Méthodologie d'identification et d'évitement des cycles de gel du processeur pour l'optimisation de la performance du logiciel sur le matériel

Perrin Njoyah Ntafam

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTE UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Perrin NJOYAH NTAFAM

Thèse dirigée par **Frédéric PÉTROT**
Et co-encadrée par **Alain CLOUARD**

préparée au sein du **Laboratoire Techniques de l'Informatique et de la Microélectronique pour l'Architecture des systèmes intégrés**
et de **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Méthodologie d'identification et d'évitement des cycles de gel du processeur pour l'optimisation de la performance du logiciel sur le matériel

Avoidance and identification methodology of processor stall cycles for software-on-hardware performance optimization

Thèse soutenue publiquement le **20 Avril 2018**,
devant le jury composé de :

Pr. Frédéric ROUSSEAU
Professeur, Université Grenoble Alpes, TIMA, France, Président
Pr. Pascal SAINRAT
Professeur, Université Paul Sabatier, IRIT, France, Rapporteur
Pr. Tanguy RISSET
Professeur, INSA–Lyon, CITI, France, Rapporteur
Dr. Nicolas VENTROUX
Ingénieur/chercheur HDR, CEA LIST, France, Examinateur
Pr. Frédéric PÉTROT
Professeur, Université Grenoble Alpes, TIMA, France, Directeur de thèse
Mr. Alain CLOUARD
Systems & Platforms Solutions Manager, STMicroelectronics, France, Co-Encadrant de thèse

Remerciements

C'est rempli d'émotion que je souhaite par ces quelques mots exprimer ma gratitude aux personnes et entités qui par leur soutien ont contribué de près ou de loin à l'aboutissement de ce travail :

Je remercie mon directeur de thèse, M. Frédéric PETROT, pour m'avoir accueilli au sein de son équipe. Je lui suis également reconnaissant pour le temps conséquent qu'il m'a accordé, ses qualités pédagogiques et scientifiques, sa franchise et sa sympathie. J'ai beaucoup appris à ses côtés et je lui adresse ma gratitude pour tout cela.

J'adresse également de chaleureux remerciements à mon co-encadrant de thèse chez STMicroelectronics, M. Alain CLOUARD, pour son attention de tout instant sur mes travaux, d'avoir faciliter mon intégration dans son équipe de R&D et de m'avoir donné l'opportunité de réaliser cette thèse. Je remercie également M. Eric PAIRE pour ses conseils avisés, son écoute et son implication qui ont été prépondérants pour la bonne réussite de cette thèse. Vos énergies et la confiance que vous m'avez accordée ont été des éléments moteurs pour moi. J'ai pris un grand plaisir à travailler avec vous.

Je tiens à remercier les rapporteurs de cette thèse M. Pascal SAINRAT, et M. Tanguy RISSET, d'avoir pris le temps d'évaluer ce travail, pour leur remarques et suggestions. J'associe à ces remerciements M. Frédéric ROUSSEAU, qui a présidé la soutenance de cette thèse et M. Nicolas VENTROUX, pour leur disponibilité et pour avoir accepté d'examiner mon travail.

Mes sincères remerciements aux membres de l'équipe Special Project (« ex-SPG ») avec qui j'ai collaboré et échangé durant mes années de doctorat. Merci d'avoir faciliter mon intégration, pour les discussions qui ont pu enrichir ma culture scientifique et pour les activités effectuées ensemble (Raclette, Barbecues, Karting, Resto pédagogique, foot, etc).

Au terme de ce parcours, je remercie enfin celles et ceux qui me sont chers et que j'ai quelque peu délaissés ces derniers mois pour achever cette thèse. Leurs attentions et encouragements m'ont accompagnée tout au long de ces années. Je suis redevable à mes parents, Jeannette ("le doc") et Dieudonné, pour leur soutien moral et matériel sans failles et leur confiance indéfectible dans mes choix. Mes frères et soeurs, Roger, Audrey, Aurel, et Carnot, qui ont toujours trouvé les mots justes pour m'encourager et me faire comprendre que je pouvais y parvenir ("futur docta"). J'ai une pensée toute particulière pour mon papa, Dieudonné NTAFAM qui nous a quitté juste au moment où je débutais la troisième année de cette

thèse, à qui je dédie ce travail. Papa j'aurais vraiment aimé et voulu que tu sois là pour me voir réaliser et franchir cette étape importante de ma vie car tu méritais cela. Enfin, à ma copine Rebecca qui m'a toujours soutenu et qui me supporte depuis déjà quelques années. Merci encore d'être toujours là (la Schminov) ! Je n'oublierai pas mes amis que j'ai rencontrés et qui m'ont également apporté leur soutien. Merci à vous !

« *Ad astra per apera* »

« *Sic itur ad astra* »

Table des matières

Liste des figures

Liste des tableaux

Chapitre 1

Introduction

Sommaire

1.1 Contexte général des travaux

L'électronique enfouie dans les équipements grand public et industriels rend des services d'une complexité inimaginable il y a encore quelques années, à l'aide de systèmes dont le coût doit être maîtrisé et dont l'évolution doit être possible, par l'utilisation conjointe de matériel et de logiciel. Dans ce contexte, ll est nécessaire de trouver des méthodologies qui permettent d'évaluer la bonne intégration logiciel/matériel, c'est–à–dire une utilisation optimale des capacités du matériel par le logiciel en développement, une fois que l'ossature [1] de l'architecture matériel est figée dans le processus de développement d'un système sur puce.

Pour fournir au développeur logiciel une simulation du SoC en amont et observable, une modélisation utilisant un niveau d'abstraction adéquat pour améliorer la compréhension et l'utilisation des capacités d'un système sur puce (Electronic System Level (ESL)) est employée. Cette simulation est dans de nombreux cas développée en C++ en utilisant la bibliothèque de modélisation SystemC (IEEE1666, http ://accellera.org/) qui est largement connue comme une des techniques de conception au niveau système (CLOUARD et collab. [2003]). Les plateformes virtuelles sont conventionnellement utilisées pour du prototypage virtuel, l'exploration d'architectures, la vérification au niveau Register Transfert Level (RTL) en fournissant des bancs de tests et des modèles de références, ou encore pour le développement pré–silicium du logiciel. Il existe ainsi plusieurs solutions d'automatisation de la conception électronique (Electronic Design Automation (EDA)) pour le développement des plateformes virtuelles, ainsi que des standards tels que présentés par GERSTLAUER et SCHIRNER [2010] et des méthodes de modélisation présentées et développées, notamment par PETROT et collab. [2011]. Estimer précisément la performance du logiciel nécessite l'utilisation de modèles précis. Ces modèles obtenus à partir du RTL peuvent être disponibles assez tôt dans le processus de conception d'un SoC, en utilisant des éléments de bibliothèque (processeur, mémoire, ...) et des éléments configurés à l'aide d'outils (interconnexion ou bus).

1.2 Problématique globale

Les plateformes virtuelles de simulation précises au niveau du cycle (Précise au niveau du cycle –ou *Cycle Accurate* (CA)) sont utilisées pour effectuer la phase d'étude d'architecture pré–silicium du matériel, ainsi que pour les estimations du temps d'exécution du logiciel. Il existe des outils qui se focalisent sur l'architecture du processeur et qui permettent d'effectuer l'estimation de la performance du logiciel (MARTIN [2008], LEUPERS et collab. [2011],

1. Ossature = backbone

SANCHEZ et KOZYRAKIS [2013]).

Il existe quelques outils qui permettent d'estimer la performance du logiciel (ANDERSON et collab. [1997]). Cependant ces outils ne proposent pas une approche documentée et facile à mettre en œuvre pour l'estimation et l'optimisation, suffisamment tôt (une fois l'ossature de l'architecture du futur SoC figée) de la performance du logiciel sur le matériel en s'appuyant sur des plateformes virtuelles SoC précises au niveau du cycle.

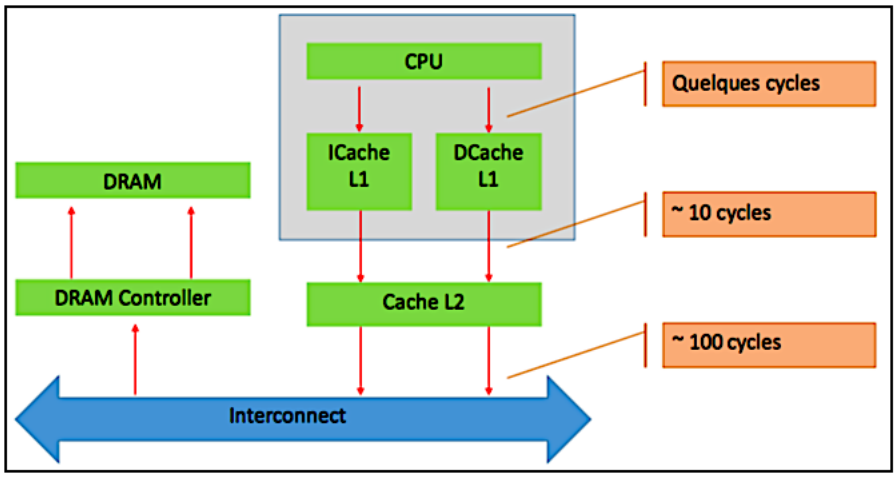

FIGURE 1.1: Hiérarchie mémoire d'un SoC simple ou un sous–système d'un SoC complexe

Par ailleurs, la figure 1.1 présente une approximation de l'ordre de grandeur de la latence (cycles) entre les différents niveaux de la hiérarchie mémoire dans un SoC simple ou un sous–système d'un SoC complexe. On observe évidemment qu'entre le processeur et le cache L_1, et entre le cache L_1 et le cache L_2, les latences sont assez faibles. En revanche, la latence est assez élevée entre le dernier niveau de cache et la mémoire ($\approx$ 100 cycles). Ainsi, en cas d'échec de lecture d'une donnée dans cache L2, la requête de lecture de la donnée est transférée au contrôleur mémoire. Cette requête mettra au minimum 100 cycles (en fonction de la charge de transfert et la latence du bus) pour que la ligne de cache contenant la donnée lue soit copiée en mémoire cache (L_1 et/ou L_2 en fonction des architectures). Dans cette situation, on se rend compte que le problème d'accès à la mémoire est crucial. Cependant l'on y est contraint puisque le matériel est fixe c'est–à–dire que les marges de manœuvre sur l'évolution possible du matériel sont réduites. L'absence d'une donnée à un instant dans les caches de la hiérarchie mémoire peut ainsi empêcher le logiciel de s'exécuter rapidement.

En effet, la définition des métriques pertinentes et la façon dont un développeur logiciel pourrait les utiliser pour optimiser la performance de son logiciel (par exemple en terme de temps d'exécution ou de cycles de gel du processeur générés) sont considérées aujourd'hui comme un acquis, alors qu'en réalité, cela demeure presque un savoir–faire artisanal. En

général, pour l'optimisation du logiciel, les développeurs logiciel comptent sur les options d'optimisation du compilateur et sur des programmes soigneusement développés.

Concernant les programmes soigneusement développés, un ingénieur d'intégration logiciel/matériel n'est pas toujours par exemple un expert en rendu audio-video, en vision par ordinateur, ... Par ailleurs, les experts ne sont pas souvent à la disposition de l'ingénieur d'intégration pour un travail commun. L'ingénieur en charge de l'intégration a donc besoin d'une méthodologie et d'outils pour son travail d'optimisation de la performance du logiciel.

En ce qui concerne les compilateurs, il est possible aujourd'hui de leur fournir typiquement en option de compilation des tailles de caches, la profondeur du tampon d'écriture, la latence d'accès à la mémoire principale et la bande passante, la latence de pré–chargement et le nombre de pré–chargements simultanés autorisés, ...

Cependant, le compilateur n'a qu'une idée abstraite (résumée sous forme de fonctions analytiques de coûts) du comportement de l'architecture. Même si cela est approprié pour certaines fonctionnalités architecturales telles que les caches, cela ne garantit pas qu'une décision d'optimisation produira un gain, par exemple en terme de temps d'exécution. Pour confirmer cela, la documentation (GCC-OPTIONS [2017]) du compilateur GNU Compiler Collection (GCC) sur les différentes options de compilation qui contrôlent l'optimisation du logiciel souligne que sa stratégie intégrée de pré–chargement peut accélérer ou ralentir l'exécution d'un programme.

Enfin, du fait que les ingénieurs s'occupent très souvent de codes logiciel dont ils ne connaissent au départ ni la structure, ni le détail, ils ont besoin d'aide pour localiser précisément les endroits dans lesquels le logiciel perd du temps à cause de comportements difficiles à évaluer avant l'intégration logiciel/matériel.

1.3 Plan de la thèse

Le reste du document est organisé comme suit :

- Le chapitre 2 (*Problématique*) présente la problématique de manière détaillée et permet de circonscrire concrètement le problème auquel nous voulons apporter une solution dans cette thèse. Ce problème est présenté sous forme de questions à la fin de ce même chapitre.
- Le chapitre 3 (*État de l'art*) présente un état de l'art sur les techniques utilisées aujourd'hui pour résoudre la problématique. C'est–à–dire les techniques d'identification et d'évitement des cycles de gel du processeur afin de faciliter l'optimisation de la performance du logiciel sur la puce.

- Le chapitre 4 (*Exploitation des modèles précis au niveau du cycle pour l'identification et la correction des gels du processeur*) présente la première approche empirique que nous proposons pour résoudre non-intuitivement le phénomène de gel de processeur dû à l'absence de données en mémoire cache.
- Le chapitre 5 (*Exploitation de traces d'exécution pour l'identification et la quantification des gels du processeur*) présente une approche qui s'appuit sur les traces du bus et des différentes valeurs du registre PC (program Counter) pour identifier les instructions du logiciel gelées et quantifier leurs contributions au rallongement du temps global d'exécution du logiciel.
- Le chapitre 6 (*Expérimentation*) détaille les résultats obtenus suite à l'expérimentation des deux approches proposées.
- Le dernier chapitre 7 (*Conclusion générale et perspectives*) conclut le document par une synthèse des deux approches et ouvre le débat sur d'autres pistes de recherche non couvertes par nos travaux.

1.4 Références

ANDERSON, J. M., L. M. BERC, J. DEAN, S. GHEMAWAT, M. R. HENZINGER, S.-T. A. LEUNG, R. L. SITES, M. T. VANDEVOORDE, C. A. WALDSPURGER et W. E. WEIHL. 1997, «Continuous profiling : Where have all the cycles gone ?», *ACM Trans. Comput. Syst.*, vol. 15, n° 4, doi :10.1145/265924.265925, p. 357–390, ISSN 0734-2071. URL http://doi.acm.org/10.1145/265924.265925. 3

CLOUARD, A., K. JAIN, F. GHENASSIA, L. MAILLET-CONTOZ et J.-P. STRASSEN. 2003, «Systemc», chap. Using Transactional Level Models in a SoC Design Flow, Kluwer Academic Publishers, Norwell, MA, USA, ISBN 1-4020-7479-4, p. 29–63. URL http://dl.acm.org/citation.cfm?id=886362.886366. 2

GCC-OPTIONS. 2017, «Options that control optimization», https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. [En ligne ; dernier accès le 03 Octobre 2017]. 4

GERSTLAUER, A. et G. SCHIRNER. 2010, «Platform modeling for exploration and synthesis», dans *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, ISSN 2153-6961, p. 725–731, doi :10.1109/ASPDAC.2010.5419794. 2

LEUPERS, R., L. EECKHOUT, G. MARTIN, F. SCHIRRMEISTER, N. TOPHAM et X. CHEN. 2011, «Virtual manycore platforms : Moving towards 100+ processor cores», dans *2011 Design, Automation Test in Europe*, ISSN 1530-1591, p. 1–6, doi :10.1109/DATE.2011.5763121. 2

MARTIN, G. 2008, «Multi-processor soc-based design methodologies using configurable and extensible processors», *J. Signal Process. Syst.*, vol. 53, nº 1-2, doi :10.1007/s11265-007-0153-7, p. 113–127, ISSN 1939-8018. URL http://dx.doi.org/10.1007/s11265-007-0153-7. 2

PETROT, F., N. FOURNEL, P. GERIN, M. GLIGOR, M. M. HAMAYUN et H. SHEN. 2011, «On mpsoc software execution at the transaction level», *IEEE Design Test of Computers*, vol. 28, nº 3, doi :10.1109/MDT.2010.118, p. 32–43, ISSN 0740-7475. 2

SANCHEZ, D. et C. KOZYRAKIS. 2013, «Zsim : Fast and accurate microarchitectural simulation of thousand-core systems», dans *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, ACM, New York, NY, USA, ISBN 978-1-4503-2079-5, p. 475–486, doi :10.1145/2485922.2485963. URL http://doi.acm.org/10.1145/2485922.2485963. 3

Chapitre 2

Problématique

Sommaire

2.1 Introduction

Un SoC est une combinaison de logiciel et de matériel interagissant dans le but de réaliser un ensemble de fonctions spécifiques et bien définies pour des domaines variés tels que l'automobile, l'Internet des objets –ou *Internet of Things* (IoT) ou encore les téléphones mobiles. Les fonctionnalités de ces systèmes doivent être fournies avec un niveau de performance satisfaisante. Cependant, avec l'augmentation de la complexité de l'architecture des SoCs et de leurs capacités de traitement, il devient important de pouvoir concevoir des stratégies efficaces et rapides qui permettent au logiciel de mieux utiliser ces capacités disponibles.

