'évaluent pas si l'utilisation de l'extension est efficace en performance ou énergie d'un point de vue global.
- la variabilité des tailles de données et du flux de contrôle : par exemple pour le traitement d'image, en fonction de la taille d'une image, l'intensité de l'utilisation de l'extension peut varier. De même, si un objet est détecté dans une image, certaines fonctions du programme qui potentiellement utilisent les extensions vont être exécutées.

Ces facteurs jouent un rôle important et ne sont pas facilement maitrisable par le programmeur avec les outils actuels.

L'idéal serait de permettre au système d'en savoir plus sur le comportement des applications. Des classifications d'application existent, comme dans le rapport de l'université de Berkeley [ABC+06] qui permet de mieux apprécier le parallélisme dans les applications. Mais les résultats de ces travaux ne sont pas tous communs aux extensions. Certaines caractéristiques d'applications sont utiles pour les extensions vectorielles mais pas pour les extensions FPU. De plus, avec l'apparition dans le commerce des PAMP (e.g. big.LITTLE et les autres plateformes décrites Section 2.2.1), d'autres études ont été réalisées pour améliorer le placement dynamique des tâches. Un concept prédominant est l'analyse dynamique des applications pour les placer sur le meilleur cœur au meilleur moment. Ces applications sont analysées pour évaluer leur potentielle intensité en mémoire ou en calcul. Elles sont ainsi placées en conséquence pour optimiser la performance et l'énergie. L'analyse du comportement permet d'extraire des phases [SR11, SZD04, DTD03]. Ces phases sont exploitées dynamiquement par l'ordonnanceur, comme par exemple dans les travaux [VCJE+12, FSSP09] où les auteurs estiment la performance sur les différents types de cœurs à partir du taux d'absence dans les caches, de l'intensité des accès mémoires et intensité de calcul. **Par contre, ces études et solutions ne s'intéressent pas aux extensions matérielles car tous les cœurs des plateformes asymétriques en performance possèdent les mêmes extensions.** Des études sur la plateforme Bulldozer (qui partage des extensions entre deux cœurs) s'intéressent à l'utilisation globale de l'extension FPU [BBSG11, RKK14], mais cela reste une analyse à gros grain.

Dans ce chapitre, les profils d'utilisation des extensions sont analysés dans des applications de l'état de l'art. Le but est d'analyser et de quantifier l'utilisation de ces extensions, c'est à dire quand, à quelle fréquence et pendant combien de temps elles sont utilisées. Pour analyser

ces comportements, les expérimentations s'appuient sur les extensions de deux architectures parmi les plus utilisées aujourd'hui : les extensions vectorielles SSE et AVX de l'architecture x86 d'Intel et l'extension de calculs sur données à virgule flottante (FPU) de ARM.

Le chapitre s'organise comme suit : la Section 3.2 présente les applications étudiées. La Section 3.3 décrit les méthodes d'expérimentations pour analyser les profils d'utilisation des extensions. La Section 3.4 présente et analyse les résultats en se concentrant sur l'utilisation globale par application et sur la variabilité de l'utilisation pendant l'exécution.

3.2 Applications étudiées

La thèse s'intègre dans un contexte de système embarqué et de HPC. Ces deux domaines sont de plus en plus similaires car ils doivent supporter des charges de travail de plus en plus hétérogènes. De plus, l'espace des applications et la diversité en objectifs de calcul (température, efficacité énergétique, performance) est large. Il est donc nécessaire de ne pas analyser qu'un type d'application mais plutôt un ensemble assez représentatif de l'espace global. Les programmes sélectionnés sont des applications et des noyaux de l'état de l'art actuellement utilisés dans le commerce et l'industrie. Ils proviennent des suites d'applications *Parsec* [Bie11], *MiBench* [GRE+01], *SDVBS* [VAJ+09], *Polybench* [Pou12], *WCET* [GBEL10] et *fbench* [Wal04]. Voici une description de ces différentes suites :

La suite *Parsec* [Bie11] (V2.1) est une suite mise en œuvre et maintenue par l'université de Princeton. Elle représente des charges de travail émergentes et a été conçue pour être représentative de programmes à mémoire partagée de nouvelle génération pour les puces multiprocesseurs. La version actuelle de la suite contient 13 programmes parallélisés issus de nombreux domaines tels que la vision par ordinateur, l'encodage vidéo, l'analyse financière, la physique et le traitement d'image.

La suite *MiBench* [GRE+01] est mise en œuvre et maintenue par l'université du Michigan et est représentative d'applications utilisées actuellement dans les systèmes embarqués. Les applications sont divisées en six catégories : l'automatique et le contrôle industriel, le réseau, la sécurité, dispositifs consommateur, la bureautique et les télécommunications. Toutes les sources des programmes sont disponibles en langage C.

La suite *SDVBS* [VAJ+09] (San-Diego Visual Benchmark Suite) contient diverses applications issues de domaines tels que l'analyse d'image, la compréhension d'image (localisation), la détection, le suivi d'objets et la vision stéréo. Chaque application est disponible en code C ou MATLAB. La description du programme en code C emploie un usage minimal des pointeurs et des constructions simples, ce qui permet une parallélisation plus facile et une indépendance face aux plateformes.

La suite *Polybench* [Pou12] est une suite de noyaux développés par l'université de l'Ohio. Elle contient 30 noyaux tels que la multiplication de matrice, le calcul de corrélation, de covariance, la décomposition Gram-Schmidt. Les programmes sont disponibles en code C.

La suite *WCET* [GBEL10] est une suite d'applications collectées de différents groupes de recherche et de vendeurs d'outils logiciels à travers le monde. Chaque application est disponible en code C.

La suite *fbench* [Wal04] est une application complète d'un algorithme de lancer de rayon (optical design raytracing algorithm) développé par John Walker. L'application utilise de façon intensive le calcul sur des données à virgule flottante (incluant des fonctions trigonométriques). La suite est disponible en code C.

Les Tableaux 3.1 et 3.2 présentent les applications et noyaux de chaque suite qui ont été adaptés pour être exécutés sur les plateformes que nous utilisons. Toutes les applications de chaque suite n'ont pas été adaptées car certaines applications ne contenaient pas d'utilisation de données flottantes ou n'étaient pas vectorisables. La suite *Parsec* a été utilisée pour l'étude sur x86. Pour l'étude de la FPU sur ARM, les suites *MiBench*, *SDVBS* et *polybench* ont été utilisées. Ces suites d'applications ont été choisies car elles permettent de représenter une grande diversité d'applications avec différentes utilisations des extensions. De plus, leur disponibilité en langage C permet de les utiliser sur différents types de plateformes tels que des simulateurs, des plateformes embarquées à faibles ressources et avec différents types de jeux d'instructions. Au cours de la thèse, ces applications ont été adaptées pour être exécutées sur les différentes plateformes utilisées et pour être analysées avec différentes méthodes, telles que celles décrites dans la section suivante.

3.3 Méthodes d'expérimentations

Pour analyser l'utilisation des extensions dans les applications de l'état de l'art, deux études ont été menées. La première analyse les extensions vectorielles (SIMD) SSE et AVX de l'architecture Intel x86. La deuxième prend en compte l'extension de calculs sur données à virgule flottante (FPU) de l'architecture ARM armv7. Les deux méthodes d'expérimentations sont présentées ci-dessous, puis le procédé de validation des processus d'expérimentations est détaillé dans une dernière partie.

3.3.1 Méthodes pour étudier l'utilisation de x86 SSE* & AVX

Pour analyser l'exécution des applications sur x86, l'outil *Pin* [LCM+05] a été utilisé. Il a été choisi car il permet d'étudier en détails l'exécution d'applications directement en natif sur un processeur, sans passer par un simulateur. Il rajoute des surcoûts lors de l'exécution, mais cela reste plus rapide qu'un simulateur.

Plus en détails, l'outil *Pin* utilise l'instrumentation binaire dynamique pour les jeux d'instructions IA-32 et x86-64. Il permet la création d'outils en C++ d'analyse dynamique du programme. Ces outils sont appelés *pintools*. Un *pintool* comprend l'instrumentation, des routines d'analyse et des routines de rappel. Dans un premier temps, les routines d'instrumentation sont appelées lorsque le code est sur le point d'être exécuté, et ainsi elles permettent l'insertion des routines d'analyse. Les routines d'analyse sont ensuite appelées lorsque le code qui leur est associée est exécuté. Les routines de rappel ne sont appelées que lorsque certaines conditions sont remplies, ou quand un certain événement a eu lieu. *Pin* fournit une API pour l'instrumentation à différents niveaux d'abstraction, d'une instruction à l'ensemble d'un module binaire. Il prend également en charge les routines de rappel pour de nombreux événements tels que les chargements de bibliothèques, les appels systèmes, les signaux/exceptions et les événements de création de fils d'exécution (threads). *Pin* effectue l'instrumentation en prenant le contrôle du programme juste après qu'il soit chargé dans la mémoire. Puis, en utilisant de la recompilation juste-à-temps 'Just-In-Time' (JIT), il recompile de petites sections de code binaire. De nouvelles instructions pour effectuer des analyses sont ajoutées au code recompilé.

Parsec	Blacksholes	Option pricing with Black-Scholes Partial Differential Equation
	Bodytrack	Body tracking of a person
	Canneal	Simulated cache-aware annealing, routing cost optimisation
	Dedup	Next-generation compression with data deduplication
	Fluidanimate	Fluid dynamics for animation purposes with Smoothed
	Swaption	Pricing of a portfolio of swaptions
	Vips	Image processing
MiBench	basicmath	Auto. / Industrial
	bitcount	Auto. / Industrial
	qsort	Auto. / Industrial
	susan (corners)	Auto. / Industrial
	susan (edges)	Auto. / Industrial
	susan (smoothing)	Auto. / Industrial
	jpeg	Consumer
	stringsearch	Office
	dijkstra	Network
	patricia	Network
	dijkstra	Network
	CRC32	Network / Telecomm.
	sha	Network / Security
	FFT	Telecomm.
	IFFT	Telecomm.
SDVBS	Localization	Robot Localization / Image Understanding
	Mser	Maximally Stable Regions (MSER) / Image Analysis
	Multincut	Image Segmentation / Image Analysis
	Sift	Feature Tracking Motion / Tracking / Stereo Vision
	SVM	Support Vector Machines (SVM) / Image Understanding
	Texture synt.	Texture Synthesis / Image Processing and Formation
	Tracking	Feature Tracking Motion / Tracking and Stereo Vision

TABLE 3.1 – Résumé(1/2) des applications et noyaux utilisés dans la thèse

polybench	2mm	2 Matrix Multiplications (D=A.B ; E=C.D)
	3mm	3 Matrix Multiplications (E=A.B ; F=C.D ; G=E.F)
	adi	Alternating Direction Implicit solver
	atax	Matrix Transpose and Vector Multiplication
	bicg	BiCG Sub Kernel of BiCGStab Linear Solver
	cholesky	Cholesky Decomposition
	correlation	Correlation Computation
	covariance	Covariance Computation
	doitgen	Multiresolution analysis kernel (MADNESS)
	durbin	Toeplitz system solver
	dynprog	Dynamic programming (2D)
	fdtd-2d	2-D Finite Different Time Domain Kernel
	fdtd-apml	FDTD using Anisotropic Perfectly Matched Layer
	gauss-filter	Gaussian Filter
	gemm	Matrix-multiply C=alpha.A.B+beta.C
	gemver	Vector Multiplication and Matrix Addition
	gesummv	Scalar, Vector and Matrix Multiplication
	gramschmidt	Gram-Schmidt decomposition
	jacobi-1D	1-D Jacobi stencil computation
	jacobi-2D	2-D Jacobi stencil computation
	lu	LU decomposition
	ludcmp	LU decomposition
	mvt	Matrix Vector Product and Transpose
	reg-detect	2-D Image processing
	seidel	2-D Seidel stencil computation
	symm	Symmetric matrix-multiply
	syr2k	Symmetric rank-2k operations
	syrk	Symmetric rank-k operations
	trisolv	Triangular solver
	trmm	Triangular matrix-multiply
WCET (FP only)	fft	1024-point Fast Fourier Transform using the Cooly-Turkey algorithm
	lms	LMS adaptive signal enhancement
	ludcmp	LU decomposition algorithm
	minvert	Inversion of floating point matrix
	qsort-exam	Non-recursive version of quick sort algorithm
	qurt	Root computation of quadratic equations
	select	Select the Nth largest number in a floating point array
	sqrt	Square root function implemented by Taylor series
	st	Statistics program.

TABLE 3.2 – Suite du résumé(2/2) des applications et noyaux utilisés dans la thèse

Pour cette étude, un *pintool* a été créé pour analyser dynamiquement le profil de l'utilisation des instructions vectorielles. Pour chaque *basic bloc* (un *basic bloc* est un morceau de programme défini comme une suite d'instructions contenant seulement un saut situé à la fin du bloc) exécuté dans un programme, le *pintool* analyse le nombre d'instructions vectorielles et la distance entre elles. La distance est calculée à travers tous les *basic blocs*, pas seulement à l'intérieur d'un *basic bloc* (la distance n'est pas remise à zéro entre deux *basic blocs*). Un fichier de sortie est aussi généré puis analysé par des scripts *perl* et *gnuplot*, comme montré en figure 3.1. Pour pouvoir analyser de longues applications, les données ont du être compressées.

L'étude a été réalisée sous Linux *2.6.32-5-amd64* et sur des processeurs de type *x86_64 AMD Opteron(TM) Processor 6276* avec 2 Mo de cache L1 et sachant exécuter les jeux d'instructions suivants : *sse sse2 ssse3 sse4a sse4_1 sse4_2 avx 3dnowprefetch*. L'extension SSE est sous-divisée en sous-ensembles dus à l'évolution du jeu d'instructions. L'extension SSE calcule jusqu'à 16*128bits. L'extension AVX travaille sur des données allant jusqu'à 16*256bits.

Dans cette étude, nous considérons seulement l'utilisation avec des données *vectorielles* de SSE et AVX. Une utilisation avec des données scalaires est possible, mais nous la considérons ici comme un biais introduit par Intel et les développeurs de compilateurs. Pour cela, dans l'outil *pintool* réalisé, il y a une distinction entre les instructions ayant un mnémonique exprimant un usage vectoriel (finissant par exemple par "PS","2PI", ou commençant par "P") et les instructions ayant un mnémonique exprimant un usage scalaire (finissant par exemple par "SI" ou "SD"). L'outil prend donc aussi en compte les instructions mémoires associées à l'extension.

La suite d'applications Parsec a été compilée avec les compilateurs de l'état de l'art suivants : GNU GCC 4.7 et ICC 13.1. Les options de compilation pour GCC sont `-m64 -Ofast -flto -march=native -funroll-loops -opt-prefetch`, comme conseillé par intel [Inta]. Les options de compilation pour ICC sont `-O3 -funroll-loops -opt-prefetch -fpermissive -fno-exceptions`. L'option `march=native` permet l'utilisation des extensions de l'AMD Opteron citées ci-dessus.

Plus d'informations sur l'auto-vectorization de GCC sont détaillées dans le document [GNUb] et plus d'informations sur les boucles qui ne sont pas encore vectorisables sont détaillées dans ce rapport de bogue [GNUa].

3.3.2 Méthodes pour étudier l'utilisation de armv7 NEON & VFP

Pour étudier les profils d'utilisation de l'extension FPU de l'ensemble d'instructions armv7 de ARM, le simulateur Gem5 [BBB+11] a été utilisé. Au moment de la réalisation de ces expériences, il n'y pas d'outils tel que *Pin* pour les plateformes ARM. Le simulateur Gem5 a été sélectionné car il permet d'exécuter rapidement des applications grâce à son mode de simulation "system call emulation". Ce mode permet d'exécuter facilement des binaires (compilés pour contenir toutes les librairies statiquement) comme sur une plateforme linux mais sans simuler la couche Linux totalement. Les appels systèmes sont transférés à l'hôte Linux. Ainsi la simulation est rapide et simple. De plus, c'est un outil open source avec une communauté active à laquelle participe l'entreprise ARM. Enfin, le fait d'avoir l'accès aux sources de Gem5 permet de pouvoir adapter l'outil à nos besoins.

Dans Gem5, le modèle de processeur a été configuré pour correspondre étroitement à un ISA ARMv7 Cortex A9 avec NEON (contenant l'unité de virgule flottante), ayant une fréquence d'horloge à 1 Ghz, 32 Ko de cache L1 instruction/data et 512 Ko de cache L2. Pour cela, le modèle de processeur par défaut de l'architecture a été ajusté pour correspondre au manuel de référence technique du processeur [ARM12], tel que décrit dans le document [ECC14] (par exemple pour adapter la taille de pipeline et la latence des instructions).

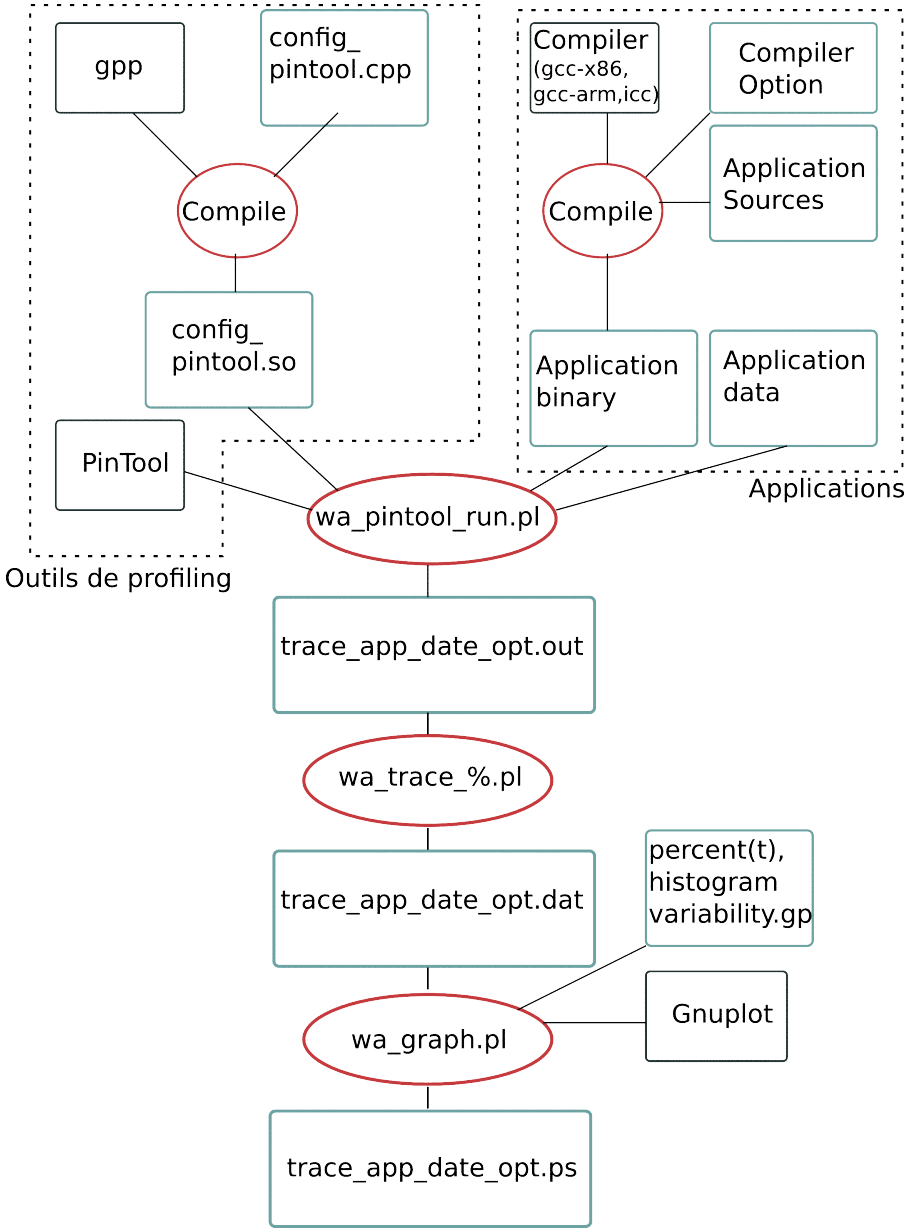


FIGURE 3.1 – Procédure pour analyser l'utilisation des extensions lors de l'exécution

Pour indiquer au compilateur GCC d'utiliser l'extension FPU, les options de compilation suivantes ont été utilisées `-mfpu-abi=softfp -mfpu=neon -fno-tree-vectoriser -O3`. La fonctionnalité vectorielle de l'extension NEON pour les données de type entier n'a pas été utilisée pour se concentrer uniquement sur des données flottantes. En outre, la fonctionnalité vectorielle NEON pour les calculs à virgule flottante n'a pas été activée afin d'être conforme à la norme IEEE 754 ISO.

L'objectif de cette étude n'est pas de montrer si une extension est correctement exploitée par le compilateur ni de reprogrammer les applications pour exploiter manuellement les extensions. Mais plutôt de faire ressortir le profil d'utilisation globale de ces extensions en utilisant une chaine d'outils classique. Par ailleurs, exploiter manuellement les extensions demande un trop grand effort au programmeur et rend le code non-portable et peu réutilisable. Les programmes sont compilés avec les options nécessaires pour utiliser les extensions.

Les études analysent l'ensemble des applications et ne se concentrent pas seulement sur les régions d'intérêt (ROI) qui encadrent seulement les calculs spécifiques. Les parties d'application qui ne sont pas directement liées à l'algorithme principal font souvent une mauvaise utilisation des extensions. Pour voir les différentes phases de l'utilisation de l'extension, toutes les parties d'une exécution de l'application sont prises en compte, par exemple les phases d'initialisation, les entrées/sorties, les transferts mémoires et les structures de données de configuration.

3.3.3 Validation des méthodes

Cette section décrit le protocole utilisé pour valider le processus d'expérimentation.

Validation de la compilation et du type d'application

À la compilation, la transformation d'un code scalaire en un code vectorisé est principalement impacté par deux paramètres : l'activation des optimisations de l'auto-vectorisation du compilateur et la structure d'un programme qui facilite sa transformation en code vectoriel. Pour les extensions de type FPU ou cryptographique, c'est surtout l'utilisation explicite de certains types de données ou de certaines bibliothèques qui force l'utilisation de l'extension.
Pour réaliser cette validation, trois types d'applications sont utilisés :

- Une application codée explicitement avec des instructions vectorielles.

```
1 //explicit data type
  typedef int v4si __attribute__ ((vector_size (16)));
3 int main(int argc, char *argv[])
  {
5     v4si c;
      v4si a = {1,2,3,4};
7     v4si b = {3,2,1,4};
      c = a > b; //operation with explicited vectors
9     printf("%d, %d, %d, %d\n", c[0], c[1], c[2], c[3]);
      return 0;
11 }
```

  Les données typées `v4si` indiquent au compilateur d'utiliser l'extension vectorielle.

- Une application dont les données sont explicites mais codée de façon régulière, facilement transformable par le compilateur en instructions vectorielles.

```
#define M 32
#define N 48
int main(int argc, char *argv[])
{
    int a[M][N]; int i,j;
    int x = atoi(argv[1]);

    //GCC auto-vectorize supports multidimensional arrays
    for (i=0; i<M; i++) {
        for (j=0; j<N; j++) {
            a[i][j] = x;
        }
    }
    for (i=0; i<M; i++) {
        for (j=0; j<N; j++)  {
            printf ("%d ",a[i][j]);
          }
        printf ("\n");
    }
    return 0;
}
```

Dans ce code, la boucle qui parcours le tableau 2D est régulière et l'opération réalisée n'a pas de dépendance entre les cycles de boucles, donc le compilateur optimise facilement le calcul en utilisant des instructions vectorielles.

- Une autre application trop irrégulière pour qu'il ne soit possible d'utiliser des instructions étendues. (Non vectorisable avec GCC 4.5)

```
#define M 8
int main(int argc, char *argv[])
{
    int a[M] = {0,1,2,3,4,5,6,7}; int b[M] = {7,6,5,4,3,2,1,0};
    int i;
    // unvectorized loop
    for(i=M-2; i>=0; i--)
        a[i+1] = a[i] + b[i];

    for (i=0; i<M; i++) {
        printf ("%d ",a[i]);
      }
    printf ("\n");

    return 0;
}
```

Dans ce code, le parcours du tableau et les opérations internes de la boucle sont trop complexes pour que le compilateur détecte des schémas pour vectoriser.

Après avoir compilé ces programmes, la vérification a été réalisée grâce aux détails des commentaires de l'outil de compilation (en mode *verbose*) et grâce à la décompilation du binaire final. La détection des boucles est notifiée dans les commentaires du compilateur et l'apparition (ou la non-apparition) des instructions vectorielles dans le code est vérifiable dans le code décompilé.

Validation des outils de profilage et de mise en forme

Une fois les processus de compilation validés, les binaires ont été utilisés pour vérifier les outils de profilage.

Dans un premier temps, nous vérifions que les instructions étendues sont bien détectées au profilage et que les outils de profilage ne rajoutent pas de fausses informations. La validation se fait avec une entrée contenant des instructions étendues et avec une entrée qui n'en contient pas. Dans un second temps, la validation vérifie la cohérence des traces. L'exécution en parallèle de plusieurs tâches, comme par exemple pour les applications de la suite *Parsec*, peut produire des traces non cohérentes si ces tâches écrivent en même temps dans le fichier de sortie. Il est donc nécessaire de gérer l'accès au fichier pour garder une cohérence des traces.

La vérification a été réalisée en analysant les fichiers intermédiaires (visible figure 3.1) et les graphes finaux sur l'utilisation de l'extension. En effet, les applications utilisées pour la validation étant déjà caractérisées, il est ainsi possible de comparer le nombre d'instructions étendues calculé statiquement avec les résultats prédéfinis.

3.4 Résultats

Les résultats des deux études présentées ci-dessus sont décrits dans les deux prochaines sous-sections. La première présente l'utilisation moyenne globale des extensions FPU et vectorielle. La deuxième présente les profils et les phases d'utilisation des extensions lors de l'exécution des applications.

3.4.1 Utilisation moyenne globale des extensions

Pour avoir une vue globale de l'utilisation par rapport aux applications, le pourcentage d'utilisation sur toute l'exécution a été mesuré. Le pourcentage correspond au nombre d'instructions de l'extension par rapport au nombre d'instructions du CB.

Résultats pour les extensions SSE et AVX

L'histogramme 3.2 présente le pourcentage d'utilisation des instructions vectorielles lors de l'exécution des programmes `blacksholes` (blac), `bodytrack` (body), `canneal` (cann), `dedup` (dedu), `fluidanimate` (flui), `swaptions` (swap) et `vips`. Ces programmes ont été compilés par chaque compilateur GCC et ICC. La différence sur le pourcentage d'instructions vectorielles exécutées entre les deux versions est de quelques pourcents seulement. Pour plus d'information sur la différence entre les compilateurs, le papier [MGG+11] présente les différents types de vectorisation gérés par les compilateurs usuels.

Le pourcentage global de l'utilisation se situe entre 0.01 % (pour `fluidanimate`) et 15 % (pour `blacksholes`).

Résultats pour l'extension FPU de ARM

Les tableaux 3.3 et 3.4 présentent le pourcentage d'utilisation de la FPU pour les applications *MiBench*, *SDVBS* et *polybench*.

Pour les applications des ensembles *MiBench* et *SDVBS*, certaines applications ont des taux moyen d'utilisation inférieur à 0.5 %, comme `dijkstra`, `crc32`, `djpeg`, `patricia` et `susan smoothing`, et certaines applications ont des taux supérieur à 30%, comme `sift` et `multincut`.

Pour les applications de l'ensemble de *polybench*, il y a deux noyaux (`dynprog` et `regdetect`) qui sont purement entiers, les restants ont un taux d'usage qui varie entre 40 % et 80 %. Le

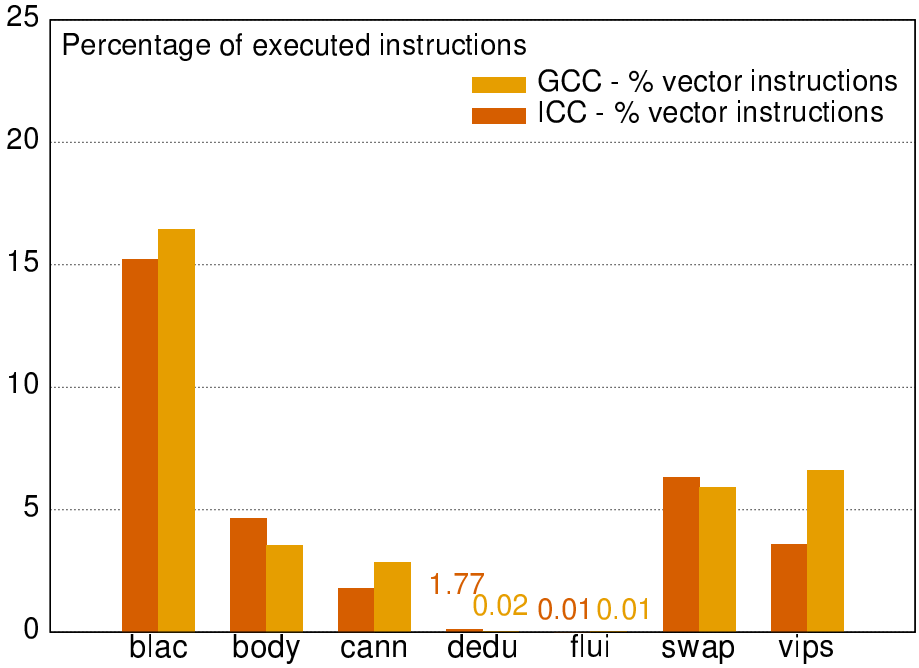


FIGURE 3.2 – Pourcentage des instructions SIMD exécutées par rapport au nombre total d'instruction, pour les applications de la suite *Parsec*.

taux d'utilisation est très élevé car cette suite contient principalement des noyaux de calculs et non des applications complètes. Il faut noter que ces noyaux sont ensuite intégrés dans des applications plus larges.

En résumé, d'une application à une autre, l'utilisation de l'extension est très variable. Et dans un même domaine d'application tel que l'analyse d'image, il y a aussi de la diversité d'utilisation entre chaque application. Afin de déterminer si l'utilisation des extensions intra-application est aussi variable, la section suivante présente les résultats de l'analyse de l'utilisation des extensions au cours de l'exécution.

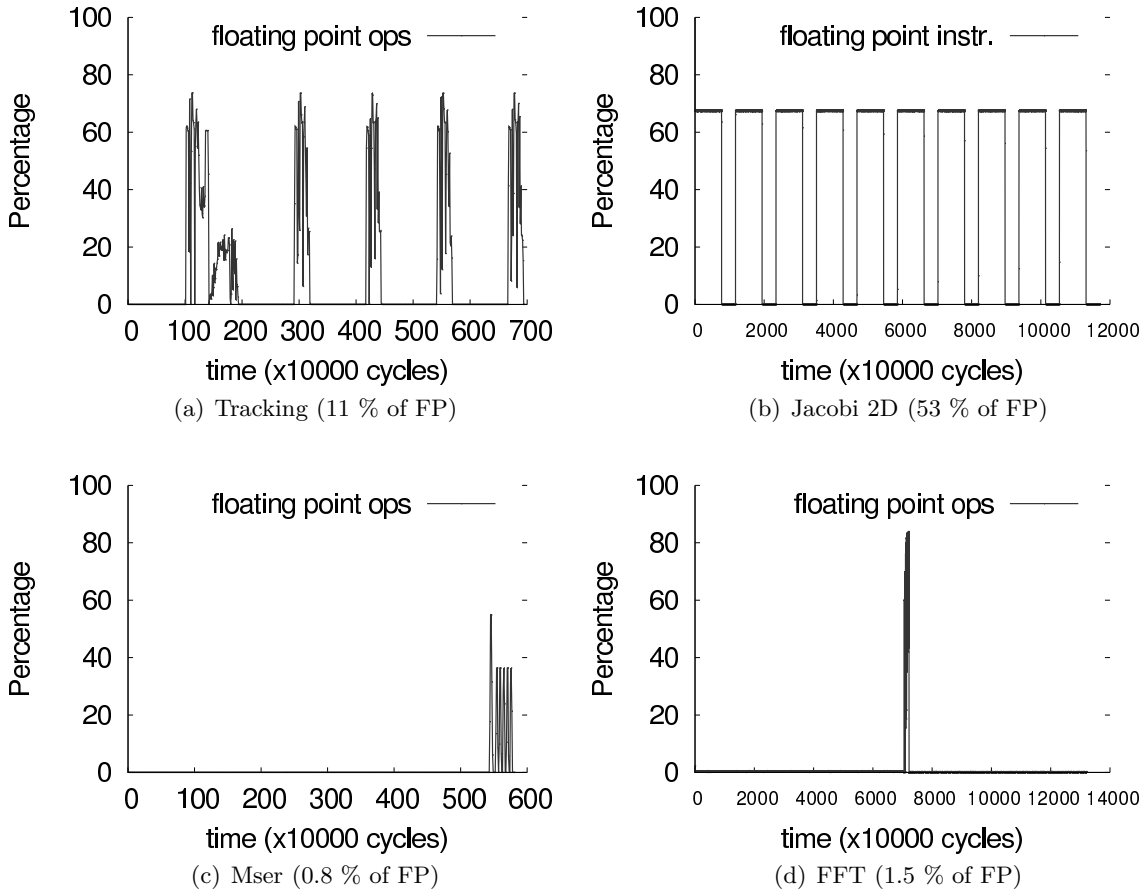
3.4.2 Profils d'utilisation au cours de l'exécution

Au cours de l'exécution d'une application, celle-ci utilise souvent différents types de données et procède à différents traitements plus ou moins réguliers. Ces différentes étapes vont impacter l'utilisation des extensions.