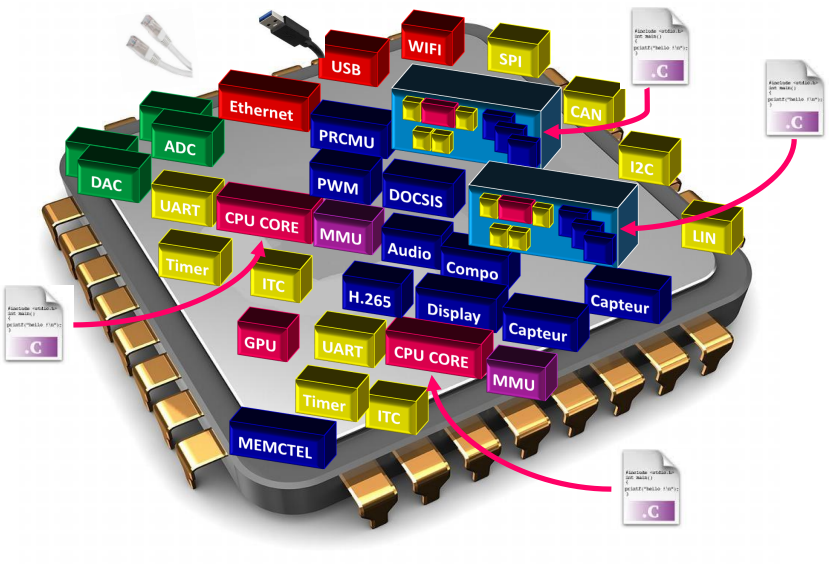

FIGURE 2.1: Exemple de systèmes sur puce complexes

La figure 2.1 présente un exemple de SoC, intégrant quelques modules et sous-systèmes ou blocs matériels, également appelés Intellectual Property (IP), que l'on peut retrouver aujourd'hui dans de tels systèmes. Elle donne aussi un aperçu de la complexité atteinte par ces produits.

Au cours de l'exécution d'une application sur un tel système, les requêtes adressées au contrôleur mémoire suite à l'indisponibilité de la donnée lue au niveau des caches ($L_1 \ldots L_n$), ne peuvent être servies immédiatement en un seul cycle. Ceci du fait de la différence significative de puissance de traitement existante entre les modules de calcul et le sous-système mémoire (communément appelé « memory wall problem ». KANG et collab. [2003], WULS et MCKEE [1995]). Cette perturbation résulte non seulement de la latence d'accès à travers l'architecture de l'interconnexion et le contrôleur mémoire, mais aussi de la contention causée par les requêtes simultanées d'accès à la mémoire par d'autres maîtres (aussi appelés ini-

tiateurs) du SoC. Ce cumul de sollicitations au niveau du bus (ou interconnexion) provient du contrôleur mémoire qui traite moins rapidement les requêtes ; comparé au processeur et aux autres IPs, et de la quantité de données à transmettre par intervalle de temps. Tout ceci entraîne une accumulation des transferts sur le bus, et une inefficacité de certains modules, puisqu'ils restent en attente active de ressources en provenance de la mémoire, et notamment la mémoire principale. Cette attente se matérialise principalement par des cycles de gel de données ou d'instructions au niveau du processeur dont les durées s'accentuent avec le temps.

Dans le cadre de cette thèse, nous nous intéressons principalement à identifier des moyens permettant de réduire l'inefficacité causée par les gels du processeur dus à la latence d'accès aux données sur le chemin mémoire.

2.2 Cause des cycles de gel du processeur

Dans un processeur à jeu d'instructions réduit Reduced Instruction Set Computing (RISC) BLEM et collab. [2013] pipeliné moderne, bon nombre d'instructions du jeu d'instructions s'exécutent en environ un cycle. Cependant, certaines instructions sont parfois contraintes de s'exécuter après une attente de plusieurs cycles. Ceci notamment du fait des latences d'accès à la mémoire à travers les différents niveaux de la hiérarchie mémoire. Cet effet est accentué en présence de plusieurs initiateurs en concurrence pour l'accès à cette ressource.

Par ailleurs, dans un processeur qui exécute les instructions d'un programme dans l'ordre (« In-Order »), ces cycles d'exécution supplémentaires peuvent provenir de causes variées au sein du pipeline. Par exemple une mauvaise prédiction de branchement, une contention sur un opérateur, ... Dans cette thèse, nous ne nous intéressons pas à ces problèmes spécifiquement internes au pipeline d'instructions.

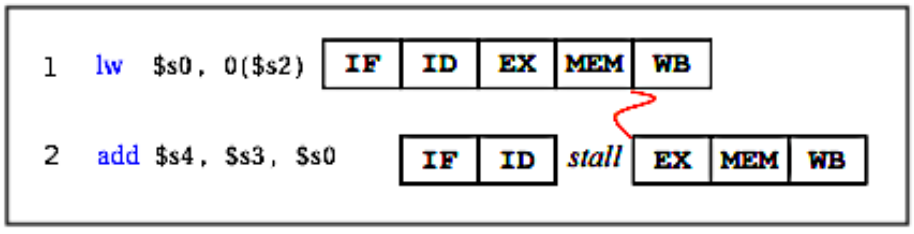

FIGURE 2.2: Exemple d'un gel du pipeline provenant de la dépendance de données entre les instructions *1←2* dans un processeur pipeline à cinq étages basique. **IF** =Instruction Fetch, **ID** = Instruction Decode, **EX** = Execute, **MEM** = Memory access, **WB** = Register Write Back.

En revanche, ils peuvent aussi provenir d'un chargement dans les registres internes du

processeur d'une valeur stockée en mémoire principale, ce qui nous intéresse ici. La donnée va prendre plusieurs cycles pour être copiée dans une ligne de cache (L_1 et / ou L_n en fonction des architectures) et être rendue disponible dans le registre cible. Dans cette situation, plusieurs cas de figure sont possibles. Dans la figure 2.2 le processeur est gelé sur une lecture en mémoire principale s'il y a un échec dans le cache L_1.

Dans le deuxième exemple un peu plus complexe, présenté dans la figure 2.3, le processeur est capable d'exécuter plusieurs instructions simultanément parmi une suite d'instructions et de détecter l'absence de dépendances entre les instructions *1, 2, 3, et 4* (processeur superscalaire). Du fait de la dépendance existante entre l'instruction *ligne 10* et l'instruction *ligne 1*, le processeur est gelé sur l'instruction (*ligne 10*) qui utilise le registre cible de la lecture en mémoire effectuée effectuée par l'instruction de la *ligne 1*.

Dans le troisième exemple, le code 2.4 qui est décrit en pseudo instructions, pour une microarchitecture sans étages de pipeline inter-bloqués *Microprocessor Without Interlocked Pipeline Stages (MIPS)*, donne un exemple de dépendances entre instructions. On suppose que la lecture effectuée à la *ligne 1* fait un échec dans les caches intégrés au processeur.

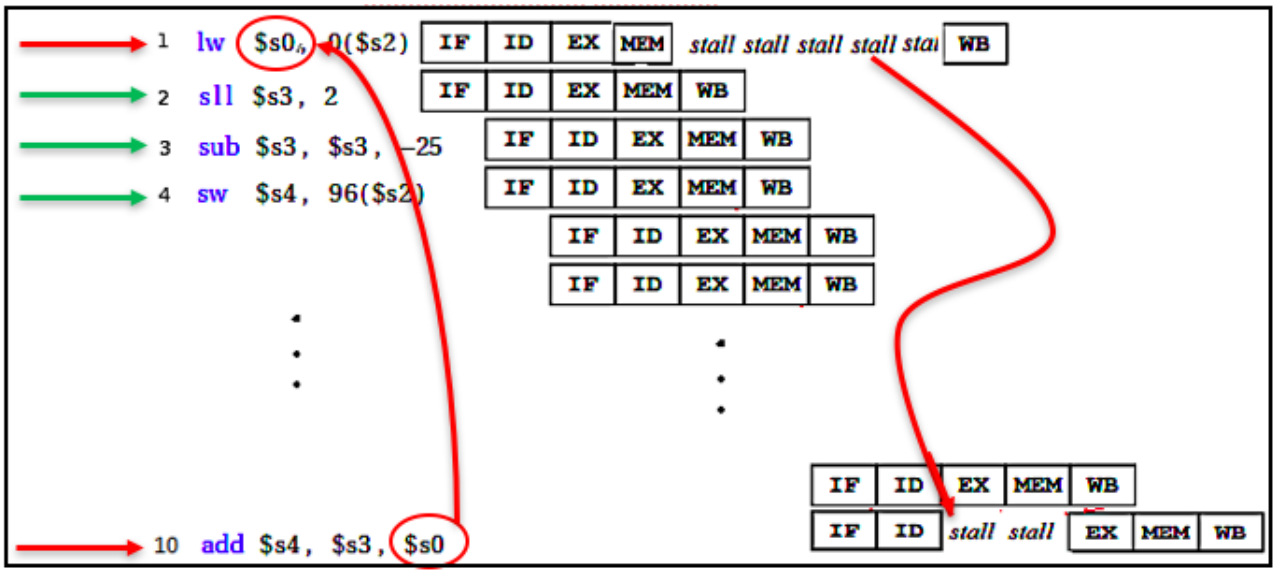

FIGURE 2.3: Exemple d'un gel du pipeline provenant de la dépendance de données entre les instructions *1*←*10* dans le code du programme dans un processeur superscalaire.

Nous notons une relation de dépendance entre deux instructions par le signe ←. Ainsi, x←y se lira : **y** dépend de **x**.

Du fait des dépendances existantes entre les instructions de lecture en mémoire principale (instructions *1*←*2* . . . *4*←*8*), les gels dus à l'absence des données sont produits à certains étages du pipeline comme le montre la figure 2.3. On remarque également que le gel du pro-

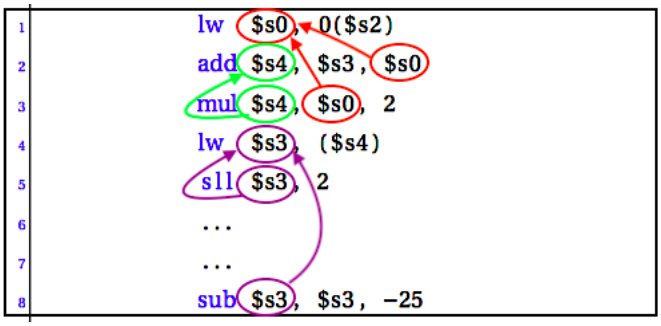

FIGURE 2.4: Pseudo code en assembleur pour architecture MIPS avec exécution pouvant produire des cycles de gel du processeur

cesseur est produit sur les instructions qui réutilisent des données lues mais qui n'ont pas encore été acheminées dans les caches du processeur.

Sans chercher à inventorier toutes les causes possibles d'un gel du processeur (Central Processing Unit (CPU)) ni en apporter une définition complète et générale pour tout type de processeur, nous nous intéressons dans cette thèse aux gels de processeurs « In-Order » de type RISC pipelinés. Et nous dirons qu'un processeur de ce type est gelé quand il est en attente active de données nécessaires pour continuer l'exécution d'un programme. Cette attente est active parce que le processeur ne fait rien d'utile pour le programme en cours d'exécution, consomme de l'énergie et produit de la chaleur pendant cette période.

Notons qu'ici le gel ne survient pas lors de l'exécution de l'instruction de chargement qui produit un échec dans le cache, mais lorsqu'une instruction tente d'accéder au registre qui est la cible du chargement et que celui-ci n'est pas encore à jour car l'accès mémoire n'est pas fini (2.3). C'est précisément un état durant lequel il ne fait rien d'utile pour le programme en cours d'exécution.

La figure 2.5 montre un exemple d'application simple s'exécutant sur une machine équipée d'un processeur « In-Order ». On suppose comme contexte initial que les caches de données sont vides. On remarque que lorsque le processeur émet une requête de lecture d'une donnée en mémoire principale, un échec se produit dans le cache, et le CPU est obligé d'attendre la fin de l'étape de chargement de celle-ci et qu'elle soit disponible avant d'adresser une nouvelle requête. Dans cette situation (caches vides, processus séquentiels, ...), on constate que ces cycles de gel rallongent le temps d'exécution de l'application et donc qu'il serait utile de détecter ce phénomène et de trouver un moyen de gagner des cycles.

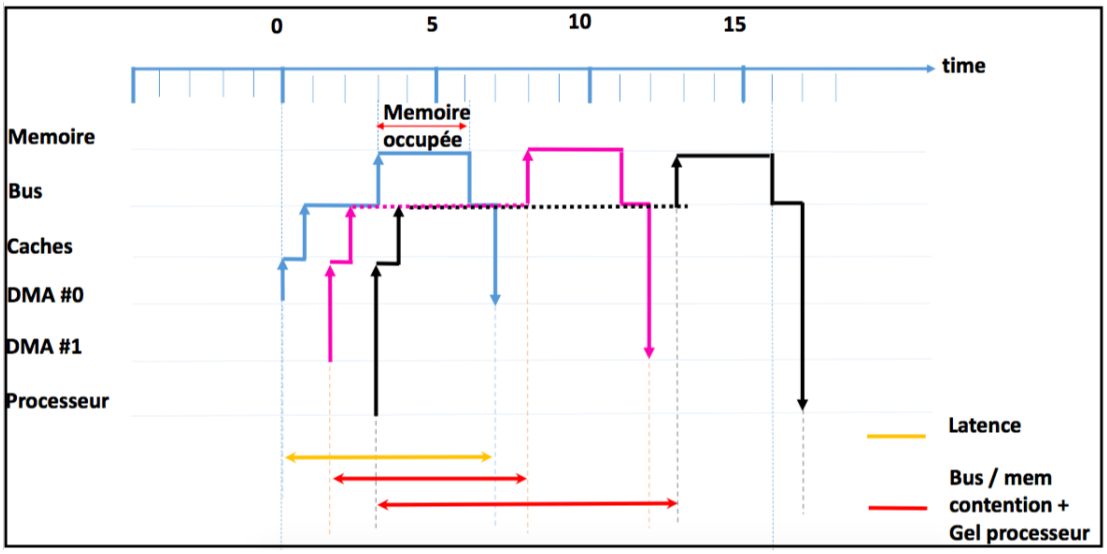

FIGURE 2.5: Gel d'un processeur « In-Order » dû à l'absence de données (caches vides, processus séquentiels) qui produit une latence importante

2.3 Impact d'un gel de processeur sur le temps d'exécution global du programme

Les cycles de gel augmentent évidemment le nombre moyen de Cycles Par Instruction –ou *Cycles Per Instruction* (CPI) du processeur. Cette augmentation dépend non seulement du taux d'échec (proportion de requêtes du CPU dont la donnée lue est indisponible en mémoires caches), mais aussi du coût de l'échec c'est-à-dire du nombre moyen de cycles nécessaires pour rapatrier la ligne de cache manquante. En cas d'échec sur un cache L_1, ce nombre de cycles de gel peut être très élevé (plusieurs centaines de cycles), s'il faut aller chercher la ligne de cache qui intègre ladite donnée dans une mémoire principale (Dynamic Random–Access Memory (DRAM)).

La figure 2.6 présente la hiérarchie mémoire dans un système ou un sous-système et donne une approximation de l'ordre de grandeur de la latence (cycles) entre le CPU et les différents niveaux du chemin mémoire. En s'appuyant sur cette hièrarchie mémoire simple, l'équation (2.1) (HENNESSY et PATTERSON [2000])présente le temps moyen d'accès à la mémoire principale dans un système qui intègre deux niveaux de caches (L_1 et L_2). On remarque que la pénalité au niveau du cache L_1 est proportionnelle au taux d'échec dans le cache L_2 (équation (2.2)).

Il existe plusieurs types d'échecs dans les caches de données.

— Les échecs obligatoires : ils surviennent lors du premier accès à un bloc de données. Le programme n'a jamais sollicité une donnée de ce bloc. Dans cette situation les don-

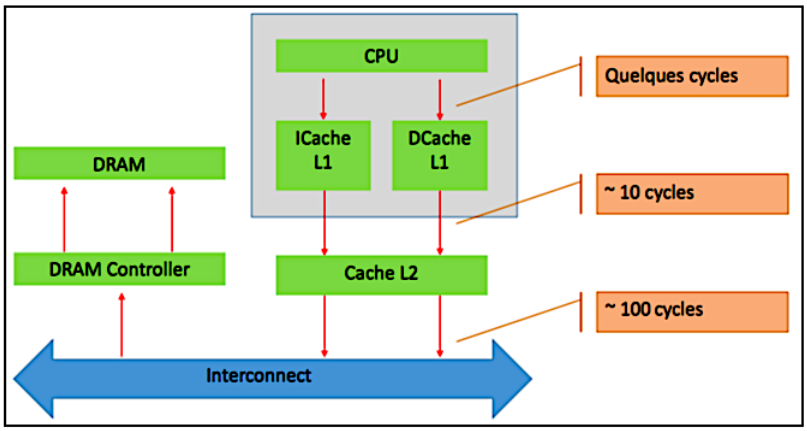

FIGURE 2.6: Hiérarchie mémoire dans un SoC simple ou un sous–système d'un SoC complexe.

nées seront obligatoirement copiées en mémoire cache.

— Les échecs dus à la capacité : ils surviennent lorsque l'application utilise activement plus de données que le cache L_1 peut en contenir. Des blocs sont alors éjectés du cache parce que le cache ne peut pas contenir toutes les données nécessaires à l'exécution du programme.

— Les échecs dus à des conflits : dans le cas des blocs set-associatifs ou directement mappés, ils surviennent lorsque plusieurs blocs sont mappés sur un même ensemble ou ligne de cache.

Il existe plusieurs stratégies qui permettent de réduire l'occurrence de ces différents échecs. Notamment l'augmentation de la taille des caches, l'associativité, les optimisations faites par le compilateur, ...

Dans cette thèse, nous supposons que le matériel est fixe et que notre seule marge de manœuvre reste au niveau logiciel. Nous nous positionnons après que le compilateur ait déjà effectué au mieux toutes les optimisations nécessaires qui permettent de corriger l'apparition des échecs sus-mentionnés. Notamment le réordonnancement des procédures sans modification de l'exactitude du programme, le déroulement et/ou la fusion de boucles, l'amélioration de la localité temporelle et spatiale (par exemple les calculs sur les données d'un tableau peuvent être effectués sur les données situées dans la même ligne de cache plutôt que de parcourir aveuglement le tableau pas à pas), ... HENNESSY et PATTERSON [2000] L'idée est de trouver des stratégies qui permettent de réduire les effets négatifs de la pénalité du L_1 en réduisant le taux d'échec au niveau du cache L_2. TRUONG et collab. [1998] ont effectué des travaux pour améliorer le comportement des mémoires caches sur des structures de données allouées dynamiquement. Pour cela il convient d'identifier précisément les instructions du programme qui sont susceptibles de produire des échecs dans le cache L_2 pendant

l'exécution du programme.

$$\text{TAMM} = \text{Temps\_Hit\_L}_1 + \text{Taux\_Echec\_L}_1 \times \text{Penalite\_Echec\_L}_1. \tag{2.1}$$

$$\text{Penalite\_Echec\_L}_1 = \text{Temps\_Hit\_L}_2 + \text{Taux\_Echec\_L}_2 \times \text{Penalite\_Echec\_L}_2. \tag{2.2}$$

TAMM : Temps Accès Moyen à la Mémoire

Par ailleurs, dans un SoC la durée des gels du processeur peut être très longue. Ceci est dû d'une part à la latence du contrôleur de la DRAM, et la largeur du bus. Associées à la dépendance existante entre les registres des différentes instructions du programme qui lisent les données en mémoire, et les instructions du programme qui effectuent des traitements sur ces données lues d'autre part. Cette dépendance entre les registres utilisés par les différentes instructions de lecture de l'application génère dans certaines situations des aléas de données (« *Load Data Hazards* ») tel que présenté dans la figure 2.3.