Variabilité de l'utilisation

Les graphiques figures 3.3 et 3.4 représentent l'utilisation de l'extension FPU au cours de l'exécution. Chaque point du graphique est le pourcentage d'utilisation pour une moyenne sur 10K instructions. Les figures 3.3 et 3.4 présentent les graphes pour les applications de *SDVBS* `mser`, `sift`, `tracking`, de l'ensemble *MiBench* `basicmath`, `susan corner` et `fft`, et de l'ensemble *polybench* `bicg` et `jacobi2D`; les autres applications sont détaillées dans l'annexe C.

On peut remarquer que certaines applications telles que `Tracking` et `jacobi 2D`, ont différentes phases distinctes. Par exemple, pour `Tracking`, la première phase n'utilise pas du tout l'unité flottante, puis la deuxième phase entre les points 100 et 200, utilise jusqu'à 70% du temps l'unité flottante, et ainsi de suite pour chaque image traitée.



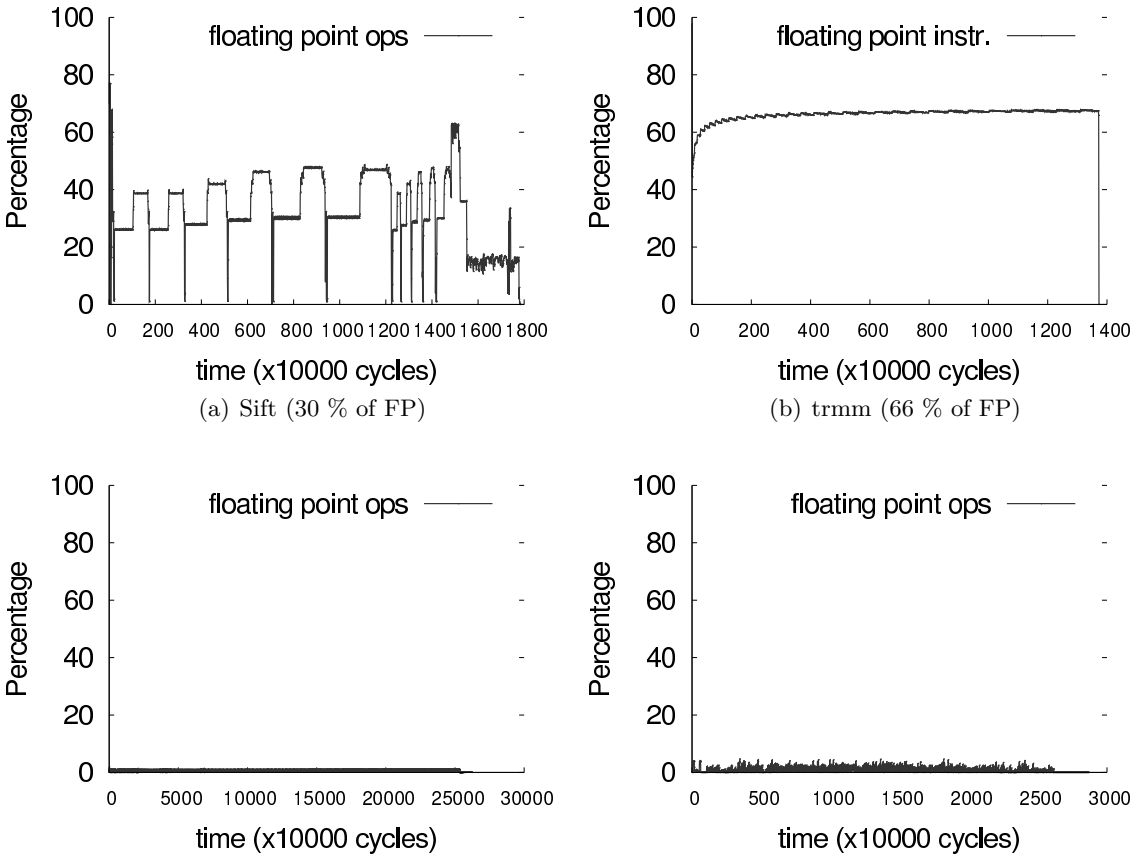
(a) Tracking (11 % of FP)

(b) Jacobi 2D (53 % of FP)

(c) Mser (0.8 % of FP)

(d) FFT (1.5 % of FP)

FIGURE 3.3 – Profils 1/2 - Utilisation de l'unité flottante (FPU) de ARM pendant l'exécution. Chaque point du graphique est une moyenne sur 10K cycles. Il y a des phases de programmes distinctes pour l'utilisation de la FPU.



(a) Sift (30 % of FP)

(b) trmm (66 % of FP)

(c) Basicmath (0.7 % of FP)

(d) Susan Corner (0.7 % of FP)

FIGURE 3.4 – Profils 2/2 - Utilisation de l'unité flottante (FPU) de ARM pendant l'exécution. Chaque point du graphique est une moyenne sur 10K cycles. Il y a des phases de programmes distinctes pour l'utilisation de la FPU.

Application	Pourcentage FPU
dijkstra	0.39
crc32	0.39
djpeg	0.42
patricia	0.42
susan s	0.47
svm	0.50
bitcnts	0.51
basicmath	1.25
mser	1.29
susan c	1.38
fft	1.48
qsort	1.51
susan e	1.59
fft Inv	1.83
cjpeg	2.59
search	5.16
texture synt	6.09
sha	13.67
tracking	16.56
localization	22.77
sift	32.16
multincut	49.98

TABLE 3.3 – Pourcentage d'utilisation de la FPU lors de l'exécution. Applications des suites *Mibench* et *SDVBS*.

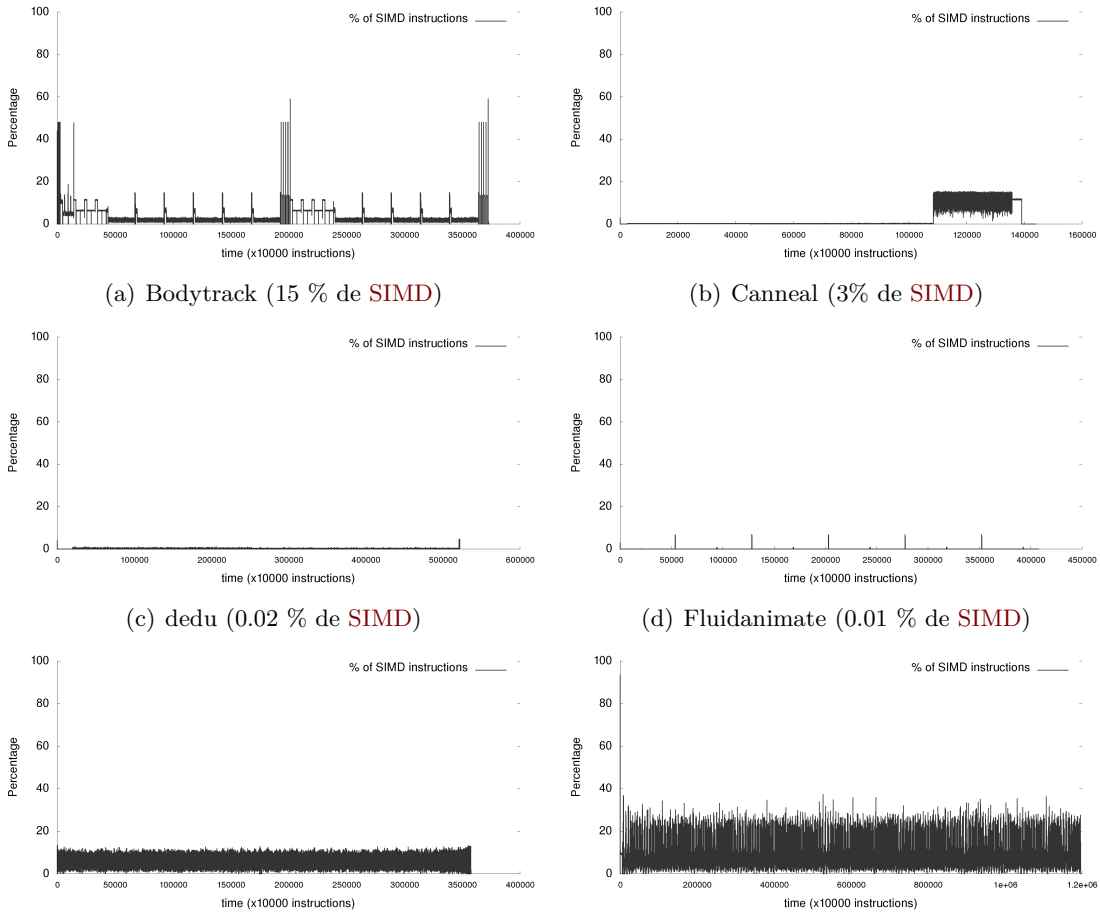
Application	Pourcentage FPU
dynprog	0.00
regdetect	0.00
ludcmp	41.66
durbin	43.70
covariance	44.50
correlation	44.66
doitgen	48.46
gramschmidt	49.63
cholesky	52.45
jacobi-2d-imper	53.42
jacobi-1d-imper	55.13
3mm	58.28
2mm	61.09
mvt	62.46
trisolv	62.91
fdtd-2d	63.53
symm	63.57
gemm	63.64
syrk	65.61
trmm	66.21
lu	66.50
adi	67.03
atax	67.61
gemver	70.28
floyd-warshall	71.19
seidel-2d	71.74
fdtd-apml	76.26
bicg	76.61
gesummv	78.14
syr2k	78.60

TABLE 3.4 – Pourcentage d'utilisation de la FPU lors de l'exécution. Applications de l'ensemble *polybench*.

Pour certaines applications telles que `MSER` et `FFT`, il n'y a qu'une seule phase qui utilise fortement l'extension lors de l'exécution. Si l'application est exécutée en boucle cela revient à avoir un pattern comme celui de `jacobi 2D`. Pour `mser`, lorsqu'il n'y a pas la phase qui utilise fortement l'extension, aucune instruction de l'extension n'est exécutée, par contre pour `FFT` il y a une faible utilisation de l'extension.

Pour les applications `sift`, `jacobi2D`, `basicmath` et `susan corner` l'utilisation est continue pendant toute l'exécution. Par contre le pourcentage d'utilisation peut être très différent, car `sift` a une moyenne de 30 % et `susan corner` a une moyenne de 0.7 %.

L'utilisation de l'unité vectorielle au cours de l'exécution des applications de la suite *Parsec* est montrée en figure 3.5. Les profils d'utilisation sont similaires à ceux de l'extension FPU, mais avec des sections moins intenses et plus de dispersion. Par exemple, il y a un profil de type créneaux, visible dans `bodytrack` et `canneal`, un profil d'utilisation très faible, visible dans `dedu` et `fluidanimate` et un profil intense tout le long de l'application, visible dans les applications `swap` et `vips`.



(a) Bodytrack (15 % de SIMD)

(b) Canneal (3% de SIMD)

(c) dedu (0.02 % de SIMD)

(d) Fluidanimate (0.01 % de SIMD)

(e) Swap (6 % de SIMD)

(f) Vips (7 % de SIMD)

FIGURE 3.5 – Utilisation de l'unité vectorielle x86. Chaque point du graphique est une moyenne sur 10K cycles. Il y a des phases de programmes distinctes pour l'utilisation de l'unité SIMD.

	Info	size of burst 100% I_{base}		
	total instructions	$\bar{x}$	σ	max
black	1.2G	18	137	508K
body	4.0G	90	3.4K	6M
cann	1.5G	185	22K	51M
dedu	5.6G	100K	1.8M	270M
flui	4.3G	19K	1.1M	212M
swap	3.8G	29	499	32K
vips	13G	57	8K	3M

TABLE 3.5 – Statistiques sur le nombre d'instructions consécutives n'utilisant pas l'extension SIMD x86 (seulement les I_{base}) - Applications de l'ensemble *Parsec*

Pour analyser plus précisément l'utilisation des extensions au cours de l'exécution des applications, la section suivante présente les résultats par rapport à la distance, en nombre d'instructions, entre deux instructions utilisant une extension matérielle.

Distance d'utilisation

La distance entre deux instructions vectorielles peut être mesurée en nombre d'instructions basiques (instructions qui ne correspondent pas aux extensions, appelées I_{base}). Ces sessions d'exécution d'instructions I_{base} nous intéressent car cela va nous permettre, par exemple, d'optimiser l'utilisation de l'extension matérielle. Par exemple, pour le cas où on souhaiterait éteindre l'extension (*power-gating*), ou transférer l'exécution de l'application d'un CE à un CB. Ces actions ont un coût, il est donc nécessaire que ces sections d'exécution soient assez grandes, de manière à contrebalancer ce coût.

Pour les applications *Parsec*, la distance entre deux instructions vectorielles a été mesurée grâce à l'outil *pintool*, à l'aide de la méthode qui est décrite en section 3.3.1. Le Tableau 3.5 présente les statistiques qui ressortent de cette étude. La première colonne est le nombre total d'instructions exécutées. La colonne $\bar{x}$ est la moyenne de la distance entre deux instructions vectorielles, c'est-à-dire le nombre d'instructions basiques qui se suivent. La colonne σ est l'écart type, et la colonne *max* est la plus grande distance mesurée.

Pour les applications `blacksholes`, `bodytrack`, `canneal`, `swaptions` et `vips`, la moyenne est inférieure à 200 instructions. Cela montre que l'utilisation est répartie tout au long de l'application. Par contre pour les applications `bodytrack`, `canneal` et `vips`, l'écart type et la section maximale sont grands, montrant des opportunités d'optimisation.

Si le coût pour déplacer une tâche d'un CE vers un CB est estimé à plusieurs centaines voire milliers de cycles, la section sans instruction étendue doit être largement supérieure. Seulement `fluidanimate` et `dedup` sont des applications présentant un comportement adéquat pour être optimisées sans transformation.

3.4.3 Discussion

Dans cette étude, deux moyens d'utilisation des extensions ont été testés. Le premier moyen est l'utilisation d'extension introduit par le compilateur (ex. pour le calcul vectoriel). Le programmeur n'a pas exprimé explicitement l'utilisation de la vectorisation, mais le compilateur, dans un processus d'optimisation, a essayé d'exploiter au maximum l'extension vectorielle. Le deuxième moyen est l'utilisation explicite (par exemple pour le calcul sur virgule flottante). Le programmeur déclare explicitement des variables avec un type `float` ou `double` et fait appel à

des fonctions de librairies de type flottantes, qui font donc référence à des opérateurs de type flottants, ainsi transformés en instructions spécialisées.

Au final, quelle que soit la méthode d'utilisation de l'extension, leur intensité d'utilisation globale dans les applications est souvent faible : la plupart des applications (excepté les noyaux) utilisent moins de 10 % du temps les extensions matérielles.

Par contre, d'après l'étude des noyaux *polybench* et l'analyse de la variabilité de l'utilisation au cours de l'exécution des applications, nous remarquons aussi des phases distinctes d'utilisation intense ou d'utilisation faible (voire de non-utilisation).

Les phases d'utilisation intense (ou les applications à très forte utilisation comme `sift` et `trmm`), vont être extrêmement impactées par le fait d'avoir ou non l'extension appropriée sur le cœur. Par contre les phases d'exécution où l'extension n'est pas utilisée, comme dans `Tracking` ou `Jacobi 2D`, vont pouvoir être exécutées sur un CB sans perdre en performance et en économisant le cout énergétique statique de l'extension. Les phases de faible utilisation sont les phases les moins efficaces car le surcoût énergétique de l'extension n'est pas contrebalancé par les gains en temps. Des applications qui ont un taux moyen d'utilisation faible comme `basicmath`, `susan corner`, `vips` et `swaptions`, et une distance moyenne faible (moins de 60 instructions entre deux instructions vectorielles pour `vips` et `swaptions`), ne vont pas être efficaces avec des méthodes dynamiques de migration. Il faudrait d'autres méthodes pour transformer le code, et ainsi pouvoir exécuter certaines parties (qui n'utilisent pas un CE efficacement) sur un CB. Ces hypothèses sont testées et analysées chapitre 5 et chapitre 6.

Pour résumer, l'apparition de ces phases va permettre l'optimisation en alternant l'utilisation de l'extension quand cela est nécessaire. Mais d'après les résultats, certaines applications n'ont pas de phases distinctes, ce qui va demander de gérer les applications à différents niveaux de granularité. Le chapitre suivant explore cette piste.

3.5 Perspectives

L'étude menée dans ce chapitre pourrait s'étendre à d'autres types d'applications. La même étude pourrait être réalisée en considérant d'autres points de vue, par exemple en analysant l'utilisation des extensions dans les différents fils d'exécution (*thread*) d'applications parallèles. Ainsi nous pourrions vérifier si dans les applications parallèles (type dataflow) il serait possible d'associer certains *threads* à certains types de cœurs. Sur ce sujet, une étude [SP09] a été réalisée pour l'application parallèle *JPEG*, en optimisant certains thread pour seulement certains types de cœurs. Cette étude montre des résultats prometteurs pour cette approche et pourrait être étendue avec les applications étudiées ici.

Une autre perspective serait d'analyser l'utilisation de plusieurs extensions en même temps. Dans notre étude, seulement une extension a été activée, mais il serait aussi intéressant d'en activer plusieurs en même temps, pour ainsi voir si les extensions sont utilisées simultanément. Cela permettrait de mieux choisir les types de cœurs à implémenter et leur répartition dans les architectures asymétrique en fonctionnalités.

3.6 Conclusion

Ce chapitre explore les applications de l'état de l'art et analyse les différents profils d'utilisation des extensions matérielles. Notre objectif est de vérifier si les profils de l'utilisation des extensions par les applications permettent des optimisations de gestion de ressources.

D'après nos résultats pour les extensions vectorielles et FPU étudiées, différents types d'utilisation sont distincts dans l'ensemble des applications et certains sont exploitables pour optimiser l'utilisation des extensions. Par exemple, des périodes de forte ou faible utilisation émergent

clairement. Dans ces cas de figure, des optimisations sont envisageables. Néanmoins, il existe des cas où les facteurs intermédiaires tels que le compilateur ou le chemin de contrôle peuvent rendre le profil d'utilisation difficilement maitrisable par le programmeur.

Dans certaines applications, des périodes avec un taux d'utilisation très faible sont mises en évidence. Les questions suivantes peuvent se poser : est-ce vraiment utile d'utiliser l'extension dans ces cas ? Et d'un point de vue énergétique, quelles en sont les conséquences ? De plus, les différentes tailles de périodes observées dans les applications montrent qu'il faut agir à différents niveaux de granularité. Enfin, des méthodes matérielles et logicielles existent déjà pour exploiter plus efficacement les extensions matérielles. En s'appuyant sur ces méthodes existantes, le chapitre suivant étudie ces questions et explore les principales solutions à différents niveaux de granularité, afin de comprendre les avantages et les limites de chacune, et ainsi obtenir des informations supplémentaires pour proposer une solution adéquate.

Chapitre 4

Gestion de l'extension FPU dans les multi-cores

Sommaire

4.1 Introduction

Les extensions matérielles intégrées dans les cœurs sont souvent faiblement utilisées et ont un coût énergétique et surfacique non-négligeable. Dans le Chapitre 3, il a été montré qu'il y a différents profils d'utilisation : utilisation faible, utilisation intense, utilisation divisée en phases ou utilisation totalement dispersée. Ces caractéristiques peuvent être exploitées statiquement ou dynamiquement, et à différents niveaux de granularité. En optimisant l'utilisation des extensions, des gains en énergie peuvent être réalisés, mais cela peut aussi impliquer des coûts en performances.

Le but de cette étude est d'explorer les gains et les coûts de différentes solutions, pour ainsi en savoir plus sur la manière de gérer les extensions dans un multi-cœur. Pour cela, trois solutions inspirées de l'existant sont modélisées et appliquées sur des applications de l'état de l'art. Ainsi, il est possible de voir leurs différences et leurs limites, et avoir des informations pour rechercher d'autres solutions de gestion des extensions.

Chacune des solutions étudiées agit à son niveau de granularité : application, ordonnanceur et instruction. Les contraintes sont d'une part de partir de binaires qui pourraient s'exécuter sur un multi-cœur symétrique habituel. Cela permet de réutiliser des binaires existants, d'être transparent vis-à-vis de l'utilisateur et de faciliter la programmation. Une deuxième contrainte est de ne pas modifier le code *en-ligne* afin d'éviter la mise en place de processus complexes au moment de l'exécution. Les solutions au niveau application et ordonnanceur sont basées sur une architecture multi-cœur asymétrique en fonctionnalités (FAMP), où pour rappel les extensions ne sont implémentées que sur certains cœurs. Il y a donc des cœurs basiques (CB) sans extensions et des cœurs étendus (CE) avec extensions. Les applications vont être placées sur un type de cœur en fonction de leurs besoins en extension au début de l'application (pour le niveau de granularité application) ou à chaque événement de l'ordonnanceur (pour le niveau ordonnanceur). La solution au niveau instructions modélise une solution de type *power-gating*, où automatiquement l'extension s'éteint s'il n'y a pas d'instructions pour l'extension pendant une certaine période.

L'extension FPU de ARM est utilisée comme support d'expérimentation. Sa forte accélération et son utilisation dans la plupart des applications permettent d'avoir un cas d'étude assez représentatif.

Le fait d'utiliser un CB et d'éteindre un cœur avec une extension ou juste d'éteindre l'extension permet d'économiser en énergie. Mais le déplacement des applications ou l'attente de l'allumage ajoute une dégradation en performance. Ces gains et ces surcoûts vont être étudiés et comparés entre eux.

Le chapitre s'organise comme suit : la Section 4.2 présente les solutions étudiées pour chaque niveau de granularité. La Section 4.3 décrit les méthodes d'expérimentations pour mesurer la performance et l'énergie lors de cette étude. La Section 4.4 présente et analyse les résultats pour chaque niveau de granularité et les compare.

4.2 Solutions étudiées

Dans cette section, les trois solutions, de la plus gros grain à la plus fin grain, sont présentées L'objectif de chaque solution est de sélectionner les bons instants temporels où il faut utiliser l'extension ou ne pas l'utiliser. Leurs caractéristiques et leur intégration dans le système global sont expliquées. Ces trois solutions ont été choisies car elles sont les plus utilisées aujourd'hui et elles interviennent à différents niveaux de granularité : application, ordonnanceur et instruction.

Les trois niveaux de granularité sont présentés dans les sections suivantes.

4.2.1 Niveau de granularité application

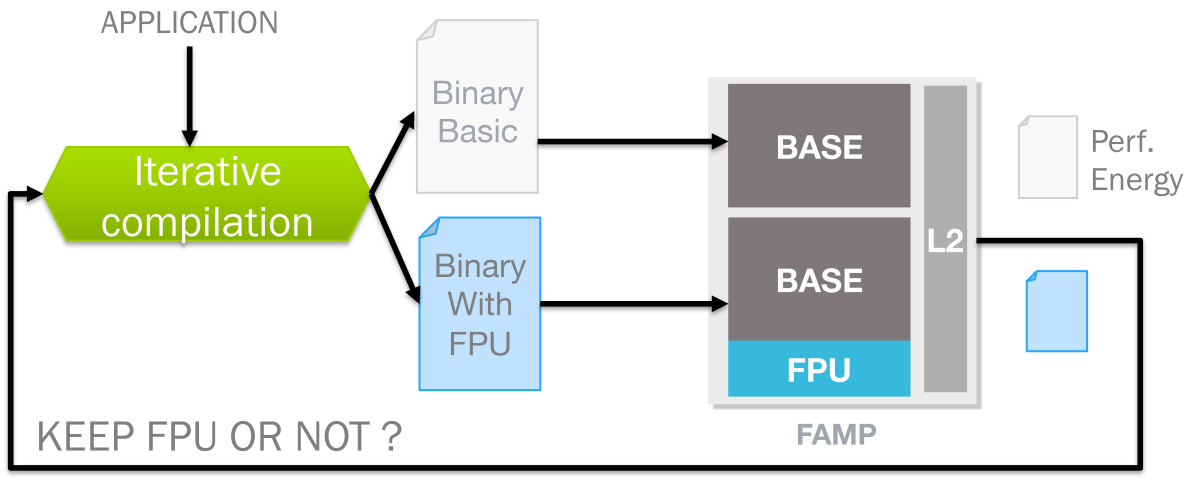


FIGURE 4.1 – Description de la solution pour gérer la FPU au niveau application

Au niveau application, le choix d'utiliser ou de ne pas utiliser la FPU est fait au moment de la compilation, et cela va agir sur toute l'application. La figure 4.1 présente le fonctionnement de la solution. Cette solution s'appuie sur le principe de la compilation itérative. C'est-à-dire que l'application est compilée avec différentes options puis elle est testée sur la plateforme. En fonction des mesures de performance et de la consommation énergétique, la meilleure configuration est sélectionnée pour ensuite être utilisée en ligne. Sur la figure 4.1, deux binaires avec et sans l'utilisation de la FPU sont créés et exécutés sur les cœurs correspondants. Après avoir exécuté le binaire compilé avec les options FPU sur un cœur FPU puis le binaire sans les options FPU sur un cœur sans FPU, la meilleure version (en fonction des objectifs énergétique et/ou performance) est sélectionnée. Au moment de l'exécution réelle du système, les applications utilisant la FPU peuvent être annotées de façon à indiquer que l'extension sera utilisée (ou non) pendant l'exécution. Si l'annotation indique l'utilisation de la FPU, le gestionnaire d'application doit faire exécuter l'application sur un cœur avec une FPU, dans le cas contraire, le gestionnaire peut la placer sur n'importe quel cœur (et en particulier un CB).

Cette solution est donc statique avec un choix pour toute l'application. Même s'il y a des phases utilisant la FPU, ces phases sont transformées en phases basiques en utilisant des appels à la librairie d'émulation. La transformation de toutes les phases peut impliquer une forte dégradation en performance. Le niveau de granularité suivant s'intéresse donc aux phases internes de l'application et apporte plus de dynamisme.

4.2.2 Niveau de granularité ordonnanceur

Pour la solution au niveau ordonnancement, le choix d'utiliser la FPU est fait à chaque événement de l'ordonnanceur. Dans le contexte d'une architecture SMP, les cœurs sont partagés entre plusieurs applications et le système donne l'accès aux ressources pour chaque application seulement par période. Ces périodes sont appelées *quantum*.

Le principe de cette solution est qu'à chaque nouveau *quantum*, l'ordonnanceur peut choisir de mettre l'application sur un cœur avec ou sans FPU. Pour cela, l'application est compilée

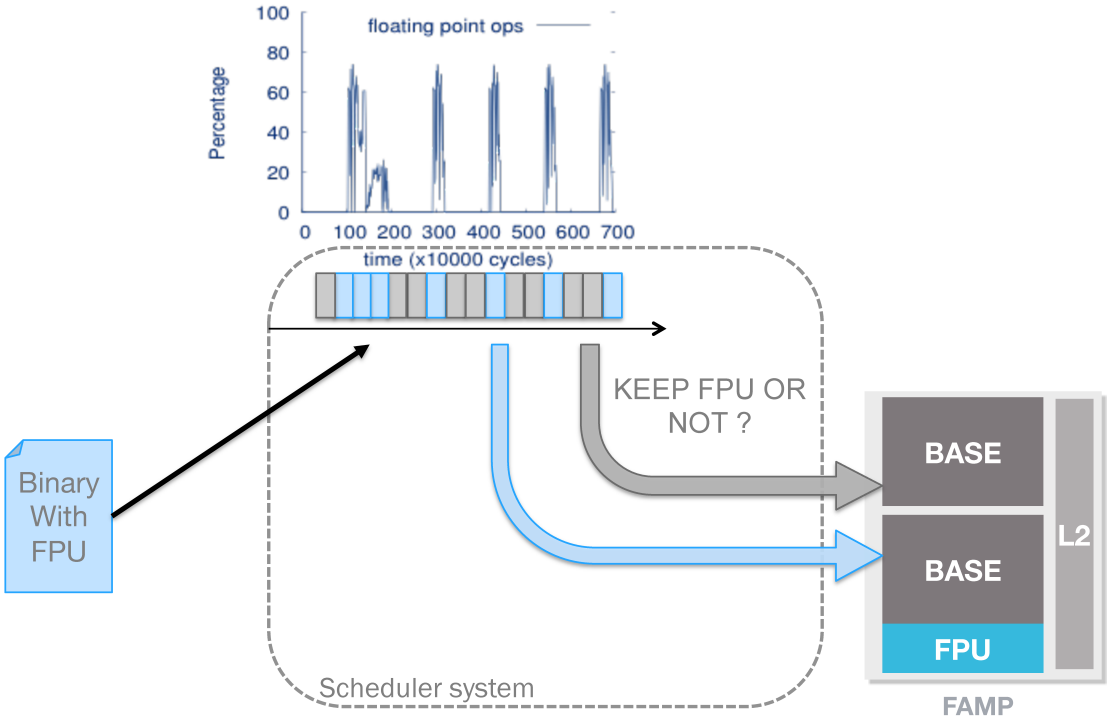


FIGURE 4.2 – Description de la solution pour gérer la FPU au niveau ordonnanceur

comme si elle était toujours exécutée sur un cœur avec une FPU. Ensuite, lors de l'exécution, si l'ordonnanceur détecte que la FPU n'est pas utilisée dans le prochain *quantum* de temps alloué, et ainsi place l'application sur le cœur adéquat. Dans le cadre de cette étude la prédiction de l'utilisation de la FPU est réalisée par un prédicteur parfait (dit *oracle*). C'est-à-dire que la prédiction ne se trompera pas sur l'utilisation. Normalement, dans le cas d'une erreur de prédiction, le cœur sans l'extension lève une interruption lorsqu'il doit exécuter une instruction qu'il ne connait pas. Ainsi l'ordonnanceur peut reprendre la main et replacer la tâche sur un cœur avec l'extension. Cette méthode est décrite dans les travaux [LBK+10]. Dans le cadre de cette étude, pour éviter le biais d'erreur de prédiction, l'ordonnanceur simulé a accès aux futures instructions exécutées.

La figure 4.2 présente le fonctionnement de la solution au niveau ordonnanceur. **L'efficacité de cette solution dépend du profil d'utilisation des extensions dans les applications et de l'habileté à capturer les phases ne contenant que des instructions basiques.** Sur la figure 4.2, le graphique montrant l'utilisation de l'exécution de `tracking` est visible, et les *quantum* de l'ordonnanceur sont représentés sous le graphique par les carrés bleus et gris. Les carrés bleus correspondent aux *quantum* où des instructions FPU ont été détectées. Ils sont donc exécutés sur le cœur avec une FPU. Les carrés gris sont les *quantum* contenant seulement des instructions basiques. En fonction de la largeur de ces carrés, ce qui correspond à la taille du *quantum*, les phases basiques sont plus ou moins détectées. Dans l'étude, nous faisons varier la taille du *quantum* pour en analyser les limites.

La taille du *quantum* ne doit pas être trop petite car les coûts de migration deviendraient trop importants. Afin d'avoir une gestion encore plus fin grain, la solution suivante agit au niveau instructions.

4.2.3 Niveau de granularité instructions

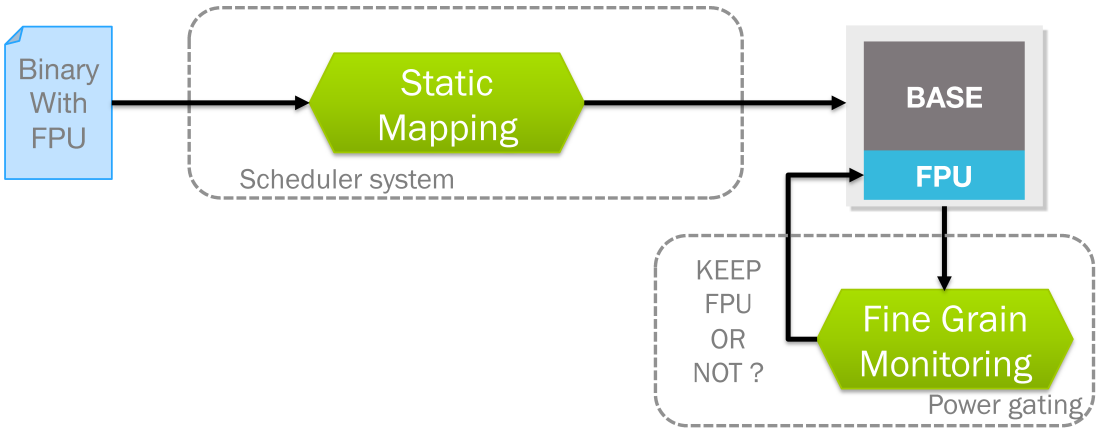


FIGURE 4.3 – Description de la solution pour gérer la FPU au niveau instructions

Le niveau instructions est réalisé par une solution de type *power-gating*, c'est-à-dire de pouvoir allumer ou éteindre dynamiquement l'extension matérielle. Cette solution est réalisée par une machine d'état mise en œuvre matériellement dans un contrôleur autonome. Contrairement au *clock-gating* qui ne réduit que la consommation énergétique dynamique, le *power-gating* permet de réduire l'énergie statique et principalement celle des registres. Mais comparée aux autres solutions, elle ajoute un coût surfacique (de 20 % à 50 % de la surface de chaque registre) et de la complexité dans le cœur [KFA+07].

Néanmoins, vu qu'elle agit directement dans le pipeline du cœur, elle peut être très réactive. La figure 4.3 présente le fonctionnement de la solution au niveau instruction. Le binaire est placé sur un seul cœur de façon statique, mais cela peut aussi fonctionner avec un autre type d'ordonnanceur plus dynamique. La solution analyse les instructions à venir et elle choisit d'éteindre ou d'allumer la FPU en conséquence.

Elle est utilisée dans les plateformes multi-cœurs symétriques pour réduire la consommation statique des extensions. Par exemple, le design du cortex A9 de ARM propose cette possibilité [ARM12]. Cette solution réduit la consommation dynamique et statique de l'extension mais n'adresse pas le problème de l'occupation de surface.

Pour étudier ces trois solutions, la méthode d'expérimentations est décrite dans la section suivante.

4.3 Méthodes d'expérimentations

Le processus d'expérimentation est présenté en figure 4.4. Il permet de simuler la performance et le coût énergétique de l'exécution des applications avec les méthodes étudiées. L'évaluation de la performance et de l'énergie est divisée en trois étapes consécutives : la simulation de performance, la simulation de la méthode de gestion de la FPU et la simulation énergétique. Des outils de simulations sont utilisés car il n'existe pas encore d'architecture FAMP en production.

Pour la simulation de performance, l'outil *Gem5* est utilisé pour simuler un Cortex A9 de ARM ayant une fréquence d'horloge à 1 Ghz. Les détails sur la configuration de *Gem5* sont les mêmes que ceux indiqués dans la Section 3.3.2. Pour chaque simulation, il est possible de configurer la taille des blocs d'analyse. Par exemple, il est possible d'avoir les statistiques d'exécution

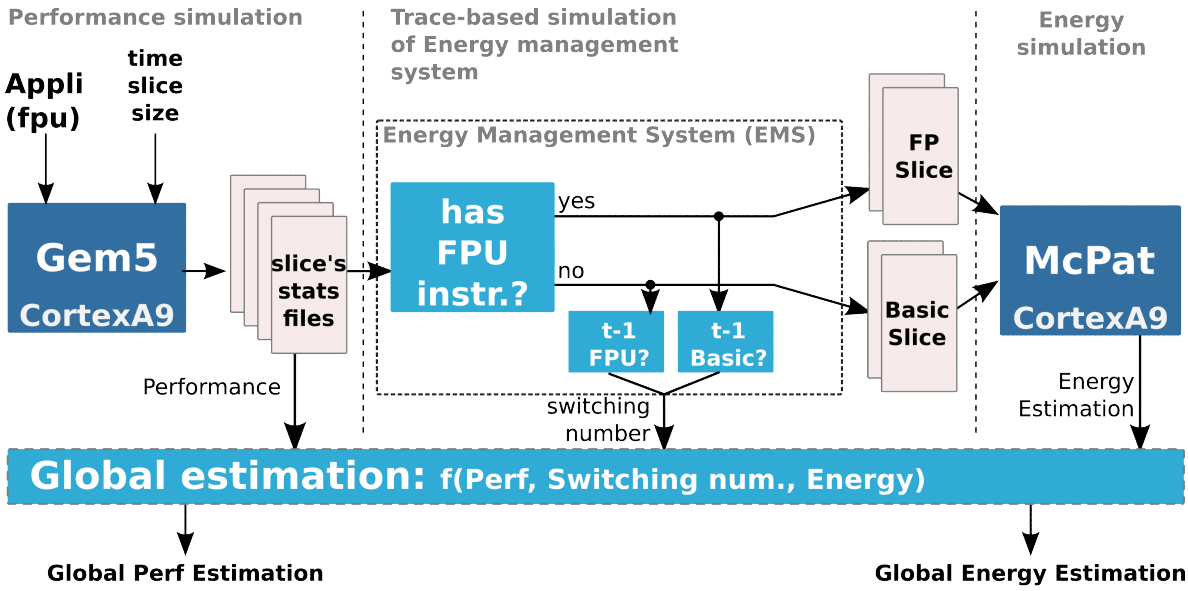


FIGURE 4.4 – Méthode d'évaluation. Simulation d'une architecture multi-cœur asymétrique en fonctionnalités (FAMP) avec système de gestion d'énergie (EMS). L'EMS est basé sur une simulation sur trace d'exécution.

pour toute l'application, ou par tranche de 100 cycles, 1K cycles, 10K cycles, etc. Cela permet de simuler les tailles de *quantum* de l'ordonnanceur et du power-gating. Toutes ces données sont ensuite transmises à la partie qui simule les systèmes de gestion de la FPU.

Les différents systèmes de gestion de la FPU (EMS) sont modélisés grâce à une technique de simulation sur trace (trace-based simulation). En parcourant les traces d'exécution et les données récupérées par *Gem5*, l'EMS sélectionne quelle partie du code est exécutée sur un cœur avec FPU (ou sur un CB). En fonction de la méthode sélectionnée, il y a plus ou moins de *quantum* à gérer, mais les méthodes sont similaires : si dans le *quantum* il y a au moins une instruction pour la FPU alors le *quantum* est considéré flottant (FP slice), sinon il est considéré basique (Basic Slice). Lorsqu'une application a deux *quantum* qui se suivent et qui ne vont pas sur le même type de cœur, une commutation (*switching number*) est comptabilisée (bloc *t-1 FPU ?* et *t-1 basic ?* dans le schéma). Chaque commutation va rajouter un coût en performance et en énergie. Pour le niveau application, ce module de gestion n'est pas utilisé car l'application est considérée comme une seule grande section.

Les données sont ensuite utilisées par le simulateur de consommation d'énergie.

La simulation de l'énergie consommée est réalisée avec l'outil *McPat* [LAS+13] créé par HP. Un cortex A9 de ARM y est configuré grâce aux fichiers `config` et `stats` de *Gem5*. La température est configurée à $85°C$. Seulement l'énergie du cœur sans les caches est considérée. Pour réaliser un modèle d'architecture FAMP, le modèle interne du cœur de type "embarqué" de *McPat* a été modifié. Une option de configuration a été ajoutée pour avoir le choix de simuler un cœur (de base similaire) avec ou sans une FPU. *McPat* a deux étapes : une de configuration de la plateforme et une de simulation. La partie configuration prend environ une minute. Il faudrait normalement exécuter *McPat* pour chaque fichier de `stats` extrait de *Gem5*, ce qui est très long surtout lorsque que la granularité d'actions est à 1K cycles (c'est à dire un fichier de stats tous les 1K cycles) soit des milliers de fichiers. Pour accélérer la simulation, un outil

d'assemblage de fichier `stats` a été réalisé. Il permet d'assembler tous les fichiers de stats en seulement deux paquets (celui tournant sur un CB et celui tournant sur un cœur avec FPU). Les données dans les fichiers `stats` sont de différents types (ex. boolean, valeur et pourcentage), ainsi le traitement a été différent selon la nature de chaque statistique (additionner les variables, faire des moyennes ou encore prendre la valeur haute). Ainsi l'outil *McPat* est exécuté seulement deux fois (une fois par type de cœur), ce qui permet d'accélérer significativement la simulation.

Au final, pour mesurer la performance et l'énergie consommée par l'exécution totale, les commutations ou les coûts d'allumage de la FPU sont ajoutés aux résultats finaux. Au niveau ordonnanceur, seulement les coûts de commutations additionnelles sont ajoutés, les coûts de changement de contexte à chaque fin de *quantum* dus à l'intervention du système d'exploitation ne sont pas ajoutés. En effet, étant donné que les résultats sont comparés à un multi-cœur symétrique, ces coûts sont les mêmes dans les deux cas. Ces coûts sont expliqués dans la partie résultats. Pour la suite, nous les appelons $T_{SwitchCost}$.

Nous allons présenter maintenant les équations qui nous permettent d'estimer la performance et la consommation énergétique globales.

L'équation 4.1 suivante permet de calculer la performance globale :

$$\widehat{Perf_{global}} = \sum T_{slices} + NbSwitch * T_{SwitchCost} \tag{4.1}$$

La variable T_{slices} est la performance pour chaque slice récupérée grâce à l'exécution dans Gem5. $NbSwitch$ est le nombre de commutation réalisées pour cette application.

L'équation 4.2 suivante permet de calculer l'énergie globale :

$$\widehat{Energy} = ECore_{basic} + ECore_{FPU} + NbSwitch * \overline{E_{SwitchCost}} \tag{4.2}$$

$ECore_{basic}$ et $ECore_{FPU}$ sont les consommations énergétiques calculées avec *McPat* pour chaque cœur pour une application donnée. Étant donné que la commutation implique surtout des transferts de données (couteux en énergie), la constante $\overline{ESwitchCost}$ a été estimée comme étant le plus haut pic d'énergie fois le temps estimé à commuter $TSwitchCost$.

La section suivante présente les résultats obtenus avec le protocole qui vient d'être décrit.

4.4 Résultats

Les résultats en terme d'énergie et de performance pour les trois niveaux de granularité sont présentés dans cette section. Ces résultats sont comparés à une architecture type SMP contenant des cœurs ayant chacun une FPU. Enfin, une discussion sur ces trois niveaux de granularité est donnée.

4.4.1 Gestion de la FPU au niveau application

Au niveau de granularité *application*, le système choisit d'utiliser ou non la FPU pour toute l'application. Deux versions binaires ont été réalisées : l'une compilée avec l'option de compilation activant l'utilisation des instructions de la FPU (`-mFPU-abi=softfp`), et la deuxième compilée avec l'option pour utiliser la librairie d'émulation de calcul sur données à virgule flottante (`-mFPU-abi=soft`).

		FPU op.	Performance	Energy
I	dijkstra	0.0000	0.00	−26.76
I	search	0.0000	0.00	−28.24
I	sha	0.0000	0.00	−26.84
I	crc32	0.0000	0.00	−33.14
f	susan s	0.0346	−1.64	−26.17
f	bitcounts	0.0002	0.26	−23.91
f	patricia	0.0265	0.25	−29.98
f	mser	0.79	7.15	−20.85
f	svm	0.22	3.34	−23.00
f	susan c	0.65	14.45	−10.27
f	susan e	0.93	19.08	−5.25
F	basicmath	0.69	57.47	18.17
F	qsort	1.13	48.73	13.93
F	fft inv	1.40	33.16	3.67
F	localization sma	1.72	36.29	6.58
F	texture	6.40	113.35	40.00
F	tracking	11.36	143.98	49.67
F	localization sqcif	23.55	303.57	67.47
F	sift	29.47	391.54	74.86

TABLE 4.1 – Niveau Application - cœur étendu contre cœur basic. En pourcentage (%). $Perf = \frac{P_{Basic} - P_{Extended}}{P_{Extended}}$; $Energy = \frac{E_{Basic} - E_{Extended}}{E_{Extended}}$

Le tableau 4.1 présente les résultats de la comparaison des deux versions pour chaque application. La première colonne est le pourcentage global de l'utilisation de la FPU. La deuxième colonne est le gain en performance obtenu en utilisant un cœur avec une FPU. La troisième colonne est le gain en énergie à utiliser un cœur avec FPU comparé à un cœur sans FPU. Le gain est calculé grâce aux résultats obtenus avec *McPat*, en utilisant le modèle cœur sans extension ou le modèle de cœur avec une extension.

Par exemple, pour `basicmath`, il y a 0.69% d'instructions à virgule flottante 'Floating Point' (FP). En utilisant la version FPU, il y a un gain en performance de 57% et un gain en efficacité énergétique de 18%. Le gain est important par rapport au taux d'utilisation car certaines instructions FP peuvent accélérer jusqu'à 1000x.

Pour synthétiser, trois catégories apparaissent :

La première catégorie, notée (I) dans le tableau 4.1, correspond aux applications n'utilisant que des données entières, telles que `dijkstra`, `search`, `sha` et `crc32`.

Ces applications n'ont pas d'accélération lors de l'utilisation d'une FPU et sont évidemment plus efficaces sur un cœur sans FPU. Nous notons qu'il est possible que dans d'autres cas (ex. données d'entrées différentes) ces applications utilisent la FPU. De plus dans d'autres applications plus complexes, comme par exemple, avec du JITs ou de la méta programmation, une analyse comme celle-ci au niveau application pourrait devenir difficile à généraliser. Pour réaliser une étude plus précise au niveau application, il faudrait donc vérifier qu'avec les données de tests utilisées, toutes les sections du programme sont bien parcourues (couverture de code).

La deuxième catégorie, notée (f) dans le tableau 4.1, correspond aux applications ayant un usage de la FPU insuffisant pour être efficace en énergie, telles que `bitcounts`, `patricia`, `fft`,

`mser` et `svm`. Elles ont une meilleure performance lorsqu'elles sont exécutées sur un cœur avec FPU, mais ont une meilleure efficacité énergétique lorsqu'elles sont exécutées sur un CB. Elles perdent entre 5 % et 29 % en énergie à être sur un cœur avec FPU. Le temps gagné grâce à l'accélération ne contrebalance pas le surcoût de l'énergie statique de l'extension.

Il faut noter que `susan smoothing` est aussi plus rapide sur un CB. Cela est principalement dû aux instructions pour convertir les données en type *double* et aux transferts de données vers les registres flottants.

Par contre, le fait d'utiliser la librairie d'émulation, peut amener des coûts en performance importants, par exemple les performances de l'application `susan edges` sont dégradées de 19% lorsqu'elle est exécutée sur un cœur sans FPU.

La troisième catégorie, notée (F) dans le tableau 4.1, correspond aux applications qui sont plus performantes et plus efficaces en énergie sur un cœur avec une FPU, telles que `basicmath`, `texture` et `sift`. Leur accélération est si importante que le temps gagné permet de contrebalancer le surcoût de l'énergie statique de la FPU, et ainsi réduire la consommation globale en énergie.

Par contre, pour cette dernière catégorie (F), il y a encore des applications qui ont un taux d'utilisation très faible (moins de 2 %). De plus pour les applications (f) comme `susan edges`, le fait d'agir au niveau application implique une dégradation en performance assez importante lorsqu'on veut optimiser l'énergie. Il est donc nécessaire de gérer la variation de l'utilisation de l'extension à un niveau plus fin pour pouvoir mieux optimiser ces cas. La section suivante présente les résultats pour la solution qui agit au niveau ordonnanceur.

4.4.2 Gestion de la FPU au niveau ordonnanceur

	Quantum (cycles)	100K		1M		10M		100M	
		Overhead(OH)	E.gains	OH	E.gains	OH	E.gains	OH	E.gains
f	susan s	0.2	18.0	0.2	17.0	0.2	8.0	0.0	0.0
f	bitcounts	1.0	17.8	0.7	14.5	0.1	5.1	0.0	0.0
f	patricia	0.0	0.4	0.0	0.4	0.0	0.0	0.0	0.0
f	mser	1.3	19.2	0.9	18.3	0.0	0.0	0.0	0.0
f	svm	0.0	20.8	0.0	20.8	0.0	20.7	0.0	0.0
f	susan c	1.4	−1.9	0.0	0.0	0.0	0.0	0.0	0.0
f	susan e	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
F	basicmath	0.0	0.2	0.0	0.1	0.0	0.0	0.0	0.0
F	qsort	0.2	6.7	0.0	6.6	0.0	6.3	0.0	3.5
F	fft inv	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
F	localization small	2.3	−0.2	0.0	0.1	0.0	0.0	0.0	0.1
F	texture	1.9	−1.1	0.0	0.0	0.0	0.0	0.0	0.0
F	tracking	3.6	9.6	0.0	3.2	0.0	0.0	0.0	0.0
F	localization sqcif	0.3	−0.1	0.0	0.0	0.0	0.0	0.0	0.0
F	sift	2.1	−1.6	0.0	0.0	0.0	0.0	0.0	0.0
	average	0.0	5.9	0.0	5.4	0.0	2.7	0.0	0.2

TABLE 4.2 – Niveau Ordonnanceur avec l'approche de commutation. Le coût de communication est admis à 20μs. En pourcentage (%) $Overhead = \frac{Full - Opt}{Full}$

Au niveau de granularité ordonnanceur, le système de gestion peut faire commuter une

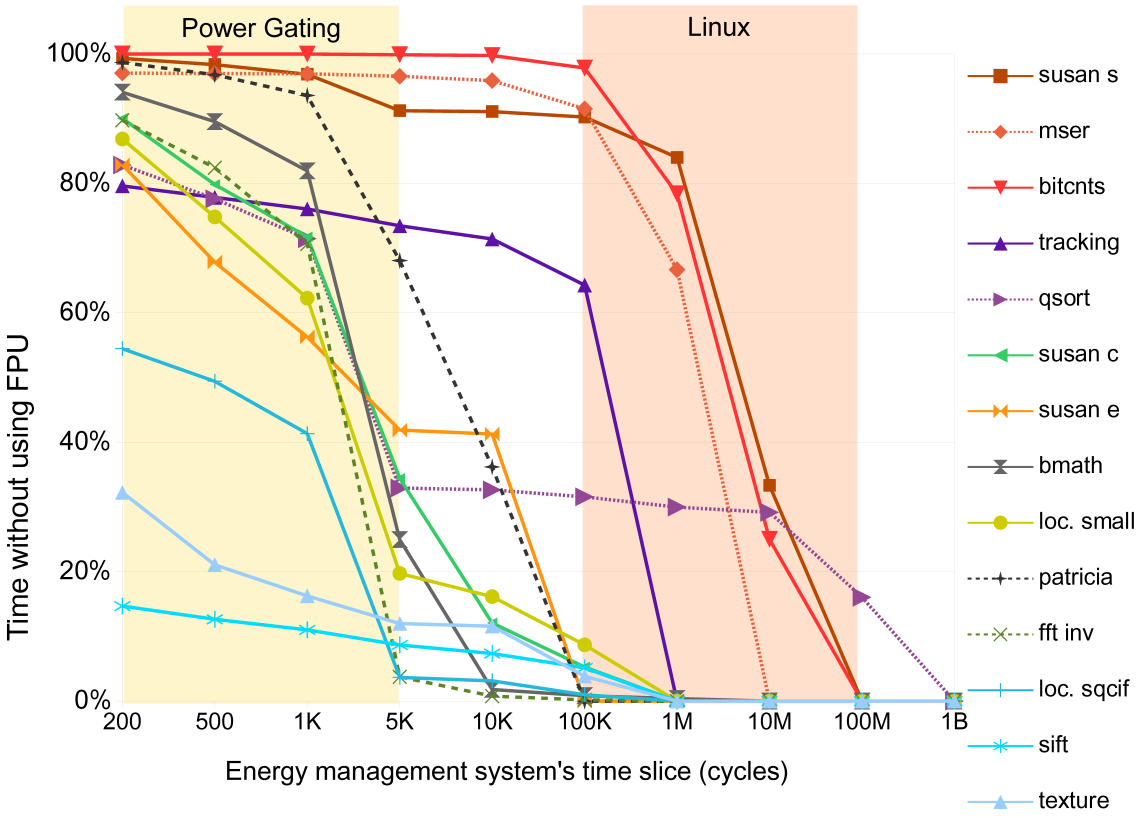


FIGURE 4.5 – Pourcentage du temps d'exécution pour lequel le système de gestion d'énergie détecte des phases basiques (sans aucune instruction pour la FPU).

application d'un cœur à un autre à chaque évènement de l'ordonnanceur et ainsi éteindre les cœurs qui ne sont pas utilisés. L'objectif est de pouvoir exploiter les phases d'utilisation de la FPU qui apparaissent dans les applications, telles que celles montrées dans la figure 3.3 du Chapitre 3.

La capture de ces phases dépend de la granularité des *quantum*. Plus la taille des *quantum* est fine, plus l'application est divisée en petites parties et plus les phases purement basiques sont capturées. Les phases purement basiques sont composées seulement d'instructions basiques, donc exécutables sans extension. Ainsi, les phases basiques détectées peuvent être placées sur un cœur sans FPU.

La figure 4.5 présente le pourcentage de l'application pour lesquelles des phases basiques sont détectées en fonction de la granularité d'action. Dans les ordonnanceurs actuels comme celui de Linux, la granularité (la taille des *quantum* i.e. le temps alloué pour un thread sur un cœur sans être interrompu) est entre 1ms et 100ms.

D'après la figure, les applications qui étaient détectées au niveau application, sont aussi bien gérées avec cette solution. Par exemple, les applications `susan s`, `bitcnts` et `mser` s'exécute à plus de 70 % du temps sur un CB avec un *quantum* de 1ms. Par contre pour les applications `susan edge` et `susan corner`, les phases basiques ne sont pas détectées. Ce qui correspond aux profils de l'utilisation décris par leurs graphiques figures 3.3 et 3.4. Par contre, avec cette solution, les application `tracking` et `qsort` ont des phases basiques détectées, donc sont exécutées une partie du temps sur un CB.

Le tableau 4.2 présente les gains énergétiques et les surcoûts de performance pour les différentes tailles de quantum. Dans ce tableau les applications utilisant seulement des données entières ne sont pas prises en compte (`dijkstra, search, sha, crc32`), étant donné qu'elles sont tout le temps sur un CB (comme au niveau application). Pour les autres applications, globalement, les gains d'énergie sont moins importants qu'au niveau application, par contre les surcoûts en performance sont plus faibles. Par exemple, `mser` est exécuté en consommant 18 % moins d'énergie pour un surcoût de seulement 0.9 % contrairement à 21 % avec 7 % de surcoût en performance pour la solution au niveau application. Les surcoûts en performance sont plus faibles car les parties utilisant la FPU ne sont pas transformées en appels à la librairie d'émulation, mais sont exécutées sur un cœur avec FPU. Cela permet de garder toute l'accélération produite par l'extension. Par exemple pour l'application `mser`, la dernière partie de son graphique figure 3.3 est bien exécutée sur un cœur avec FPU alors que toute la première partie du début est exécutée sur un cœur sans FPU. De plus, `qsort` et `tracking` qui sont aussi exécutées une partie de leur temps sur un CB, sont plus efficace en énergie (respectivement 6.6 % et 3.2 %) pour un surcoût inférieur à 0.01 %.

Pour les autres applications, les phases basiques ne sont pas capturées avec une granularité entre 1ms et 100ms.

Pour voir à plus fin grain, même si actuellement les ordonnanceurs actuels n'agissent pas à cette granularité, les applications ont été étudiées avec un *quantum* d'ordonnanceur à 0.01ms (10K sur la figure 4.5). Avec cette configuration, plus de phases basiques peuvent être capturées, comme pour `basicmath`, qui voit son temps sur un CB passer de 0 % à 65%, ou encore `localisation` et `sift` qui passent de 0 à plus de 10 %. Malheureusement, le coût de la commutation empêche l'amélioration de l'efficacité énergétique de l'exécution. Par exemple, l'application `sift` est commutée 3 fois mais au final, elle ne reste que 5 % du temps total de l'exécution sur un CB, ce qui donne un surcoût en temps d'exécution de 2.1 % sans économiser en énergie. Dans ce cas, l'application devrait être exécutée totalement sur un CE.

Une autre solution qui présente moins de surcoût en performance et qui permet d'être plus

fin grain est d'utiliser le *power-gating*. Cette solution est explorée dans la section suivante.

4.4.3 Gestion de la FPU au niveau instructions

	1μs	1K		5K		10K
		Overhead(OH)	E.gains	OH	E.gains	OH	E.gains
f	susan s	3.8	14.6	0.0	18.4	0.0	18.4
f	bitcounts	0.0	19.3	0.0	19.3	0.0	19.3
f	patricia	7.9	10.1	7.9	4.3	4.4	2.3
f	mser	0.1	21.8	0.1	21.7	0.1	21.6
f	susan c	19.0	−11.5	5.2	0.0	0.5	1.9
f	susan e	9.6	−1.6	0.2	8.6	0.1	8.6
F	basicmath	19.0	−9.4	5.8	−2.9	0.1	0.2
F	qsort	13.7	4.8	0.1	7.2	0.0	7.2
F	fft inv	29.7	−27.3	0.8	−0.4	0.0	0.0
F	loc. small	22.9	−20.	.0	3.0	0.1	3.5
F	texture	3.3	−1.2	0.1	2.5	0.1	2.5
F	tracking	1.1	15.5	0.5	15.8	0.1	15.9
F	loc. sqcif	18.3	−15.9	0.1	0.7	0.1	0.7
F	sift	1.0	0.9	0.3	1.4	0.0	1.5
	Average	10.7	0.0	1.6	7.10	0.4	7.4

TABLE 4.3 – Approche Power-gating. Le coût d'allumage de la FPU est admis ici à 1μs (1000 cycles). En pourcentage (%).

	0.2μs	200		500		1K		5K		10K
		Overhead(OH)	E.gains	OH	E.gains	OH	E.gains	OH	E.gains	OH	E.gains
f	susan s	0.76	19.10	0.75	18.91	0.75	18.61	0.0(	18.47	0.0(	18.45
f	bitcnts	0.01	19.35	0.01	19.35	0.01	19.34	0.01	19.33	0.01	19.31
f	patricia	1.58	20.94	1.58	20.50	1.58	19.77	1.58	13.91	0.88	7.64
f	mser	0.07	21.90	0.03	21.95	0.03	21.92	0.03	21.87	0.03	21.76
f	susan c	9.82	5.39	4.09	11.27	3.80	9.94	1.04	5.82	0.10	2.45
f	susan e	14.03	−2.09	5.87	6.14	1.92	9.21	0.03	8.77	0.02	8.74
F	basicmath	3.79	15.37	3.79	14.36	3.79	12.67	1.16	3.87	0.02	0.37
F	qsort	4.62	11.60	3.09	12.67	2.73	7.46	0.01	7.26	0.01	7.21
F	fft inv	6.10	10.73	5.94	9.36	5.94	6.78	0.15	0.47	0.0(	0.03
F	loc. small	10.07	4.60	6.42	7.29	4.58	7.17	0.19	4.13	0.03	3.64
F	texture	10.23	−7.50	2.33	1.25	0.67	2.56	0.02	2.65	0.02	2.61
F	tracking	1.57	15.57	0.84	16.24	0.22	16.75	0.10	16.36	0.03	16.04
F	loc. sqcif	4.16	5.69	4.09	4.76	3.66	3.69	0.03	0.87	0.02	0.78
F	sift	1.88	0.59	0.91	1.40	0.20	1.96	0.06	1.71	0.01	1.53
	Average	4.91	10.09	2.84	11.82	2.13	11.27	0.32	8.96	0.08	7.90

TABLE 4.4 – Approche Power-gating. Le coût d'allumage de la FPU est admis ici à 0.2μs (200 cycles). En pourcentage (%).

Au niveau instructions, pour économiser de l'énergie, la solution de *power-gating* peut allumer ou éteindre la FPU d'un cœur. Le coût pour chaque allumage de la FPU est estimé

entre 0.2μs (200 cycles avec un processeur de fréquence 1Ghz) et 1μs (1000 cycles). Ces coûts sont dominés par des aspects logiciels (sauvegarde des registres), plutôt que matériels (allumage physique de la FPU). Le coût à 0.2 μs considère des techniques de rétention des données dans les registres très rapides [KFA+07].

Le système de gestion va éteindre la FPU si dans les N prochaines instructions, il n'y a pas d'instructions pour la FPU. Pour observer les limites de cette solution, la taille N de cette fenêtre d'instructions a été sélectionnée entre 200 et 10K instructions. Comme indiqué dans la méthode d'expérimentation, la méthode de simulation du *power-gating* est basée sur des traces d'exécution, il est donc possible de connaître les instructions futures. Néanmoins, dans une réalisation concrète, un système de prédiction doit être ajouté. Dans cette étude, il a été choisi de garder un prédicteur dit *oracle* pour observer les gains maximums. Évaluer la précision des prédicteurs n'est pas le but de cette étude.

Premièrement, la figure 4.5, qui représente le temps passé sur un CB, montre que les phases basiques sont mieux capturées avec une granularité plus fine. Par exemple avec une granularité de 500 cycles, la plupart des applications permettent d'éteindre plus de 70 % du temps la FPU (excepté pour `sift`, `texture` et `localization_sqcif` qui sont respectivement à 15 % et 54 %).

Les tableaux 4.3 et 4.4 présentent les différentes configurations. Les colonnes représentent les surcoûts en performance et les gains en énergie pour les différentes tailles de fenêtre d'instructions. Dans ces tableaux les applications utilisant seulement des données entières (`dijkstra, search, sha, crc32`) ne sont pas prises en compte, étant donné qu'elles sont tout le temps sur un CB (comme au niveau application).

Le tableau 4.3 détaille les économies d'énergie et les pertes de rendement pour différentes tranches de temps avec un coût de démarrage de la FPU de 1 μs. Avec ce coût de démarrage, les résultats sont meilleurs lorsqu'une fenêtre de 10K-instructions est utilisée. Avec cette configuration, la solution est plus optimale que la solution au niveau ordonnanceur. Par exemple, on observe que les applications *susan e*, *tracking* sont bien plus efficaces. Par contre, si la fenêtre est réduite, le surcoût en performance réduit l'économie d'énergie pour certaines applications. Dans le tableau, les chiffres négatifs montrent que pour certaines applications, l'énergie consommée augmente par rapport à la plateforme à laquelle on se compare (SMP). Avec un coût de *power-gating* trop élevé, d'après les résultats, l'efficacité de cette solution est dépendante de l'application.

Le tableau 4.4 détaille les gains d'énergie et les pertes de rendement pour les différentes tranches de temps avec un coût de mise sous tension de 0.2 μs. Avec ce coût, les économies d'énergie sont intéressantes, avec une moyenne de 16 % avec une surcharge de 2 %. Pour les applications `patricia`, `susan corners`, les résultats sont meilleurs que dans le cas avec 1 μs de temps d'allumage. Pour les application `texture` et `sift`, presque aucun gain en énergie n'est réalisé, malgré le fait que leur utilisation globale de l'extension est sous les 30 %.

En conclusion, les solutions au niveaux instructions permettent des économies d'énergie sur plus d'application que les autres solutions. Mais cette solution nécessite aussi des mécanismes agressifs et intrusifs pour réduire les coûts pour d'allumage de la FPU, et cela implique une occupation de surface accrue. De plus, pour les applications telles que `sift, susan e` ou `texture`, cette solution ne se montre toujours pas efficace.

4.4.4 Discussion

D'après l'analyse des résultats des solutions et des différentes configurations, des points clés ressortent pour la gestion de la FPU.

Premièrement, d'après les résultats et la figure 4.5, deux pas de granularité apparaissent pour optimiser l'efficacité des applications sur des architectures multi-cœurs sans modifier le

code.
Le premier pas de granularité est proche de 1M cycles. A cette granularité, pour certaines applications (c'est-à-dire 4 sur 15 parmi les applications flottantes), les phases ne contenant que des instructions basiques de certaines applications sont déjà capturées et permettent des économies d'énergie. Pour ces applications, une solution au niveau application de type compilation itérative est plus économe en énergie, mais les applications peuvent être fortement ralenties car toutes les instructions utilisant l'extension sont émulées. Afin de prendre avantage des applications qui sont déjà compilées pour l'ensemble de l'ISA, et de continuer à exploiter l'accélération de l'extension, la solution dynamique au niveau ordonnanceur semble la plus adaptée. A l'étape de granularité 1M et pour les applications touchées, les économies d'énergie sont environ de 20 %, elles sont donc proches du maximum théorique du modèle (qui correspond à l'énergie de fuite de l'extension).

Le deuxième pas de granularité est proche de 1K cycles. A cette granularité, la solution de *power-gating* est efficace grâce à son faible surcoût en performance. Avec une solution matérielle agressive (activation de l'extension en 200 cycles) et une tranche de temps de 500 cycles, environ toutes les applications sont impactées et ont des économies d'énergie : 9 sur 15 applications ont entre 10 % et 20 % d'économies d'énergie. Mais la taille de la fenêtre d'observation des futures instructions a une forte incidence sur la performance. En effet, la performance peut être considérablement réduite si la taille de la fenêtre de coupe n'est pas suffisamment grande par rapport au coût d'allumage de la FPU. Un autre inconvénient vient du fait que le *power-gating* augmente la surface de la puce.

Cependant, dans une plateforme SMP, l'intégration du *power-gating* peut être lourd (ajout surfacique de 20 % à 50 % pour chaque registre, plus de la complexité dans le cœur [KFA+07]). Dans un SMP, la FPU est dupliquée dans chaque noyau, donc il ne vaut mieux pas rajouter plus de surface par FPU. Mais dans une architecture FAMP, il est possible d'avoir moins d'extensions FPU, donc une surface disponible plus grande par extension (plutôt que des FPU de petites tailles sur chaque cœur). Le *power-gating* pourrait être mis en œuvre en plus des deux solutions étudiées.