Aujourd'hui les compilateurs sont capable de réordonnancer statiquement les instruc-

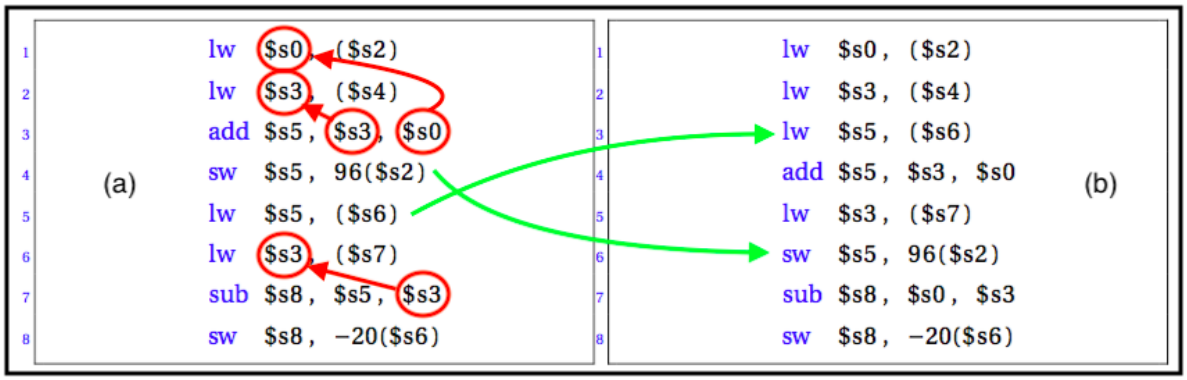

FIGURE 2.7: (a) : Pseudo code en assembleur pour architecture MIPS produisant un aléa de données. (b) : Pseudo code en assembleur pour architecture MIPS reorganisé par le compilateur pour éviter un aléa de données.

tions d'un programme pour éviter la production des aléas de données STROUT et collab. [june 2003]. Cette réorganisation tient compte des dépendances habituelles (aléa de données, de structure, ...). Le code 2.7(a) présente un exemple de code pouvant générer des aléas de données sur un processeur RISC « In–Order ». Les gels du pipeline du processeur sont observés entre les instructions adjacentes qui ont une dépendance entre les registres qui doivent être remplis avec des données qui proviennent de la mémoire principale (*instructions 2 ← 3, instructions 6 ←7*). Le compilateur a réorganisé les instructions 2.7(b) en insérant l'instruction de lecture en mémoire principale de la *ligne 5* du code 2.7(a) à la *ligne 3* du code 2.7(b). Aussi, la *ligne 4* du code 2.7(a) est insérée à la *ligne 6* du code 2.7(b).

Les processeurs qui exécutent les instructions du programme dans le désordre (« Out–of–Order ») effectuent la réorganisation dynamique des instructions du programme grâce à des modules matériels qui implémentent les algorithmes d'allocation de ressources à l'exécution. Par exemple le « scoreboarding » ou Tomasulo TOMASULO [1967].

Cependant, les 2 méthodologies de réorganisation du code (compilateur et matériel) présentent quelques limites puisqu'il reste toujours des aléas de données qui influencent considérablement le temps d'exécution global du programme même après réordonnancement. Ceci implique que le CPI de l'instruction gelée est augmenté du nombre de cycles supplémentaires produit par le gel.

Dans l'équation (2.5) obtenu en s'inspirant des cours de SHAABAN [2005], on remarque que le temps d'exécution global d'une application varie en fonction de trois composantes. La première est le temps supplémentaire ajouté par les instructions du programme qui sont gelées à cause du chargement d'une ligne de cache. La seconde représente le temps dû à d'autres effets notamment du pipeline. Par exemple un défaut de prédiction de branchement. Et enfin la troisième est le temps d'exécution idéal. On voit donc l'impact des cycles de gel du processeur dus aux accès non immédiatement disponibles aux données, sur le temps d'exécution d'un programme.

Dans notre travail, nous nous intéressons à réduire l'influence de la première composante de l'équation (2.5).

$$\text{CPI}(i)_{\text{Réel}} = \text{A}(i) + \text{B}(i) + \text{CPI}(i)_{Ideal} \tag{2.3}$$

Avec

— A : Nombre de cycles de gel dus à une lecture en mémoire

— B : Nombre de cycles ajoutés par d'autres effets (Branchement, ...)

— i : L'instruction

Concernant le temps d'exécution globale de l'application, on pose :

$$\text{Temps d'Exécution Idéal (TEI)}_{\text{P}} = \sum_{i=1}^{\text{N}} \text{CPI}(i)_{Ideal} \times Temps\_1cycle \tag{2.4}$$

$$\text{Temps d'Exécution Réel (TER)}_{\text{P}} = (\sum_{j=1}^{\text{J}} \text{A}(j) + \sum_{k=1}^{\text{K}} \text{B}(k)) \times Temps\_1cycle + \text{TEI} \tag{2.5}$$

Avec

— P : Le Programme en cours d'exécution

— N : Le nombre total d'instructions du programme

— j : L'instruction gelée à cause d'une lecture en DRAM

— J : Le nombre d'instructions gelées à cause d'une lecture en DRAM, $0 \leq J \leq N$

— k : L'instruction gelée à cause d'autres effets

— K : Le nombre d'instructions qui mènent à des gels à cause d'autres effets, $0 \leq K \leq N$

Dans la suite du document, l'utilisation du terme « gel » réfèrera à un gel du processeur dû à l'indisponibilité de données en cache L_1.

2.4 Identification des instructions produisant des gels

Un gel du processeur est évidemment produit par une instruction du programme. Aujourd'hui les processeurs modernes intègrent plusieurs étages de pipeline. Et en fonction de ce nombre d'étages et du nombre d'unités de calcul du processeur, retrouver l'instruction en cours d'exécution à tout instant donné n'est pas trivial. En effet, plusieurs instructions étant « en vol » simultanément (typiquement une par étage de pipeline), il n'est pas forcément trivial d'identifier celle qui est à la source d'un gel.

À matériel fixe et sans changements majeurs de la structure du code, il est aujourd'hui possible dans une simulation CA d'obtenir une bonne approximation du nombre total de cycles de gel du processeur. Ceci en s'intéressant précisément aux registres internes du processeur situés dans l'unité de monitoring de la performance (Performance Monitoring Unit (PMU)) du CPU. Ici, par simulation CA , nous ne faisons pas référence aux simulations sous gem5 bien qu'il soit un simulateur de référence (BINKERT et collab. [2011]). Car sur cet environnement de simulation, les modèles de processeurs disponibles sont le plus souvent génériques. Ces modèles ne représentent pas les particularités de la microarchitecture d'un processeur donné. Ils sont en mesure d'exécuter le jeu d'instructions du processeur, mais ne disposent pas d'interfaces bus. Par exemple le Cortex–A53 qui est un modèle de processeur sur architecture ARM–V8 générique.

Cependant, identifier précisément les instructions du programme qui génèrent les cycles de gel demeure un réel challenge. Ces instructions qui produisent des gels dans le pipeline du processeur sont non seulement une réelle pénalité pour la performance du processeur, mais sont aussi difficilement identifiables précisément par un compilateur, à cause de la complexité à simuler rapidement ou à avoir un modèle analytique correct qui représente le comportement temporel d'une hiérarchie mémoire.

Aujourd'hui il est typiquement possible de fournir en options à un compilateur la taille des caches, la profondeur des buffers d'écriture, la latence d'accès à la mémoire principale,

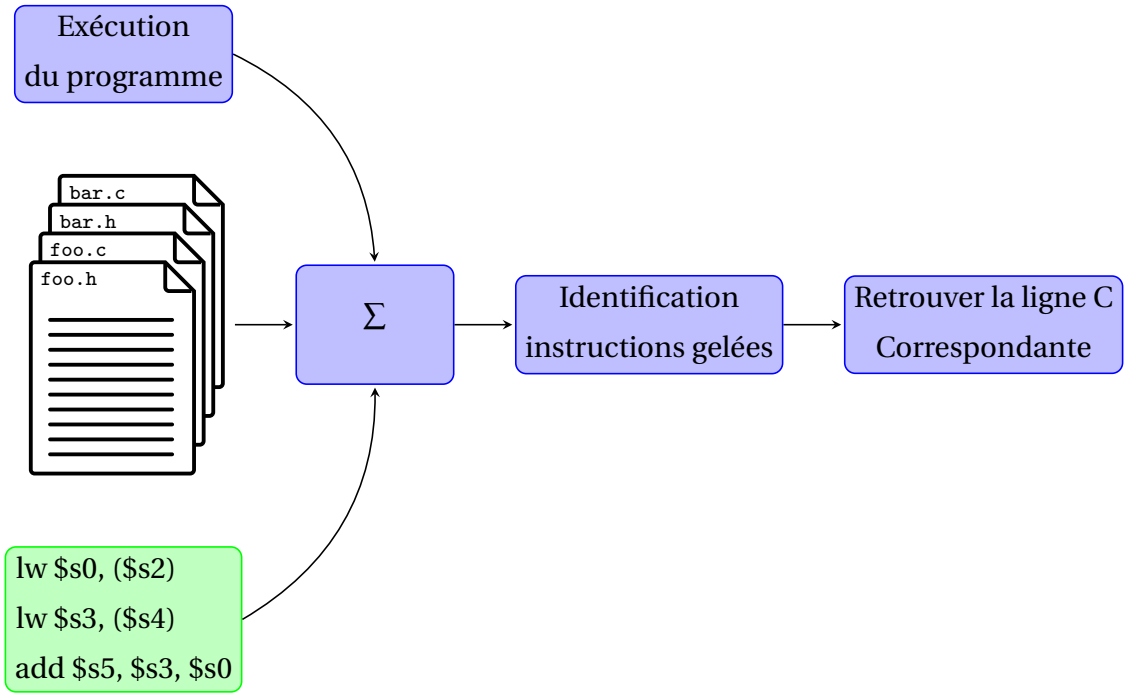

FIGURE 2.8: Vue globale d'un processus d'identification des instructions qui produisent des cycles de gel du processeur en cours de l'exécution d'un programme.

la largeur du bus, ... Malgré cela, le compilateur a au mieux une idée assez abstraite, concentrée sous forme de fonctions analytiques de coût, du comportement de l'architecture globale du SoC. Ces informations ne sont pas suffisantes pour prendre les bonnes décisions permettant l'identification précise des instructions qui produiront des cycles de gel.

En outre, comme présenté sur la figure 2.8 comment à partir du code cross-compilé d'un programme, peut-on retrouver les instructions assembleur qui provoquent des gels et conjointement l'instruction correspondante en langage de haut niveau (C par exemple) ayant provoquée chacun d'entre eux? Aujourd'hui il existe des outils qui permettent à partir de l'adresse d'une instruction de retrouver la ligne correspondante dans le programme écrit en langage de haut niveau. La question revient donc à identifier le bon PC associé à un gel sur du code effectivement exécuté par le processeur et de faire ensuite appel à un outil qui permet de retrouver la ligne correspondante dans le programme à partir du PC identifié, puis d'analyser la ligne de programme, potentiellement complexe. La figure 2.8 présente succinctement les entrées que nous estimons nécessaires pour l'identification des instructions produisant les gels du processeur. Ce processus part de trois entrées : l'exécution du programme, le code désassemblé ou le code effectivement exécuté par le processeur, et le code en langage de haut niveau du programme. Pour produire en sortie les instructions (liste des PCs) qui ont généré de la latence supplémentaire au TER global du programme.

2.5 Identification des gels pertinents à réduire

Partant du constat selon lequel les capacités de traitement du processeur n'évoluent plus en suivant la loi de Moore, comme le dit *Tom Simonite* dans le « MIT Technology Reviews » : *« Shrinking transistors have powered 50 years of advances in computing—but now other ways must be found to make computers more capable »* le 23 Mai 2016 SIMONITE [May 13, 2016]. La marge de manœuvre pour améliorer l'utilisation des capacités matérielles disponibles sur un SoC réside entre autre sur le logiciel et sur son adaptation au matériel. Les ingénieurs ont besoin d'avoir une idée assez claire du gain de performance qu'ils pourraient obtenir, mais aussi de la nécessité à effectuer des modifications sur le logiciel.

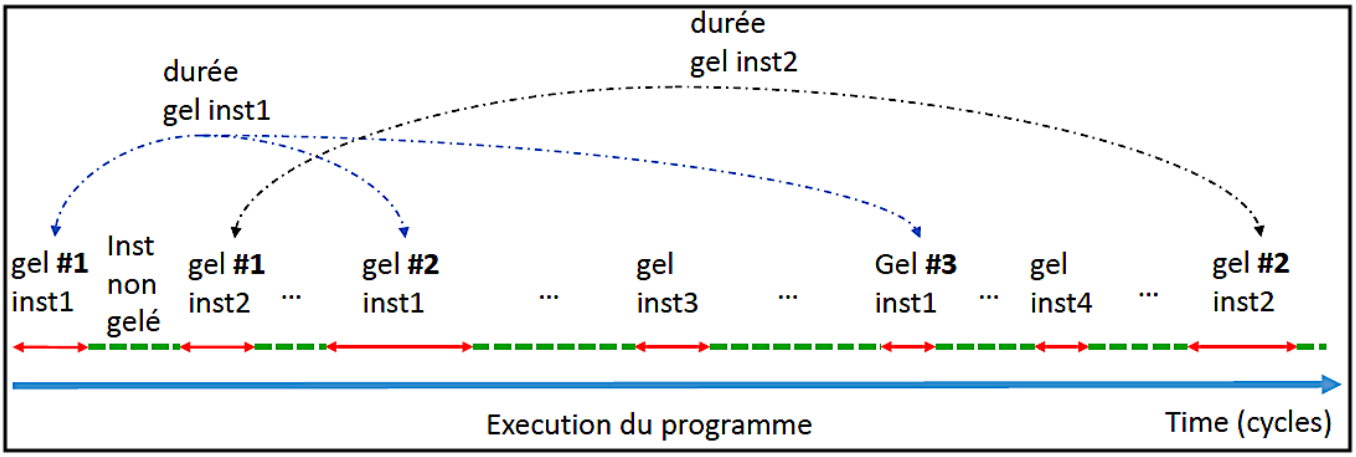

FIGURE 2.9: Illustration du concept de variabilité de la durée des différents gels intermédiaires pour une même instruction au cours de l'exécution du programme

Pour apporter des indices aux ingénieurs sur la répartition des gels au cours de l'exécution du programme, il est important de quantifier pour chacun des gels leurs contributions au rallongement du temps d'exécution global d'un programme en attribuant à chacun des gels un « poids ». Par attribuer un « poids », nous signifions quantifier le nombre de cycles ajoutés au TER global de l'application. Ceci pour chacun des gels du processeur dû à l'indisponibilité de données nécessaires à la poursuite de l'exécution dans les registres utilisés par l'instruction, et en mémoires cache. Plus simplement, ceci permet d'estimer la durée de chaque gel. Une même instruction peut être gelée plusieurs fois avec des durées plus ou moins longues pour chacun des gels de la même instruction au cours de l'exécution d'un programme. Ceci est illustré dans la figure 2.9. On voit que les *inst1* et *inst2* sont gelées 3 fois (respectivement 2 fois). La durée totale du gel de *inst1* est le cumul des durées intermédiaires des gels sur cette même instruction (*gel #1 inst1*, *gel #2 inst1*, *gel #3 inst1*). On remarque également que la durée de chaque gel est variable. Cette variabilité provient de

faits diverses. Par exemple une contention au niveau du bus. De manière plus générale telle que défini dans l'équation (2.6), il est alors possible d'estimer la durée du gel d'une instruction par sommation des contributions des différents gels de la même instructions au cours du temps. Ceci permet d'avoir une vision sur la pertinence de chaque gel. Le gain total potentiel en termes de TER total de l'application est obtenu également par sommation de la durée du gel de chaque instruction gelée identifiée. Confére équation (2.7).

$$\text{Nombre Cycle Gel}_{instruction} = \sum_{i=1}^{\text{K}} \text{durée Gel}_i \tag{2.6}$$

$$\text{Gain Potentiel} = \frac{(\sum_{i=1}^{\text{K}} \text{durée Gel}_i) \times \text{N}}{\text{TER}} \times 100 \tag{2.7}$$

Avec :

— K : Le nombre de fois qu'une même instruction est gelée

— N : Le nombre d'instructions qui ont été gelées pendant l'exécution du programme

— i : Une occurence du gel sur une même instruction

Ainsi, l'effort des ingénieurs pour améliorer la performance du logiciel sera plus orienté, ciblé et concentré sur les gels ayant un impact important c'est–à–dire présentant une grande latence.

Le fait d'avoir une idée du gain potentiel permet également de prendre rapidement des décisions notamment sur la nécessité de poursuivre la correction des gels afin d'optimiser ou pas la performance du logiciel sur la puce.

2.6 Conclusion partielle

Une représentation simplifiée sous forme de machine d'états donnée en figure 2.10 met en évidence le comportement du processeur en fonction de la disponibilité des données accédées. On constate que la réduction du nombre d'évènements de type e_3 tout en augmentant le pourcentage d'apparition d'évènements de type e_2 serait bénéfique pour l'exécution d'un programme. Pour cela il faudra identifier précisément les instructions de l'application qui les génèrent.