Un autre point est l'impact entre le type d'accélérateur et la méthode. Par exemple, une extension SIMD qui accélère seulement entre x2 et x8 (par rapport à plus de x100 pour FPU) n'aura pas la même gestion de granularité pour une efficacité optimale. L'utilisation d'une solution statique peut être plus utile qu'une solution dynamique dans ce scénario. Cela permettrait d'éviter de payer un coût de transfert pour gagner seulement une accélération de x4 pour une courte période de l'application.

Par rapport à l'état de l'art, tous ces résultats sont donnés par rapport à une plateforme SMP classique avec les cœurs contenant tous une FPU. Les gains en énergie sont très prometteurs, surtout avec la minimisation de la surface des cœurs basiques (ce qui va augmenter le pourcentage des extensions).

Ces travaux pourraient être comparés à des architectures PAMP comme le *big.LITTLE*. Néanmoins, étant donné que les deux asymétries (PAMP et FAMP) n'agissent pas sur les mêmes optimisations, il serait nécessaire de faire une étude sur l'intérêt à mettre en œuvre les deux asymétries.

Cette étude peut être extrapolée au cas d'architectures où la FPU est partagée, car les profils d'utilisation et la granularité de gestion du partage va aussi influencer l'efficacité des applications (par exemple l'attente pour la FPU, le coût d'allumage de la FPU).

Une limitation de cette étude provient du fait que l'on utilise des modèles et non des im-

plémentations réelles. Mais actuellement, cela n'est pas réalisable car il n'y a pas d'architecture FAMP en production. Cependant, grâce à cette modélisation, il est possible de comparer les 3 études et d'avoir des informations intéressantes sur la cohésion entre les profils de l'utilisation et la granularité d'action.

Granularité	Énergie sauvée	Surcoût en perf.	Surface
Application	9 %	7 %	réduit
Ordonnanceur	6 %	1 %	réduit
Instruction	12 %	3 %	ajoute

TABLE 4.5 – Résumé des trois approches (la moyenne ne comprend pas les applications qui ne travaillent que sur des données entières telles que `dijkstra`, `search`, `sha` et `crc32`)

En conclusion, les deux solutions extrêmes en granularité ont des résultats intéressants mais leurs contraintes posent un inconvénient majeur pour la scalabilité des multi-cœurs. Le tableau 4.5 résume les gains en énergie, le surcoût en performance et les conséquences au niveau surface. L'approche au niveau application apporte de bons gains en énergie et en surface, mais implique un fort surcoût en performance, principalement dû au fait de retirer toutes les instructions FPU de l'application, même celles conduisant à une forte accélération. L'approche au niveau instruction permet plus de gains énergétiques avec moins de surcoût en performance mais cela rajoute plus de surface que cela en enlève. L'approche ordonnanceur est la solution avec le moins de surcoût en performance mais aussi le moins de gains énergétiques. Par contre, l'approche ordonnanceur à un fort potentiel. D'une part, car le processus de changement de contexte est déjà mis en place dans les systèmes d'exploitation courants, et d'autre part, car cela ne demande pas d'ajout de matériel complexe. La solution fonctionne bien pour `tracking` et `Mser`, mais pour les autres applications à fort usage et faible accélération, les phases d'utilisation basique ne sont pas détectées. Dans ce chapitre, l'accent a été délibérément mis sur des solutions qui ne modifient pas le code lors de l'exécution, mais il est possible que la modification du code (hors-ligne ou en-ligne) puisse permettre d'avoir plus de phases à exécuter sur un CB sans trop perdre en performance. Par exemple, cela pourrait permettre de remplacer, réorganiser ou émuler des instructions à virgule flottante inefficaces et ainsi de maximiser le temps passé sur des cœurs basiques. Cette hypothèse va être étudiée dans les prochains chapitres.

4.5 Perspectives

Pour l'étude réalisée dans ce chapitre, les perspectives peuvent être d'étendre l'étude à d'autres extensions telles que les extensions vectorielles et cryptographiques. Le *power-gating* étant une solution qui pourrait être aussi utilisée dans n'importe quel type de plateforme en plus de la solution plus gros grain mise en œuvre, il serait intéressant d'analyser des prédicteurs et des solutions qui réduiraient les coûts d'allumage.

D'autres niveaux de granularité un peu plus transversaux pourraient être étudiés. Ces niveaux de granularité pourraient être non pas d'un point de vue temporel mais plutôt au niveau fonctionnalité et déroulement du programme, comme par exemple aux niveaux fils d'exécution (*thread*), tâches ou fonctions. Ainsi, l'utilisation de l'extension pourrait être définie manuellement à la programmation comme dans le papier [SP09] ou par un processus de détection d'utilisation à la compilation pour chaque fils d'exécution/tâche/fonction. Dans notre cas, les applications utilisées ne sont pas assez parallélisées pour réaliser cette étude au niveau fils d'exécution et tâche. Pour le faire au niveau fonction, cela demande un changement complexe au niveau compilateur et système d'exploitation, ce que nous avons voulu éviter. Une autre

solution qui pourrait être étudiée est celle de se placer au moment de la compilation JIT. Par exemple, en gérant la FPU au moment où la machine virtuelle compile le code à exécuter. Cela permettrait de retarder la compilation au dernier moment et d'exploiter les informations en ligne. Dans le cadre de cette thèse, pour être plus générique et moins dépendant des solutions fonctionnelles, nous avons choisi de faire une analyse avec des niveaux de granularité temporels qui étaient déjà pris en compte dans les architectures et outils actuels.

4.6 Conclusion

Ce chapitre étudie les gains énergétiques et les surcoûts en performance de trois méthodes pour gérer plus efficacement l'utilisation d'une extension matérielle de type FPU.

Sous la contrainte de ne pas faire des modifications en ligne du code, les méthodes se basent sur une plate-forme asymétrique avec une approche statique de compilation itérative, une approche dynamique légère de commutation au niveau ordonnanceur et une approche fin grain de type *power-gating*.

Les résultats montrent que la compilation itérative et l'attribution statique de code à des cœurs spécifiques peut économiser de l'énergie (moyenne de 9 %) par rapport à un multi-cœur où tous les cœurs ont une FPU, au prix d'une perte en performance jusqu'à 19 %. L'approche dynamique au niveau ordonnanceur peut économiser la même quantité d'énergie (jusqu'à 21 %, moyenne de 6 %) avec un coût de performance négligeable (moins de 1 %). Lorsque la surface de silicium n'est pas un problème, la solution de type *power-gating* est la solution la plus efficace pour économiser de l'énergie. Cette solution a une moyenne en économies d'énergie de 12 %. Néanmoins, les solutions de *power-gating* nécessitent des techniques agressives pour allumer/éteindre rapidement la FPU, ce qui peut affecter la mise à l'échelle des multi-cœurs.

Au final, même si un spectre large de solution et de niveau de granularité a été étudié, certaines applications ne sont pas impactées par ces solutions. Or, leur utilisation de l'extension est faible. Ces solutions réussissent à économiser de l'énergie quand elles captent des phases de non utilisation de l'extension, ce qui est très dépendant du profil de l'utilisation. Il suffit qu'il y ait seulement une instruction dans la prochaine fenêtre d'instructions, et cette fenêtre devra être exécutée sur un cœur avec une FPU allumée, même si l'instruction n'accélère que peu. Pour pouvoir avoir un placement plus optimal, il ne faut pas se limiter à l'apparition des instructions mais à l'intérêt réel en performance et en consommation d'énergie à utiliser l'extension. Ceci permet à la solution de gestion de tâches de choisir l'utilisation de la FPU en fonction de ces données.

Dans le premier chapitre, il a été montré que différents types d'utilisation de l'extension FPU et SIMD apparaissaient au cours d'une même application. Dans ce chapitre, le but était d'observer les conséquences aux niveaux énergie, performance et surface lors de la gestion de la FPU à plusieurs niveaux de granularité. Les résultats montrent que pour certaines applications ayant un taux très faible d'utilisation de l'extension, il n'est pas possible d'avoir des gains énergétiques intéressants. Cela est causé par l'apparition d'instructions flottantes qui forcent l'ordonnanceur à placer la tâche sur un cœur avec FPU. Nous pensons que le choix d'utiliser les extensions doit être fait en temps réel et devrait être guidé non par le fait que ces instructions sont présentes dans le code, mais aussi par d'autres métriques. Ces métriques permettraient d'estimer "l'intérêt" d'activer une extension (ou de commuter une tâche) en associant des gains et des coûts aux différents choix qui se présentent à l'ordonnanceur. Pour valider cette hypothèse, dans le prochain chapitre, nous proposons une méthode pour estimer l'intérêt de la FPU en temps réel, puis le chapitre suivant étudie l'impact que peut avoir cette approche d'ordonnancement.

Chapitre 5

Estimateur d'accélération par extension matérielle

Sommaire

5.1 Introduction

Dans ce chapitre, nous présentons un modèle qui permet d'estimer l'accélération fournie par une extension matérielle représentée par un ensemble d'instructions spécialisées. Cette estimation peut par exemple permettre au compilateur ou au système d'exploitation d'avoir plus d'information afin d'effectuer des meilleurs choix d'utilisation d'une extension. En effet, cette estimation permet d'éviter, dans une certaine mesure, de commettre des erreurs dans les choix liés à l'utilisation de l'extension et d'impliquer donc des réductions en performance trop importantes.

Par exemple, dans le cadre d'un multi-cœur asymétrique en fonctionnalité, le système d'exploitation doit optimiser le placement des tâches sur les différents types de cœurs en fonction de l'utilisation des extensions. Le fait de minimiser l'utilisation d'un cœur avec extensions matérielles peut permettre des gains en énergie, comme présenté dans le chapitre précédent. Par contre, le cas contraire peut être aussi vrai, puisque si trop d'instructions spécialisées sont remplacées par des routines d'émulation, la dégradation peut être très importante (typiquement entre 20x-50x pour du calcul flottant).

Dans un scénario classique, une application contenant des instructions étendues ne peut pas en soi être exécutées sur un CB. Cette limitation peut être surmontée et plusieurs études fournissent des solutions pour placer une tâche sur n'importe quel cœur indépendamment des extensions matérielles utilisées. Ils sont basés sur la réécriture de binaires [GNL13, DVT12], la virtualisation des extensions [NDR+11], ou encore la traduction binaire [GNVL14]. Ces techniques sont détaillées dans la Section 2.4. L'intérêt de pouvoir exécuter une application sur n'importe quel cœur permet de donner au système d'exploitation une plus grande flexibilité pour placer les tâches. Mais cette liberté doit être utilisée avec parcimonie pour éviter de trop perdre en performances. L'accélération potentielle d'une tâche qui utilise une extension devrait pouvoir être connue avec précision par le système d'exploitation, afin d'optimiser dynamiquement le placement de celle-ci. Ainsi, il pourrait détecter les phases qui sous-utilisent l'extension matérielle et les placer sur un CB. Étant donné que ces phases n'utilisent que peu la FPU, les performances de celles-ci seraient peu impactée.

Dans le cadre d'un multi-cœur asymétrique en fonctionnalité, de la flexibilité dans le placement de tâche et l'estimation de l'accélération fournie par une extension peuvent permettre des optimisations mono-tâche et multi-tâche :

- Optimisation mono-tâche : la flexibilité dans le placement de tâches peut permettre au système d'exploitation de choisir le bon cœur pour la bonne tâche et ainsi optimiser localement la consommation d'énergie ou la performance . Le prix d'une extension est un surplus en consommation statique. Cependant si l'accélération de l'extension atteint un certain seuil, l'énergie économisée par le gain de temps équilibre le surplus d'énergie statique. Dans le cas contraire, l'exécution de la tâche sur un CB permet une meilleure efficacité énergétique. Donc, en connaissant l'accélération et le seuil propre à l'extension, le système d'exploitation peut économiser de l'énergie en plaçant une tâche sur un cœur avec une extension, uniquement lorsque l'accélération dépasse ce seuil.
- Optimisation multi-tâche : la flexibilité dans le placement de tâches peut permettre au système d'exploitation d'optimiser l'exécution globale. Par exemple, si un CB est disponible, sans flexibilité de placement il ne sera que peu utilisé. Mais avec plus de flexibilité et de l'information sur l'utilisation des extensions, le système d'exploitation peut sélectionner la tâche la moins accélérée parmi les tâches en attente d'un cœur avec une extension, et l'exécuter sur un CB. Le système d'exploitation peut également utiliser la flexibilité pour permettre à des tâches plus prioritaires de fonctionner sur un CE tout en exécutant les

autres tâches sur des CB. Cela permet d'avoir moins de perte de performance que lors de l'attente pour un CE. Différents exemples de placement de tâches sont présentés en Figure 5.1. Dans ce scénario, les tâches T1,T2,T3,T4 sont des tâches utilisant une extension. Le placement (a) utilise les CB seulement pour les tâches qui au départ n'utilisent pas l'extension. Dans le placement (c), les tâches T3 et T4 sont placées sur des CB afin de gagner en performance d'un point de vue global (même si au final T3 et T4 ont un temps d'exécution supérieur). Dans le placement (b), les tâches sont presque toutes placées sur des CB (sauf T2 qui subirait une pénalisation excessive à être sur un CB). Ce placement est globalement moins performant mais la somme des énergies consommées est plus faible. Dans les cas (b) et (c), nous faisons l'hypothèse qu'il existe un moyen technique qui permet de rendre les tâches exécutables sur un CB, par exemple en émulant les calculs flottants en utilisant des bibliothèques adaptées ou en transformant le code à la volée.

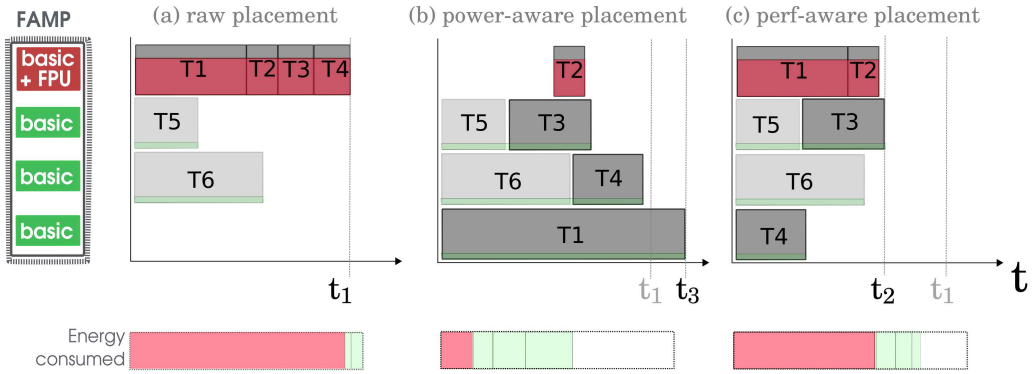


FIGURE 5.1 – Le système d'exploitation a besoin de flexibilité dans le placement de tâches pour permettre plusieurs points de fonctionnement

Pour estimer l'accélération d'une extension, une méthode triviale exploite l'utilisation relative de cette extension (le pourcentage sur l'ensemble de l'exécution de la tâche). Cette solution est par exemple utile pour équilibrer les applications sur une plate-forme asymétrique avec des extensions partagées, comme dans l'architecture bulldozer d'AMD [BBSG11] où deux cœurs partagent une FPU. Avec cette solution, deux tâches qui utilisent intensivement la FPU seraient placées sur deux cœurs qui ne partagent pas la même FPU. Nous pensons que dans le cas d'un FAMP, l'intensité d'utilisation ne suffit pas. Imaginons, par exemple, le cas où deux tâches utilisent la FPU. La première tâche utilise pendant 10 % de son temps d'exécution des instructions qui accélèrent la multiplication sur données flottantes. Si ces instructions sont émulées sur une unité de calcul sur données entières, le CB dans notre cas, le temps d'exécution de la tâche augmentera de façon considérable. Dans le même exemple, la deuxième tâche fait appel pendant 10 % de son temps d'exécution à des instructions d'addition sur données flottantes. Même si le pourcentage d'utilisation de la FPU est équivalent, les pertes de performances dues à une émulation seront significativement inférieures. En se basant sur cette hypothèse nous allons proposer une modèle qui permet d'estimer la dégradation des performances lors de l'émulation (quelque soit le moyen technique) d'instructions étendues. Ainsi un ordonnanceur pourra s'appuyer sur ce modèle pour effectuer un placement de tâche optimisé.

Pour cette étude nous avons utilisé l'extension FPU. Les raisons sont son existence dans la plupart des processeurs et sa forte hétérogénéité dans son jeu d'instructions. Dans ce chapitre, nous démontrons que l'estimation de l'accélération à partir de l'intensité d'utilisation n'est pas précise. L'erreur absolue moyenne est d'environ 68 % avec un écart type de 129 %. Au lieu de ce modèle d'estimation trivial, nous proposons une méthode d'estimation plus précise basée sur la

combinaison d'instructions FPU exécutées. D'après les résultats, ce modèle à une erreur absolue moyenne de 14 % avec un écart-type de 39 %. Par rapport à la solution grossière, l'erreur est réduite d'un facteur de 4,8.

Ensuite, les deux modèles ont été comparés avec plus de détails pour mesurer avec plus de précision les différences d'estimation. Il est montré qu'avec le modèle qui se base sur le pourcentage d'utilisation, l'erreur peut amener l'ordonnanceur à faire de mauvais choix.

Le chapitre s'organise comme suit : la Section 5.2 décrit les méthodes d'expérimentations. La Section 5.3 démontre que la solution triviale qui utilise le pourcentage d'utilisation n'est pas une solution fiable pour estimer l'accélération de la FPU. La Section 5.4 propose et valide une nouvelle méthode pour estimer l'accélération de la FPU. La Section 5.5 compare les deux solutions et présente les variations d'accélération de la FPU au cours des différentes applications.

5.2 Méthodes d'expérimentations

Afin de tester ces modèles et d'effectuer une comparaison, nous nous sommes appuyés sur le simulateur *Gem5* [BBB+11]. Un cœur ARM a été émulé avec le mode "system call emulation".

Pour observer l'utilisation de la FPU à un niveau fin grain, les statistiques de l'exécution sur *Gem5* sont générées pour chaque évènement d'ordonnanceur simulé (par exemple tous les 0.01 ms ou 1 ms).

Les 93 applications étudiées sont principalement issues des suites logicielles suivantes : MiBench [GRE+01], SDVBS [VAJ+09], WCET [GBEL10], fbench [Wal04] et Polybench [Pou12]. Ces suites sont présentées plus en détails Section 3.2.

Les applications ont été compilées avec GCC 4.5.1 et avec les options suivantes `-mfpu=neon -O3 -fno-tree-vectorize -mfpu-abi=softfp`. L'option `- mfpu=neon` permet d'indiquer au compilateur d'utiliser l'extension NEON (contenant la FPU). L'option `-fno-tree-vectorize` empêche le compilateur d'utiliser l'extension vectorielle (aussi dans l'extension NEON). Cela permet de concentrer l'expérience seulement sur l'extension FPU. L'option `-mfpu-abi` assure la compatibilité du programme généré avec des bibliothèques compilées sans FPU.

5.3 Estimation grossière de l'accélération

Sur une plate-forme hétérogène, l'une des fonctions du système d'exploitation est de trouver le meilleur cœur pour chaque fil d'exécution. Une solution pour allouer de façon efficace un cœur à une tâche est d'estimer le temps d'exécution de celle-ci pour chaque cœur de l'architecture. Cette solution peut être lourde si nous considérons tous les effets de l'architecture interne de chaque cœur.

Or, le temps d'exécution d'une tâche dépend d'une part des portions de l'application codées en instructions de base de l'architecture, et d'autre part des portions qui utilisent des instructions spécialisées. Pour le cas d'une architecture FAMP, tous les cœurs partagent la même architecture de base. Nous faisons donc l'hypothèse que les parties qui utilisent seulement les instructions de base ne sont pas impactées si les instructions spécialisées sont exécutées telles quelles, ou bien émulées/transformées pour être exécutées sur un cœur sans l'extension. Des effets de bords peuvent avoir lieu, liés à l'état des registres ou au changement de contexte. Mais afin de garder un modèle compact, nous ne prenons pas en compte ces effets dans notre étude. Ainsi, en s'appuyant sur ce modèle, l'ordonnanceur peut sélectionner le cœur seulement en partant d'un temps d'exécution d'un des cœurs, et ensuite estimer l'accélération de l'extension.

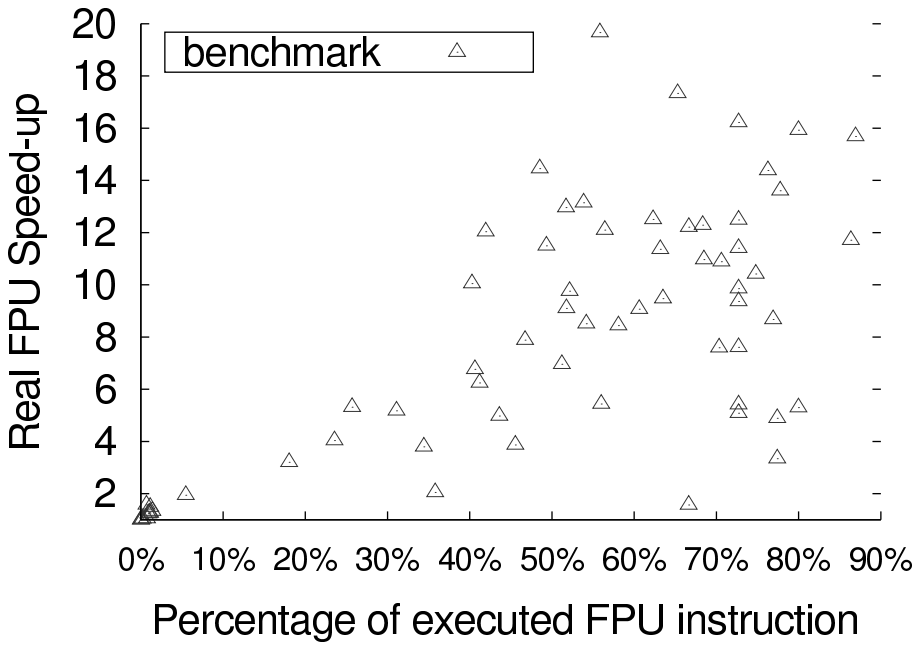


FIGURE 5.2 – Accélération de l'extension ARM FPU(NEON) en fonction du pourcentage d'utilisation. Chaque point △ correspond à une application (Ces applications correspondent à celle détaillées Section 3.2).

Définissons l'accélération (*speedup*) tel que :

$$speedup = \frac{t_{BasicVersion}}{t_{FPUVersion}} \tag{5.1}$$

Les variables $t_{BasicVersion}$ et $t_{FPUVersion}$ sont les temps d'exécution de la même application sur les deux cœurs différents (ici avec et sans FPU).

Une solution triviale pour estimer la différence d'exécution peut être d'utiliser le pourcentage global d'utilisation de la FPU, comme le montre l'équation suivante :

$$\widehat{speedup} = f(\%FPU) \tag{5.2}$$

L'équation 5.2 est montrée en Figure 5.2 pour toutes les applications utilisées (voir Section 3.2). Chaque point correspond à une application. L'ordonnée représente l'accélération et l'abscisse le pourcentage d'instructions FPU exécutées dans toute l'application. Une tendance linéaire est visible, mais le nuage de points est très dispersé. Par exemple, à 70 % d'utilisation de FPU, nous observons une application qui a une accélération de 2x et une autre application qui a une accélération de 18x.

Sans considérer cette première observation, si nous supposons que la fonction f définie dans l'équation 5.2 peut être représentée par une fonction linéaire, nous définissons l'accélération estimée comme suit :

$$\widehat{speedup} = \beta_0 + \beta_1 * \%FPU \tag{5.3}$$

Une régression linéaire permet de calculer les paramètres β_0 et β_1 de l'équation 5.3. Pour la configuration avec le Cortex A9 et la FPU (comme détaillée Section 3.3.2), les valeurs calculées

Class	β Gem5
FloatAdd	31.99163
FloatMult	21.03304
FloatNegAbs	-19.67249
FloatDiv	76.51138
FloatSqrt	154.88496
FloatAddDouble	46.91159
FloatDivDouble	406.40989
FloatSqrtDoubl	1855.97785
CoProMemRead	-12.13329
CoProMemWrite	32.34266
MemRead	-0.07841
MemWrite	0.16073

TABLE 5.1 – Exemple de paramètres β pour certaines classes de de dégradation. Valeurs pour la calibration du modèle sur un ARM Cortex A9 avec FPU(NEON) dans l'outil Gem5.

sont respectivement de 18,2083 et -0,2126. Lorsque ce modèle est utilisé pour estimer le gain en vitesse et qu'il est comparé avec la véritable accélération, l'erreur absolue moyenne de tout l'ensemble des applications est de 68 % avec un écart type de 129 %.

D'après cette étude, l'accélération ne peut pas être estimée avec précision avec ce modèle de premier ordre. Dans la section suivante, nous proposons une solution plus précise.

5.4 Estimation fine de l'accélération

Le gain en vitesse d'exécution fourni par une extension dépend du pourcentage d'utilisation mais aussi de la complexité des instructions qui doivent être émulées. Par exemple, la fonction utilisant des données entières qui émule l'instruction flottante `sqrt` nécessite plus d'instructions qu'une simple fonction qui émule l'instruction flottante `add`.

Afin d'améliorer la précision du modèle d'estimation de l'accélération, nous nous intéressons dans la suite, non pas seulement au pourcentage, mais aussi à l'impact de chaque classe d'instructions de l'extension. Le modèle de l'estimateur est présenté en premier, puis plus de détails sont donnés sur son étalonnage, sa validation et sa mise en œuvre.

5.4.1 Modèle de l'estimateur

Pour définir le modèle de l'estimateur, dans un premier temps, nous définissons l'accélération (*speedup*) comme étant :

$$speedup = \frac{t_{BasicVersion}}{t_{FPUVersion}} \qquad \text{(5.1 revue)}$$

$$\widehat{speedup} = \frac{t_{FPUVersion} + \widehat{degradation}}{t_{FPUVersion}} \qquad (5.4)$$

Dans l'équation 5.4, la *degradation* représente le temps additionnel pour exécuter l'application sur un CB plutôt que sur un cœur avec une FPU. Comme cette dégradation est principalement

due à des instructions en virgule flottante qui sont émulées, pour estimer la dégradation nous proposons le modèle suivant :

$$\widehat{degradation} = \sum_i (NbExecInstr_{Class_i} \times \beta_i) \tag{5.5}$$

L'équation 5.5 estime la dégradation en associant un poids β à chaque classe d'instruction. Ces poids font référence au coût d'émulation de la classe d'instruction sur CB. La dégradation est donc la somme du nombre d'apparitions d'instructions de chaque classe multiplié par le coût d'émulation associé. Pour calculer les paramètres β, une régression linéaire a été utilisée. Comme le procédé d'étalonnage est réalisé à partir de données d'exécution réelle, le modèle prend ainsi en compte le coût d'émulation des instructions spécialisées mais aussi le comportement complexe de l'exécution dans les valeurs des paramètres β.

Les résultats de cet étalonnage pour le modèle de processeur utilisé (détaillé Section 3.3.2) sont présentés dans le tableau 5.1. Une valeur élevée de β signifie que la dégradation de la performance (coût d'émulation) est élevée. Les valeurs proches de 0 signifient que ces classes d'instructions ont un faible impact. Des valeurs négatives de β signifient que ces classes d'instructions impliquent un ralentissement et non une accélération.

Considérant les valeurs des paramètres β et la fonctionnalité des classes, les coûts d'émulation des classes sont bien modélisés par la régression. Par exemple, l'instruction `FloatSqrtDouble` a un poids élevé de dégradation, parce que l'émulation de cette fonction coûte beaucoup d'instructions entières. Nous voyons aussi que les instructions de mémoire (`MemRead` et `MemWrite`) sur des données entières ont des valeurs proches de 0. Leurs valeurs sont faibles car elles sont indépendantes de l'extension FPU et ne nécessitent pas d'émulation. Nous notons également que la valeur de la classe `FloatNegAbs` est une valeur négative. En effet, l'émulation des instructions `Neg` et `Abs` consiste à utiliser seulement *une* instruction de type entier. Ces instructions modifient seulement le bit de signe, sans aucun transfert de données supplémentaires ni aucune modification. La même fonction en instruction FP nécessite plusieurs cycles d'exécution et souvent des transferts de données entre registres entiers et registres à virgule flottante.

Dans la FPU il y a une seconde unité de gestion de transfert mémoire correspondant aux instructions `FloatMemRead` et `FloatMemWrite`. D'après les résultats, par rapport à l'émulation en utilisant l'unité mémoire de base, l'unité de mémoire FP semblerait permettre une écriture plus rapide, mais une lecture plus lente.

5.4.2 Validation du modèle

Le modèle a été validé avec 93 applications contenant des tests unitaires et des applications complètes. Les applications sont extraites des ensembles détaillés dans la section 3.2.

Intégration du comportement de la FPU et de l'émulation

Les tests unitaires sont composés seulement d'un ou plusieurs types d'opérations FPU sur un large tableau de données à virgule flottante. Cela permet de n'utiliser qu'une partie de la FPU à la fois et de n'estimer que l'accélération de certaines opérations. Dans ces tests unitaires, le pourcentage d'instructions FPU est supérieur à 80 %. Le reste des instructions concerne principalement des instructions de contrôles. Afin de prendre en compte dans notre modèle les effets indirects, par exemple créés par des enchaînements de certaines instructions, nous avons recours à des applications complètes. En calibrant le modèle à l'aide d'applications complètes, ces effets vont être pris en compte dans les β. Par exemple, l'enchaînement de plusieurs opérations FPU sur les mêmes données amortit le coût de transfert mémoire (surtout lorsqu'il y a différents

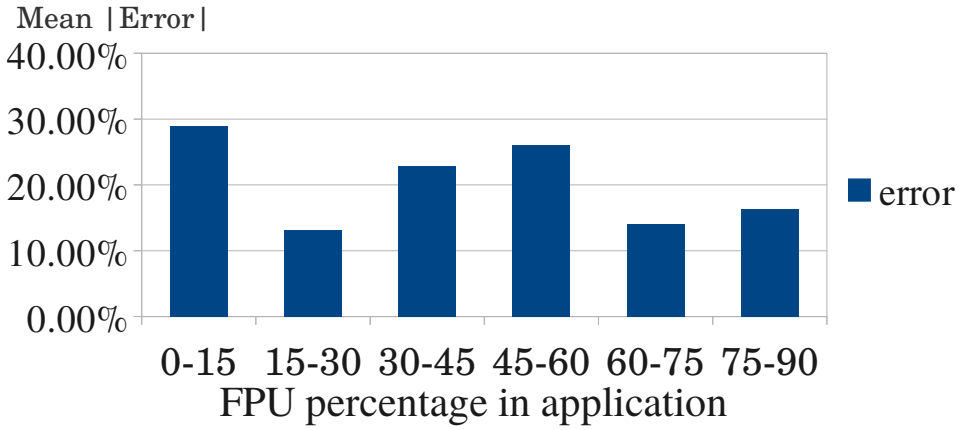


FIGURE 5.3 – Erreur du modèle pour estimer la dégradation. L'erreur n'augmente pas avec le taux d'utilisation de la FPU.

types de données, par exemple flottant et entier). Lors de l'émulation ces coûts de transfert sont moindres car les données seront potentiellement déjà dans les registres, par conséquent, aucun transfert de registres entiers vers flottants, ni de conversion. Un autre exemple d'effet complexe peut être l'exploitation des plusieurs voies de calculs dans l'extension (par exemple une voie pour la mémoire et une voie pour le calcul en parallèle). Avec les tests unitaires, ces voies ne sont peut être pas exploitées, cependant, dans une application complète ces voies peuvent être amenées à être plus exploitées, car l'utilisation sera hétérogène.

Erreur du modèle

Afin de calculer l'erreur du modèle, chaque application a été exécutée sur un CE. Ensuite la dégradation a été estimée avec le modèle proposé. Cette estimation a été comparée avec les résultats d'une exécution réelle sur un CB.

Pour l'ensemble des applications, la moyenne des valeurs absolues de l'erreur de la dégradation réelle par rapport à la dégradation estimée ($deg.$ vs $\widehat{deg.}$) est de 22 %. L'erreur moyenne (non absolue) est de 0,02 %. Elle est proche de zéro car nous calculons ici l'erreur sur l'ensemble des applications qui ont permis de calculer la régression linéaire, et l'objectif de celle-ci est de trouver les β tels que l'erreur moyenne soit proche de zéro.

En comparant l'erreur absolue moyenne pour différents pourcentages d'utilisation de FPU, la figure 5.3 montre que celle-ci est stable par rapport au pourcentage d'instructions Flottantes utilisées lors de l'exécution. Avec un modèle trop sensible, l'erreur ne serait pas constante. Par exemple, si le modèle ne prenait pas en compte un facteur de dégradation pour chaque instruction flottante, l'erreur augmenterait avec le pourcentage.