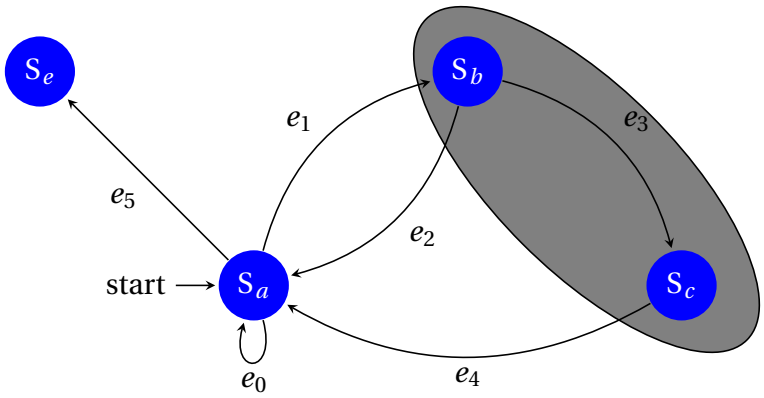

FIGURE 2.10: Représentation simplifiée pour mettre en évidence le comportement du processeur en fonction de la disponibilité des données accédées sous forme de machine d'états

Etat S_a : Exécution de l'instruction en cours

Etat S_b : Attente de données des mémoires caches

Etat S_c : Processeur gelé, attente de données

Etat S_e : Idle ou éteint

e_0 : Données disponibles dans les registres

e_1 : Données indisponibles dans les registres

e_2 : Données disponibles en mémoires caches ($L_1 et/ou\ L_n$)

e_3 : Données indisponibles en caches

e_4 : Données à nouveau disponibles

e_5 : Fin du traitement de l'exécution

Ainsi, au vu de ce qui a été présenté précédemment dans ce chapitre, plusieurs questions auxquelles les travaux de cette thèse visent à répondre se dégagent :

➢ Q1 : Comment identifier les instructions du programme qui génèrent des cycles de gel du processeur et la latence produite ? Dans les cas suivants :

- Visibilité totale sur les registres internes et concernés du processeur, et ceux-ci contiennent des informations sur les gels.
- Aucune visibilité sur les registres concernés du processeur, ou pas de registres existants fournissant des informations sur les gels.

➢ Q2 : Comment mesurer la durée de chaque gel, pour connaître l'importance de son impact ? Et notamment :

- Sur le nombre de cycles potentiels (pourcentage) qu'un ingénieur pourrait gagner sur le temps d'exécution total de l'application, s'il améliore l'utilisation des capacités du matériel par le logiciel.

- Sur la nécessité pour un ingénieur de (continuer à) chercher à optimiser l'utilisation du matériel par le logiciel.

➢ Q3 : Comment utiliser au mieux l'effort d'optimisation, en le concentrant sur les gels ayant le plus d'impact ?

➢ Q4 : Qu'est-il possible d'automatiser pour minimiser l'effort du développeur lorsqu'il cherche à améliorer la performance de son logiciel par réduction du nombre et la durée (cycles) des gels ?

2.7 Références

BINKERT, N., B. BECKMANN, G. BLACK, S. K. REINHARDT, A. SAIDI, A. BASU, J. HESTNESS, D. R. HOWER, T. KRISHNA, S. SARDASHTI, R. SEN, K. SEWELL, M. SHOAIB, N. VAISH, M. D. HILL et D. A. WOOD. 2011, «The gem5 simulator», *SIGARCH Comput. Archit. News*, vol. 39, nº 2, doi :10.1145/2024716.2024718, p. 1–7, ISSN 0163-5964. URL http://doi.acm.org/10.1145/2024716.2024718. 16

BLEM, E., J. MENON et K. SANKARALINGAM. 2013, «Power struggles : Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures», *Proceedings - International Symposium on High-Performance Computer Architecture*. 9

HENNESSY, J. L. et D. A. PATTERSON. 2000, *Computer Architecture a Quantitative Approach*, Harcourt ASIA PTE LTD. 12, 13

KANG, J.-Y., S. GUPTA, S. SHAH et J.-L. GAUDIOT. 2003, «An efficient pim (processor-in-memory) architecture for motion estimation», *Proceeding of IEEE 14th International Conference on Application-Specific Systems, Architectures and Processors (ASAP2003)*, p. 273–283. 8

PORPODAS, V., A. MAGNI et T. M. JONES. 2015, «Pslp : Padded slp automatic vectorization», *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

SHAABAN. 2005, «Instruction pipeline review», http ://meseec.ce.rit.edu/eecc551-winter2005/551-11-30-2005.pdf. [En ligne ; dernier accès le 7 Juillet 2017]. 15

SIMONITE, T. May 13, 2016, «Moore's law is dead. now what ?», https://www.technologyreview.com/s/601441/moores-law-is-dead-now-what/. [En ligne ; dernier accés le 25 juillet 2017]. 18

STROUT, M. M., L. CARTER et J. FERRANTE. june 2003, «Compile-time composition of run-time data and iteration reordering», *Proceeding of 2003 ACM SIGPLAN conference on Programming Language DEsign and Implementation*. 14

TOMASULO, R. M. 1967, «An efficient algorithm for exploiting multiple arithmetic units», *IBM Journal of Research and Developement*, vol. 11, p. 25–33. 15

TRUONG, D. N., F. BODIN et A. SEZNEC. 1998, «Improving cache behavior of dynamically allocated data structures», *Proceeding of the 8th IEEE International Conference on Parallel Architecture and Compilation Techniques*, p. 322. 13

WULS, W. A. et S. A. MCKEE. 1995, «Hitting the memory wall : Implications of the obvious», *Computer Architecture News*, vol. 23, p. 20–24. 8

ZHANG, Z. 2005, «Scoreboarding and tomasulo algorith», http://users.utcluj.ro/~sebestyen/_Word_docs/Cursuri/SSC_course_5_Scoreboard_ex.pdf. [En ligne; dernier accés le 25 juillet 2017].

ZHONG, Y., M. ORLOVICH, X. SHEN et C. DING. June 2004, «Array regrouping and structure splitting using whole program reference affinity», *Proceeding of 2004 ACM SIGPLAN conference on Programming Language Design and Implementation*.

Chapitre 3

État de l'art

Sommaire

3.1 Introduction

Mesurer la performance d'une application sur un système est une activité cruciale effectuée de manière conjointe entre les ingénieurs matériel et les ingénieurs logiciel. Pour ce qui est des ingénieurs logiciel, ils ont besoin d'identifier les portions critiques d'un programme qui contribuent à la dégradation de la performance globale du système. La dégradation de la performance peut être due à plusieurs facteurs. Parmi ces facteurs, on s'intéresse dans cette thèse aux cycles de gel du processeur dus à l'indisponibilité des données dans le cache L_1 du processeur.

Identifier les instructions du programme qui engendrent ces cycles de gel du processeur n'est pas une activité facile. Il en est de même pour la correction de ces cycles de gel du processeur. Théoriquement, l'apparition d'un gel du processeur apporte une occasion d'optimiser la performance du logiciel sur le matériel. Il existe aujourd'hui différentes techniques qui permettent d'apporter une solution à la correction ou à l'évitement des gels du processeur. Cependant, bien que ces techniques utilisées toutes seules soient efficaces dans certains cas d'étude, elles présentent quelques limitations. Nous analyserons sans être exhaustifs, quelques–unes de ces techniques.

3.2 Formules analytiques et générateurs de trafic pour l'estimation de performance

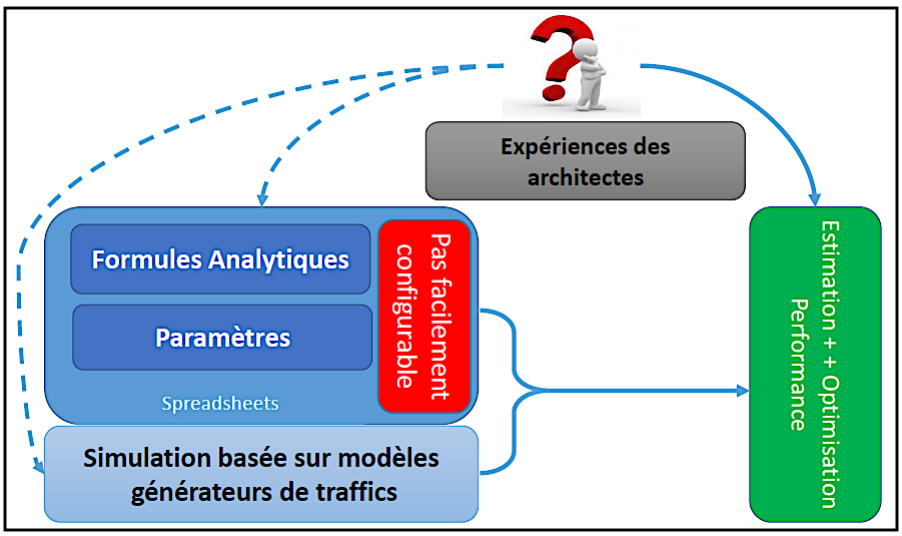

FIGURE 3.1: Flot basé sur des formules analytiques et des générateurs de trafics pour l'estimation et l'optimisation de la performance du logiciel sur la puce en amont d'un projet de développement SoC.

La figure 3.1 présente le flot d'estimation de la performance du logiciel sur puce, utilisé par les architectes. On peut observer dans ce flot les différents outils utilisés (*flèches pointillées*) par les architectes pour concevoir des systèmes performants. Dans cette optique, avant que le futur SoC ne soit disponible physiquement, les architectes ont utilisé pendant longtemps des modèles et/ou des formules purement analytiques (SIMONSON et HE [2005]). Associés à l'expérience des architectes systèmes, acquise lors de la conception des générations précédentes du produit en cours de développement, cette approche s'est avérée pendant longtemps suffisante au regard de la simplicité relative des produits. Ces formules et paramètres sont principalement répertoriés dans des outils de type tableur. Cette approche n'exécute pas l'application mais en tient compte par le biais des hypothèses faites sur son exécution.

Par ailleurs, ces formules fruits de l'expérience de l'architecte se comptent aujourd'hui par centaines par feuille de tableur pour chaque nouvelle conception de SoC et sont devenues quasiment impossibles à comprendre de nouveau et à faire évoluer pour des modifications majeures à apporter aux architectures (évolution de génération ou variante dans une famille de SoCs). Elles ne sont pas facilement configurables, par exemple pour définir une nouvelle variante de la topologie de l'interconnexion dans un SoC.

Pour dimensionner les interconnexions (largeur du bus, profondeur des FIFO, ...), le recours à la simulation est à présent bien établi. Les simulations sont conçues à partir de modèles RTL (très lents, de l'ordre du kHz), ou de modèles précis au niveau du cycle (CA) du bus, et/ou de blocs générateurs de trafics. Un tel générateur de trafic est paramétrable et produit des transactions qui tentent de représenter le trafic généré sur le bus par un bloc/IP maître (par exemple un Accès Direct à la mémoire –ou Direct Memory Access (DMA)), ou par un processeur exécutant un programme (SCHERRER et collab. [2006]).

Dans les deux cas, l'utilisation des formules analytiques tout comme les simulations basées sur des générateurs de trafics, la méthodologie n'inclut pas une exécution réelle du programme sur la cible SoC. Elle ne fournit donc pas une trace réelle d'exécution dans le SoC. Ceci peut engendrer principalement deux effets négatifs :

— Un sur-dimensionnement du produit final : avec des capacités supérieures à celles mentionnées dans les documents de spécification. Ce qui peut conduire à des coûts de fabrication élevés à cause de l'augmentation de la surface du silicium induite.

— Un sous-dimensionnement du produit final : avec des capacités inférieures à celles attendues. Ce qui peut conduire au mécontentement des clients, avec pour conséquence directe la perte de parts de marché considérable, liés à la perte de crédibilité de l'entreprise.

Le flot typique de conception d'un SoC est présenté dans la figure 3.2. Les activités de conception s'appuyant sur les formules analytiques et les générateurs de trafics sont effectuées à l'étape « architecture backbone [1] SoC » (étape numéro *2*). Ce flot commence par la définition de l'architecture globale du SoC à partir de la spécification de besoin : nombre et type des blocs/IPs, existence de chemins de données entre eux, d'un point de vue fonctionnel; une première estimation de dimensionnement peut éventuellement être faite dans un tableur.

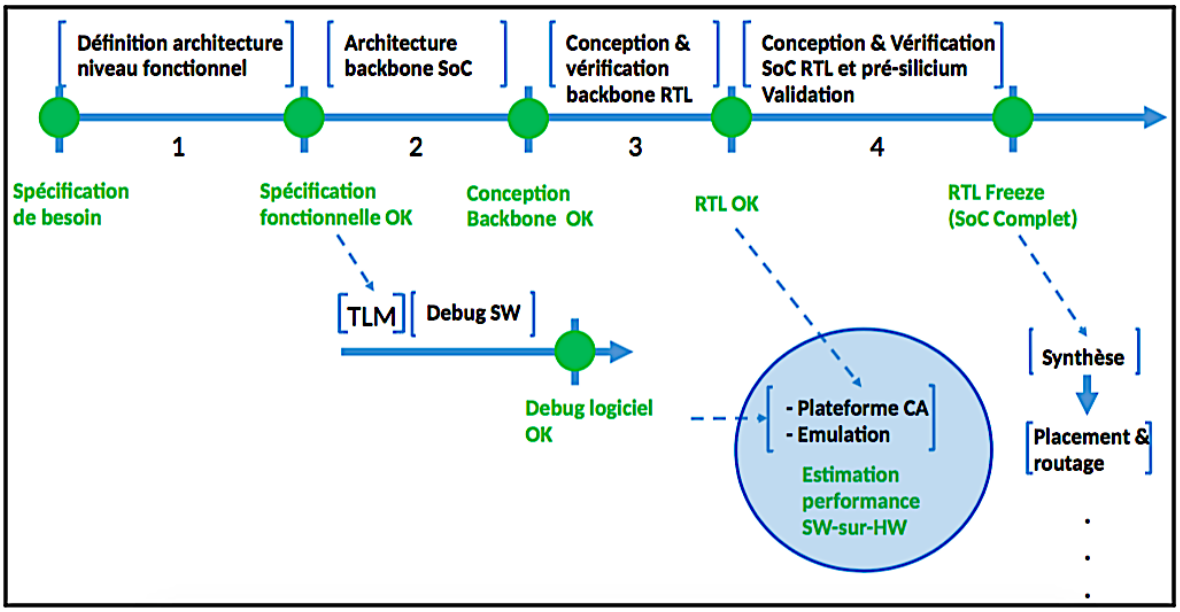

FIGURE 3.2: Flot typique de conception d'une puce

Dans la seconde étape de ce flot, la topologie de l'interconnexion, qui est au cœur des trafics dans le SoC, est modélisée. Les activités de définition d'ossature (backbone), s'appuyant sur la simulation du modèle d'interconnexion stimulé par les générateurs de trafics, sont effectuées à cette étape « architecture backbone SoC ».

Il est à noter que dans cette étape certains périphériques non–critiques d'un point de vue trafic ne sont pas pris en compte au moment de l'étude de l'architecture d'interconnexion. L'étude à cette étape n'inclut pas l'exécution du logiciel et se focalise juste sur l'étude du chemin principal processeur —> interconnexion —> mémoires (DRAM, flash, ...) qui constitue l'ossature minimale du SoC. Sur la figure 3.3, l'ossature minimale est représentée par la partie délimitée sur la figure. L'étude à cette étape n'inclut pas non plus l'exécution du logiciel, qui n'est en général pas disponible dans cette phase amont, et est représenté par des générateurs de trafic.

Considérons par exemple une interconnexion entre 2 ports maîtres et 2 ports esclaves

1. backbone = ossature

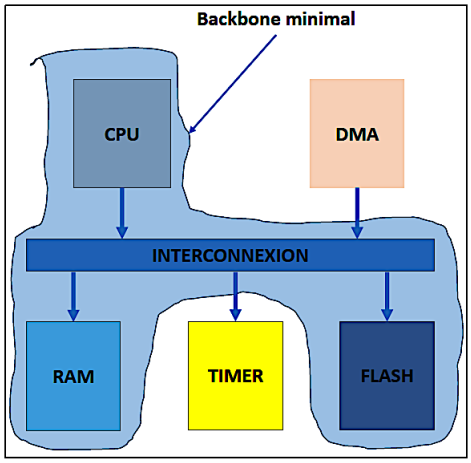

FIGURE 3.3: Exemple de plateforme montrant la délimitation du périmètre du backbone minimal d'un SoC

telle que présentée dans la figure 3.4.

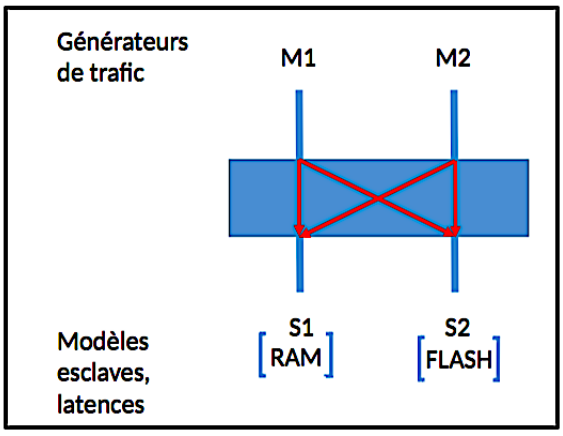

FIGURE 3.4: Exemple simple de topologie de l'interconnexion dans le bus entre 2 ports maîtres et 2 ports esclaves

Dans cette situation simple, il est possible d'effectuer un maillage complet entre les 2 catégories de ports pour définir les différents chemins d'interconnexion. Le nombre de ports étant réduit, cela reste possible.

En revanche, lorsqu'il y a plusieurs maîtres et esclaves (on en compte des dizaines dans les SoCs complexes), il faut trouver les chemins critiques et importants car il n'est pas efficace d'effectuer un maillage complet entre les différents ports. Ceci prendrait trop de place sur la plaquette de silicium et augmenterait ainsi les coûts de fabrication. De plus, le placement–routage serait très compliqué voire impossible, avec des longueurs excessives de chemins critiques.