L'erreur absolue respective par rapport à l'ensemble de l'exécution (t_{base} vs $t_{extended} + \widehat{deg.}$) est réduite à 9 % avec un écart-type de 16 %. L'erreur absolue moyenne de la dégradation est de 14 % avec un écart-type de 39 %. **Cette erreur est 4,8 fois inférieure au modèle basé sur l'utilisation relative de pourcentage.**

Ce niveau d'erreur d'estimation de l'accélération pour une approche système est à notre avis acceptable. De plus, la mise en œuvre de l'approche est d'une complexité faible, elle peut donc être utilisée en-ligne. Une discussion sur l'implémentation de ce modèle est présentée dans la section suivante.

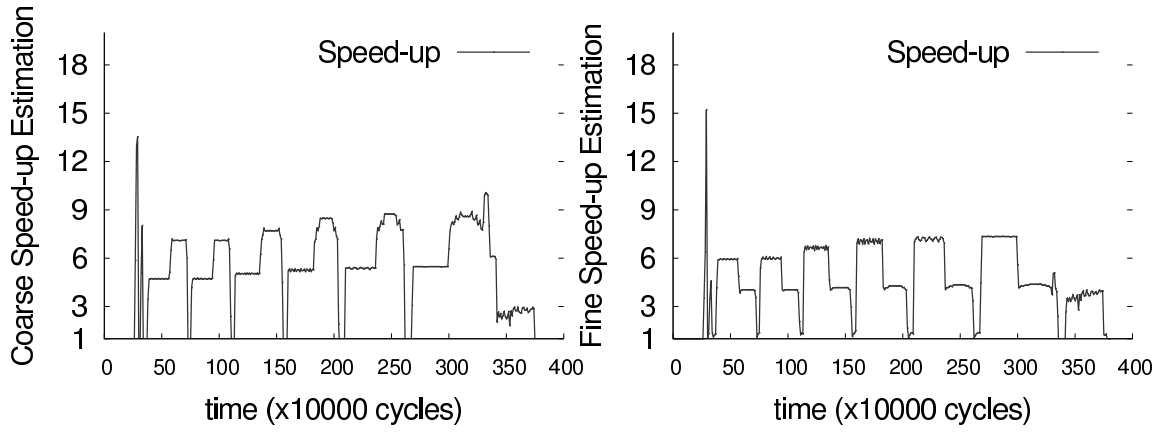
5.4.3 Implémentation

Le modèle d'estimation de l'accélération peut être utilisé au moment de la compilation ou en temps réel pendant l'exécution. Dans les deux cas, il faut d'abord récupérer les données sur l'utilisation de l'extension, puis appliquer les équations 5.4 et 5.5 en utilisant les valeurs des β calibrés à l'avance pour une technologie donnée (ex. FPU NEON, Tableau 5.1). Cette calibration doit être faite pour chaque type d'extension, en utilisant un ensemble de tests et d'applications comme cela a été fait dans les sections précédentes. Plus cet ensemble représente l'espace de calcul possible, plus le modèle sera précis.

Pour la compilation, les données peuvent être obtenues à l'aide d'outils d'analyse et de processus de compilation itérative [Fur04, HB06, PAS03]. En comptant le nombre d'apparitions de chaque type d'instructions, le calcul des équations 5.4 et 5.5 peut être effectué. Le compilateur peut alors choisir de continuer à utiliser l'extension ou la bibliothèque d'émulation.

Pour l'utilisation en-ligne, la mise en œuvre nécessite d'une part un moniteur qui récupère les données de l'exécution en cours et d'autre part, un module d'analyse qui permet d'estimer l'accélération pour enfin la transmettre à l'ordonnanceur. La partie estimation peut être réalisée sous forme d'un module logiciel dans l'ordonnanceur du système d'exploitation (ou du logiciel d'exécution). Le module est de faible complexité car il utilise seulement l'équation 5.4, l'équation 5.5 et la table des β pré-calibrés.

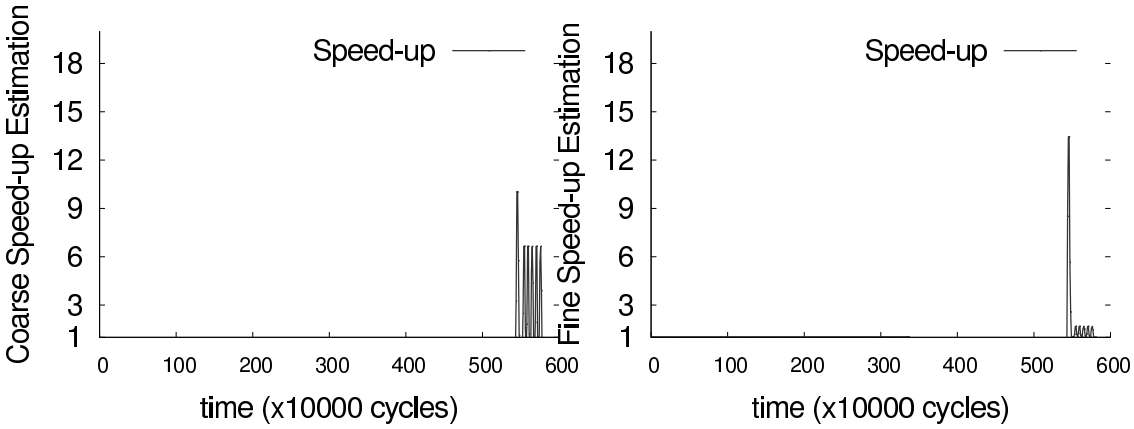
5.5 Comparaison des deux modèles



(a) Sift(SDVBS) Coarse Speed-up Estimation (b) Sift(SDVBS) Fine Speed-up Estimation

FIGURE 5.4 – Estimation de l'accélération avec la méthode grossière (a) et la méthode fine (b) pour l'application `Sift` (Chaque point du graphique est l'estimation pour un quantum de 10K cycles).

Dans cette partie, nous allons comparer les deux modèles à un niveau de granularité fin (correspondant à des quantum de 10K cycles lors de l'exécution). Ceci est intéressant par rapport à la comparaison de l'estimation sur l'application totale, car la différence entre les deux modèles est moins lissée par les larges sections d'instructions entières. Les figures 5.4 et 5.5 présentent les estimations effectuées par les deux modèles à chaque quantum (10k cycles) de l'exécution.



(a) Mser(SDVBS) Coarse Speed-up Estimation (b) Mser(SDVBS) Fine Speed-up Estimation

FIGURE 5.5 – Estimation de l'accélération avec la méthode grossière (a) et la méthode fine (b) pour l'application `Mser` (Chaque point du graphique est l'estimation pour un quantum de 10K cycles).

La figure 5.4 montre l'accélération de la FPU lors de l'exécution de `Sift` (SDVBS). Des phases distinctes sont visibles dans l'exécution. Une phase est elle-même divisée en deux sous-phases. La première sous-phase contient un nombre important d'instructions à faible accélération. Dans la deuxième phase, au contraire, un nombre plus faible mais à plus fort potentiel d'accélération est présent.

Une inversion des créneaux est visible entre les deux versions. Avec l'estimation grossière, le créneau commence par une accélération de 4x puis continue avec une accélération de 7x. Avec le modèle précis, le créneau commence avec une accélération plus élevée (6x) puis continue avec une accélération plus faible (4x). Le premier cas est dû au fait qu'il y a un fort pourcentage mais contenant des instructions avec une faible accélération, et inversement pour la deuxième partie du créneau. Cet exemple, bien qu'il ne soit pas représentatif de tous les cas possibles, démontre que la méthode basée uniquement sur le pourcentage global d'instructions peut donner des résultats aberrants. Si cette estimation est utilisée pour placer deux tâches en parallèle dans un système multitâche, l'ordonnanceur serait très probablement amené à effectuer le pire choix de placement possible.

Pour l'application `mser`(SDVBS), dont la trace d'exécution est visible sur la figure 5.5, le modèle gros grain estime dans la phase finale une accélération élevée, mais finalement nous constatons que l'accélération est faible. En fonction des deux estimations, le comportement de l'ordonnanceur peut varier. Par exemple, sur l'application `mser`, avec l'estimation grossière l'ordonnanceur ne prendrait pas la décision de placer la fin de l'application sur un CB. Or, d'après l'estimation plus fine, dans un contexte d'optimisation énergétique, l'ordonnanceur devrait placer la tâche sur un CB. Une mauvaise estimation pourrait aussi amener à placer une application sur un CB, ce qui impliquerait trop de dégradation en performance (et, passé le seuil temporel, une dégradation en énergie).

Pour conclure, les figures 5.4 et 5.5 montrent que l'erreur de la méthode d'estimation gros grain n'est pas répartie également à travers l'exécution de l'application. L'erreur peut être très élevée seulement sur une certaine partie de l'application. Ces exemples montrent aussi qu'il est

nécessaire d'avoir un système précis et une solution de placement de tâches dynamique.

5.6 Perspectives

Les perspectives possibles pour ces travaux sont dans un premier temps l'amélioration de la calibration du modèle à l'aide d'un ensemble plus grand d'applications. Actuellement, nous utilisons 93 applications, cependant l'espace d'utilisation de l'extension n'est pas totalement représenté. En augmentant la richesse des applications, le modèle pourrait être plus performant.

Ce modèle a été appliqué à un modèle de processeur simulé, pour l'extension FPU. D'autres études peuvent être réalisées pour d'autres processeurs et pour d'autres extensions. En fonction de leur accès à la mémoire, et à leur intégration dans la structure du CB, le modèle devra peut être prendre en compte d'autres variables. Par exemple il serait intéressant de prendre en compte la métrique du nombre d'arrêt du cœur à cause d'un manque de données (*stall*). Lors d'un nombre important d'arrêts, une extension qui calcule rapidement n'a pas d'intérêt car le cœur est en attente. Dans ce cas, il est peut-être plus intéressant d'utiliser un CB. Ces arrêts sont dépendants des données et de la hiérarchie mémoire, et non des instructions. Il s'agit donc d'une métrique nouvelle pour le modèle.

Une autre amélioration consiste à considérer des classes plus abstraites que seulement les classes d'instructions (en utilisant des outils tels qu'une analyse en composantes principales (PCA)). Pour réduire le coût surfacique de surveillance, dû aux compteurs matériels, il faudrait réduire le nombre de classes. Néanmoins, cela aurait un effet négatif sur la précision de l'estimation. Une étude pourrait être réalisée pour trouver le bon compromis. Dans la même idée, une autre possibilité est de prendre en compte l'enchaînement des classes d'instructions comme une variable. Comme expliqué en Section 5.4.2, l'enchaînement des instructions peut aussi influencer la performance, ce qui n'est pas pris en compte explicitement dans le modèle. Néanmoins, ces informations peuvent être coûteuses à surveiller, donc il faut trouver le bon compromis afin de garder une solution utilisable à l'exécution.

Une dernière perspective est d'avoir un processus de calibration qui permettrait de re-calibrer les poids du modèle en ligne. En effet, des événements extérieurs au système d'exploitation comme le vieillissement, le changement des processeurs, ou des options de configuration activeraient des changements sur les variables β du modèle (Équation 5.5). Ainsi, pour pouvoir avoir un procédé rapide en ligne, il faudrait définir un ensemble minimum d'applications qui représenterait l'espace de calcul nécessaire pour calibrer le modèle. Dans notre étude, nous utilisons un ensemble d'applications pour la calibration, mais celui-ci n'est pas optimal.

5.7 Conclusion

Le fait de ne pas utiliser une extension, d'une part en l'éteignant (ou en exécutant l'application sur un CB), et d'autre part en utilisant des librairies remplaçant ses fonctions, permet d'économiser de l'énergie, tant que la dégradation en performance n'est pas trop forte. Pour éviter ces cas, ce chapitre étudie comment estimer l'accélération d'une extension matérielle à partir de son utilisation.

Nous avons montré que le fait de se baser sur le pourcentage d'utilisation de la FPU pour en déterminer l'accélération potentielle n'est pas une méthode adéquate. L'erreur absolue moyenne du modèle sur l'ensemble des applications est de 68 % avec un écart type de 129 %. Pour avoir une meilleure précision tout en permettant sa mise en œuvre en ligne, nous proposons d'estimer l'accélération en se basant sur l'apparition de classes d'instructions. Notre solution estime l'accélération avec une erreur de 14 %, ce qui est 4,8 fois plus précis que la solution gros grain. De plus, avec une étude grain fin, nous avons montré que le fait d'utiliser seulement le

modèle gros grain pouvait générer des erreurs importantes qui donnent d'ailleurs des estimations trompeuses.

Pour conclure, ce chapitre présente un nouveau modèle utilisable en ligne pour estimer l'accélération pour une extension. Dans les chapitres précédents, il a été montré que certaines applications avaient une utilisation de la FPU peu efficace quelle que soit l'approche. Or l'approche au niveau ordonnanceur sur une architecture FAMP est potentiellement intéressante. Mais si celle-ci se restreint à l'apparition des instructions, elle a tendance à privilégier un placement qui favorise le cœur doté d'une FPU. En utilisant le module présenté dans ce chapitre le système d'exploitation peut gagner en flexibilité de placement tout en contrôlant la perte de performance. Ainsi, il est aussi possible d'optimiser la consommation en énergie. Pour vérifier ces hypothèses, le chapitre suivant étudie la flexibilité de placement et les gains en énergie grâce à l'implémentation du modèle d'estimation présenté dans ce chapitre sur une architecture FAMP.

Chapitre 6

Ordonnancement relaxé et intelligent pour processeur asymétrique en fonctionnalités

Sommaire

6.1 Introduction

Quelle que soit l'architecture, un système d'exploitation a plusieurs objectifs à satisfaire : des objectifs globaux comme un budget énergétique et des objectifs locaux comme une qualité de service. Il a aussi des contraintes à respecter dues à la plateforme comme par exemple éviter de faire surchauffer certaines parties de la puce. Ce problème devient majeur avec l'augmentation de la densité de transistors au mm^2. Tous les circuits qui constituent la puce ne peuvent pas être allumés en même temps, principalement pour des raisons de dissipation thermique. Certaines parties doivent donc être éteintes, ce qui amène à une sous-utilisation globale du circuit. Dans la littérature, on appelle ce phénomène `dark-silicon` [EBSA+11]. Tous ces objectifs vont être plus facilement atteignables si le système d'exploitation possède de la flexibilité dans l'utilisation du multi-cœur, et en particulier sur le placement des applications sur les cœurs.

Dans le contexte d'une *architecture asymétrique en fonctionnalités*, cette flexibilité de placement peut être contrainte par l'asymétrie du jeu d'instructions et des applications, qui ont besoin de certaines extensions. Nativement, une application qui utilise l'extension doit obligatoirement être exécutée sur un cœur avec cette extension. Cette contrainte peut créer des goulots d'étranglement sur les processeurs contenant les extensions. De plus cela empêche d'exploiter les autres processeurs disponibles.

Afin de placer les tâches sur n'importe quel cœur, quelle que soit l'utilisation de l'extension, des solutions existent comme le multi-versionning, la traduction binaire ou l'émulation. Ces solutions sont détaillées dans l'état de l'art section 2.4. Mais, pour permettre au système d'exploitation de respecter une qualité de service ou un budget énergétique, ce dernier doit pouvoir placer les tâches sur un CB de façon raisonnée. Nous avons développé un modèle, présenté en section 5.4 qui nous permet d'estimer la dégradation lors de l'exécution sur un CB. En l'intégrant dans l'ordonnanceur et en limitant la dégradation à un seuil maximal, cela permet à l'ordonnanceur de savoir quelles sont les applications qui sont déplaçables sur un CB tout en respectant une qualité de service donnée. Nous appelons cette approche un *ordonnanceur relaxé*.

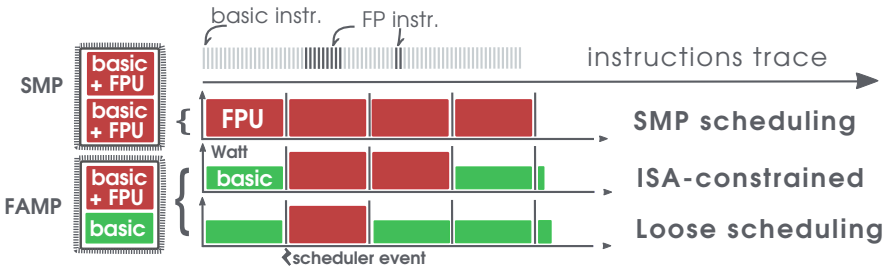


FIGURE 6.1 – Approches pour placer les tâches sur des architectures SMP et FAMP. L'approche ordonnanceur relaxé (Loose scheduling) peut permettre plus de flexibilité et des gains en énergie.

Dans schéma de la figure 6.1, l'ordonnanceur relaxé est comparé avec l'approche contrainte par l'ISA et l'approche classique SMP.

La première approche correspond à une plate-forme SMP classique. Toutes les sections de code sont placés sur un CE. Cette solution n'a pas de surcoût logiciel (ordonnanceur ou émulation), mais la plateforme SMP est coûteuse en énergie et en surface. Dans la seconde approche, contrainte par l'ISA, l'ordonnanceur place certaines sections de code sur un CB. Cependant, les parties d'application qui contiennent au moins une instruction de FP doivent être exécutées sur un CE. Cela pourrait être inefficace si l'utilisation de l'extension est faible, comme par exemple dans le troisième bloc d'instructions de la figure. Dans la troisième approche,

que nous appelons `ordonnancement relaxé`, quelques sections de code peuvent être placées sur un CB, même si elles contiennent des instructions utilisant l'extension. La troisième section de code se trouve dans ce cas de figure. Par contre, pour la deuxième section de code, l'utilisation de l'extension a été jugée utile, donc la section reste sur un CE.

Dans ce chapitre, notre objectif est de simuler ce système et ainsi d'observer la flexibilité acquise sous une certaine limite de *dégradation autorisée*. Ce chapitre explore aussi la dégradation globale due à ce système et, en tant que cas d'étude, les gains en énergie possibles.

Le chapitre s'organise comme suit : la Section 6.2 explique les limites d'un placement de tâches contraint par le jeu d'instructions. La Section 6.3 détaille le système complet qui permet d'avoir un ordonnanceur plus flexible. La Section 6.4 décrit les méthodes d'expérimentations. La Section 6.5 présente les résultats d'un ordonnanceur qui n'est plus contraint par l'ISA. La flexibilité de placement, le surcoût en performance que cela implique et les gains en énergie potentiels grâce à cette flexibilité sont ainsi quantifiés.

Pour clarifier certains termes utilisés dans ce chapitre voici tout d'abord les définitions de la flexibilité et des différentes dégradations.

Définitions : flexibilité et dégradations

Dans cette section, les termes *flexibilité*, *dégradation autorisée* et *dégradation globale* sont expliqués.

La *flexibilité* est la capacité de l'ordonnanceur à placer une tâche sur n'importe quel cœur dans un système multi-cœur. Plus la flexibilité est grande, plus les binaires peuvent être placés partout par l'ordonnanceur. Par conséquent, l'ordonnanceur a la liberté d'optimiser plusieurs objectifs, tels que : la performance, l'énergie, la priorité et/ou la température.

La *dégradation autorisée* est exprimée sous la forme d'un seuil. Elle est utilisée à chaque événement de l'ordonnanceur. Elle représente le surcoût maximal que le système autorise lors du placement d'un *quantum* sur un CB plutôt que sur un CE. Si la valeur de la dégradation estimée pour le futur *quantum* (grâce au modèle Section 5.4) est inférieure au seuil, l'ordonnanceur place la tâche sur un CB. Ainsi, cela permet à l'ordonnanceur de respecter un objectif de performance. En fonction du contexte, le système d'exploitation peut augmenter ou réduire le seuil de *dégradation autorisée*.

La *dégradation globale* est le surcoût en performance de l'ensemble de l'application. Cette dégradation est due au coût de fonctionnement d'une partie de l'application sur un CB plutôt que sur un CE et au coût de commutation de l'application entre les cœurs. La *dégradation globale* est souvent inférieure à la *dégradation autorisée* parce que tous les *quanta* de l'application ne sont pas exécutés sur un CB.

6.2 Limites de l'ordonnancement contraint par l'ISA

Lorsqu'une application est originalement compilée pour une plate-forme multi-cœur symétrique (dont chaque cœur contient chaque extension) ou dans le cadre de l'architecture FAMP avec une approche unifiée (détaillée section 2.4), une instruction étendue peut apparaître à tout moment. Pour que l'application soit exécutée sur un architecture FAMP, l'ordonnanceur sélectionne le cœur en tenant compte de l'apparition de ces instructions étendues. Par exemple,

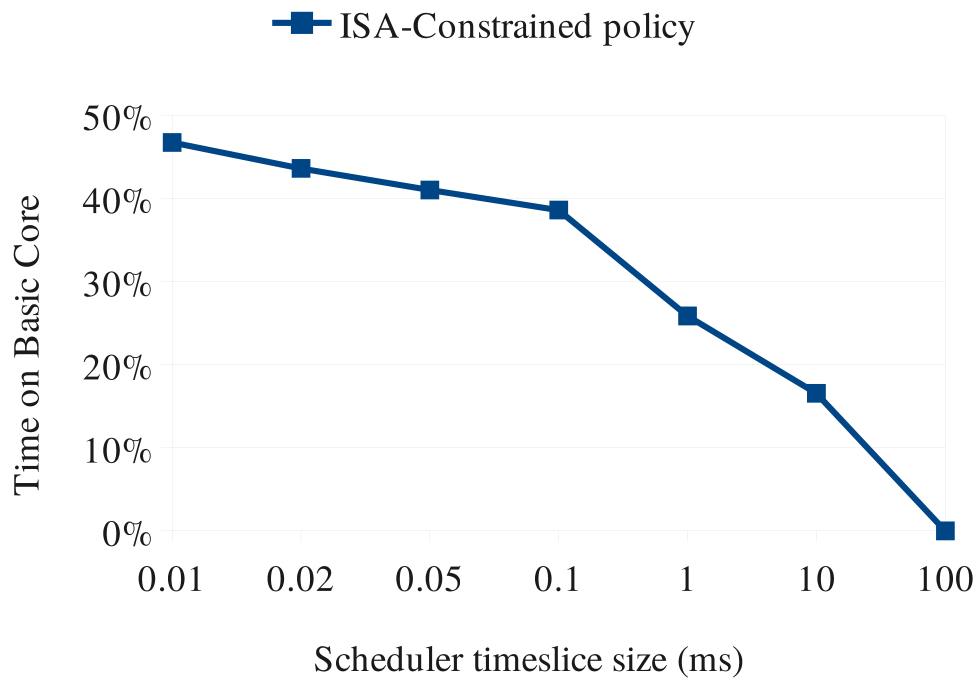


FIGURE 6.2 – Temps d'exécution moyen passé sur un CB en fonction de la granularité d'action de l'ordonnanceur. Simulation sur 12 applications des suites *SDVBS* et *miBench* avec un ordonnanceur contraint par l'ISA et un "oracle". Comme le système a accès aux futures instructions à exécuter, le prédicteur ne peut pas se tromper. Cela permet de voir les gains maximums et d'éviter les biais.

à chaque événement de l'ordonnanceur, l'utilisation de l'extension est prédite. Si l'application utilise l'extension dans le prochain *quantum* (c'est-à-dire que le *quantum* contient au moins une instruction étendue), l'ordonnanceur place l'application sur un CE. Dans le cas inverse, l'application est placée sur un CB jusqu'au prochain événement de l'ordonnanceur. Lors d'une mauvaise prédiction, le CB lève un drapeau d'"instruction inconnue", puis l'ordonnanceur bascule l'application sur un cœur avec l'extension demandée. Une solution proposée dans le papier [LBK+10], peut être utilisée pour migrer la tâche quand une erreur survient. Comme l'apparition de l'instruction étendue force l'ordonnanceur à déplacer la tâche, nous avons nommé ce type de modèle : un ordonnanceur *contraint par l'ISA*.

Cependant, en respectant ces règles, l'ordonnanceur perd beaucoup en flexibilité de placement. L'ordonnanceur contraint par l'ISA a deux problèmes clés :

- Premièrement, cette approche ne prend pas en compte le fait que, dans certains cas, l'utilisation des instructions de l'extension est inefficace. Le placement de tâche est basé sur un choix fait hors ligne par le compilateur et le développeur. Le développeur choisit de compiler avec ou sans les options qui permettent d'utiliser l'extension. Ensuite, le compilateur a pour objectif d'utiliser l'extension à son maximum. Le compilateur a été développé pour augmenter les performances sur une plateforme SMP, où les extensions sont toujours disponibles. Il ne possède pas de système pour évaluer l'intérêt d'utiliser ou non les extensions. Il en résulte que certaines parties de code contiennent des instructions étendues mais au final n'utilisent pas assez efficacement l'extension. Ces sections ne sont pas assez accélérées par l'extension et cela ajoute un surcoût énergétique (énergie statique de l'extension). Pour résumer, le processus de placement de tâche repose sur des choix hors ligne inexacts.
- Deuxièmement, l'ordonnanceur contraint par l'ISA est dépendant de la taille du *quantum* de l'ordonnanceur. Plus le *quantum* est petit, plus les sections de code sans instructions étendues sont détectées. Les systèmes d'exploitation courants ont un *quantum* entre 1 ms à 100 ms. A ce niveau de granularité, les phases sans instructions étendues sont peu détectées. L'étude au niveau de la granularité section 4.2.2 montre bien la contrainte, et plus précisément la figure 4.5 qui détaille le temps passé sur un CB pour chaque application. Pour résumer cette limitation, la figure 6.2 montre le temps d'exécution moyen d'une application sur un CB en fonction de la taille du *quantum*. À 1 ms, le pourcentage de temps d'exécution sur un CB est seulement de 26 %. Avec un *quantum* plus fin (0.01ms), le pourcentage passe à 47 %. Cependant, la moyenne d'utilisation de l'extension pour toutes les applications est de 7 %. C'est-à-dire que pendant 93 % du temps les applications pourraient être exécutées sur un CB (dans le cas d'une architecture FAMP). En se basant sur cette observation, on en conclut que le temps passé sur un CB avec la solution contrainte par l'ISA n'est pas optimal.

Pour conclure, un ordonnanceur contraint par l'ISA repose sur des choix fait hors ligne et sa flexibilité de placement est faible par rapport à l'utilisation réelle de l'extension. Les sections suivantes étudient le coût et l'intérêt d'avoir une flexibilité de placement donnée par un ordonnanceur relaxé. Nous allons aussi présenter un modèle de système doté d'un ordonnanceur relaxé.

6.3 Multi-cœur asymétrique avec ordonnanceur relaxé

Cette section décrit plus en détails la mise en place de l'ordonnanceur relaxé sur un multi-cœur asymétrique en fonctionnalités. La solution n'a pas été intégralement mise en œuvre pour

l'étude car celle-ci a été réalisée sur un simulateur. Néanmoins cela permet de présenter notre vision et de comprendre l'intégration du système dans son ensemble.

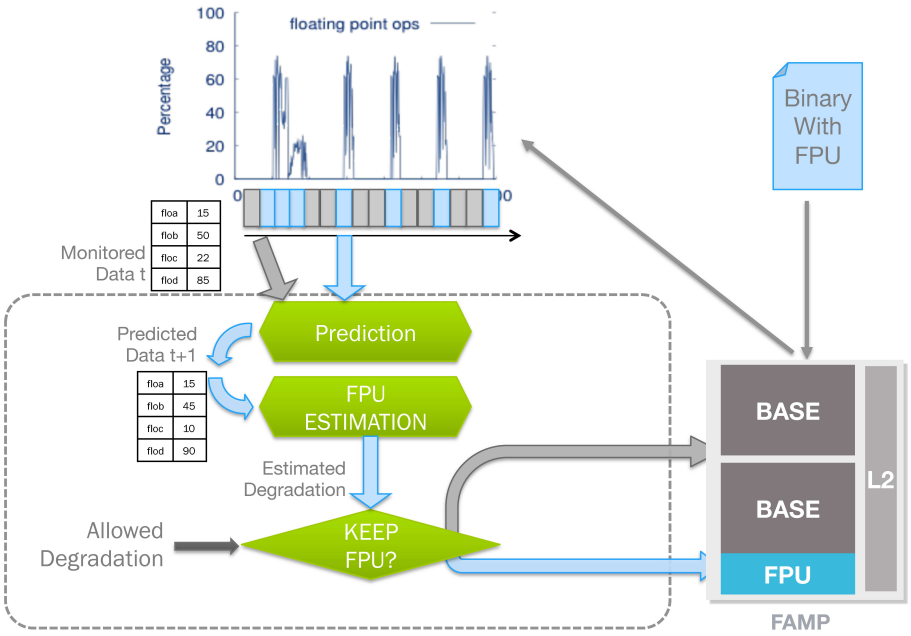


FIGURE 6.3 – Système avec le module en ligne plus flexible et plus efficace pour le placement de tâches.

La figure 6.3 présente le système d'un point de vue fonctionnel, avec l'ajout du module permettant l'ordonnancement relaxé.

Notre solution peut s'intégrer dans un système d'exploitation préemptif tel que Linux. Dans ce type de système d'exploitation, l'ordonnanceur doit périodiquement choisir quelle tâche placer sur quel cœur. Comme un cœur est souvent partagé par plusieurs tâches, ces tâches accèdent aux cœurs seulement pour des tranches de temps données, appelées *quantum*. À chaque fin de *quantum*, un événement réveille l'ordonnanceur et celui-ci lance un processus pour placer une autre application en attente du cœur. C'est lors de ce processus que le module permettant l'ordonnancement relaxé est ajouté. Ainsi, l'ordonnanceur ne prend pas seulement en compte l'apparition des instructions, mais il récupère aussi les données sur l'exécution, prédit les données pour le *quantum* suivant, et estime ainsi la dégradation due au fait de ne pas utiliser l'extension. Si la dégradation estimée est en dessous du seuil de *dégradation autorisée*, alors l'ordonnanceur sait qu'il peut placer l'application sur un CB. Ensuite le placement se fait en fonction du contexte global (ex. les autres applications) et des objectifs de l'ordonnanceur.

L'application doit pouvoir être exécutable sur les deux types de cœurs (avec et sans FPU). Le système présenté est indépendant de la méthode choisie. Plus de détails sur les solutions possibles de l'état de l'art sont donnés section 2.4.

Lorsqu'il y a une commutation de tâche d'un cœur à un autre, il faut gérer la consistance de l'exécution. En fonction de la méthode sélectionnée pour représenter le code, il peut y avoir des problèmes de synchronisation et de gestion de données liés à l'extension. Par exemple, pour une extension FPU, les données stockées dans les registres de l'extension flottante peuvent être utilisées plus tard, donc il est nécessaire que ces données soit sauvegardées. Avec une solution multi-version de code, des points de synchronisation peuvent être définis, comme décrit par

exemple dans le papier [HBC05]. Il y a aussi la possibilité d'utiliser de la traduction binaire pour gérer les données, tel que décrit dans le papier [GNVL14].

Pour la mise en œuvre du système, trois éléments sont nécessaires : une partie moniteur qui récupère les données de l'exécution, une partie prédicteur et une partie estimateur. Ensuite, une dernière composante effectue le choix de placement à partir du seuil de *dégradation autorisée* et de l'estimation de la dégradation calculée. Les parties moniteur, prédicteur et estimateur sont détaillées dans les sections suivantes.

6.3.1 Moniteur

Pour la partie surveillance, les données d'entrée du modèle doivent représenter au mieux la suite de l'exécution de l'application. Une possibilité est d'utiliser les données des derniers *quantum*s exécutés.

Pour récupérer les données de l'exécution, des compteurs peuvent être utilisés pour mémoriser les apparitions de chaque classe d'instructions spécialisées.

Dans le cas d'un cœur avec une FPU, les compteurs peuvent être matériels et incrémentés automatiquement à chaque instruction FPU exécutée. Par exemple dans le Cortex A9, il y a une unité de surveillance de performance programmable qui contient 6 registres matériels et 58 événements. Pour l'instant, seulement un sous-ensemble des événements associés à certaines classes d'instructions des extensions est possible de connecter aux compteurs matériels (ex. le nombre de cycle, le nombre de données manquantes dans le cache et le nombre d'instructions FPU exécutées). Dans le cadre de notre modèle, il faudrait rajouter, par exemple, des événements en fonction de l'exécution de classes d'instructions de l'extension.

Dans le cas d'un CB, une solution logicielle à base de compteurs mémoire est ajoutée à la bibliothèque d'émulation pour compter le nombre de remplacements d'instructions spéciales.

Ces solutions permettent de réduire les surcoûts et seulement quelques cycles sont ajoutés à chaque *quantum* pour obtenir les données à partir des registres. Ces données peuvent être ensuite stockées dans le contexte d'exécution de la tâche pour pouvoir être utilisées par le prédicteur et l'estimateur.

6.3.2 Prédicteur

Un prédicteur est nécessaire car l'ordonnanceur doit optimiser la suite de l'exécution et pas seulement celle de la section qui vient de s'exécuter. Or, les données récupérées sont celles de la section précédente (*monitored data t* sur la figure 6.3). Dans l'ordonnanceur, il faut donc ajouter un module de prédiction qui récupère les données de l'exécution de la tâche à placer, et qui prédit la suite de l'exécution.

Pour réaliser un prédicteur simple, une hypothèse consiste à dire que le prochain *quantum* est semblable au dernier. C'est-à-dire que l'usage de l'extension va impliquer la même accélération. Cette hypothèse est suffisante lorsque les applications ont des phases distinctes de comportement. L'hypothèse de la similarité est alors fausse seulement aux changements de phases. Des prédicteurs utilisant plus d'informations du passé (pas seulement le dernier *quantum*) peuvent être explorés afin d'avoir une prédiction plus réaliste. De multiples recherches ont déjà été réalisées dans le domaine de la prédiction de branchement [Smi81, BL93, SJSM96, Sez11]. Des solutions de ce domaine peuvent être testées par rapport aux profils de l'utilisation des extensions.

Pour la mise en œuvre du prédicteur, on note qu'il y a deux points d'entrées. La première possibilité est d'effectuer une prédiction juste après avoir récupéré les données de l'exécution.

C'est-à-dire que le prédicteur traitera un vecteur contenant un historique du nombre d'apparitions de chaque classe d'instructions de l'extension (tel que montré dans la figure 6.3). Cette solution sera précise mais coûteuse en temps de calcul et en espace mémoire. La deuxième possibilité est d'effectuer la prédiction après avoir estimé la dégradation à partir du nombre d'apparitions de chaque classe d'instructions. Donc le prédicteur traitera un scalaire (la dégradation). Cette solution sera moins précise mais sera moins consommatrice en calculs et place mémoire.

6.3.3 Estimateur

L'estimateur a pour but de calculer la dégradation relative à la non utilisation d'une extension. Il est implémenté après la récupération des données du moniteur et/ou du prédicteur (*Predicted Data t+1* sur la figure 6.3). Il peut être basé sur le modèle défini en section 5.4. Ainsi, grâce à l'utilisation de l'équation 5.4, l'équation 5.5 et la table des β calibrés, l'estimateur est réalisable en seulement quelques dizaines de cycles. Ensuite, l'ordonnanceur peut évaluer la dégradation estimée et sélectionner le cœur adéquat pour le prochain *quantum* d'exécution.

Le système complet a été simulé pour extraire la flexibilité, le surcoût en performance et les potentiels gains en énergie par rapport à un ordonnancement contraint par l'ISA. La méthode d'expérimentation, basée sur de la simulation, est détaillée dans la prochaine section.

6.4 Méthodes d'expérimentations

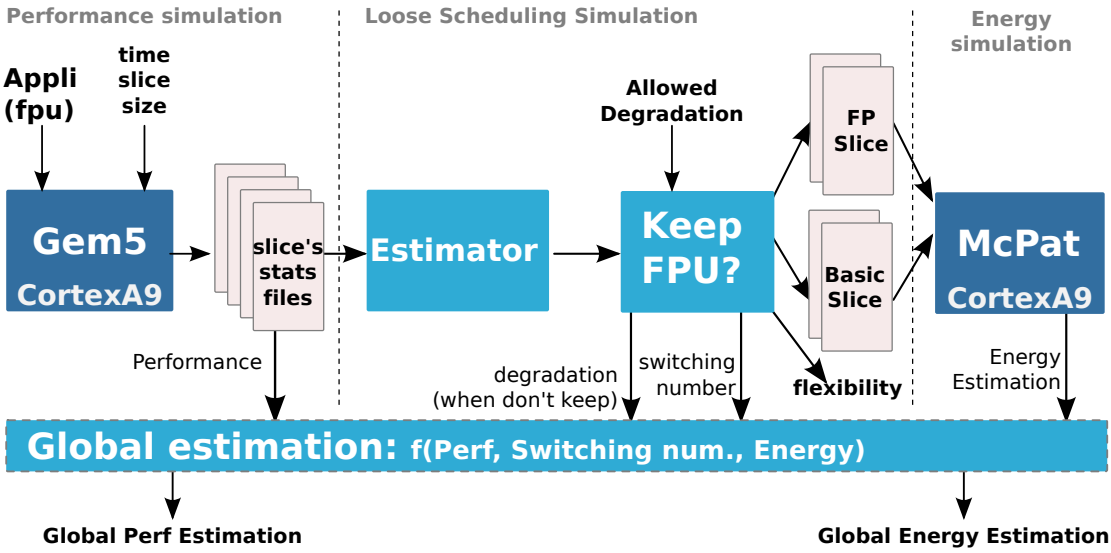


FIGURE 6.4 – Méthodologie d'expérimentation. Simulation d'une architecture multi-cœur asymétrique en fonctionnalités (FAMP) pour évaluation du gain potentiel d'un ordonnanceur relaxé

Cette section décrit la méthode d'expérimentations pour simuler une plateforme FAMP avec un ordonnanceur relaxé. Les applications utilisées proviennent des ensembles miBench et SDVBS détaillés Section 3.2.

Pour la simulation de la plateforme, la configuration est similaire à celle utilisée dans le Chapitre 4 et décrite en Section 4.3. La méthode s'appuie sur les outils Gem5, McPat et le

modèle de l'ordonnanceur relaxé décrit dans la section suivante.

6.4.1 Simulation de l'ordonnanceur relaxé

Le modèle global de simulation du système est montré en figure 6.4. Ce modèle se compose de trois parties : la simulation de la performance, la simulation de l'ordonnanceur relaxé et la simulation de la consommation énergétique.

Gem5 est utilisé pour simuler les applications. Elles sont simulées avec l'utilisation de l'extension FPU. Des statistiques sont collectées à chaque *quanta* d'ordonnancement simulé. Nous notons que ces données sont similaires à celles récupérées d'un moniteur dans une plateforme matérielle, puisqu'il s'agit par exemple du nombre de cycles et du nombre d'apparitions de chaque type d'instructions.

Pour chaque *quantum*, l'estimateur calcule la dégradation de performance que subirait le *quantum* en n'utilisant pas l'extension FPU. Cette dégradation est comparée au seuil *dégradation autorisé* choisi, afin de savoir si le *quantum* est considéré comme étant exécuté sur un cœur FPU ou sur un CB. Si le CB est sélectionné, alors le temps d'exécution sera le temps mesuré avec une FPU plus la dégradation de performance estimée.

Nous définissons l'équation suivante qui permet de calculer le temps d'exécution global d'une application :

$$\begin{aligned} \widehat{Tglobal} = TCore_{basic} + TCore_{fpu} \\ + \widehat{Degradation} + NbSwitch * \overline{SwitchNbCycles} \end{aligned} \tag{6.1}$$

Le résultat $\widehat{Tglobal}$ est l'estimation du temps d'exécution en nombre de cycles. La variable $\widehat{Degradation}$ est la somme, en nombre de cycles, de toutes les différences de temps d'exécutions des *quanta* placés sur un CB plutôt que sur un CE. Ce facteur de dégradation peut avoir des valeurs positives dans certains cas (comme par exemple l'application *susan smoothing* dont les résultats sont détaillés section 4.4.1 et Tableau 4.1).

La variable $NbSwitch$ est le nombre de fois où l'application a commuté entre un CB et un CE, ou l'inverse.

La variable $SwitchNbCycles$ est le coût en performance dû à la commutation. Il est estimé à 20 μs (soit 20k cycles à 1 Ghz), comme expliqué dans le document de ARM sur l'architecture multi-cœur asymétrique big.LITTLE [ARM11]. Cette référence est satisfaisante dans notre cas, car les architectures que nous simulons sont similaires. Cependant, il pourrait y avoir une différence en temps de commutation entre un CB et un CE. Contrairement à un CE qui contient une extension, dans le cas d'un CB il n'est pas nécessaire de recharger les registres de l'extension. Cela peut potentiellement réduire le temps de commutation. Une autre différence est que l'architecture *big.LITTLE* présentée dans ce papier est divisée en deux ensembles, l'un de quatre gros cœurs et l'autre de quatre petits cœurs. Lors de la commutation, le déplacement se fait d'un ensemble à un autre. Dans notre cas la commutation est seulement d'un processeur à un autre processeur. Les coûts pourraient être donc plus faibles, car il y a une quantité de données plus faible à transférer. Il est tout de même difficile d'estimer avec précision la différence, nous considérons donc l'estimation de 20 μs, comme pire cas.

Une estimation de la consommation énergétique est calculée par l'outil McPat à partir des données de simulation, issues de l'ensemble des traces exécutées sur un CE et un CB. Ensuite, les résultats générés par McPat sont utilisés pour calculer la consommation énergétique globale

selon l'équation suivante :

$$\begin{aligned} \widehat{Energy} = & \ ECore_{basic} + ECore_{fpu} \\ & + \widehat{Degradation} * \overline{ECore_{Basic}Emulating} \\ & + NbSwitch * \overline{ESwitchCost} \end{aligned} \tag{6.2}$$

Nous nous appuyons sur la figure 6.5 pour expliquer le modèle décrit par l'équation 6.2. Les variables $ECore_{basic}$ et $ECore_{fpu}$ représentent les valeurs de consommation énergétique, calculées par McPat pour chaque section de code, sur le cœur CB ou le cœur CE. Ces valeurs sont calculées de façon distinctes pour chaque *quantum* d'exécution des applications. Comme montré en figure 6.5, ces estimations sont utilisées directement pour calculer la consommation énergétique des intervalles 1,2,4. Dans l'intervalle 1, une section de code qui utilise des instructions FPU est exécutée sur un cœur doté d'une FPU, la consommation d'énergie est donc estimée à $ECore_{fpu(1)}$. Ensuite, pour les intervalles 2 et 4, des sections de code qui n'utilisent que des instructions entières sont exécutées sur un CB. L'outil nous fournit les estimations de chaque section $ECore_{basic}(2)$ $ECore_{basic}(4)$, en s'appuyant sur son modèle de cœur CB.

Lorsqu'une émulation d'instructions flottantes a lieu comme c'est le cas pour l'intervalle 3, l'estimation de la consommation énergétique est corrigée par le facteur $\widehat{Degradation} * \overline{ECore_{Basic}Emulating}$. La consommation énergétique dans ce cas est représentée par la somme de deux termes : $ECore_{basic(3)} + \widehat{Degradation}_{(3)} * \overline{ECore_{Basic}Emulating}$. La consommation énergétique $ECore_{basic}(3)$ est liée aux instructions entières contenues dans la section de code, montrées en gris clair dans la figure 6.5. Les instructions flottantes, montrées en gris foncé, sont émulées et donc converties en instructions entières (traits bleus).

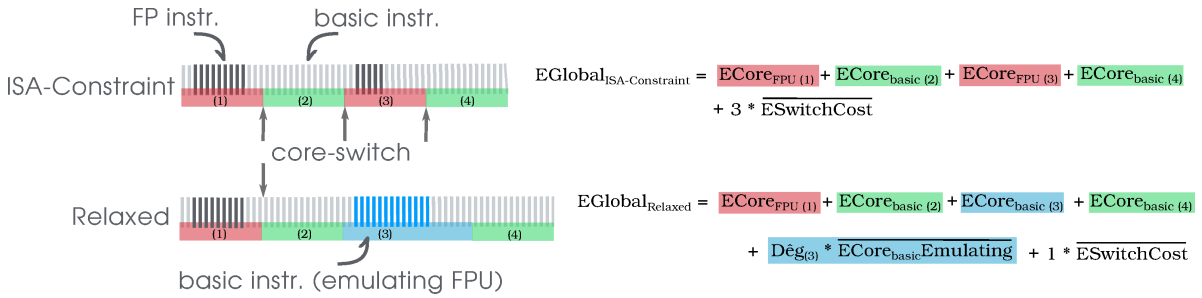


FIGURE 6.5 – Exemples d'ordonnancement afin d'expliquer l'estimation de l'énergie globale consommée.

Afin d'estimer la consommation énergétique de la section émulée, nous utilisons la constante $\overline{ECore_{Basic}Emulating}$ qui est la consommation moyenne en énergie d'un CB lorsqu'il est en train d'exécuter des fonctions d'émulation d'instructions flottantes. Cette valeur a été mesurée grâce à l'émulation des tests unitaires qui ont un pourcentage d'instructions flottantes supérieur à 80 %.

Ensuite, cette valeur d'énergie moyenne par cycle émulé ($\overline{ECore_{Basic}Emulating}$) est multipliée par le nombre de cycles nécessaires à l'émulation, estimé par $\widehat{Degradation}$, afin de calculer le surplus de consommation énergétique dû à l'émulation.

La variable $\overline{ESwitchCost}$ est l'énergie consommée lorsqu'il y a commutation d'une application d'un cœur vers un autre cœur. Cette valeur est estimée à $\lceil ECore_{Basic}/cycle \rceil * SwitchNbCycles$. Le pire cas est utilisée car la commutation de cœurs demande une activité importante de sauvegarde de contexte et nettoyage/remplissage du pipeline. Dans l'exemple en figure 6.5, le nombre de commutations est égal à 3 avec l'approche contrainte par l'ISA et seulement égal à 1 pour l'approche relaxée.

Dans ce modèle, l'estimation de $\widehat{Degradation}$ est utilisée pour estimer le temps d'exécution sur un CB. Pour avoir la valeur exacte de la dégradation, il aurait fallu implémenter la solution pour permettre d'avoir le code disponible sur les deux cœurs à "n'importe quel moment", puis implémenter l'ordonnanceur dans Linux. Ne connaissant pas le potentiel de cette solution, l'étude s'est dans un premier temps concentrée sur la démonstration du potentiel de cette solution. Les futurs travaux prévoient la réalisation d'une plateforme FAMP avec l'ordonnanceur relaxé. L'erreur de ce modèle pour calculer la dégradation est de 14 %, comme expliqué dans la section 5.4.2. Ce qui est, à notre avis, une précision suffisante pour une simulation système avec un but exploratoire.

Nous présentons dans la section suivante une étude où nous avons fait varier les valeurs des différentes variables, telles que la dégradation autorisée et la taille du *quantum* de l'ordonnanceur.

6.5 Résultats

Dans cette section, les résultats de l'exploration d'un *ordonnanceur relaxé* sont présentés. Ces résultats détaillent :

- la flexibilité de placement acquise en autorisant une dégradation locale par *quantum* à exécuter,
- le surcoût dû à cette dégradation locale autorisée et le déplacement des tâches,
- les gains potentiels en énergie grâce à la flexibilité acquise.

Les résultats sont comparés à l'ordonnanceur contraint par l'ISA décrit en section 6.2. Nous avons fait varier la *dégradation autorisée* lors du placement d'une tâche sur un CB. Le but est de voir la flexibilité gagnée en fonction de cette *dégradation autorisée*. Cette étude a été réalisée avec plusieurs tailles de *quantum* pour en voir l'impact sur la flexibilité et les gains.

Le fait d'étudier le gain potentiel en énergie est intéressant car il y a un point limite à ne pas dépasser dans la dégradation locale autorisée. En effet, une exécution lente sur un CB peut être plus coûteuse en énergie qu'une exécution rapide sur un CE. Ce principe est décrit figure 1.4.

6.5.1 Flexibilité de placement

La flexibilité de placement d'une tâche sur les différents cœurs est représentée par le pourcentage de temps pour lequel les applications peuvent être placées sur un CB. Dans cette étude, nous avons fait varier la taille du *quantum* de l'ordonnanceur et la *dégradation autorisée* afin d'observer l'impact sur la flexibilité de placement.

Le temps d'exécution moyen sur un CB (en pourcentage) pour les applications étudiées est présenté figure 6.6. L'axe des abscisses est la *dégradation autorisée* maximale par *quantum* lors de l'exécution sur un CB.

Nous notons que les résultats avec une *dégradation autorisée* de 0 % sont similaires à un ordonnanceur contraint par l'ISA, visible en figure 6.2. Pour mémoire, dans le modèle présenté en section 6.2, cet ordonnanceur attribue une tâche sur un CB seulement s'il n'y a que des instructions basiques. Nous comparons nos résultats à ce modèle que nous considérons comme référence.

Un premier constat est que plus la *dégradation autorisée* est élevée, moins la taille des *quanta* de l'ordonnanceur impacte la flexibilité. Avec une dégradation autorisée de 0 %, l'écart de la flexibilité entre une fine granularité d'ordonnancement et une grossière granularité est de 60 %, tandis qu'avec une *dégradation autorisée* à 100 %, l'écart est de seulement de 10 %.

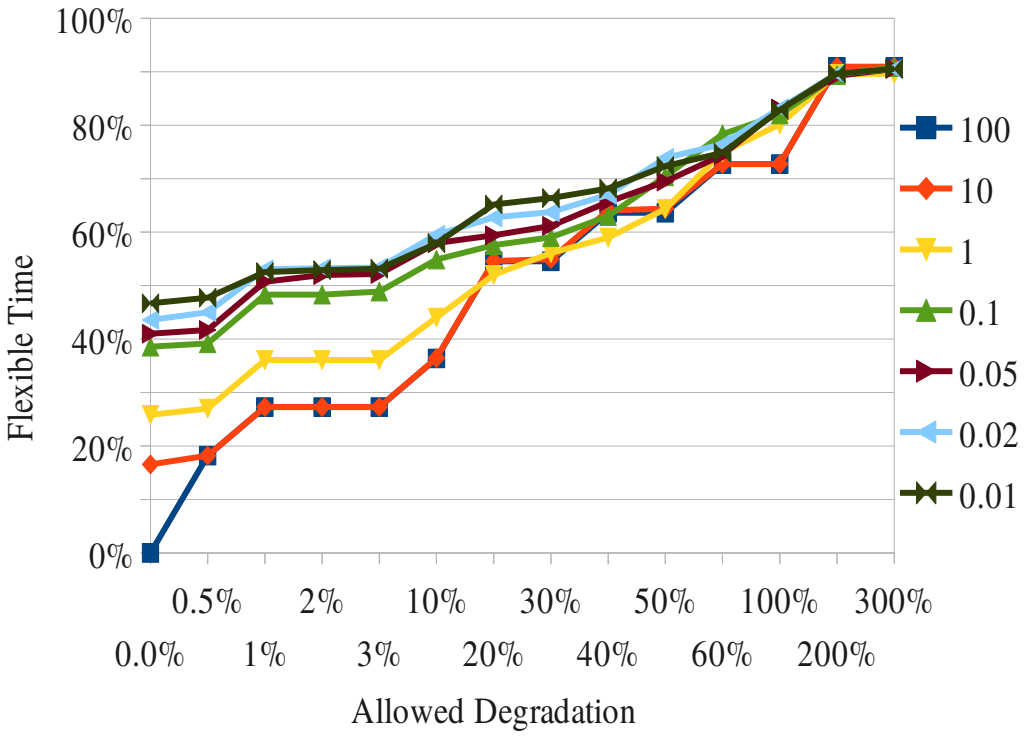


FIGURE 6.6 – Pourcentage moyen du temps où l'ordonnanceur a la flexibilité de choisir n'importe quel cœur (basique ou étendu) pour la prochaine section de code à exécuter. Flexibilité mesurée pour différentes dégradations autorisées et différentes tailles de *quantum* d'ordonnanceur.

Avec une dégradation autorisée de 0,5 %, la flexibilité est faiblement impactée. Cependant, l'ordonnanceur avec un *quantum* de 100 ms, qui avait au départ 0 % de flexibilité, en a maintenant 20 %. Avec 1 % de la dégradation, quelle que soit la taille des *quanta*, la flexibilité de l'ordonnanceur est augmentée d'environ 10 %. Un deuxième écart est proche de 20 % de la dégradation. À ce stade, la flexibilité de l'ordonnanceur est comprise entre 50 % à 65 %.

En explorant les extrêmes de la taille des *quanta*, nous remarquons que le fait d'avoir des *quanta* de 0,01 ms par rapport à 0,02 ms n'a pas d'impact sur la flexibilité, et de même pour la taille de 10 ms par rapport 100 ms (sauf lorsque la *dégradation autorisée* est à 0 %).

Dans l'ordonnanceur équitable 'Completely Fair Scheduler' (CFS) de Linux, la taille d'un *quantum* varie entre 0.75 ms et 5 ms et peut augmenter jusqu'à 100 ms. Lorsque le *quantum* est à 100 ms et qu'il n'y a pas de *dégradation autorisée* (0 %), l'ordonnanceur n'a pas de flexibilité, donc toutes les tâches s'exécutent sur un CE. Mais avec 20 % de dégradation autorisée et un *quantum* de 0,1 à 100 ms, l'ordonnanceur a maintenant plus de 50 % de flexibilité. En permettant une dégradation locale de 100 %, la flexibilité de placement atteint 80 %.

En conclusion, sans modifier la structure de l'ordonnanceur, notre module améliore fortement la flexibilité pour placer les tâches sur les cœurs. Par contre, cette flexibilité de placement implique un surcoût sur la performance globale, causée par la dégradation de certains *quanta* et par la commutation des tâches sur les cœurs. Ce surcoût est étudié dans la section suivante.

6.5.2 Surcoût en performance

Le surcoût en performance est dû à la somme des dégradations autorisées sur certains *quanta*, et par la commutation des tâches sur les cœurs. Le processus de commutation apparaît quand une tâche est déplacée d'un CB à un CE (ou l'inverse). Ce processus implique le temps de rallumer un cœur, le stockage et la restauration de l'état de l'architecture et les effets de cache. Pour mémoire, ce coût en performance est estimé à 20 μs comme expliqué section 6.4.

La figure 6.7 représente la dégradation globale moyenne en fonction de différents seuils de dégradation autorisée. Les coûts de commutation sont inclus. Avec un seuil de *dégradation autorisée* très bas, le système global a des gains en performance. Les gains en performance sont dus aux instructions qui sont plus coûteuses lorsqu'elles sont exécutées sur la FPU plutôt qu'avec une fonction logicielle d'émulation, comme les instructions de classes `FloatMemRead` et `FloatNegAbs` (voir la section 5.4.2 pour plus de détails).

En se plaçant à l'échelle de la taille de *quantum* de Linux, la dégradation globale correspondant à une *dégradation autorisée* de 20 % est de 2 %. Dans le cas d'un besoin de flexibilité plus agressive, l'ordonnanceur de Linux peut atteindre 80 % du temps flexible avec un surcoût global de 12 %. Pour atteindre 90 % de flexibilité, la dégradation globale monte à 30 % lorsque la taille des *quanta* configurée est grande. Mais avec une granularité plus fine, les *quanta* les plus accélérés sont exécutés sur un cœur FPU et ainsi la dégradation globale est réduite à moins de 20 %.

Pour rappel, dans cette exploration, certains surcoûts n'ont volontairement pas été pris en compte, comme l'erreur de prédiction et le coût d'avoir le code disponible pour un CB. Ce dernier peut être coûteux en utilisant la compilation juste-à-temps, mais peut être géré efficacement en utilisant la méthode *faute & réécrire* [GNVL14], ou d'autres méthodes de gestion de versions de binaires (détaillées Section 2.4). Ces surcoûts n'ont pas été pris en compte dans cette étude afin de voir les gains maxima atteignables avec la flexibilité.

En conclusion, un ordonnanceur relaxé permet à un système de type Linux d'avoir 50 % de temps flexible dans le placement de tâche. Ce qui était impossible avec un ordonnanceur

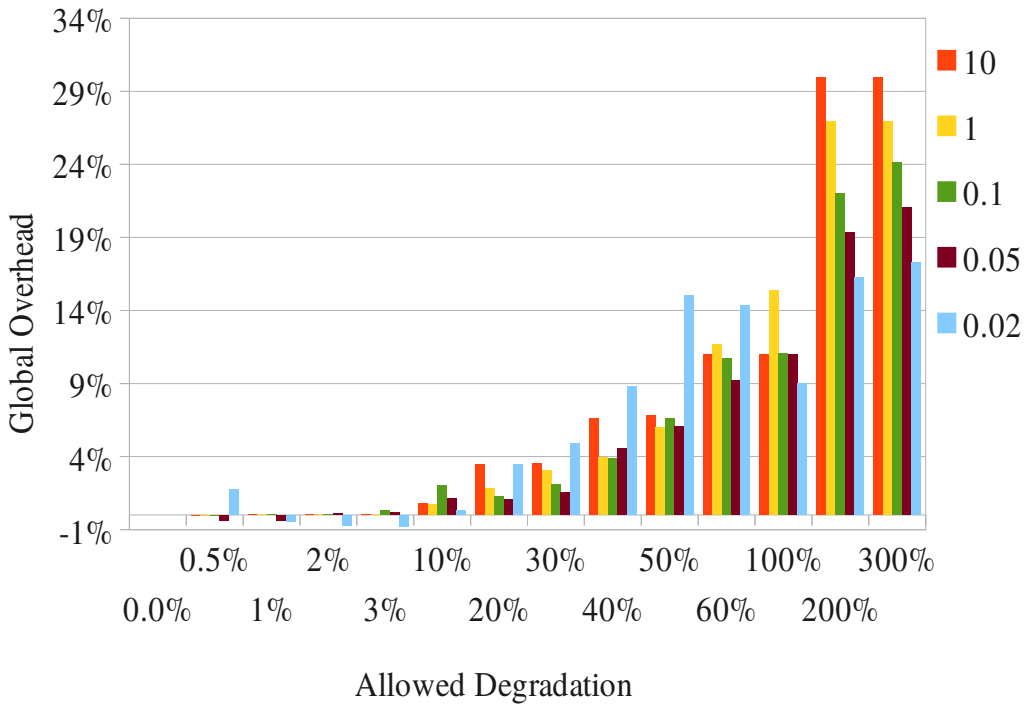


FIGURE 6.7 – *Dégradation globale* pour différentes *dégradations autorisées* et tailles de *quantum*.

contraint par l'ISA même en affinant, de la granularité. La dégradation moyenne globale pour avoir cette flexibilité reste faible, étant donné qu'elle est inférieure à 3 %. Cette flexibilité disponible est très prometteuse pour permettre au système d'exploitation de rester performant tout en répondant aux autres objectifs multicritères qui lui sont demandés. Pour voir les possibilités de gains obtenus grâce à cette flexibilité, le cas de l'efficacité énergétique a été étudié. Les résultats sont présentés dans la section suivante.

6.5.3 Cas d'étude : efficacité énergétique

La flexibilité permet le placement d'un plus grand nombre d'applications sur un CB. De cette manière, l'énergie consommée par la plateforme pour exécuter toutes les applications peut être réduite. En effet, en plaçant les tâches sur un CB, les CE peuvent être désactivés de sorte que la consommation d'énergie statique est réduite. Toutefois, il existe un compromis énergétique entre une exécution lente sur un CB et une exécution rapide sur un CE. Ainsi, un mauvais choix de seuil de *dégradation autorisée* peut réduire l'efficacité énergétique. Un point fonctionnel optimal peut être défini.

Dans cette section, les résultats sur l'efficacité énergétique sont présentés pour différentes dégradations autorisées et pour différentes tailles du *quantum* de l'ordonnanceur. Les résultats avec l'ordonnanceur relaxé sont comparés avec les résultats de l'ordonnanceur contraint par l'ISA.

Comparaison avec un ordonnanceur contraint par l'ISA

Dans la figure 6.8, l'effet de la flexibilité sur l'efficacité énergétique est présenté. Les résultats sont normalisés par rapport à la valeur à 0 % de dégradation et un *quantum* à 100 ms.

Tout d'abord, la figure montre qu'une plus grande *dégradation autorisée* et la réduction de la taille du *quantum* impacte la consommation énergique, de plus, la forme convexe des lignes signifie qu'il y a un point optimal. La valeur du seuil de dégradation autorisée optimale en termes de consommation d'énergie est de près de 20 %. Cette valeur est dépendante de l'extension. Plus l'extension a un coût énergétique élevé, plus la valeur du seuil peut être élevée. Lorsque le seuil de dégradation autorisée dépasse 20 %, la *dégradation globale* est trop importante et cela réduit les gains en énergie, dans notre cas d'étude.

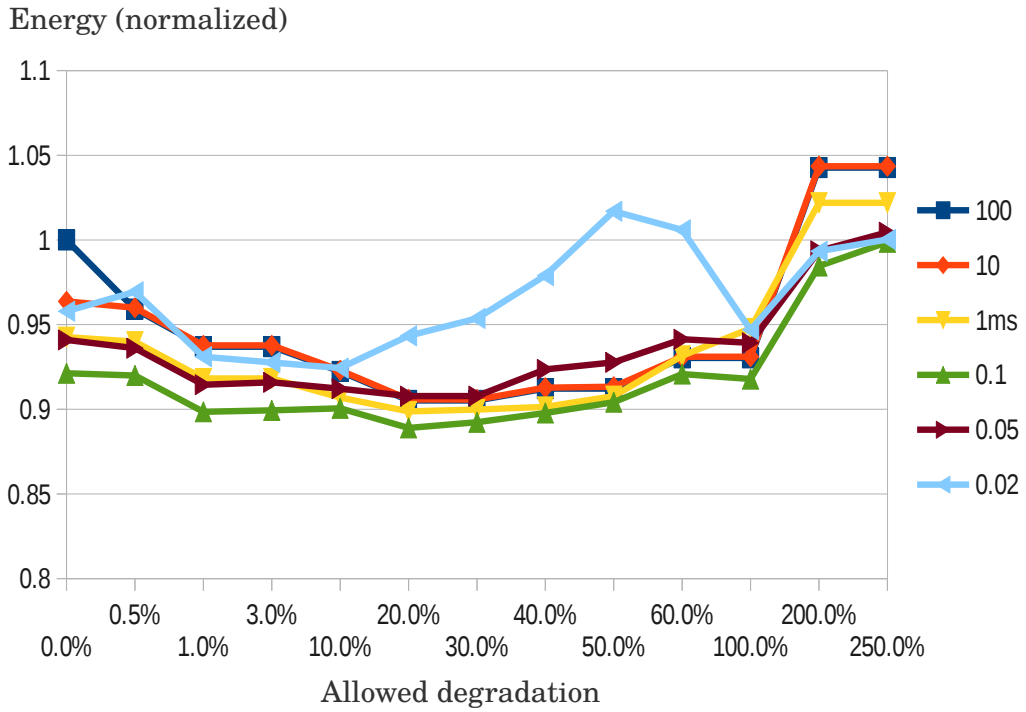


FIGURE 6.8 – Énergie moyenne consommée pour différentes dégradations autorisées et différentes tailles de *quantum*. Le point de fonctionnement le plus efficace est à 20 % de *dégradation autorisée* avec un *quantum* de taille 0.1 ms.

Pour ce point de fonctionnement optimal en termes de consommation énergétique, l'économie d'énergie est de 11 % en moyenne par rapport à la solution avec 100 ms et contraint par l'ISA (équivalent à un multi-cœur symétrique, les *quantum*s sont si grands qu'une application n'a pas de périodes basiques détectées). Le surcoût moyen en performance globale sur l'ensemble de l'application, montré en figure 6.7, est d'environ 2 %. Par comparaison, un ordonnanceur contraint par l'ISA (soit 0 % de la dégradation autorisée sur la figure) sur l'ordonnanceur de Linux permet seulement 5 % d'économie d'énergie. L'ordonnanceur relaxé permet deux fois plus d'économie d'énergie.

Sur la figure 6.8 la courbe bleue clair, correspondant à une taille de *quantum* à 0,02 ms, monte à partir de 10 % puis redescend à 100 %. Ce comportement est dû au fait que d'une part le coût de commutation est de 0,02 ms, et d'autre part qu'il y a beaucoup de commutations entre les deux cœurs. À 100 %, la dégradation autorisée est tellement élevée que les applications passent beaucoup plus de temps sur un CB, ce qui réduit le nombre de commutations.

Pour conclure ce cas d'étude, réalisé sur une simulation des ARM Cortex A9 avec/sans

l'extension NEON FPU, et avec une granularité d'un ordonnanceur type CFS de Linux, le point optimal de fonctionnement est obtenu avec un dégradation autorisée de 20 %. Grâce à la flexibilité apporté par l'ordonnanceur relaxé, l'économie d'énergie est doublée par rapport à un ordonnanceur contraint par l'ISA, pour un surcoût en performance inférieur à 3 %.

6.6 Discussion et perspectives

Nous avons réalisé une étude exploratoire d'ordonnanceur relaxé sur une plateforme asymétrique en fonctionnalités. Les résultats montrent que la solution a du potentiel et nous présentons une discussion autour de certains points que l'étude ne traite pas et qui pourraient être explorés pour la suite des travaux.

Valorisation des extensions

Les résultats présentés dans cette étude se basent sur une extension qui reste faiblement coûteuse en énergie et surface par rapport à un CB. Dans le futur, les extensions auront une complexité de plus en plus importante. Une architecture FAMP permettra d'ajouter des points de fonctionnement encore plus performants, mais aussi d'apporter des économies en surface et en énergie encore plus importantes par rapport aux architectures SMP.

L'architecture FAMP ouvre une porte pour de nouvelles extensions. En effet, le fait de devoir obligatoirement dupliquer l'extension sur chaque cœur dans un SMP réduit le nombre d'extensions potentiellement intégrables. Mais dans une architecture FAMP, l'extension pourrait être intégrée avec parcimonie.

Mise à l'échelle de la solution

L'étude réalisée considère le cas d'une asymétrie avec seulement une extension. Néanmoins, la solution proposée peut être étendue pour plusieurs extensions. Avec n extensions, il faut calculer une estimation pour chaque type de cœurs. Ces cœurs peuvent contenir 0,1,2..n extensions. De plus, une extension peut être utilisée dans une librairie d'émulation d'une autre extension. Donc, dans le pire cas, le nombre d'estimations à réaliser est $\sum_{k=0}^{n} \binom{n}{k} = 2^n$. Par contre, si les extensions sont indépendantes entre elles (les extensions ne sont pas utilisées dans les librairies d'émulation), le nombre d'estimations à réaliser correspond au nombre d'extensions. Ensuite pour les processeurs où il manque des extensions, il suffit de faire la somme des dégradations en fonction des extensions indisponibles. Pour réduire le nombre d'estimations , il pourrait également être efficace d'avoir des informations dans l'entête de l'application (et dans le contexte d'exécution), permettant de savoir si l'application est sujette à utiliser certains types d'extensions.