Des outils spécialisés pour l'étude d'architecture de l'interconnexion (topologie gra-

phique du bus, algorithmes de routage, ...) permettent d'effectuer plusieurs configurations alternatives, et de générer automatiquement le RTL et les modèles CA correspondants. Le modèle d'interconnexion, paramétrable par l'architecte pour des boucles rapides d'analyse pour le choix de la configuration, et l'outil de simulation associé, sont typiquement spécifiques du type d'interconnexion utilisé. Les types d'applications qui seront exécutées sur le futur SoC sont modélisés sous forme de générateurs représentant le trafic des processeurs pour marché visé (Application-Specific Integrated Circuit (ASIC) ou Application-Specific Standard Product (ASSP)). Par exemple le profil du trafic d'entrées–sorties est modélisé, car une application de décodage vidéo ne produira pas le même profil de trafic qu'une application de décodage audio.

La qualité de service de l'interconnexion pour les applications modélisées par les générateurs de trafic, peut être ainsi évaluée par analyse des résultats de simulation, comme par exemple les temps d'accès d'un port maître à un port esclave. L'interconnexion entre les différents composants maîtres et les composants esclaves est alors dimensionnée, notamment le nombre de noeuds de routage, la largeur et priorité des canaux inter–noeuds, les tailles des différentes FIFOs). A l'issue de cette étape de conception d'ossature, les bonnes valeurs des paramètres nécessaires à la configuration de l'interconnexion sont figées : l'interconnexion est définie (« interconnect freeze »).

Parallèlement à l'étape 2 dans le flot de conception, une fois que l'architecture fonctionnelle du SoC a été définie en étape 1, les modèles fonctionnels de type Transaction Level Modeling (TLM) des principaux composants de la plateforme sont développés. La simulation TLM peut inclure un modèle du processeur qui ne tient pas compte des cycles d'exécution (Instruction Accurate (IA)) et sur lequel il est facile de connecter un débogueur. Le développement et le débogage du logiciel peut donc commencer en parallèle du développement matériel. Cette simulation, disponible tôt, est rapide en temps d'exécution car plus abstraite que les modèles de niveau cycle. Elle utilise la table d'adressage (« address map ») définie par les architectes lors de l'étape *1*. Le logiciel du projet peut alors être mis au point sans attendre la disponibilité du modèle RTL.

Puis, lors de l'étape 3, le RTL généré lors de la précédente étape (2) est assemblé avec le RTL des contrôleurs mémoire (flash, DRAM) pour construire le modèle RTL de l'ossature. Un exemple est présenté en figure 3.5. Cet exemple comporte 2 processeurs ARM Cortex–A53. L'estimation de la performance du logiciel développé et débogué lors de la simulation TLM peut débuter, pour la partie du logiciel qui réalise des accès mémoire sans accès à des périphériques particuliers. Ceci afin d'estimer la performance applicative et la comparer à la spécification initiale du besoin.

Pour cette exécution à des fins d'estimation de performance, les modèles RTL des dif-

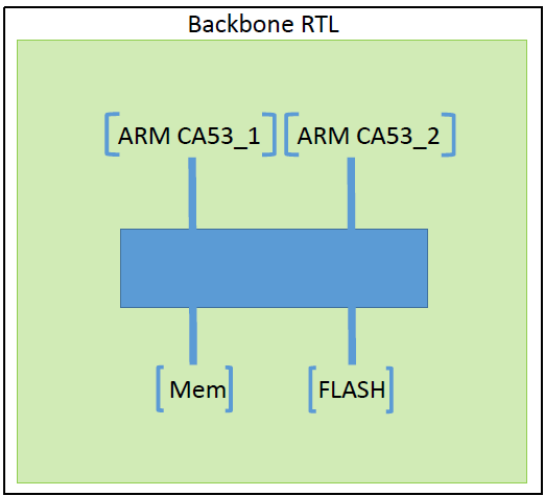

FIGURE 3.5: Exemple simple de backbone RTL

férents composants de l'ossature du SoC peuvent être transformés automatiquement par des outils existants (ARM-CYCLE-MODELS-STUDIO [2017]) pour produire leurs modèles précis au niveau du cycle (CA). De même pour certains blocs/IPs périphériques existants, ils peuvent alors être ajoutés à la plateforme pour permettre l'exécution des portions supplémentaires du logiciel qui nécessitent ces blocs périphériques (tel que par exemple certaines entrées–sorties) ; ou si la présence de leurs modèles respectifs est utile pour exécuter le code du logiciel sans effort de simplification.

Une première utilisation de la méthodologie proposée dans cette thèse se situe à ce niveau du flot, lorsqu'est disponible l'architecture de l'ossature minimale (processeur & interconnexion & mémoires), puis l'architecture de l'ossature complétée avec des blocs maîtres/esclaves utiles pour l'étude de trafic. L'estimation de performance du logiciel (réel développé sur plateforme TLM, ou codes de benchmarks représentatifs de la future application) permet de confirmer la définition de l'ossature, ou dans le cas contraire permet à l'architecte de se rendre compte suffisamment tôt dans le projet SoC du besoin de reprendre l'étude d'architecture avec des paramètres différents (tailles de FIFO par exemple), voire avec une topologie différente. Et ceci avant l'étape suivante d'intégration du RTL complet du SoC, étape 4.

L'étape *4* de ce flot consiste en la conception et la vérification du RTL du SoC complet, avec tous les blocs (IPs), puis en la validation pré–silicium du RTL avec le logiciel. Lorsque le RTL du SoC est 100% vérifié, et suffisamment validé avant silicium (le boot de Linux peut être un critère pour certains SoCs), les étapes classiques suivantes sont effectuées : tape–out ou PG–tape, synthèse, placement–routage, fabrication des masques, ...

A partir de ce moment, le matériel du SoC est figé (sauf dans le cas de Field Programmable Gate Array (FPGA) embarqué), et les performances du futur produit ne peuvent être

améliorées par les clients du fabriquant de SoC que grâce à des optimisations logicielles. La méthodologie proposée dans cette thèse trouve alors une seconde application.

Que ce soit pendant l'étape 3 du développement du SoC pour identifier son architecture optimale d'ossature en utilisant un logiciel ou benchmark optimisé, ou lors de l'optimisation du logiciel final du client sur le silicium, nous cherchons un moyen aidant à placer correctement dans le code certaines instructions de pré–chargement de données. Ceci pour contribuer à optimiser le logiciel pour une meilleure performance du système complet : logiciel sur matériel du SoC. Pour un tel moyen, nous utilisons dans cette thèse le prototypage virtuel en nous basant sur la simulation avec des modèles précis au niveau du cycle (CA), capable d'exécuter l'application cible (niveau encerclé dans le flot de conception 3.2), c'est–à–dire à la fin de la phase 3 *: conception backbone RTL*. Ceci nous permet d'exploiter d'une manière nouvelle (l'aide au placement des instructions de pré–chargement) l'observabilité fournie par de tels modèles pour estimer et optimiser la performance du logiciel sur la puce, en amont et en aval de la disponibilité du silicium.

3.3 Plateformes virtuelles précises au niveau du cycle

Nous avons choisi d'utiliser les modèles précis au niveau du cycle (CA) dans l'approche développée. Traditionnellement, de tels modèles, s'ils sont développés manuellement, sont très coûteux en temps, sont fréquemment délivrés avec du retard, et avec une correspondance au RTL qui n'est pas toujours garantie, au mieux testée via des simulations. En terme de coût de développement et de date de mise à disposition des équipes, ce n'est donc pas une solution particulièrement raisonnable.

Aujourd'hui, des outils matures, disponibles industriellement, rendent possible la génération automatique des modèles CA d'un sous-système ou d'un système complet à partir d'un langage de description du matériel (Hardware Description Language (HDL)) au niveau RTL (ARM [2017]). Les plateformes virtuelles CA peuvent ainsi être conçues et assemblées aussitôt que les modèles RTL des différents blocs matériels ou IP existent et ceci avec des efforts de développement extrêmement réduits. Les modèles CA peuvent être conçus de manière à être suffisamment configurables, et permettent de faire des analyses en utilisant différentes valeurs pour leurs paramètres.

Cependant, bien que plus rapides que les simulations basées sur des composants décrits en HDL–RTL, ces modèles sont nettement plus lents que les modèles transactionnels (TLM). La figure 3.6 présente un ordre de grandeur des vitesses de simulation à différents niveaux d'abstraction. En prenant comme vitesse de référence celle du niveau RTL (×1), on observe que les modèles CA sont de l'ordre de *10 fois* plus rapides que les modèles RTL et que les

modèles TLM sont de l'ordre de *1000 à 10000 fois* environ plus rapides que les modèles CA. Bien que la simulation TLM soit rapide, et qu'il soit plus aisé d'y connecter un débogueur, elle ne fournit pas d'informations suffisamment précises sur le comportement de la micro–architecture du processeur et du reste du système et ne permet donc pas l'optimisation de performance telle que nous l'envisageons.

Avec des modèles CA suffisamment précis (rapport proche de 100% de la durée d'une transaction ou d'autres actions en simulation par rapport à la durée réelle mesurable sur une simulation HDL–RTL), les plateformes virtuelles CA apportent une valeur ajoutée pour l'évaluation de performance temporelle de l'intégration matériel/logiciel d'un SoC. Ces modèles paramétriques offrent plus de flexibilité au niveau de l'architecture globale de la puce, plus d'observabilité que le prototypage matériel et plus de réalisme que les méthodologies qui s'appuient sur des formules analytiques et/ou les générateurs de trafics, ce qui est un avantage clé pour les architectes système et les développeurs de logiciel sur puce.

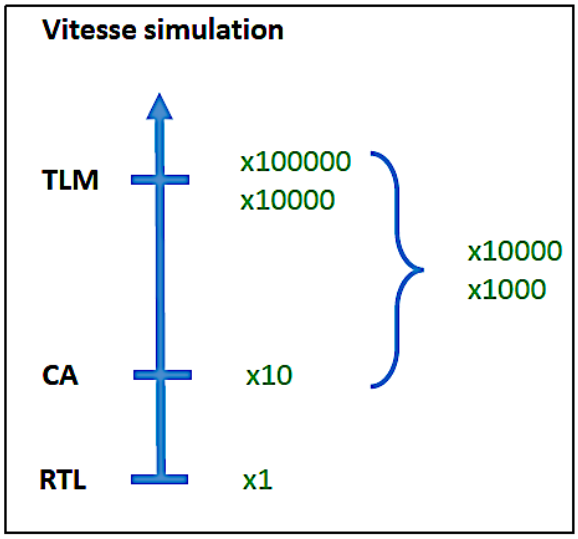

FIGURE 3.6: Vitesse de simulation entre les différents niveaux d'abstraction

En somme, les simulations du futur SoC basées sur des modèles CA peuvent être générées à partir de blocs d'IPs existants, développés au niveau d'abstraction RTL. Cette approche est particulièrement bien adaptée au modèle économique des sociétés développant des modèles d'IPs et des SoC complets pour leurs clients, puisque la réutilisation massive d'IPs existantes sera possible. Ces simulations offrent la possibilité de profiler et de monitorer le comportement interne du matériel dans différentes configurations. De plus, elles sont suffisamment rapides pour permettre l'exécution d'un programme réel ou d'un ensemble de benchmarks représentatifs des situations réelles. Ceci permet d'étudier en amont plusieurs variantes de l'architecture du SoC durant le processus de conception de la puce (ZHU

et collab. [2005], GERSTLAUER et SCHIRNER [2010]).

3.4 Réduction de l'impact d'un gel de processeur

Il existe plusieurs techniques qui permettent de limiter les cycles de gel du processeur générés suite à un échec dans le cache L_1 du processeur.

3.4.1 Processeur Out–of–Order

Les architectures de processeurs « out–of–order » sont une approche historique pour limiter les cycles de gel du processeur. Ce sont des processeurs de type superscalaire qui exécutent les instructions suivant un ordre partiel différent de l'ordre total du flux d'instructions (SMITH et PLESZKUN [1988] et TOMASULO [1967]). Ces processeurs exploitent des tampons de ré–ordonnancement (ReOrder Buffer (ROB)) pour permettre l'émission de multiples instructions dès que les données sont disponibles et lorsque les dépendances sont satisfaites (TOMASULO [1967]). Les instructions sont ordonnancées dynamiquement au cours de l'exécution. De ce fait, elles ne sont généralement pas exécutées dans l'ordre dans lequel elles ont été générées au moment de la compilation du programme source. Les instructions indépendantes peuvent être exécutées pendant le gel d'une autre instruction du programme. Par exemple, le listing de code 3.1, présente un code source simple en assembleur MIPS produit par le compilateur. On suppose que l'instruction *ligne 1* produit des échecs dans les différents niveaux de caches (L_1...L_n) de données de la hiérarchie mémoire du SoC lors de l'exécution de ce bout de code sur un processeur « in–order ». Une requête de lecture de données est alors émise en direction du contrôleur mémoire. Le processeur « in–order » est obligé de patienter sur l'instruction *ligne 2* jusqu'à la disponibilité de la donnée chargée dans le registre *$3* du processeur avant de poursuivre son traitement.

Dans le listing de code 3.2, en cas d'échec dans les caches lors de l'exécution de la lecture en mémoire effectuée dans l'instruction *ligne 1*, grâce aux algorithmes de re–ordonnancement dynamique disponible sur un processeur « Out–of–Order », l'instruction *ligne 3* du code 3.1 est exécutée avant l'instruction *add (ligne 2)* qui utilise la donnée de lecture. Dans cette situation, le cycle de gel est évité.

```
1  lw $3, 100($4)
2  add $2, $3, $4
3  sub $5, $6, $7
```

Listing 3.1: Code MIPS ad–hoc généré par le compilateur

```
1  lw $3, 100($4)
2  sub $5, $6, $7
3  add $2, $3, $4
```

Listing 3.2: Code MIPS ad–hoc ré–ordonnancé et exécuté sur processeur out–of–order

Cependant, les modules matériels qui permettent de mettre en œuvre ces différents algorithmes sont très complexes. Dans ce sillage, les architectures des processeurs « out–of–order » sont très coûteuses en terme de coût de consommation énergétique mais aussi de surface occupée sur le silicium (HILY et SEZNEC [1999]). En effet, la consommation énergétique et la surface sont proportionnelles au carré du degré du superscalaire. Pour le monde des SoCs, et en particulier pour les applications de masse, ces coûts sont trop élevés pour être acceptables. De ce fait, l'utilisation les processeurs « out–of–order » n'est pas aujourd'hui la solution adoptée par STMicroelectronics, en particulier dans le domaine de l'informatique embarquée.

Par ailleurs, même avec l'utilisation de ces processeurs, certains gels du processeur peuvent continuer de subsister. On suppose que le code fourni dans le listing 3.3 s'exécute sur un processeur « Out–of–Order » qui dispose de trois tampons de ré–ordonnancement des instructions et que la lecture en mémoire principale effectuée à la *ligne 1* produit un échec dans le cache L_1 de données du processeur. Dans ce code il existe des dépendances de données entre les registres (*colorés*) des différentes instructions. Du fait de ces dépendances le processeur ne peut exécuter aucune des instructions qui suivent la lecture en mémoire principale. Il y aura plus de tampons de ré–ordonnancement disponibles après l'analyse de l'instruction *ligne 5*. Dans cette situation, le processeur est contraint d'être gelé et d'attendre la disponibilité des données lues en mémoire principale avant de poursuivre son traitement. Ces gels ne sont pas facile à éviter et à identifier par les modules matériels intégrés au processeur ou par un développeur pour l'optimisation de son logiciel. Ces processeurs ne fournissent donc pas toujours une solution rationnelle pour l'optimisation de la performance du logiciel dans un SoC et pour l'identification des instructions du programme gelées pendant l'exécution.

```
1  lw $1, 100($4)
2  add $5, $1, $3
3  sub $4, $5, $2
4  add $7, -15, $4
5  sub $5, $7, 5
```

Listing 3.3: Code MIPS ad–hoc pour l'apparition des cycles de gel sur processeur « Out–Of–Order »

3.4.2 Processeur SMT

Le SMT est une technique qui consiste à augmenter les capacités de parallélisation des processus légers (« threads ») sur un processeur. Comparé à un processeur non–SMT, un processeur SMT est partagé (le pipeline, les unités de calcul, les caches) entre plusieurs processus légers. Une implémentation de cette technique a été faite par Intel sous le nom d'hyper–threading (MARR et collab. [2002]). Schématiquement l'hyper–threading consiste à créer deux processeurs logiques mappés sur un seul cœur d'un processeur physique. Chaque processeur logique a ses propres registres de données et de contrôle, et partage les éléments du cœur du processeur physique. Ainsi, deux processus légers peuvent être entrelacés cycle par cycle ou quasiment sur le même cœur physique ce qui peut augmenter la performance du processeur (TIAN et collab. [2010]).

Cependant, les performances obtenues grâce à l'hyper–threading peuvent être significativement dégradées du fait des échecs qu'il peut générer dans les caches de données. Par exemple, un processeur qui dispose de 32 Kilo Octet (KO) de cache de données et deux processus légers qui opèrent sur 27 KO de données chacun, on peut distinguer deux cas possibles. 1) Les deux processus légers s'exécutent individuellement sur les deux processeurs logiques. Pour chacun des processus légers, le cœur du processeur physique est capable de copier la totalité de données sur lesquelles ils opèrent en mémoire cache. 2) Les deux processus s'exécutent au même moment. Ils nécessitent alors 54 KO de cache pour stocker les données. Les échecs dans le cache de données seront fréquemment observés. Ce qui fait que l'hyper–threading n'est pas toujours la solution pour l'optimisation de l'exécution d'un programme.

Par ailleurs, les processeurs SMT n'apparaissent pas aujourd'hui comme solution à STMicroelectronics pour ses SoCs. Rappelons que STMicroelectronics cherche à faire des circuits rapides, de la plus petite taille possible. Aussi, l'implantation de ce type de processeurs nécessite une surface de silicium importante. Le style de programmes que nous cherchons à optimiser dans cette thèse sont en général des programmes qui s'exécutent sur un seul processus léger (« mono–thread »). La solution offerte par le SMT pourrait être utile mais elle serait coûteuse pour la compagnie.

3.4.3 Compilation

La compilation est aujourd'hui une technologie mature qui propose de très nombreuses optimisations. La compilation est ainsi également une alternative pour l'optimisation de la performance du logiciel sur le matériel en réduisant les risques d'occurrences d'un gel lors de l'accès à une donnée. Par exemple, GROSSMAN [2000] propose de faire en sorte que le

compilateur réordonne les instructions de manière à ce qu'elles soient placées explicitement dans des unités fonctionnelles avec files d'attente. Bien que cela soit moins complexe qu'un processeur out–of–order, cela nécessite l'ajout de files d'attente matérielle, ce qui en pratique représente tout de même une surface assez importante sur le silicium.

Le compilateur peut procéder à la détection et au ré–ordonnancement des dépendances entre les instructions du programme dans le but d'éviter l'apparition des cycles de gels du processeur. Du fait des dépendances qui peuvent exister entre certaines instructions de lecture de données en mémoire et d'autres instructions qui utilisent au travers des registres ces données lues en mémoire, le compilateur peut s'arranger pour éloigner la lecture d'une donnée de son utilisation. Le compilateur fait au mieux pour garantir la non–apparition de cycles de gel du processeur car il n'a qu'une connaissance partielle de l'architecture du matériel sur lequel le programme sera exécuté. Bien qu'il soit aujourd'hui possible de fournir à un compilateur les informations sur l'architecture du matériel cible, en fournissant en option de compilation les tailles de caches, les latences d'accès à la mémoire ainsi que la bande passante du bus, le compilateur n'a qu'une idée abstraite du comportement du matériel sur lequel le programme sera exécuté. Il ne peut donc pas toujours prendre les bonnes décisions d'optimisation de la performance pour l'exécution d'un programme sur une architecture cible.

Par ailleurs, sur les programmes qui manipulent des structures de données de type tableau et les fonctions d'accès de ces tableaux sont des fonctions affines des itérateurs de boucles (boucles à contrôle affine (FEAUTRIER [1992], MORVAN [2013])), les compilateurs intègrent aujourd'hui des algorithmes complexes qui font des analyses polyédriques des différents accès aux données dans le programme compilé pour les optimiser et éviter ainsi la production des gels du processeur au moment de l'exécution. Le code 3.4 présente un exemple d'accès aux éléments d'une structure de données de type tableau régulièrement espacées sous la forme de fonctions affines des indices de boucles.

```
1  for(i = 0; i < 10; i++){
2    Z[i] *= Z[i+1] + Z[i-1];
3  }
```

Listing 3.4: Exemple de code avec accès à des adresses espacées sous la forme de fonctions affines des indices de boucles

En effet, les polyèdres permettent de traiter les problèmes d'ordonnancement, de placement et d'allocation pour les parties du programme à contrôle statique (IRIGOIN et collab. [2012]) en effectuant des transformations dans le code source du programme au moment de la compilation. Cependant, malgré les optimisations de placement des instructions du programme

de manière à garantir la meilleure utilisation du matériel sur lequel il sera exécuté, certaines instructions du programme se retrouvent très souvent gelées en attente de données, car le compilateur n'a pas connaissance des instructions susceptibles d'être gelées au moment de la compilation. Très souvent le compilateur ne connaît pas exactement les dépendances entre les instructions au moment de la compilation.

D'autres techniques de compilation basées sur les traces d'exécution du programme peuvent être utilisées pour considérer l'architecture du matériel cible sur lequel le programme s'exécute (TALBOT et collab. [2012], SCHNEIDER et GROSS [2006]).

Il existe d'autres moyens pour réduire l'impact du gel de processeur sur l'exécution d'un programme.

3.4.4 Techniques de pré–chargement de données

Une approche pour faire face au problème de gel du processeur est l'utilisation de mémoires caches pour y copier les données en avance de façon à ce qu'elles soient disponibles au moment où elles seront requises par le CPU. Bien que les caches soient utiles, ils ne sont pas forcément une bonne solution s'ils ne sont pas bien utilisés. Il existe diverses techniques qui permettent de masquer les cycles de gel du processeur dus à l'absence de données dans le cache L_1 du processeur. Le pré–chargement de données est une des stratégies utilisées pour rapprocher les données du processeur avant qu'elles ne soient accédées. Ceci peut également avoir des effets destructifs sur la performance du logiciel, tels que développés en détails par SAAVEDRA et collab. [1994] qui présentent un modèle de performance pour l'analyse de l'efficacité et des limites du pré–chargement de données. Leur modèle tient compte des effets liés au comportement du programme, au protocole de cohérence de cache et au modèle de consistance de la mémoire. La figure 3.7 extraite de BYNA et collab. [2008] montre cinq questions fondamentales que toutes les stratégies de pré–chargement doivent aborder.

A la question quoi précharger, il est question d'identifier le type de structure de données (tableau, pointeur, ...) à précharger pour prédire l'occurrence d'un échec dans le cache L_1 de données. Le pré–chargement doit être effectué au moment opportun.

— Si le pré–chargement est fait trop tôt : cela peut entrainer le remplacement de certaines lignes de caches utiles qui devront être utilisées dans un futur proche, entrainant une pollution du cache. Les lignes de cache pré–chargées peuvent aussi avoir une grande probabilité d'être remplacées avant qu'elles ne soient utilisées.

— Si le pré–chargement est fait trop tard : dans ce cas il est inutile et ne permet pas d'éviter les cycles de gel du processeur dus à l'indisponibilité des données dans le cache.

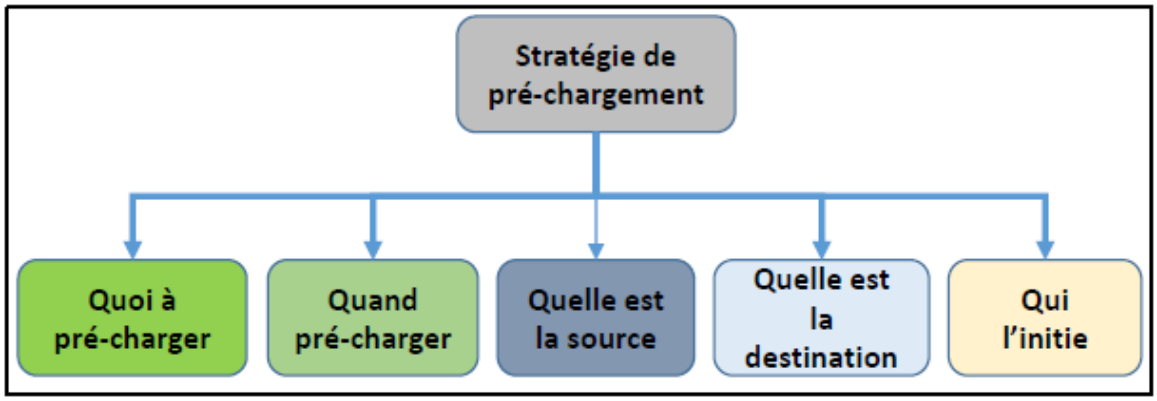

FIGURE 3.7: Questions fondamentales abordées par les stratégies de pré–chargement (BYNA et collab. [2008])

Les données pré–chargées proviennent fréquemment de la mémoire principale et sont destinées à remplir une ligne de cache (L_1 et/ou L_n en fonction des architectures). L'initiateur peut être le processeur, un DMA, ...

VANDERWIEL et LILJA [2000] donnent une définition du pré–chargement de données, et grâce à une étude, effectuent une comparaison entre différentes stratégies de pré–chargement de données dans le cadre d'un processeur mono-cœur. Le pré–chargement peut être dirigé par le compilateur, le matériel, le logiciel, ou par le matériel et le logiciel (hybride).

Le pré–chargement dirigé par matériel

Bien que notre travail porte sur un matériel figé qui ne contient pas de module matériel de pré–chargement, nous analyserons tout de même les approches matérielles car elles peuvent être source d'inspiration. Le pré–chargement est effectué par un module matériel qui scrute les différents accès qui génèrent fréquemment des échecs. Une méthode basée sur cette technique est le pré–chargement basé sur l'historique des accès (*history-based prediction*). C'est une stratégie de pré–chargement très souvent utilisée parmi les stratégies de pré–chargement de données dirigé par le matériel (LEE et collab. [2012]). Dans cette stratégie, une unité est utilisée pour prédire les références des futures données et émettre sous la base de la régularité d'apparition de certaines références, des instructions de pré–chargement de données. L'unité de pré–chargement observe l'historique des accès aux données ou alors l'historique des échecs dans le cache L_1 de données pour prédire les futurs accès que le processeur effectuera. Le processeur doit alors disposer à minima d'un module de pré–chargement dans lequel les algorithmes d'analyse de l'historique des accès sont implémentés sous forme de modules matériels pour faire fonctionner cette stratégie. Ceci est non-seulement coûteux en temps de développement, mais aussi en occupation de la surface du silicium.

L'exécution des instructions sur un module matériel lorsque le processeur est gelé ou inactif (« idle ») est une autre possibilité. GANUSOV et BURTSCHER [2005] proposent une méthodologie basée sur la pré–exécution des futures itérations de boucles sur un cœur inactif en utilisant le pré–chargement de données.

La stratégie d'analyse hors-ligne (Off-line analysis) est une autre approche de pré–chargement contrôlée par matériel. KIM et collab. [2002] proposent une méthode, où les modèles d'accès aux données sont analysés pour les points chauds du code qui sont fréquemment exécutés en se basant sur les modèles de prédictions de Markov.

Le pré–chargement de données dirigé par le matériel est une stratégie très importante pour des programmes à usage général, comme le prouve l'existence d'un concours international sur le pré–chargement dirigé par le matériel [2].

pré–chargement dirigé par le logiciel

Il existe plusieurs techniques pour réduire la latence du contrôleur mémoire en exécutant explicitement des instructions de pré–chargement qui rapprochent les données du processeur avant qu'elles ne soient effectivement accédées. Ces techniques de pré–chargement peuvent permettre de masquer la latence à la fois en lecture et en écriture en nécessitant relativement peu de support matériel.

La stratégie de pré–chargement dirigé par le logiciel peut utiliser des instructions de pré–chargement de données insérées soit automatiquement par le compilateur, soit manuellement par le développeur dans le code source du programme en se basant sur une analyse post-exécution. Aujourd'hui, de nombreux processeurs prennent en charge ces instructions de pré–chargement de données dans leur jeu d'instructions. Les compilateurs ou les développeurs de programmes peuvent les insérer soit comme instructions de pré–chargement de données spécifiques au processeur, soit comme routines spécifiques intégrées au compilateur.

Pour initier un tel pré–chargement, un mécanisme usuel utilisé consiste à faire exécuter par le processeur une instruction logiciel qui permet d'émettre explicitement une instruction *fetch* pour la donnée. Cette instruction spécifie notamment l'adresse de la donnée à copier dans le cache. Quand le *fetch* est exécuté, l'adresse de la donnée est transféré au contrôleur mémoire dans une transaction à travers l'interconnexion sans toutefois forcer le processeur à attendre une réponse à sa requête. Le contrôleur mémoire répond à la requête *fetch* de l'instruction de pré–chargement de la même manière qu'à une instruction normale

2. Le 2^{nd} championnat sur le pré–chargement de données remporté par Pierre MICHAUD, IRISA en Juin 2015

de lecture en mémoire, à l'exception près que la donnée pré–chargée n'est pas directement transmise au processeur mais seulement mise en cache.

Les instructions de pré–chargement initiées par logiciel peuvent avoir pour conséquence d'augmenter la fréquence des requêtes émises par le processeur en direction du système mémoire. Ceci nécessite que le contrôleur mémoire et l'interconnexion disposent d'une bande passante suffisante pour éviter de devenir un goulot d'étranglement du trafic (VANDERWIEL et LILJA [2000]). Dans notre cas, les plateformes choisies, représentatives d'une partie importante du marché des microcontrôleurs enfouis, disposent d'un seul processeur intégrant un seul cœur derrière une hiérarchie de cache ; les besoins de bande passante sont donc à priori moins importants que dans des systèmes multiprocesseurs. Néanmoins, des applications à forte bande passante (par exemple utilisant la vidéo) existent, et le trafic supplémentaire causé par les instructions de pré–chargement est à prendre en compte au même titre que les autres trafics, par un architecte SoC lors de la phase d'étude d'architecture. Cette étude peut exploiter une exécution de benchmarks logiciels sur une simulation de l'ossature du SoC.

D'autre part, les instructions de pré–chargement insérées par le développeur ou le compilateur augmentent le nombre d'instructions à exécuter par le processeur. Dans la figure 3.8, on observe qu'au lieu de traiter 3 lectures (*r1, r2, r3* en exécution normale), le processeur en traite finalement 6 (y compris les 3 lectures précédentes) parce que des instructions de pré–chargement sont insérées dans le code source du programme.

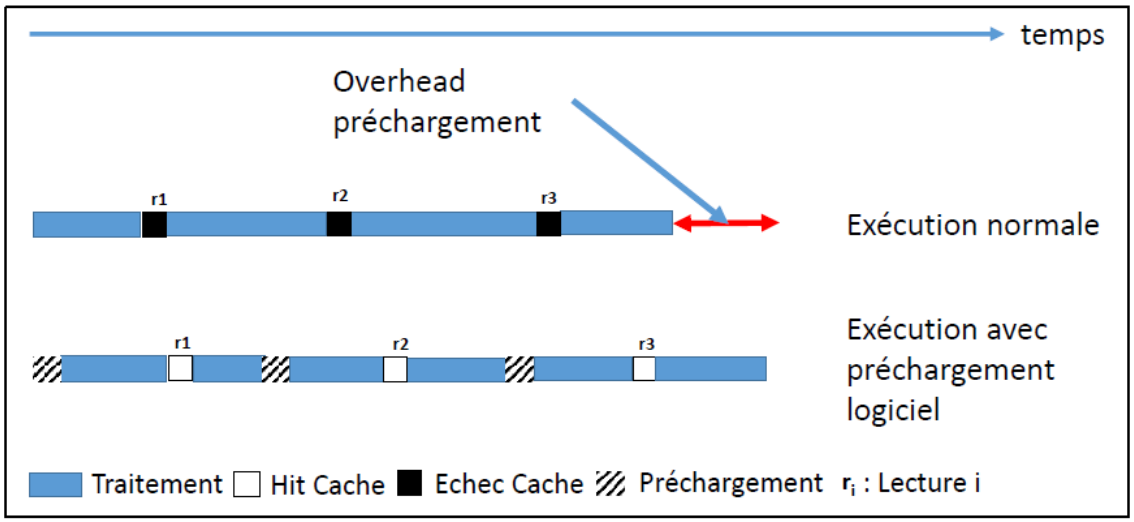

FIGURE 3.8: Surcoût de traitement créé par les instructions de pré–chargement logiciel ([VANDERWIEL et LILJA, 2000]).

La question se pose donc d'insérer ou non une instruction de pré–chargement, mais aussi de l'endroit d'insertion de cette instruction dans le programme source. En effet, tel que présenté dans NTAFAM et collab. [2016], une insertion mal placée résulte très souvent

en une dégradation de la performance du système avec, par exemple, l'augmentation des cycles de gel du processeur. Ceci du fait notamment qu'un pré–chargement mal placé peut évincer une ligne de cache utile (pollution du cache SESHADRI et collab. [2015]) ou causer de la contention sur le bus de communication par conflit avec d'autres trafics parce que les données pré–chargées n'ont pas été utiles et de nouvelles transactions sont donc obligées d'être émises pour transférer ces mêmes données.

En ce qui concerne le compilateur, il peut rajouter automatiquement (s'il estime que c'est nécessaire) les instructions de pré–chargement lors de la génération de l'exécutable du programme afin d'éviter les cycles de gel du processeur. Dans ce cas, les options de compilation adéquates sont passées au compilateur. Ces options informent le compilateur qu'il faudra générer et insérer automatiquement les instructions de pré–chargement de données en cache dans le code exécutable du programme. Pour des algorithmes simples tels que présentés dans le listing de code 3.5, on peut voir dans le code dés–assemblé équivalent 3.9 que les instructions de pré–chargement ont bien été générées et insérées automatiquement par le compilateur. On peut remarquer également les dépendances entre les instructions de pré–chargement et les instructions qui utilisent les résultats des pré–chargements.

```
1            int foo(int n, int a[]){
2              int i = 0;
3              for(i = 0; i < n; i++){
4                  a[i]+=a[i];
5              }
6             return 0;
7            }
```

Listing 3.5: Exemple simple de code sur lequel le compilateur génère et insère automatiquement des instructions de pré–chargement

Par ailleurs, cette insertion automatique des instructions de pré–chargement par le compilateur ne garantit pas toujours une meilleure performance en terme de temps d'exécution du programme optimisé. Pour le code 3.5 fourni en exemple, pour $n = 100000$ (pour que les données ne tiennent pas dans le cache de 32KO) le temps d'exécution de la version du code sans instructions de pré–chargement est considérablement plus faible que celui du code avec instructions de pré–chargement insérées automatiquement par le compilateur. Ceci confirme l'avertissement de la documentation de GCC (GCC-OPTIONS [2017]) qui stipule que les options de compilation qui contrôlent l'optimisation du logiciel par le pré–chargement de données en mémoire cache ne garantit pas une meilleure performance du logiciel en terme de temps d'exécution. C'est certainement dû au fait que les instructions ajoutées (calcul d'adresses, instructions de pré–chargement, ...) en plus pour faire fonction-