Afin d'optimiser le nombre de types de cœurs et de dimensionner l'architecture FAMP, une étude pourrait être aussi réalisée pour savoir si certaines extensions sont utilisées aux mêmes moments où bien l'inverse.

Coûts de monitoring

L'architecture FAMP réduit la surface et la consommation énergétique en n'intégrant qu'un nombre limité d'extensions. Néanmoins, il est nécessaire de ne pas contrebalancer ces gains à cause de la mise en place du système pour monitorer la plateforme (en particulier l'utilisation des extensions). Actuellement il y a déjà des compteurs matériels dans les processeurs commerciaux. Mais pour monitorer plus précisément les extensions, il est peut être nécessaire d'en ajouter certains. Ces compteurs peuvent être coûteux en surface et en consommation énergétique, car

ce sont des registres mémoire. Dans une extension, il y a souvent un banc de registres (par exemple 16 registres de 128 bits pour l'extension ARM NEON du Cortex A9, plus des registres de contrôle et configuration), donc le fait d'enlever une extension permet des gains importants. Ainsi, il est possible d'ajouter des compteurs matériels sur les autres extensions, mais il est nécessaire de les limiter pour ne pas perdre les gains en surface et en consommation d'énergie. De plus, pour réduire les surcoûts de performance il est possible d'en échantillonner la surveillance de l'utilisation logicielle sur un CB, par exemple en utilisant la librairie d'émulation monitorée seulement à certaines périodes.

Prédiction

Une suite des travaux peut être la réalisation d'un prédicteur précis et léger permettant de mieux prévoir la future accélération grâce à l'extension. Plus de détails sont fournis section 6.3.2.

Gestion de l'ISA asymétrique

Dans cette étude, la flexibilité de placement est étudiée en posant l'hypothèse qu'il existe une solution pour gérer l'exécution du code sur n'importe quel cœur. Par exemple, en s'appuyant sur les travaux qui sont discutés section 2.4. Néanmoins, des axes de recherches sont possibles pour l'amélioration de la gestion de l'asymétrie, par exemple, autour de la mise en place d'une solution de multi-versions de code ou virualisation légère spécialisée pour le cas de l'asymétrie d'ISA. Les solutions actuelles ne sont pas encore optimisées pour les architectures FAMP. Elles ne prennent pas en compte le fait qu'une partie des instructions ne sera presque jamais exécutée (émulée) sur un CB. Dans le cas de la virtualisation par exemple, celle ci peut être hybride comme décrit dans le papier de Vapor SIMD [NDR+11]. C'est-à-dire que seulement les instructions des extensions sont virtualisées. Si l'on considère que les extensions ont des instructions à accélération très faible et des instructions à accélération très forte, comme par exemple *FloatAdd* (x40) et *FloatSqrtDouble* (plus de x1000), il ne semble pas obligatoire de toutes les virtualiser. En effet, dans le cas où l'ordonnanceur ne place pas de tâches sur un CB lorsque l'accélération de celle-ci à utiliser un CE est trop forte, il est possible que les sections de code contenant des instructions à forte accélération ne soient jamais sur des CB. Par exemple, les sections contenant des instructions *FloatSqrtDouble* n'iront jamais sur des CB. Dans ce cas, il devient inefficace de les virtualiser.

En résumé, la discussion et les perspectives montrent que les instructions partagées entre les cœurs et la différence d'accélération entre les instructions d'une même extension ne sont pas assez exploitées dans les solutions actuelles pour gérer le code. Ces deux points sont des pistes pour les travaux futurs.

6.7 Conclusion

Pour étudier le compromis d'une plate-forme hétérogène et d'une couche logicielle homogène, ce chapitre explore le potentiel à avoir un ordonnancement relaxé sur une architecture FAMP. Cette approche d'ordonnancement apporte la flexibilité pour placer les tâches en cours d'exécution sur différents types de processeurs, indépendamment des extensions requises par la tâche. Grâce à cette indépendance, elle permet à l'ordonnanceur d'optimiser les objectifs tels que l'économie d'énergie.

En simulant un architecture FAMP, l'étude montre qu'un ordonnanceur relaxé permet à un système de type Linux d'avoir 50 % du temps flexible dans le placement de tâche. Ce qui

était impossible avec un ordonnanceur contraint par l'ISA, même en affinant de la granularité. La dégradation moyenne globale pour avoir cette flexibilité reste faible, étant donné qu'elle est inférieure à 3 %. Ce surcoût en temps d'exécution est faible grâce à l'estimateur de dégradation qui sélectionne les meilleurs candidats à exécuter sur le cœur de faible puissance sans perdre trop de performances.

Cette flexibilité disponible est très prometteuse pour permettre au système d'exploitation d'être performant et en même temps répondre aux autres objectifs multicritères qui lui sont demandés. Par exemple, avec 50 % de la flexibilité, qui est l'optimum pour la fonction d'économie d'énergie, cette solution augmente l'économie d'énergie de 2 fois par rapport à un ordonnanceur contraint par l'ISA.

Pour répondre aux multiples objectifs du duo ordonnanceur et processeur, il devient crucial d'avoir la puissance d'un matériel hétérogène et la simplicité et la flexibilité d'une couche homogène plus abstraite. L'approche d'un ordonnanceur relaxé sur une architecture FAMP montre un grand potentiel pour aborder ces questions.

La section suivante est une conclusion globale sur la thèse.

Chapitre 7

Conclusions et perspectives

De nos jours, les systèmes d'information ont besoin de ressources hétérogènes pour répondre à plusieurs objectifs tels que la qualité de service, l'efficacité énergétique, le respect de température de fonctionnement et la priorisation d'applications. Le processeur est une ressource clé pour permettre aux systèmes d'information de tenir ces objectifs. L'une des architectures qui semble être un bon compromis entre la puissance de l'hétérogénéité et la simplicité de l'homogénéité est l'architecture FAMP. Cette architecture est un multi-cœur dont certains cœurs contiennent des extensions spécialisées en plus. En effet, ces extensions n'étant pas utilisées tout le temps et ayant un coût énergétique et surfacique important, il peut être efficace de ne pas les intégrer dans chaque cœur. Néanmoins, cette asymétrie rend le placement des tâches contraint par l'utilisation des extensions.

Le but de la thèse est de savoir si les différents profils de l'utilisation des extensions permettent des optimisations dynamiques et si l'utilisation d'un ordonnanceur relaxé est efficace sur une architecture FAMP.

La thèse montre que les extensions sont utilisées par phases et qu'il est nécessaire d'avoir une granularité assez fine pour les exploiter. Ainsi, le fait d'agir au niveau application n'est pas efficace. Le fait d'agir au niveau instructions, avec une solution type power-gating est efficace en énergie mais ne réduit pas la surface. De plus, l'architecture FAMP avec un ordonnanceur contraint par l'ISA est trop limitée dans le placement de tâches pour être efficace. Par contre, lorsque l'ordonnanceur est libre de placer les tâches où il le souhaite, tout en prenant compte la dégradation impliquée, une meilleure efficacité peut être atteinte pour une légère perte de performance. Cette relaxation de l'ordonnanceur permet de répondre aux multi-objectifs des systèmes : qualité de service, efficacité énergétique, priorisation et température.

La prochaine section synthétise les travaux réalisés, puis la section suivante résume les perspectives de ces travaux.

7.1 Synthèse des Travaux

L'usage des extensions

L'utilisation des extensions lors de l'exécution est due à la demande manuelle du programmeur (ajout d'instructions spécialisées), à des appels de librairies, à l'utilisation de certains types de données, ou à l'optimisation automatique du compilateur. Pour extraire les profils d'utilisation, une analyse de l'exécution d'une cinquantaine applications de l'état de l'art a été réalisée. A travers toutes ces applications, des comportements ressortent : l'utilisation est variable au cours d'une même application et elle varie sous forme de phases. Ces phases sont par

exemple : un haut volume d'utilisation, pas d'utilisation du tout, ou une très faible utilisation. Les variations et les différents profils d'utilisation sont des éléments intéressants pour des optimisations dynamiques. Ces travaux ont fait l'objet d'une publication au *International Workshop on Adaptive Self-tuning Computing Systems* à Hipeac'14 [ALCC14].

Gestion de l'extension FPU dans les multi-cœurs

L'utilisation des extensions varie au cours d'une même application, donc la granularité d'action pour gérer les extensions va impacter le fait de capturer ou non les phases. La question de la granularité d'action sur un FAMP pour la gestion de l'asymétrie se pose. Une analyse des méthodes actuelles pour gérer la FPU a été réalisée et les méthodes ont été comparées avec différentes configurations de granularité. Pour comparer ces méthodes, un environnement de simulation a été mis en place. Au final, la méthode fin grain (type power-gating) présente le plus d'efficacité énergétique, mais implique des coûts de surface non négligeables. La méthode gros grain statique réduit considérablement la performance. Enfin, le niveau ordonnanceur pourrait être le plus adapté. Or nativement, ce niveau est limité par les apparitions spontanées d'instructions. En effet, l'ordonnanceur doit placer la tâche sur un CE lorsqu'une instruction étendue apparaît. Ces travaux ont fait l'objet d'une publication au *workshop Multi-Objective Many-Core Design* (MOMAC) à ARCS'15 [ALC+15a].

Estimateur d'accélération par extension matérielle

Afin de tenir compte de l'intérêt d'utiliser des extensions et pas uniquement de l'apparition des instructions, il est nécessaire d'avoir une solution en ligne qui estime l'accélération apportée par l'extension. Notre étude a montré que la solution naïve qui calcule l'accélération en fonction du pourcentage globale d'utilisation n'est pas précise. Cette solution présente une erreur absolue moyenne de 67%. Pour obtenir une estimation plus précise, un autre modèle a été proposé, permettant 4,8 fois moins d'erreur (14% d'erreur absolue moyenne). Ces travaux ont fait l'objet d'une publication au *IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip* (MCSoC-15) [ALC+15b].

Ordonnancement relaxé et intelligent pour processeurs asymétriques en fonctionnalités

Enfin, le dernier chapitre de la thèse explore une solution d'ordonnancement relaxé (sans se contraindre à l'apparition d'instructions spécialisées) sur une architecture FAMP. Le placement d'une tâche sur un CB plutôt que sur un CE peut réduire fortement la performance. Pour éviter ces cas, nous préconisions l'utilisation d'un estimation d'accélération et ainsi en borner le placement des tâches à une dégradation maximum. Une étude de l'estimation des coûts globaux et des gains potentiels en énergie grâce à la flexibilité de placement a été réalisée grâce à la mise en place d'un environnement qui simule une architecture FAMP. Les résultats montrent que l'ordonnanceur a plus de flexibilité et qu'il est aussi moins dépendant de la granularité d'action. De plus, les résultats montrent que l'énergie gagnée par rapport à un SMP est deux fois meilleure avec un ordonnanceur relaxé qu'avec un ordonnancement contraint par le jeu d'instructions. Ces travaux ont fait l'objet d'une publication en soumission et d'un dépôt de brevet [ALC+15c].

7.2 Perspectives

Dans cette thèse, *les perspectives sont détaillées à chaque fin de chapitre*. Pour résumer les perspectives, ces travaux au niveau modèle et simulation montrent que l'on peut passer à la réalisation d'une plateforme FAMP plus complète et à l'implémentation d'un ordonnanceur relaxé dans un système d'exploitation tel que Linux. Ensuite, pour continuer à explorer le potentiel de l'architecture FAMP, deux axes de recherche pourraient être envisagés sur le plan architectural et logiciel :

Sur le plan matériel, il reste des questions ouvertes sur la conception de l'architecture FAMP telles que : *Quel est le nombre optimisé d'extensions à mettre en place dans une architecture FAMP ?* Sous-dimensionner le nombre d'extensions pourrait impliquer des pertes de performances trop importantes car les tâches attendraient pour accéder aux cœurs avec extension ou trop de tâches (potentiellement fortement accélérées par l'extension) seraient exécutées sur une CB.

Doivent-elle être placées ensemble sur le même cœur ? Une extension SIMD sur données entières n'est peut-être pas utilisée lors d'une phase intense en calculs sur données flottantes (utilisant ainsi une extension FPU). Pour optimiser l'utilisation, il faudrait avoir un cœur avec une extension SIMD et un autre cœur avec seulement une extension FPU, et non un cœur avec les deux extensions.

Quel est le cœur basique (CB) minimal à avoir lorsqu'une extension est associée à un cœur ? Certaines fonctionnalités internes d'un cœur ne sont pas obligatoires si une extension peut aussi les satisfaire ou inversement. Par exemple, lorsqu'un code est régulier et bien parallélisable, celui-ci va être exploitable par une extension SIMD, donc il n'est pas nécessaire d'avoir une base de cœur qui sache exécuter plusieurs instructions en parallèle dans le désordre. Dans ce cas là, une base de cœur dans l'ordre est suffisante. L'architecture optimisée semble donc être un assemblage du modèle FAMP et PAMP.

Sur le plan logiciel, les questions ouvertes sont *quelle est la méthode de prédiction de l'utilisation des extensions la plus efficace ?* Il y a un compromis à trouver entre le surcoût de la prédiction en ligne et la précision nécessaire. Par exemple, une solution simple est de s'appuyer sur la localité temporelle et de prétendre que la prochaine section est similaire à la dernière exécutée. D'autres solutions peuvent être explorées telles que celles s'appuyant sur les méthodes de prédictions de branchements.

Comment améliorer la méthode pour rendre le code exécutable sur n'importe quel cœur ? Les solutions actuelles ne sont pas encore optimisées pour les architectures FAMP. Elles ne prennent pas en compte le fait qu'une partie des instructions ne sera presque jamais exécutée (émulée) sur un CB. En effet, dans le cas où l'ordonnanceur ne place pas de tâches sur un CB lorsque l'accélération de celle-ci à utiliser un CE est trop forte, il est possible que les sections de code contenant des instructions à forte accélération ne soient jamais sur des CB. Par exemple, les sections contenant des instructions calculant la racine carrée sur des données *double flottante* n'iront jamais sur des CB. Dans ce cas, il devient inefficace de les virtualiser.

Pour conclure, ces travaux ont contribué à montrer le potentiel de l'architecture FAMP et l'intérêt d'avoir un *ordonnanceur relaxé*. L'estimateur dynamique proposé permet à l'ordonnanceur d'être moins contraint tout en plaçant les tâches efficacement. Ces résultats n'avaient jamais été obtenus sur la plateforme considérée dans notre étude. Ils ouvrent la porte à des perspectives d'amélioration de la mise à l'échelle des multi-cœurs et apportent des réponses aux besoins multi-objectifs des systèmes d'information. Ces résultats sont très encourageants et contribuent

au développement d'une plateforme complète FAMP. Cette thèse a fait l'objet d'un dépôt de brevet, de trois communications scientifiques internationales (plus une en soumission), et contribue à deux projets européens [BEN, THI]. De plus, le laboratoire met en place la structure pour réaliser du transfert industriel en proposant une brique technologique qui se base sur les travaux de cette thèse.

Annexe A

Liste des Publications

Articles scientifiques

Flexible Task Mapping on Overlapping-ISA Multi-Core Processors
Alexandre Aminot, Yves Lhuiller, Andrea Castagnetti, Henri-Pierre Charles. en soumission

FPU Speedup Estimation for Task Placement Optimization on Asymmetric Multicore Designs
Alexandre Aminot, Yves Lhuiller, Andrea Castagnetti, Henri-Pierre Charles. MCSOC 2015 [ALC+15b]

Floating Point Units Efficiency in Multi-Core Processors
Alexandre Aminot, Yves Lhuiller, Andrea Castagnetti, Henri-Pierre Charles. MOMAC Workshop, ARCS 2015 [ALC+15a]

On the advantage of time-varying diversity of workload on functionally asymmetric multi-core
Alexandre Aminot, Yves Lhuillier, Alain Chateigner, Henri-Pierre Charles. ADAPT Workshop, Hipeac'14. [ALCC14]

Early Design Stage Thermal Evaluation and Mitigation : the Locomotiv Architectural Case
Tanguy Sassolas, Chiara Sandionigi, Alexandre Guerre, Alexandre Aminot, Pascal Vivet, Hela Boussetta, Luca Ferro, Nicolas Peltier. DATE'14. [SSG+14]

PACHA : Low Cost Bare Metal Development for Shared Memory Manycore Accelerators
Alexandre Aminot, Alexandre Guerre, Julien Peeters, Yves Lhuillier. ALCHEMY Workshop, ICCS'13. [AGPL13]

Brevets

Placement d'une tâche de calcul sur un processeur fonctionnellement asymétrique
Alexandre Aminot, Yves Lhuillier, Andrea Castagnetti, Alain Chateigner, Henri-Pierre Charles. Brevet déposé le 20 avril 2015.

Divers

Contributions projets Européens

BENEFIC - Best ENergy EFficiency solutions for heterogeneous multI-core Communicating systems, Projet CATRENE. 2013-2016
http ://benefic.tudelft.nl [BEN]

Things2Do - THIN but Great Silicon 2 Design Objects, Projet ENIAC JU. 2014-2017
http ://things2do.space.com.ro [THI]

Posters

Should extension units be in each core of a multi-core processor ?
Alexandre Aminot, Yves Lhuillier, Andrea Castagnetti, Henri-Pierre Charles. ACACES'14.

Towards a better efficiency on heterogeneous platforms : Conception of an instruction set family for virtualization in system-on-chip.
Alexandre Aminot, Yves Lhuillier, Alain Chateigner, Henri-Pierre Charles. JDD EDMSTII'14.

PACHA : a low overhead, Platform Agnostic, Close-to-HArdware programming
Alexandre Aminot, Alexandre Guerre, Yves Lhuillier, Maroun Ojail. Workshop P2012/STHORM, DATE'13

Annexe B

Bibliographie

[ABC+06] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, and othersst. The landscape of parallel computing research : A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[AGPL13] Alexandre Aminot, Alexandre Guerre, Julien Peeters, and Yves Lhuillier. Pacha : Low cost bare metal development for shared memory manycore accelerators. *Procedia Computer Science*, 18(0) :1644 – 1653, 2013. 2013 International Conference on Computational Science.

[ALB+03] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. Llva : A low-level virtual instruction set architecture. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 205–, Washington, DC, USA, 2003. IEEE Computer Society.

[ALC+15a] Alexandre Aminot, Yves Lhuiller, Andrea Castagnetti, Henri-Pierre Charles, and Henri-Pierre Charles. Floating point units efficiency in multi-core processors. In *Architecture of Computing Systems. Proceedings, ARCS 2015 - The 28th International Conference on*, pages 1–8, March 2015.

[ALC+15b] Alexandre Aminot, Yves Lhuiller, Andrea Castagnetti, Henri-Pierre Charles, and Henri-Pierre Charles. Fpu speedup estimation for task placement optimization on asymmetric multicore designs. In *Embedded Multicore/Many-core Systems-on-Chip. Proceedings, MCSoC 2015 - IEEE 9th International Symposium on*, pages 1–8, September 2015.

[ALC+15c] Alexandre Aminot, Yves Lhuiller, Andrea Castagnetti, Alain Chateigner, Henri-Pierre Charles, and Henri-Pierre Charles. Placement d'une tâche de calcul sur un processeur fonctionnellement asymétrique, April 2015.

[ALCC14] Alexandre Aminot, Yves Lhuillier, Alain Chateigner, and Henri-Pierre Charles. On the advantage of time-varying diversity of workload on functionally asymmetric multi-core. In *Proceedings of International Workshop on Adaptive Self-tuning Computing Systems*, ADAPT '14, pages 11 :11–11 :13, New York, NY, USA, 2014. ACM.

[AMD] AMD. Kaveri - multicore cpu with amd radeon graphics - http ://www.amd.com/en-us/products/processors/desktop/a-series-apu.

[AN09] Cédric Augonnet and Raymond Namyst. A unified runtime system for heterogeneous multi-core architectures. In Eduardo César, Michael Alexander, Achim Streit, Jesper Träff, Christophe Cérin, Andreas Knüpfer, Dieter Kranzlmüller, and Shantenu Jha, editors, *Euro-Par 2008 Workshops - Parallel Processing*, volume 5415 of

Lecture Notes in Computer Science, pages 174–183. Springer Berlin / Heidelberg, 2009.

[ARM11] ARM. big.little processing with arm cortex-a15 & cortex-a7. White paper, September 2011.

[ARM12] ARM. Cortex-a9 technical reference manual, June 2012. revision : r4p1.

[BBB+11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2) :1–7, August 2011.

[BBSG11] Michael Butler, Leslie Barnes, Debjit Das Sarma, and Bob Gelinas. Bulldozer : An approach to multithreaded compute performance. *IEEE Micro*, 31(2) :0006–15, 2011.

[BC06] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd Conference on Computing Frontiers*, CF '06, pages 29–40, New York, NY, USA, 2006. ACM.

[BC11] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5) :67–77, May 2011.

[BDH+10] Andre R Brodtkorb, Christopher Dyken, Trond R Hagen, Jon M Hjelmervik, and Olaf O Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1) :1–33, 2010.

[BEN] BENEFIC. European catrene project - best energy efficiency solutions for heterogeneous multi-core communicating systems http ://benefic.tudelft.nl/.

[Bie11] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[BL93] Thomas Ball and James R. Larus. Branch prediction for free. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 300–313, New York, NY, USA, 1993. ACM.

[BSM01] Tracy Braun, Howard Siegel, and Anthony Maciejewski. Heterogeneous computing : Goals, methods, and open problems. In Burkhard Monien, Viktor Prasanna, and Sriram Vajapeyam, editors, *High Performance Computing ? HiPC 2001*, volume 2228 of *Lecture Notes in Computer Science*, pages 307–318. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45307-527.

[CB99] Ernst Christen and Kenneth Bakalar. Vhdl-ams-a hardware description language for analog and mixed-signal applications. *Circuits and Systems II : Analog and Digital Signal Processing, IEEE Transactions on*, 46(10) :1263–1272, 1999.

[CHY+07] N. Clark, A. Hormati, S. Yehia, S. Mahlke, and K. Flautner. Liquid simd : Abstracting simd hardware using lightweight dynamic mapping. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 216 –227, feb. 2007.

[CJVDP08] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP : portable shared memory parallel programming*, volume 10. MIT press, 2008.

[CSH+12] N. Chitlur, G. Srinivasa, S. Hahn, P.K. Gupta, D. Reddy, D. Koufaty, P. Brett, A. Prabhakaran, Li Zhao, N. Ijih, S. Subhaschandra, S. Grover, Xiaowei Jiang, and R. Iyer. Quickia : Exploring heterogeneous architectures on real prototypes. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1 –8, feb. 2012.

[DBB07] Romain Dolbeau, Stéphane Bihan, and François Bodin. Hmpp : A hybrid multi-core parallel programming environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

[DJR01] S. Dutta, R. Jensen, and A. Rieckmann. Viper : A multiprocessor soc for advanced set-top box and digital tv systems. *Design Test of Computers, IEEE*, 18(5) :21 –31, sep-oct 2001.

[DTD03] E. Duesterwald, J. Torrellas, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, pages 220–231, 2003.

[DVT12] Matthew DeVuyst, Ashish Venkat, and Dean M. Tullsen. Execution migration in a heterogeneous-isa chip multiprocessor. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 261–272, New York, NY, USA, 2012. ACM.

[EBSA+11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[ECC14] Fernando A Endo, Damien Courousse, and Henri-Pierre Charles. Micro-architectural simulation of in-order and out-of-order arm microprocessors with gem5. In *Embedded Computer Systems : Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, pages 266–273, July 2014.

[FSSP09] Alexandra Fedorova, Juan Carlos Saez, Daniel Shelepov, and Manuel Prieto. Maximizing power efficiency with asymmetric multicore systems. *Commun. ACM*, 52(12) :48–57, December 2009.

[Fur04] Grigori G Fursin. Iterative compilation and performance prediction for numerical applications. 2004.

[GBEL10] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen wcet benchmarks : Past, present and future. In *OASIcs-OpenAccess Series in Informatics*, volume 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[GKBS13] Vishakha Gupta, Rob Knauerhase, Paul Brett, and Karsten Schwan. Kinship : Efficient resource management for performance and functionally asymmetric platforms. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 16 :1–16 :10, New York, NY, USA, 2013. ACM.

[GLS99] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI : portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[GM12] Kate Gregory and Ade Miller. *C++ AMP : Accelerated Massive Parallelism with Microsoft® Visual C++®*. " O'Reilly Media, Inc.", 2012.

[GNL13] Giorgis Georgakoudis, Dimitrios S. Nikolopoulos, and Spyros Lalis. Fast dynamic binary rewriting to support thread migration in shared-isa asymmetric multicores. In *Proceedings of the First International Workshop on Code OptimiSation for MultI and many Cores*, COSMIC '13, pages 4 :1–4 :10, New York, NY, USA, 2013. ACM.

[GNUa] GNU. http ://gcsccc.gnu.org/bugzilla/show_bug.cgi ?id=43422 (visited april2015).

[GNUb] GNU. Vectorization gcc - http ://gcc.gnu.org/projects/tree-ssa/vectorization.html (visited april 2015).

[GNVL14] Giorgis Georgakoudis, Dimitrios S. Nikolopoulos, Hans Vandierendonck, and Spyros Lalis. Fast dynamic binary rewriting for flexible thread migration on shared-isa heterogeneous mpsocs. In *Embedded Computer Systems : Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, pages 156–163, July 2014.

[GRE+01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench : A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[GRSW04] E. Grochowski, R. Ronen, J. Shen, and Hong Wang. Best of both latency and throughput. In *Computer Design : VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*, pages 236 – 243, oct. 2004.

[GS13] Vishal Gupta and Karsten Schwan. Brawny vs. wimpy : Evaluation and analysis of modern workloads on heterogeneous processors. *IPDPSW*, 2013.

[Gsc07] M. Gschwind. The cell broadband engine : Exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming*, 35(3) :233–262, 2007.

[HB06] K Heydemann and F Bodin. Iterative compilation for two antagonistic criteria : Application to code size and performance. In *Proceedings of the 4th Workshop on Optimizations for DSP and Embedded Systems, colocated with CGO*, 2006.

[HBC05] Karine Heydemann, Francois Bodin, and Henri-Pierre Charles. A software-only compression system for trading-offs between performance and code size. In *Proceedings of the 2005 Workshop on Software and Compilers for Embedded Systems*, SCOPES '05, pages 27–36, New York, NY, USA, 2005. ACM.

[hp] http ://www.euroserver project.eu/. Green computing node for european microservers.

[Inta] Intel. Options de compilation pour de la performance gcc x86 - http ://software.intel.com/en-us/blogs/2012/09/26/gcc-x86-performance-hints (visited june 2015).

[Intb] Intel. Sandy bridge - http ://software.intel.com/en-us/articles/sandy-bridge.

[JLC11] Alexandra Jimborean, Vincent Loechner, and Philippe Clauss. Handling multiversioning in llvm : Code tracking and cloning. In *WIR 2011 : Workshop on Intermediate Representations, in conjunction with CGO 2011*, Chamonix, France, April 2011.

[Kal15] Kalray. Mppa manycore http ://www.kalrayinc.com/img/pdf/flyer_mppa_manycore.pdf (visited june 2015), 2015.

[KFA+07] Michael Keating, David Flynn, Rob Aitken, Alan Gibbons, and Kaijian Shi. *Low Power Methodology Manual : For System-on-Chip Design.* Springer Publishing Company, Incorporated, 2007.

[KFJ+03] R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, and D.M. Tullsen. Single-isa heterogeneous multi-core architectures : the potential for processor power reduction. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 81 – 92, dec. 2003.

[KJT04] Rakesh Kumar, Norman P. Jouppi, and Dean M. Tullsen. Conjoined-core chip multiprocessing. In *Proceedings of the 37th annual IEEE/ACM International Sym-*

posium on Microarchitecture, MICRO 37, pages 195–206, Washington, DC, USA, 2004. IEEE Computer Society.

[KRH10] David Koufaty, Dheeraj Reddy, and Scott Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 125–138, New York, NY, USA, 2010. ACM.

[KTJ06] Rakesh Kumar, Dean M. Tullsen, and Norman P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 23–32, New York, NY, USA, 2006. ACM.

[KTR+04] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. *SIGARCH Comput. Archit. News*, 32(2) :64–, March 2004.

[KZT05] R. Kumar, V. Zyuban, and D.M. Tullsen. Interconnections in multi-core architectures : understanding mechanisms, overheads and scaling. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pages 408 – 419, june 2005.

[LAS+13] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. The mcpat framework for multicore and manycore architectures : Simultaneously modeling power, area, and timing. *ACM Trans. Archit. Code Optim.*, 10(1) :5 :1–5 :29, April 2013.

[LBK+10] Tong Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating system support for overlapping-isa heterogeneous multi-core architectures. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1 –12, jan. 2010.

[LBKH07] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 53 :1–53 :11, New York, NY, USA, 2007. ACM.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin : Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[Lis] The Green500 List. http ://www.green500.org/greenlists.

[LMSG02] Stan Liao, Grant Martin, Stuart Swan, and Thorsten Grötker. *System Design with SystemC*. Springer Science & Business Media, 2002.

[MGG+11] S. Maleki, Yaoqing Gao, M.J. Garzaran, T. Wong, and D.A. Padua. An evaluation of vectorizing compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 372–382, 2011.

[NDR+11] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. Vapor simd : Auto-vectorize once, run everywhere. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 151–160, Washington, DC, USA, 2011. IEEE Computer Society.

[Nvi] Nvidia. Tegra k1 - http ://www.nvidia.com/object/tegra-k1-processor.html.

[Nvi08] CUDA Nvidia. Programming guide, 2008.

[OOS+08] Kevin O'Brien, Kathryn O'Brien, Zehra Sura, Tong Chen, and Tao Zhang. Supporting openmp on cell. *International Journal of Parallel Programming*, 36(3) :289–311, 2008.

[PAS03] A. Pegatoquet, M. Auguin, and O. Sohier. Assembly code performance evaluation apparatus and method, July 22 2003. US Patent 6,598,221.

[PLDM13] Shruti Padmanabha, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. Trace based phase prediction for tightly-coupled heterogeneous cores. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 445–456. ACM, 2013.

[Pou12] Louis-Noël Pouchet. Polybench : The polyhedral benchmark suite. *URL : http ://www. cs. ucla. edu/˜ pouchet/software/polybench/[cited July,]*, 2012.

[RAKK13] Rance Rodrigues, Arunachalam Annamalai, Israel Koren, and Sandip Kundu. Improving performance per watt of asymmetric multi-core processors via online program phase classification and adaptive core morphing. *ACM Trans. Des. Autom. Electron. Syst.*, 18(1) :5 :1–5 :23, January 2013.

[RKBH11] Dheeraj Reddy, David Koufaty, Paul Brett, and Scott Hahn. Bridging functional heterogeneity in multicore architectures. *SIGOPS Oper. Syst. Rev.*, 45(1) :21–33, February 2011.

[RKK14] R. Rodrigues, I. Koren, and S. Kundu. Performance and power benefits of sharing execution units between a high performance core and a low power core. In *VLSI Design and 2014 13th International Conference on Embedded Systems, 2014 27th International Conference on*, pages 204–209, Jan 2014.

[Sez11] André Seznec. A new case for the tage branch predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 117–127, New York, NY, USA, 2011. ACM.

[SGS10] John E Stone, David Gohara, and Guochun Shi. Opencl : A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3) :66–73, 2010.

[SIZI11] Sadagopan Srinivasan, Ravishankar Iyer, Li Zhao, and Ramesh Illikkal. Heteroscouts : hardware assist for os scheduling in heterogeneous cmps. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '11, pages 149–150, New York, NY, USA, 2011. ACM.

[SJSM96] André Seznec, Stéphan Jourdan, Pascal Sainrat, and Pierre Michaud. Multiple-block ahead branch predictors. *SIGOPS Oper. Syst. Rev.*, 30(5) :116–127, September 1996.

[SKR07] Tyler Sondag, Viswanath Krishnamurthy, and Hridesh Rajan. Predictive thread-to-core assignment on a heterogeneous multi-core processor. In *Proceedings of the 4th Workshop on Programming Languages and Operating Systems*, PLOS '07, pages 7 :1–7 :5, New York, NY, USA, 2007. ACM.

[Smi81] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, pages 135–148, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[SP09] Hao Shen and Frédéric Pétrot. Novel task migration framework on configurable heterogeneous mpsoc platforms. In *Proceedings of the 2009 Asia and South Pacific*

Design Automation Conference, ASP-DAC '09, pages 733–738, Piscataway, NJ, USA, 2009. IEEE Press.

[SR11] T. Sondag and H. Rajan. Phase-based tuning for better utilization of performance-asymmetric multicore processors. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pages 11–20, 2011.