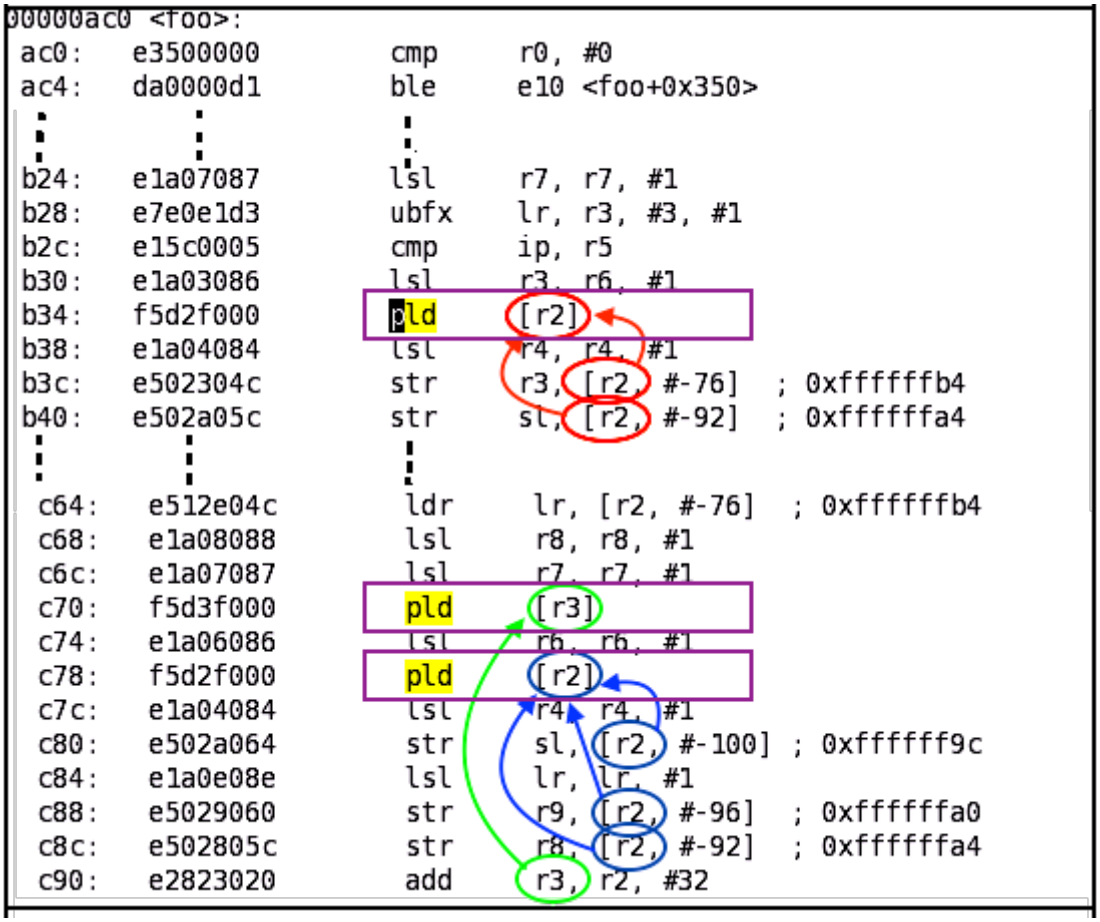

FIGURE 3.9: Code dé–assemblé pour architecture ARM de l'exemple 3.5 avec insertion automatique des instructions de pré–chargement par le compilateur GCC.

ner le pré–chargement nécessitent plusieurs cycles d'exécution qui ne sont pas compensés par les cycles de latence que le pré–chargement est censé permettre d'éviter en copiant les données en cache pour qu'elles soient disponibles au moment où elles seront requises par le CPU. De plus, les instructions ne sont probablement pas toujours insérées automatiquement au bon endroit pour garantir la disponibilité des données préchargées dans le cache au moment où elles seront requises par le processeur.

Sur d'autres programmes simples à contrôle dynamique comme par exemple le tri à bulles, dans lequel les accès aux éléments du tableau à trier se font au travers de boucles à contrôle affine, le compilateur n'est pas capable d'insérer automatiquement des instructions de pré–chargement dans le code source du programme (NTAFAM et collab. [2016]).

Pour ce qui est de l'insertion manuelle des instructions de pré–chargement de données dans le code source du programme, ceci impose aux développeurs un effort supplémentaire de travail. Cela peut nécessiter par exemple l'analyse post-exécution de traces d'accès aux données pour identifier des schémas d'accès. Cette stratégie présente ainsi deux défis ma-

jeurs.

— Premièrement, une analyse est nécessaire de la part du programmeur, ou (de préférence) du compilateur pour identifier les bons emplacements dans le code source du programme pour insérer les instructions de pré–chargement de données au bon endroit.

— Deuxièmement, il faut veiller à ce que les coûts additionnels liés au pré–chargement de données (instructions supplémentaires ajoutées, délais d'attente en mémoire plus longs, ...) n'éliminent pas les avantages de celui–ci.

Nous n'avons pas trouvé dans la littérature d'outils, ni de méthodologies clairement formalisées qui visent à 1) aider les développeurs dans cette analyse post–exécution pour identifier les instructions du programme qui produisent les gels du processeur et 2) donner des indices aux développeurs sur la possibilité de poursuivre l'optimisation de l'exécution du logiciel sur le matériel.

pré–chargement dirigé par le matériel et le logiciel (hybride)

Ces stratégies gagnent en popularité sur les processeurs avec le support multi-thread. Sur ces processeurs, les processus légers (threads) peuvent être utilisés pour exécuter des algorithmes complexes afin de prédire les futurs schémas d'accès aux données. Ces méthodes nécessitent le support du matériel pour exécuter les processus légers d'aide qui sont spécifiquement exécutés pour le pré–chargement des données. Elles ont également besoin d'un support logiciel pour se synchroniser avec le processus léger réel de calcul. Les stratégies de pré–chargement basées sur des processus légers analysent l'historique des accès aux données effectués par le processus de calcul, ou pré-exécutent des parties du code exécuté par le processus principal de calcul, qui traitent des flux de données importants (BAER et CHEN [1995]). Les stratégies de pré–chargement hybride basées sur l'historique (BYNA et collab. [2008]) analysent l'historique des accès pour prédire les futurs accès et anticiper ces accès.

Pour réduire les effets des cycles de gel du processeur sur l'exécution d'un programme, il peut être nécessaire d'identifier les instructions du programme qui sont à l'origine de la génération de ces cycles de gel qui contribuent au rallongement du temps global d'exécution du programme.

3.5 Identification des instructions allongeant le temps d'exécution

Il existe des approches s'appuyant uniquement sur des informations de profilage de l'exécution du programme et qui permettent d'identifier les sections du programme qui sont peu efficaces dans l'utilisation des capacités du matériel pour une optimisation future du programme. Par exemple, Nintendo Co., Ltd a déposé un brevet (RABIN [2017]) dans lequel il exploite les informations collectées par le biais du profilage statistique d'un programme de jeu vidéo.

gprof (GRAHAM et collab. [1982]) est un outil de profilage qui permet d'étudier le temps d'exécution de programmes implémentés en langage C/C++. Le code source du programme est instrumenté au moment de la compilation grâce à des options de compilation spécifiques afin de collecter les informations sur l'exécution du programme. **gprof** met très rapidement en lumière les parties du programme dans lesquelles l'exécution passe le plus de temps. Les statistiques sur le nombre d'appels et le temps d'exécution des différentes parties du programme qui nécessitent beaucoup de temps de traitement sont données sur la base des fonctions. Il est alors possible de concentrer les efforts d'optimisation sur les fonctions ayant des temps de traitement élevés.

La figure 3.10 présente un exemple d'informations que gprof peut fournir sur le temps d'exécution d'un programme. La première colonne montre bien que le programme passe plus de temps (environ 70 % du temps) dans la fonction *step*, et 30 % dans la fonction *nseq*. Les efforts d'optimisation peuvent alors être concentrés sur la fonction *step*.

```
Flat profile:

Each sample counts as 0.01 seconds.
 %     cumul.    self              self   total
time  seconds seconds    calls us/call us/call name
68.59    2.14    2.14 62135400    0.03    0.03 step
31.09    3.11    0.97   499999    1.94    6.22 nseq
 0.32    3.12    0.01                          main
```

FIGURE 3.10: Exemple d'informations extraites de l'outil de profilage d'exécution gprof

Cependant, l'outil ne fournit pas les informations sur les causes de la durée d'exécution. Si **gprof** permet d'évaluer les temps d'exécution, il ne dit rien sur les raisons de ces temps qui peuvent être dus à des questions algorithmiques ou d'implémentation. L'outil est utilisé

pour cibler une fonction, mais pas pour fournir des indices qui permettront à un développeur d'optimiser la fonction.

Par ailleurs, des outils comme **gcov** instrumentent le code source du programme de manière à pouvoir extraire la couverture du programme. Ils peuvent fournir des informations sur les segments de code fréquemment exécutés mais aucunes sur les temps d'exécution de ces segments de code. Il est alors possible d'identifier par exemple les instructions du code en langage de haut niveau (C/C++) qui n'ont pas été exécutées et se focaliser ainsi sur celles fréquemment exécutées.

Ces outils peuvent donner des indices sur les lignes de code qui ont une contribution forte au rallongement du temps d'exécution du programme, mais les informations fournies ne permettent pas de prendre des décisions sur la correction des cycles de gel du processeur et sur l'identification des instructions du programme qui engendrent ces cycles de gel. Il est également difficile à partir de ces informations de fournir à un développeur d'applications des indices sur les emplacements du code pour l'insertion manuelle des instructions de pré–chargement dirigé par le logiciel, et qui copieront les données dans les caches du système.

ANDERSON et collab. [1997] proposent *DIGITAL*, une infrastructure de profilage dynamique de l'exécution d'un programme sur matériel physique. Le fichier exécutable du programme n'est pas modifié et l'intégralité du système est profilée. Les données collectées incluent par exemple des informations sur l'exécution des programmes utilisateurs, sur les librairies partagées, ainsi que sur le noyau du système d'exploitation.

DIGITAL permet d'identifier de manière précise les raisons pour lesquelles un programme ou un système complet produit de mauvaises performances. Les données collectées sont stockées sur disque et analysées pour fournir pour chacune des instructions du programme, des informations sur les cycles de gel dans le pipeline du processeur engendrés par celles-ci. Quand une instruction génère des cycles de gel du processeur, l'outil identifie les raisons possibles. Par exemple, les échecs dans le cache, les erreurs de prédictions de branchement, ou une contention dans une unité fonctionnelle.

Cependant, l'outil *DIGITAL* collecte directement sur la plateforme physique et de manière intrusive les informations sur l'utilisation des capacités du matériel par le logiciel. Ceci peut ralentir l'exécution normale du programme. Par ailleurs, le fait de profiler directement la plateforme physique implique que l'outil intervient trop tard dans le flot de conception du SoC. De plus, un SoC physique ne dispose pas d'assez d'espace de stockage pour permettre le stockage des gigas d'octets de données de profilage. Car la limitation de l'espace de stockage est une contrainte forte.

Le tableau 3.1 présente un récapitulatif de quelques outils de profilage de l'exécution d'un programme ou d'un système. La colonne surcoût décrit à quel degré le profilage ralen-

tit l'exécution du programme cible. Un surcoût faible est défini arbitrairement comme étant inférieur à 20%.

La colonne *Portée* montre si l'outil de profilage se réduit à profiler une seule application ou l'ensemble de l'activité du système.

La colonne *Granularité* indique le niveau de détails de l'outil. Par exemple, l'outil *gprof* compte l'exécution d'une procédure, alors que l'outil *prof* va plus loin en fournissant le temps d'exécution de chacune des instructions du programme.

La colonne *Gel de processeur* indique la capacité et le degré de précision de l'outil à subdiviser le temps passé pour l'exécution d'une instruction en plusieurs composantes comme la latence due à un échec dans le cache L_1, le délai dû à une erreur de prédiction de branchement, ...

Outils	Surcoût	Portée	Granularité	Gels de processeur
pixie	Élevé	Application	Décompte instructions	Aucun
gprof	Élevé	Application	Décompte procédures	Aucun
jprof	Élevé	Application	Décompte procédures	Aucun
quartz	Élevé	Application	Décompte procédures	Aucun
MTOOL	Élevé	Application	Décompte instructions/temps	Pas précis
SimOS	Élevé	Système	Temps par instructions	Précis
SpeedShop(Pixie)	Élevé	Application	Décompte instructions	Aucune
VTune (dynamic)	Élevé	Application	Temps par instructions	Précis
DIGITAL	**Élevé**	**Application**	**Temps par instructions**	**Précis**
prof	Faible	Application	Temps par instructions	Aucune
iprobe	Élevé	Système	Temps par instructions	Pas précis
Morph	Faible	Système	Temps par instructions	Aucune
VTune (sampler)	Faible	Système	Temps par instructions	Pas précis
SpeedShop	Faible	Système	Temps par instructions	Pas précis
DCPI	Faible	Système	Temps par instructions	Précis

TABLEAU 3.1: Outils de profilage d'exécution de programme ou de système ([ANDERSON et collab., 1997])

On remarque dans ce tableau que les outils se divisent en deux principaux groupes. Le premier comprend les outils *pixie, gprof, jprof, quartz, MTOOL, SomOS* et l'analyseur dynamique d'Intel *VTune* (INTEL [2017]). Ces outils utilisent le support du compilateur, la modification du ou des fichiers exécutables, ou encore la simulation du programme pour collecter les données de profilage. Ils ont tous un surcoût élevé. C'est uniquement les outils basés sur la simulation qui fournissent des informations précises sur les causes et l'emplacement des gels du processeur.

Le second groupe utilise l'échantillonnage statistique pour collecter des données sur l'exécution d'un programme ou de l'ensemble d'un système. Dans ce groupe, le surcoût produit par l'instrumentation est faible pour tous les outils à l'exception de *iprobe*. *iprobe* réutilise les fonctionnalités de profilage du compilateur GCC. C'est probablement une des raisons pour lesquelles son surcoût est élevé car le compilateur GCC instrumente le code source d'un programme pour profiler son exécution. Cette instrumentation est coûteuse comme c'est le cas avec **gprof**.

3.6 Métriques d'évaluation de l'effet du gel de processeur

3.6.1 Temps d'exécution

Le temps d'exécution représente le temps écoulé entre le début d'exécution d'un programme et la fin d'exécution de celui–ci. Il est question ici du temps passé par le programme sur le processeur, indépendamment des autres programmes qui se seraient exécutés pendant le même intervalle de temps. Il permet d'effectuer une comparaison entre différentes versions d'exécution du même programme sur les mêmes données, la même architecture matérielle, la même exactitude d'exécution ... Un programme qui s'exécute en générant beaucoup de cycles de gel du processeur aura un temps d'exécution plus long comparé à sa version qui en génère moins. Le temps d'exécution permet alors d'évaluer la réduction des cycles de gel du processeur et d'estimer la vitesse (instruction/sec) d'exécution du programme.

3.6.2 Utilisation des mémoires caches (L_1...L_n)

L'utilisation des mémoires caches de données est un facteur qu'il est possible d'évaluer pour avoir l'effet des cycles de gel du processeur. Pour mémoire, les gels du processeur que nous étudions dans cette thèse sont ceux générés par l'indisponibilité de données dans le cache L_1 de données du processeur. L'utilisation des caches est donc une métrique susceptible d'influencer l'apparition des cycles de gel du processeur. Pour évaluer l'utilisation des caches, plusieurs critères peuvent être étudiés. Le nombre d'accès, le taux de réussite, le pourcentage de lignes de cache éjectées, la politique de remplacement, l'architecture du cache, ... Nous nous concentrerons dans cette thèse sur les taux de réussite et d'échec, car ce sont les informations sur lesquelles nous disposons d'une marge de manœuvre et qu'il est facile d'extraire et d'évaluer sur les différentes architectures utilisées durant cette thèse. Dans la relation 3.1, le taux de réussite représente le ratio entre le nombre d'accès pour les-

quels la donnée se trouve dans le cache et le nombre total d'accès effectués dans le même cache. Le taux d'échecs est alors donné par la relation 3.2.

$$\mathrm{T}aux\_réussite = \frac{\mathrm{N}ombre\_accès_{trouvé}}{\mathrm{N}ombre\_total\_accès} \tag{3.1}$$

$$\mathrm{T}aux\_d'échec = 1 - \mathrm{T}aux\_réussite \tag{3.2}$$

3.7 Conclusion partielle

Dans ce chapitre, nous avons présenté un ensemble de stratégies qui permettent de réduire l'effet des cycles de gel du processeur dus à l'indisponibilité des données dans le cache L_1 de données du processeur. Nous avons également évalué un ensemble d'outils qui permettent de profiler l'exécution d'un programme et d'analyser les données profilées pour produire des informations sur les cycles de gel du processeur avec suffisamment de détails pour permettre à un développeur d'optimiser l'utilisation des capacités du matériel. Finalement nous avons présenté quelques métriques d'estimation de la performance du logiciel sur le matériel en s'intéressant au temps d'exécution de celui–ci et à l'utilisation des mémoires caches de données.

Cependant, nous remarquons qu'aujourd'hui dans la littérature, nous n'avons pas pu identifier d'outils ni de méthodologies qui permettent d'avoir une vision des cycles de gel du processeur dus à l'absence de données en mémoires caches.

Dans la suite du document, nous présentons deux approches qui permettent d'obtenir plus d'informations sur les cycles de gel du processeur dus à l'absence de données dans les caches, ainsi que des informations sur les instructions qui les engendrent. Nous proposons ensuite un début de solution pour la réduction de ces cycles de gel du processeur en utilisant le pré–chargement de données dirigé par le logiciel.