[SRE12] Caroline A Schneider, Wayne S Rasband, and Kevin W Eliceiri. Nih image to imagej : 25 years of image analysis. *Nature methods*, 9(7) :671–675, 2012.

[SSAJ+09] Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. Hass : a scheduler for heterogeneous multicore systems. *SIGOPS Oper. Syst. Rev.*, 43(2) :66–75, April 2009.

[SSFP11] Juan Carlos Saez, Daniel Shelepov, Alexandra Fedorova, and Manuel Prieto. Leveraging workload diversity through os scheduling to maximize performance on single-isa heterogeneous multicore systems. *Journal of Parallel and Distributed Computing*, 71(1) :114 – 131, 2011.

[SSG+14] Tanguy Sassolas, Chiara Sandionigi, Alexandre Guerre, Alexandre Aminot, Pascal Vivet, Hela Boussetta, Luca Ferro, and Nicolas Peltier. Early design stage thermal evaluation and mitigation : The locomotiv architectural case. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 314 :1–314 :2, 3001 Leuven, Belgium, Belgium, 2014. European Design and Automation Association.

[SSKLK13] Dave Shreiner, Graham Sellers, John M Kessenich, and Bill M Licea-Kane. *OpenGL programming guide : The Official guide to learning OpenGL, version 4.3*. Addison-Wesley, 2013.

[Sta] OpenAcc Standard. Directives for accelerators http ://www.openacc-standard.org/.

[SZD04] Xipeng Shen, Yutao Zhong, and Chen Ding. Locality phase prediction. *SIGOPS Oper. Syst. Rev.*, 38(5) :165–176, October 2004.

[THI] THINGS2DO. European eniac ju project - thin but great silicon 2 design objects http ://things2do.space.com.ro/.

[VAJ+09] S.K. Venkata, I. Ahn, Donghwan Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M.B. Taylor. Sd-vbs : The san diego vision benchmark suite. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 55–64, Oct 2009.

[Var11] SMP Variable. A multi-core cpu architecture for low power and high performance. *Whitepaper-http ://www. nvidia. com*, 2011.

[VCJE+12] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). *SIGARCH Comput. Archit. News*, 40(3) :213–224, June 2012.

[Ver] Verilog. Website, http ://www.verilog.com.

[Wal04] John Walker. fbench : Floating point benchmarks. *URL : https ://www.fourmilab.ch/fbench/[cited July,]*, 2004.

[WCC+07] Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. Exochi : architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 156–166, New York, NY, USA, 2007. ACM.

[Xu10] Yu Xu. Mp3 audio decoder with thea rm neon multimedia, January 2010. http ://www.embedded.com/design/mcus-processors-and-socs/4008872/1/PRODUCT-HOW-TO-Implement-an-MP3-audio-decoder-with-the-ARM-Neon-Multimedia-extensions - available on April 2014.

Annexe C

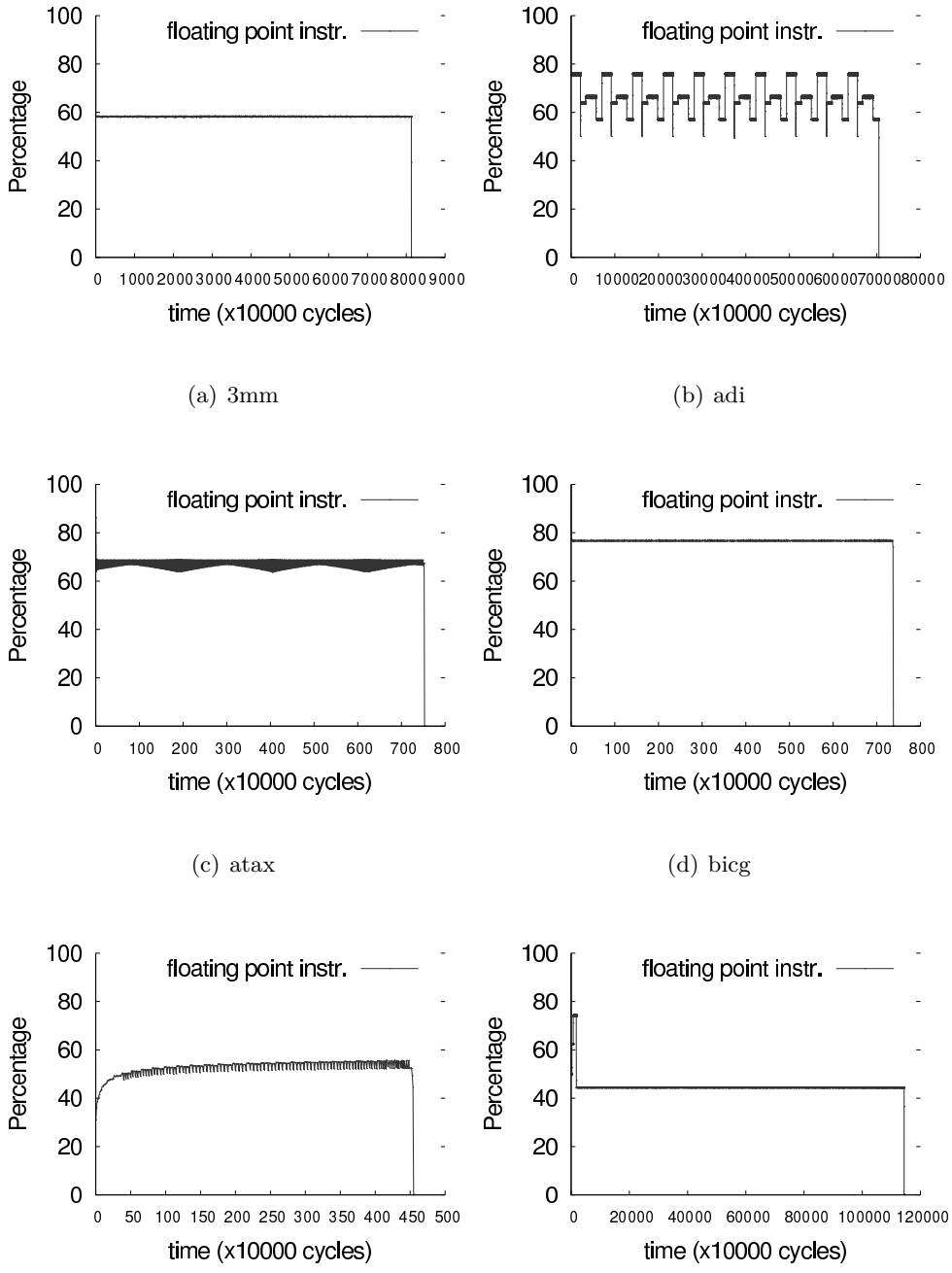
Profils d'utilisation de la FPU

Sommaire

Cette annexe présente les profils du pourcentage d'utilisation pour les benchmarks. Les détails à propos de la méthode d'expérimentations et des suites d'applications sont décrits Chapitre 3.

C.1 Suite Polybench

Les profils de la suite *Polybench* sont présentés figures C.1, C.2, C.3, C.4 et C.5.



(a) 3mm

(b) adi

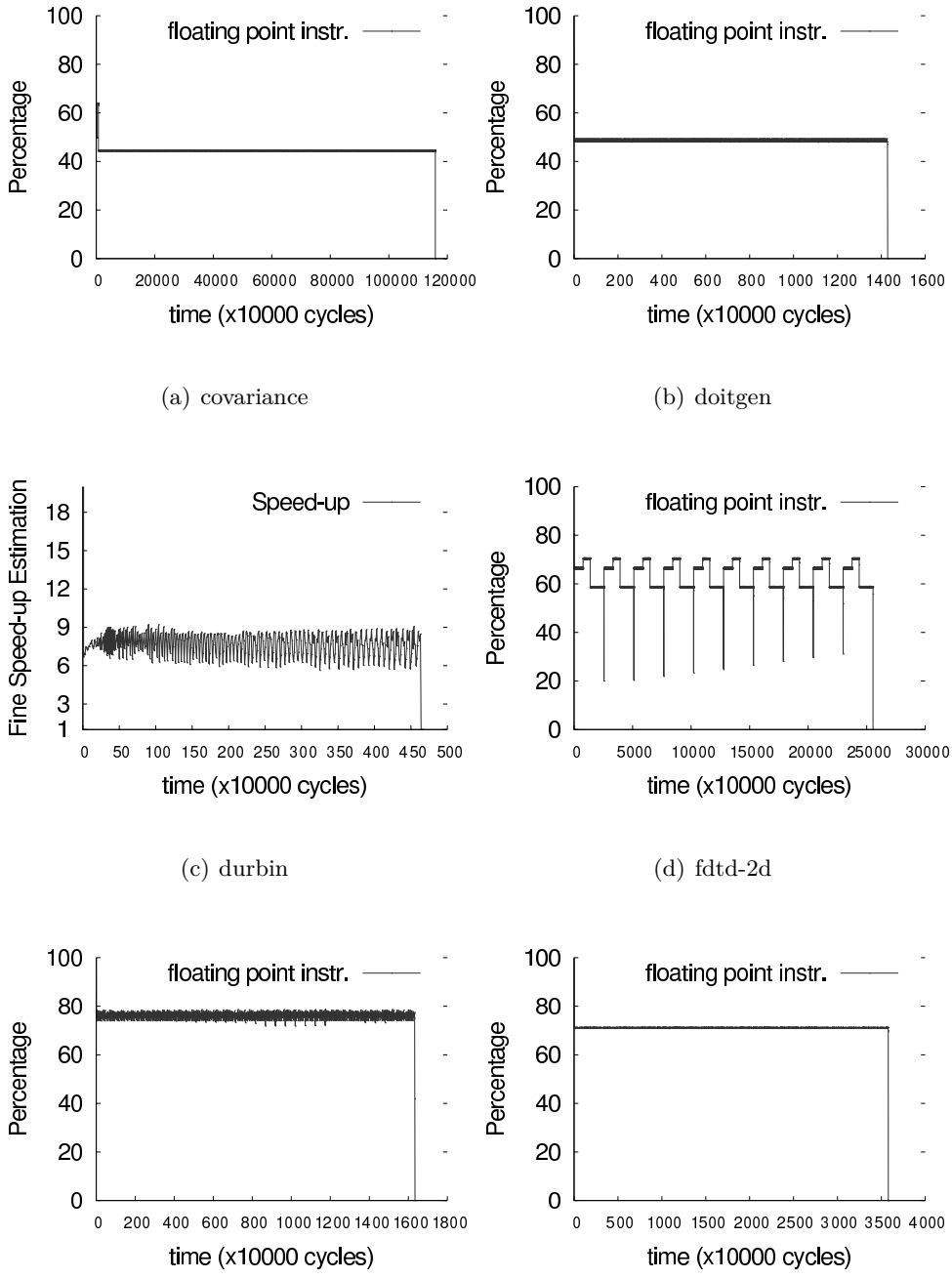
(c) atax

(d) bicg

(e) cholesky

(f) correlation

FIGURE C.1 – Profils Polybench - Utilisation de l'unité flottante (FPU) de ARM pendant l'exécution. Chaque point du graphique est une moyenne sur 10K cycles. Il y a des phases de programmes distinctes pour l'utilisation de la FPU.



(a) covariance

(b) doitgen

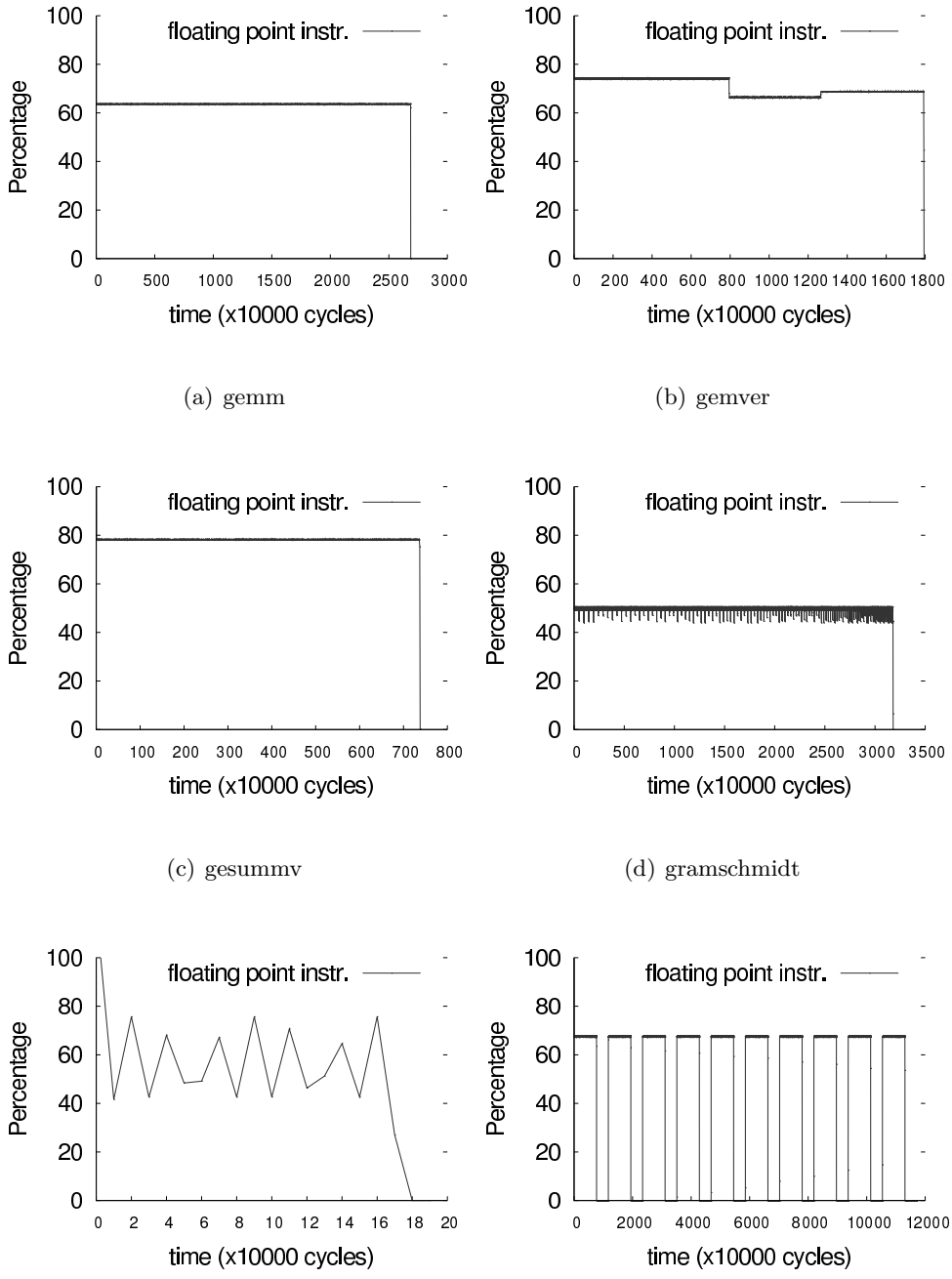
(c) durbin

(d) fdtd-2d

(e) fdtd-apml

(f) floyd-warshall

FIGURE C.2 – Profils Polybench - Utilisation de l'unité flottante (FPU) de ARM pendant l'exécution. Chaque point du graphique est une moyenne sur 10K cycles. Il y a des phases de programmes distinctes pour l'utilisation de la FPU.



(a) gemm

(b) gemver

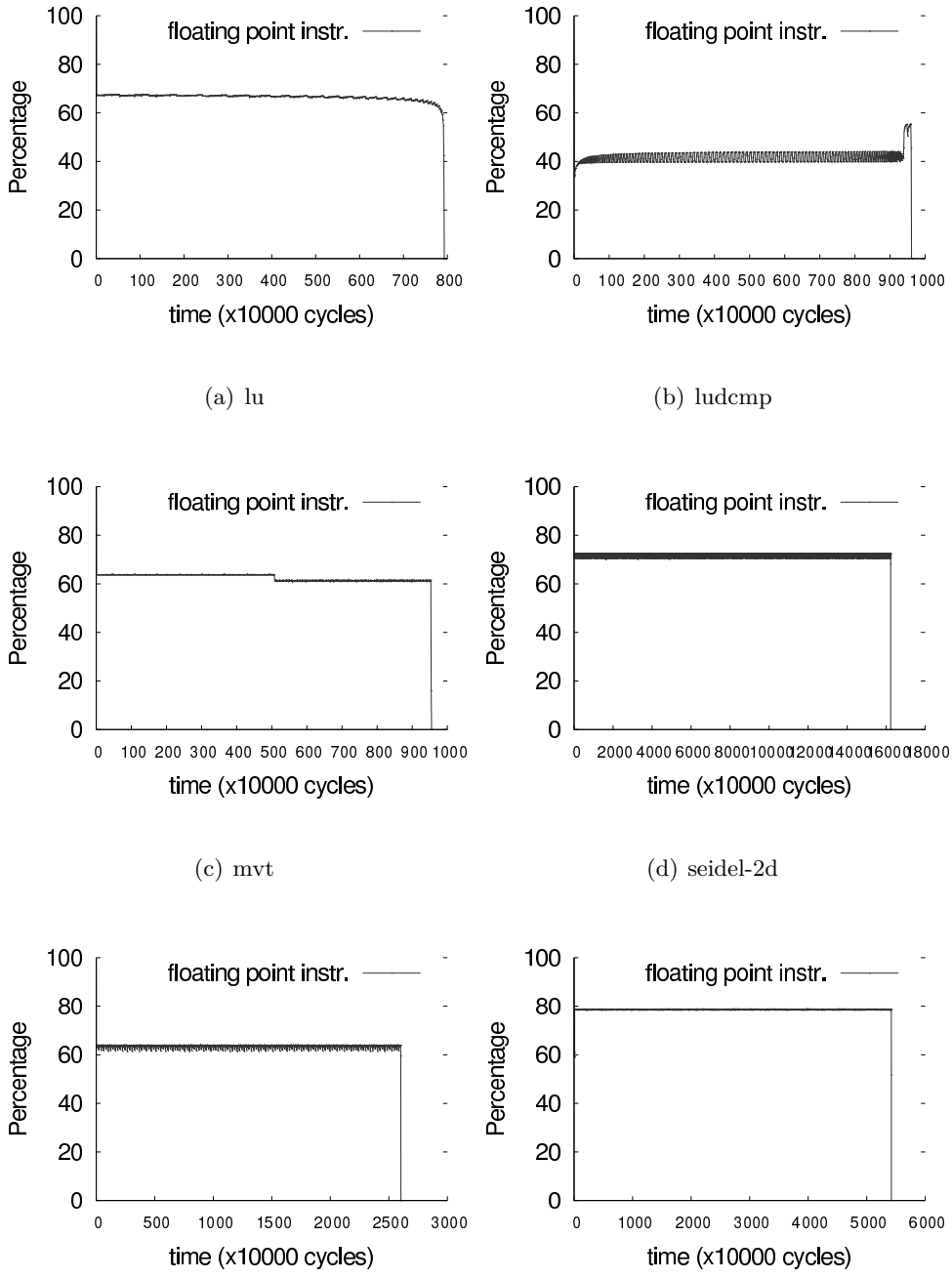
(c) gesummv

(d) gramschmidt

(e) jacobi-1d-imper

(f) jacobi-2d-imper

FIGURE C.3 – Profils Polybench - Utilisation de l'unité flottante (FPU) de ARM pendant l'exécution. Chaque point du graphique est une moyenne sur 10K cycles. Il y a des phases de programmes distinctes pour l'utilisation de la FPU.



(a) lu

(b) ludcmp

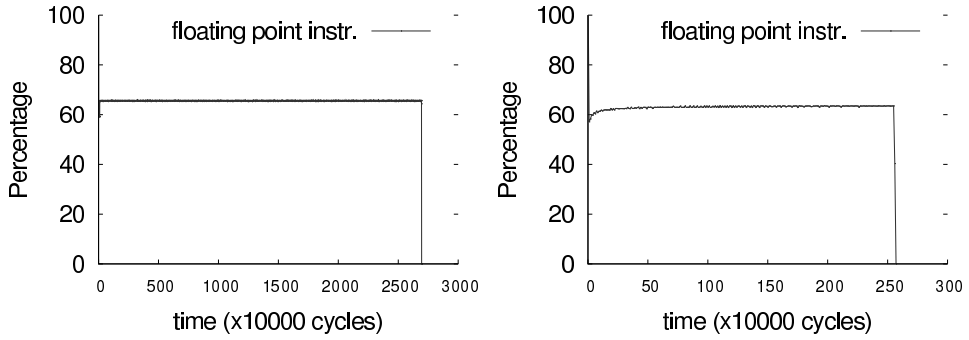
(c) mvt

(d) seidel-2d

(e) symm

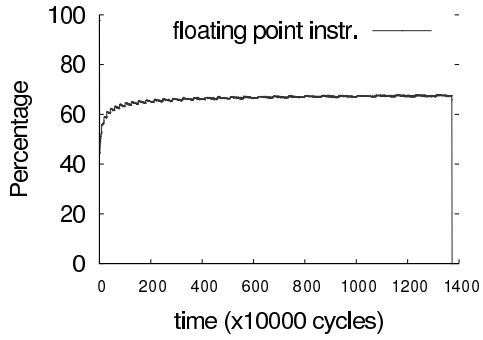
(f) syr2k

FIGURE C.4 – Profils - Utilisation de l'unité flottante (FPU) de ARM pendant l'exécution. Chaque point du graphique est une moyenne sur 10K cycles. Il y a des phases de programmes distinctes pour l'utilisation de la FPU.



(a) syrk

(b) trisolv



(c) trmm

FIGURE C.5 – Profils Polybench - Utilisation de l'unité flottante (FPU) de ARM pendant l'exécution. Chaque point du graphique est une moyenne sur 10K cycles. Il y a des phases de programmes distinctes pour l'utilisation de la FPU.

C.2 Suite SDVBS

Les profils de la suite *SDVBS* sont présentés figure C.6.

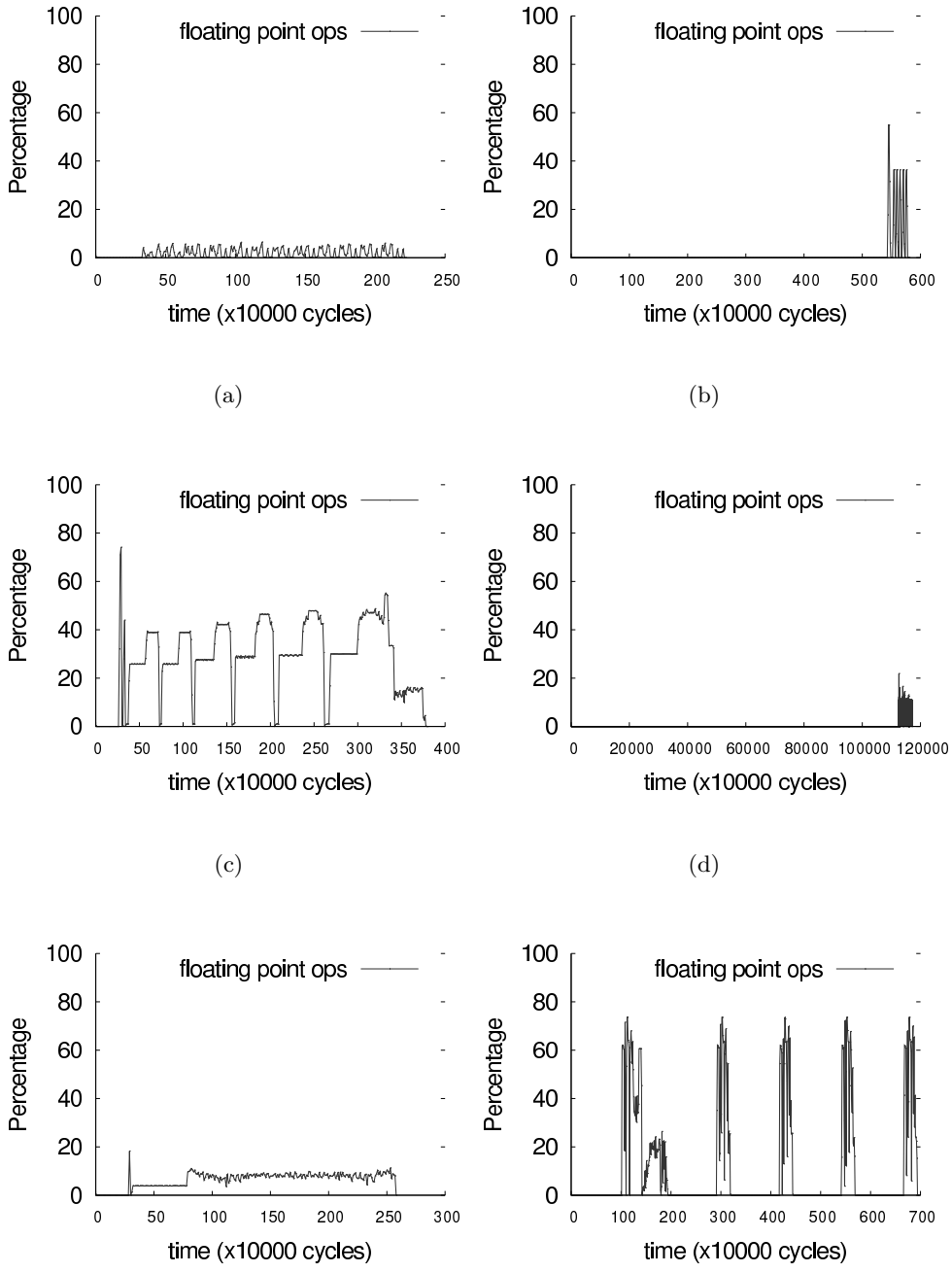
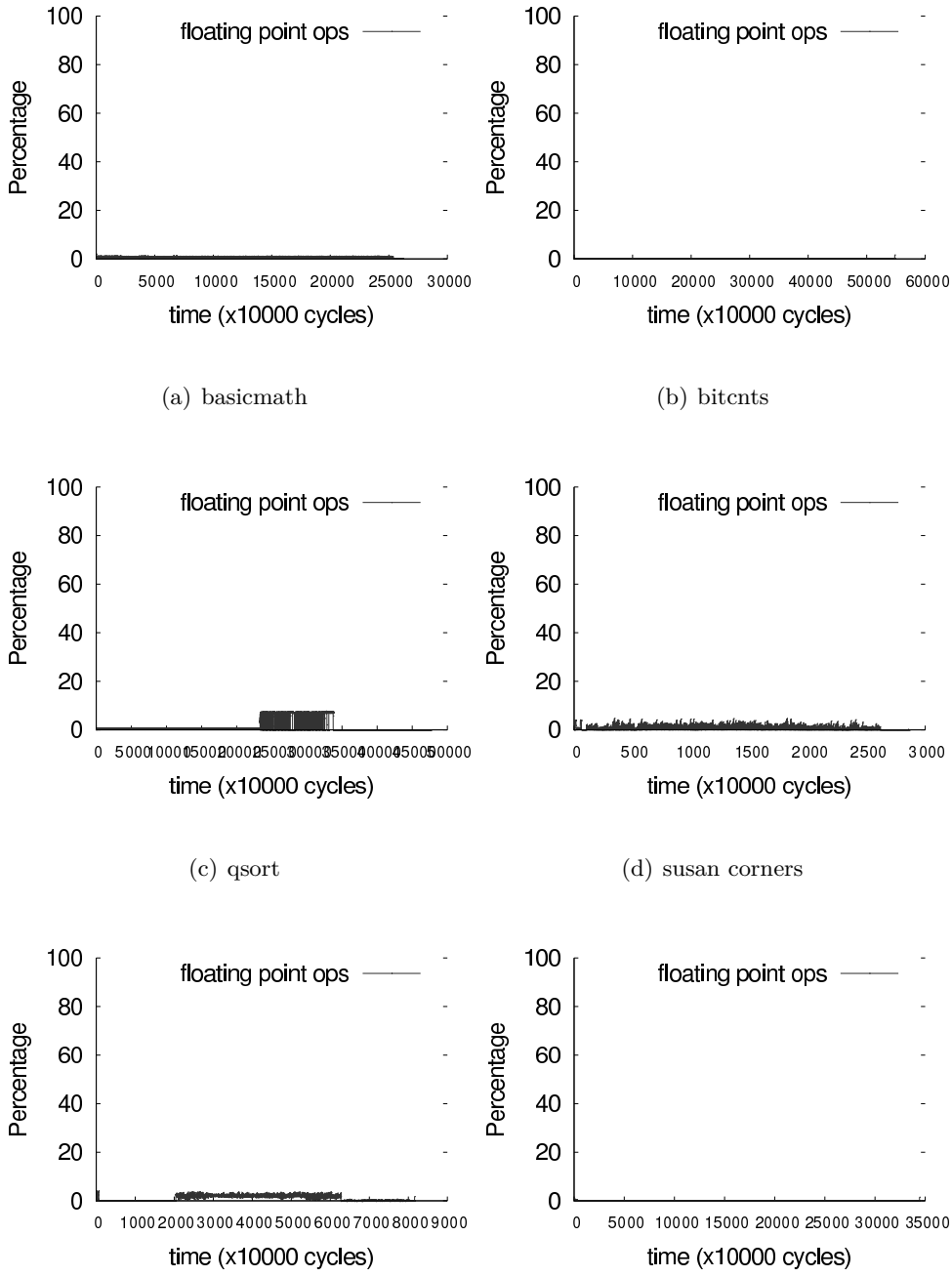


FIGURE C.6 – Profils SDVBS - Utilisation de l'unité flottante (FPU) de ARM pendant l'exécution. Chaque point du graphique est une moyenne sur 10K cycles. Il y a des phases de programmes distinctes pour l'utilisation de la FPU.

C.3 Suite Mibench

Les profils de la suite *Mibench* sont présentés figures C.7 et C.8.



(a) basicmath

(b) bitcnts

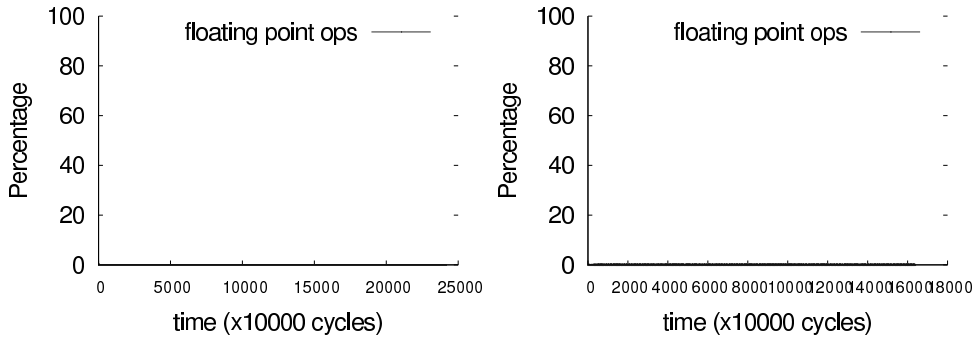
(c) qsort

(d) susan corners

(e) susan edges

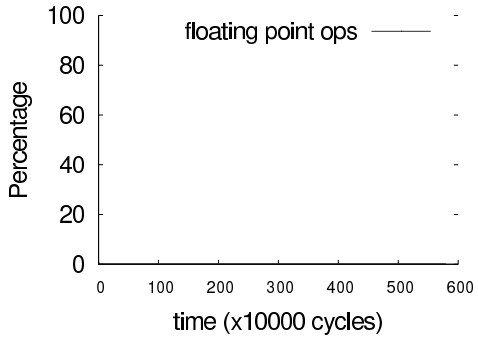
(f) susan smoothing

FIGURE C.7 – Profils Mibench - Utilisation de l'unité flottante (FPU) de ARM pendant l'exécution. Chaque point du graphique est une moyenne sur 10K cycles. Il y a des phases de programmes distinctes pour l'utilisation de la FPU.

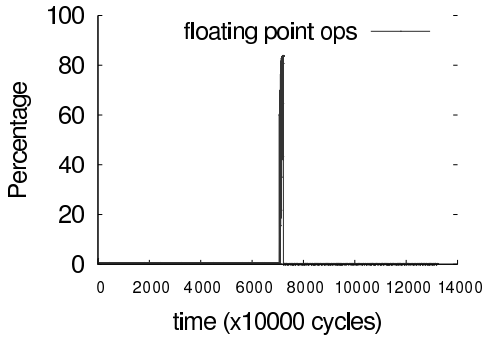


(a) dijkstra

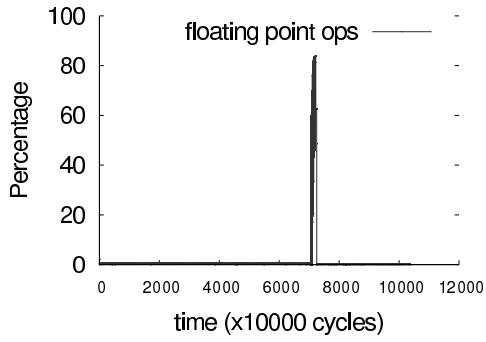
(b) patricia



(c) search



(d) FFT



(e) FFT

FIGURE C.8 – Profils Mibench - Utilisation de l'unité flottante (FPU) de ARM pendant l'exécution. Chaque point du graphique est une moyenne sur 10K cycles. Il y a des phases de programmes distinctes pour l'utilisation de la FPU.