3.8 Références

ANDERSON, J. M., L. M. BERC, J. DEAN, S. GHEMAWAT, M. R. HENZINGER, S.-T. A. LEUNG, R. L. SITES, M. T. VANDEVOORDE, C. A. WALDSPURGER et W. E. WEIHL. 1997, «Continuous profiling : Where have all the cycles gone ?», *ACM Trans. Comput. Syst.*, vol. 15, nº 4, doi :10.1145/265924.265925, p. 357–390, ISSN 0734-2071. URL http://doi.acm.org/10.1145/265924.265925. ix, 44, 45

ARM. 2017, «Implementation of Accurate Models of Arm IP», https://developer.arm.

com/products/system-design/cycle-models. [En ligne; dernier accès le 8 Août 2017]. 30

ARM-CYCLE-MODELS-STUDIO. 2017, «Cycle Model Studio», https://developer.arm.com/products/system-design/cycle-models/cycle-model-studio. [En ligne; dernier accès le 16 Décembre 2017]. 29

BAER, J.-L. et T.-F. CHEN. 1995, «Effective hardware-based data prefetching for high-performance processors», *IEEE Trans. Comput.*, vol. 44, nº 5, doi :10.1109/12.381947, p. 609–623, ISSN 0018-9340. URL http://dx.doi.org/10.1109/12.381947. 42

BYNA, S., Y. CHEN et X. H. SUN. 2008, «A taxonomy of data prefetching mechanisms», dans *2008 International Symposium on Parallel Architectures, Algorithms, and Networks (i-span 2008)*, ISSN 1087-4089, p. 19–24, doi :10.1109/I-SPAN.2008.24. vi, 36, 37, 42

FEAUTRIER, P. 1992, «Some efficient solutions to the affine scheduling problem. i. one-dimensional time», *International Journal of Parallel Programming*, vol. 21, nº 5, doi : 10.1007/BF01407835, p. 313–347, ISSN 1573-7640. URL https://doi.org/10.1007/BF01407835. 35

GANUSOV, I. et M. BURTSCHER. 2005, «Future execution : A hardware prefetching technique for chip multiprocessors», dans *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, IEEE Computer Society, Washington, DC, USA, ISBN 0-7695-2429-X, p. 350–360, doi :10.1109/PACT.2005.23. URL http://dx.doi.org/10.1109/PACT.2005.23. 38

GCC-OPTIONS. 2017, «Options that control optimization», https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. [En ligne; dernier acc'es le 03 Octobre 2017]. 40

GERSTLAUER, A. et G. SCHIRNER. 2010, «Platform modeling for exploration and synthesis», dans *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, ASPDAC '10, IEEE Press, Piscataway, NJ, USA, ISBN 978-1-60558-837-7, p. 725–731. URL http://dl.acm.org/citation.cfm?id=1899721.1899889. 32

GRAHAM, S. L., P. B. KESSLER et M. K. MCKUSICK. 1982, «Gprof : A call graph execution profiler», dans *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, ACM, New York, NY, USA, ISBN 0-89791-074-5, p. 120–126, doi :10.1145/800230.806987. URL http://doi.acm.org/10.1145/800230.806987. 43

GROSSMAN, J. P. 2000, «Cheap out-of-order execution using delayed issue», dans *Proceedings 2000 International Conference on Computer Design*, ISSN 1063-6404, p. 549–551, doi :10.1109/ICCD.2000.878338. 34

HILY, S. et A. SEZNEC. 1999, «Out-of-order execution may not be cost-effective on processors featuring simultaneous multithreading», dans *Proceedings Fifth International Symposium on High-Performance Computer Architecture*, p. 64–67, doi :10.1109/HPCA.1999.744331. 33

INTEL, V. 2017, «Intel vtune amplifier», https://software.intel.com/en-us/intel-vtune-amplifier-xe. [En ligne ; dernier accès le 30 Octobre 2017]. 45

IRIGOIN, F., M. AMINI, C. ANCOURT, F. COELHO, B. CREUSILLET et R. KERYELL. 2012, «Polyèdres et compilation», *Technique et Science Informatiques*, vol. 31, nº 8-10, doi :10.3166/tsi.31.987-1019, p. 987–1019. URL https://doi.org/10.3166/tsi.31.987-1019. 35

KIM, J., K. V. PALEM et W.-F. WONG. 2002, «A framework for data prefetching using offline training of markovian predictors», dans *Proceedings. IEEE International Conference on Computer Design : VLSI in Computers and Processors*, ISSN 1063-6404, p. 340–347, doi : 10.1109/ICCD.2002.1106792. 38

LEE, J., H. KIM et R. VUDUC. 2012, «When prefetching works, when it doesn't, and why», *ACM Trans. Archit. Code Optim.*, vol. 9, nº 1, doi :10.1145/2133382.2133384, p. 2 :1–2 :29, ISSN 1544-3566. URL http://doi.acm.org/10.1145/2133382.2133384. 37

MARR, D. T., F. BINNS, D. L. HILL, G. HINTON, D. A. KOUFATY, A. J. MILLER et M. UPTON. 2002, «Hyper-Threading technology architecture and microarchitecture», *Intel Technology Journal*, vol. 6, nº 1. URL http://download.intel.com/technology/itj/2002/volume06issue01/art01_hyper/vol6iss1_art01.pdf. 34

MORVAN, A. 2013, *Synthesis of pipelined architectures using the polyhedral model*, Theses, École normale supérieure de Cachan - ENS Cachan. URL https://tel.archives-ouvertes.fr/tel-00913692. 35

NTAFAM, P. N., E. PAIRE, A. CLOUARD et F. PETROT. 2016, «Simulation driven insertion of data prefetching instructions for early software-on-soc optimization», dans *Proceedings of the 27th International Symposium on Rapid System Prototyping : Shortening the Path from Specification to Prototype*, RSP '16, ACM, New York, NY, USA, ISBN 978-1-4503-4535-4, p. 93–99, doi :10.1145/2990299.2990315. URL http://doi.acm.org/10.1145/2990299.2990315. 39, 41

RABIN, S. 2017, «Method and apparatus for visualizing and interactively manipulating profile data», URL https://www.google.com/patents/US9576382, uS Patent 9,576,382. 43

SAAVEDRA, R. H., W. MAO et K. HWANG. 1994, «Performance and optimization of data prefetching strategies in scalable multiprocessors», *J. Parallel Distrib. Comput.*, vol. 22, nº 3, doi :10.1006/jpdc.1994.1102, p. 427–448, ISSN 0743-7315. URL http://dx.doi.org/10.1006/jpdc.1994.1102. 36

SCHERRER, A., A. FRABOULET et T. RISSET. 2006, «A generic multi-phase on-chip traffic generation environment», dans *Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors*, ASAP '06, IEEE Computer Society, Washington, DC, USA, ISBN 0-7695-2682-9, p. 23–27, doi :10.1109/ASAP.2006.5. URL http://dx.doi.org/10.1109/ASAP.2006.5. 25

SCHNEIDER, F. et T. R. GROSS. 2006, *Using Platform-Specific Performance Counters for Dynamic Compilation*, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-540-69330-7, p. 334–346, doi :10.1007/978-3-540-69330-7_23. URL https://doi.org/10.1007/978-3-540-69330-7_23. 36

SESHADRI, V., S. YEDKAR, H. XIN, O. MUTLU, P. B. GIBBONS, M. A. KOZUCH et T. C. MOWRY. 2015, «Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks», *ACM Trans. Archit. Code Optim.*, vol. 11, nº 4, doi :10.1145/2677956, p. 51 :1–51 :22, ISSN 1544-3566. URL http://doi.acm.org/10.1145/2677956. 40

SIMONSON, L. J. et L. HE. 2005, *Micro-architecture Performance Estimation by Formula*, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-540-31664-0, p. 192–201, doi :10.1007/11512622_21. URL https://doi.org/10.1007/11512622_21. 25

SMITH, J. E. et A. R. PLESZKUN. 1988, «Implementing precise interrupts in pipelined processors», *IEEE Trans. Comput.*, vol. 37, nº 5, doi :10.1109/12.4607, p. 562–573, ISSN 0018-9340. URL https://doi.org/10.1109/12.4607. 32

TALBOT, J., Z. DEVITO et P. HANRAHAN. 2012, «Riposte : A trace-driven compiler and parallel vm for vector code in r», dans *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, ACM, New York, NY, USA, ISBN 978-1-4503-1182-3, p. 43–52, doi :10.1145/2370816.2370825. URL http://doi.acm.org/10.1145/2370816.2370825. 36

TIAN, Y., C. LIN et K. HU. 2010, «The performance model of hyper-threading technology in intel nehalem microarchitecture», dans *2010 3rd International Conference on Advan-*

ced Computer Theory and Engineering(ICACTE), vol. 3, ISSN 2154-7491, p. V3–379–V3–383, doi :10.1109/ICACTE.2010.5579564. 34

TOMASULO, R. M. 1967, «An efficient algorithm for exploiting multiple arithmetic units», *IBM J. Res. Dev.*, vol. 11, n° 1, doi :10.1147/rd.111.0025, p. 25–33, ISSN 0018-8646. URL http://dx.doi.org/10.1147/rd.111.0025. 32

VANDERWIEL, S. P. et D. J. LILJA. 2000, «Data prefetch mechanisms», *ACM Comput. Surv.*, vol. 32, n° 2, doi :10.1145/358923.358939, p. 174–199, ISSN 0360-0300. URL http://doi.acm.org/10.1145/358923.358939. vi, 37, 39

ZHU, Q., R. OISHI, T. HASEGAWA et T. NAKATA. 2005, «Integrating uml into soc design process», dans *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*, DATE '05, IEEE Computer Society, Washington, DC, USA, ISBN 0-7695-2288-2, p. 836–837, doi :10.1109/DATE.2005.186. URL http://dx.doi.org/10.1109/DATE.2005.186. 31

Chapitre 4

Exploitation des modèles précis au niveau du cycle pour l'identification et la correction des gels du processeur

Sommaire

4.1 Introduction

Estimer et optimiser la performance du logiciel une fois l'ossature de l'architecture du SoC définie demeure un défi important en particulier à cause de la complexité atteinte par l'intégration du logiciel dans les nouveaux produits. Sans perte de généralité et dans le but d'être concret, nous utilisons dans la suite du document des exemples basés sur les processeurs ARM. Ces architectures disposent de pré–chargement de données dirigé par le logiciel. Cette fonctionnalité permet de précharger dans les caches du processeur une valeur qui provient de la mémoire principale. Ce mécanisme peut être explicitement contrôlé par le programmeur ou le compilateur en émettant des instructions de pré–chargement.

Ces instructions permettent d'informer le processeur qu'il aura besoin de la donnée référencée dans un futur proche (typiquement quelques cycles d'exécution). Ainsi, le processeur peut émettre une requête permettant de charger à l'avance dans ses caches (L_1 et / ou L_n en fonction des architectures) la ligne de cache contenant la donnée référencée. Ces instructions de pré–chargement s'exécutent en parallèle des traitements normaux effectués par le processeur, ce qui laisse le temps au système mémoire de transférer vers les caches les données requises. Idéalement, ces requêtes de pré–chargement devraient se terminer juste à temps pour que le processeur puisse utiliser les données copiées en cache au moment nécessaire, ce qui permettrait d'éviter ainsi le gel du processeur.

Afin de pouvoir insérer manuellement et correctement les instructions de pré–chargement dans le code source d'un programme, et éviter ou réduire l'apparition des cycles de gel lors de l'exécution de ce programme, la figure 4.1 résume le travail développé dans ce chapitre. Elle présente les différentes étapes du flot itératif proposé. Ce flot s'appuie sur l'observabilité fournie par les plateformes virtuelles précises au niveau du cycle pour la correction des cycles de gel du processeur produits par l'exécution du logiciel sur l'ossature du SoC en cours de développement. La méthode se focalise sur la correction des gels du processeur dus à la lecture d'une donnée en mémoire principale, c'est–à–dire en pratique au chargement d'une ligne de cache. Le flot commence lorsque les fichiers source du programme implémenté en langage de haut niveau (C/C++ par exemple) sont disponibles. Le programme est ensuite compilé en croisé (cross–compilation) pour produire un fichier exécutable sur l'architecture cible. Le fichier exécutable est alors exécuté sur une plateforme de simulation précise au niveau du cycle ; simulation durant laquelle les informations de profilage de la microarchitecture du processeur sont générées de manière non-intrusive : la mesure ne modifie pas l'exécution. Ces informations de profilage sont ensuite analysées manuellement pour détecter les gels du processeur, identifier les instructions qui produisent ces gels et identifier par la suite les adresses accédées prédictibles. Enfin les instructions de pré–chargement sont

insérées manuellement dans le code source du programme.

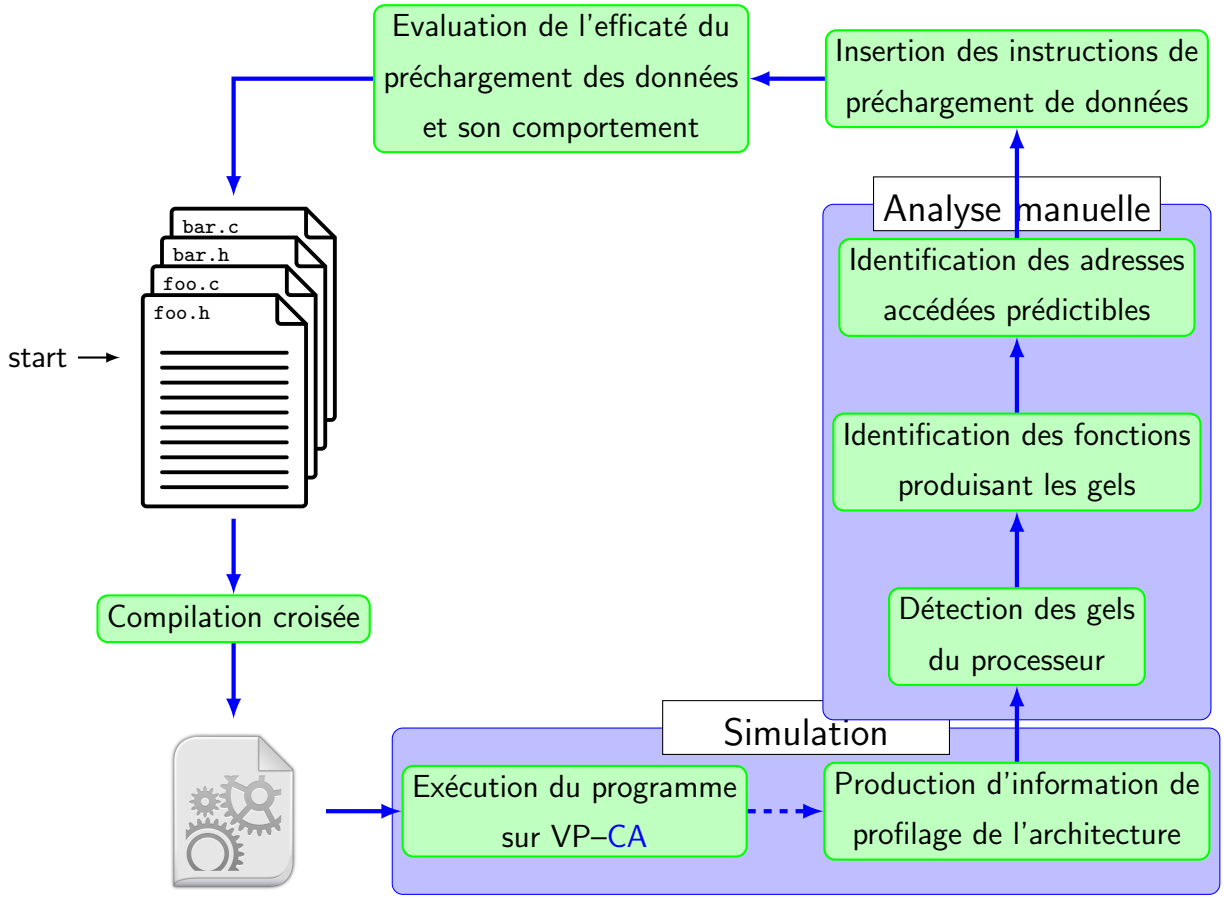

FIGURE 4.1: Flot itératif proposé pour l'identification des accès pertinents et l'insertion dans le code source du programme des instructions de pré–chargement de données en mémoires cache.

Le flot se termine par l'évaluation de l'efficacité du pré–chargement. La boucle du flot est répétée jusqu'à l'obtention d'une meilleure utilisation du cache L_1 et/ou L_n de données (en fonction des architectures de hiérarchie mémoire) pour corriger les gels du processeur, résultant en un niveau de performance applicative suffisant pour l'objectif de l'ingénieur pour son projet. Dans certaines architectures de processeur, les requêtes de chargement d'une ligne de cache copient directement le bloc de données dans le cache de données le plus proche du processeur (L_1). D'autres copient le bloc de données dans le dernier niveau de cache (Last Level Cache (LLC)). Dans cette thèse, avec le système mémoire d'un processeur de la famille Cortex Ax, les blocs de données sont directement copiés dans le cache L_1 de données. L'évaluation de l'influence de la meilleure utilisation du cache sur la performance du logiciel se concentre donc sur le cache L_1.

4.2 Production d'informations de profilage de la microarchitecture

Étant générés à partir du RTL, les modèles CA fournissent des informations aux travers des registres qui monitorent la performance (PMU) du processeur et il n'est pas possible d'accéder à des informations non prévues par le fournisseur de l'outil de conception assistée par ordinateur effectuant cette transformation (RTL –> CA). Ainsi, au cours de l'exécution du programme sur la plateforme virtuelle précise au niveau du cycle, les informations disponibles sur l'utilisation des éléments de la microarchitecture du processeur sont exclusivement celles qui sont présentes dans les registres de PMU du processeur. Ces informations sont extraites de manière non-intrusive car prévue par l'outil. Ces registres fournissent pour chaque cycle d'exécution de l'application plusieurs informations, dont le nombre d'échecs d'accès au cache de données, les cycles de gel du processeur, l'arbre d'exécution du programme (les appels de fonctions, le temps d'exécution de chaque fonction, ...), ainsi que le type (lecture, écriture) et la durée de chaque transaction que le processeur émet en direction de la mémoire principale. Certains évènements internes au CPU sont également profilés. On peut citer le nombre d'instructions décodées, le nombre d'échecs de données au niveau des caches L_1 et L_2, les adresses (virtuelles et physiques) accédées pour chaque requête de lecture de données en DRAM. Il est ainsi possible, à partir de ces informations, de trouver le nombre total de cycles d'exécution du programme exécuté.

Il est à noter que dans notre cas, l'exécution de l'application est effectuée en « bare metal » c'est–à– dire directement sur le matériel sans utiliser un système d'exploitation. Dans cette situation, le programme en code machine est chargé par le simulateur en mémoire principale et le modèle du processeur boote directement dans une fonction qui permet de configurer la plateforme simulée, puis d'exécuter le programme testé. Il est à noter également que le niveau de détails des informations fournies est très dépendant du processeur. Par exemple un processeur Intel ne fournira pas les mêmes informations qu'un processeur ARM. Et il en est de même au sein d'une même famille de processeurs. Un processeur ARM Cortex–A9 ne fournira pas les mêmes détails qu'un processeur ARM Cortex–A53.

Dans la figure 4.2, on peut observer un exemple qui présente un échantillon d'informations qu'il est possible d'extraire des registres de PMU du processeur ARM Cortex–A9 monocœur. On peut observer le nombre de cycles de gel du processeur dus à l'absence de données ou d'instructions dans les caches L_1 dédiés respectivement aux données et aux instructions, le nombre d'accès de données effectués dans le cache de L_1 de données du processeur, les évictions de données, le nombre d'échecs et les histogrammes qu'il est possible de

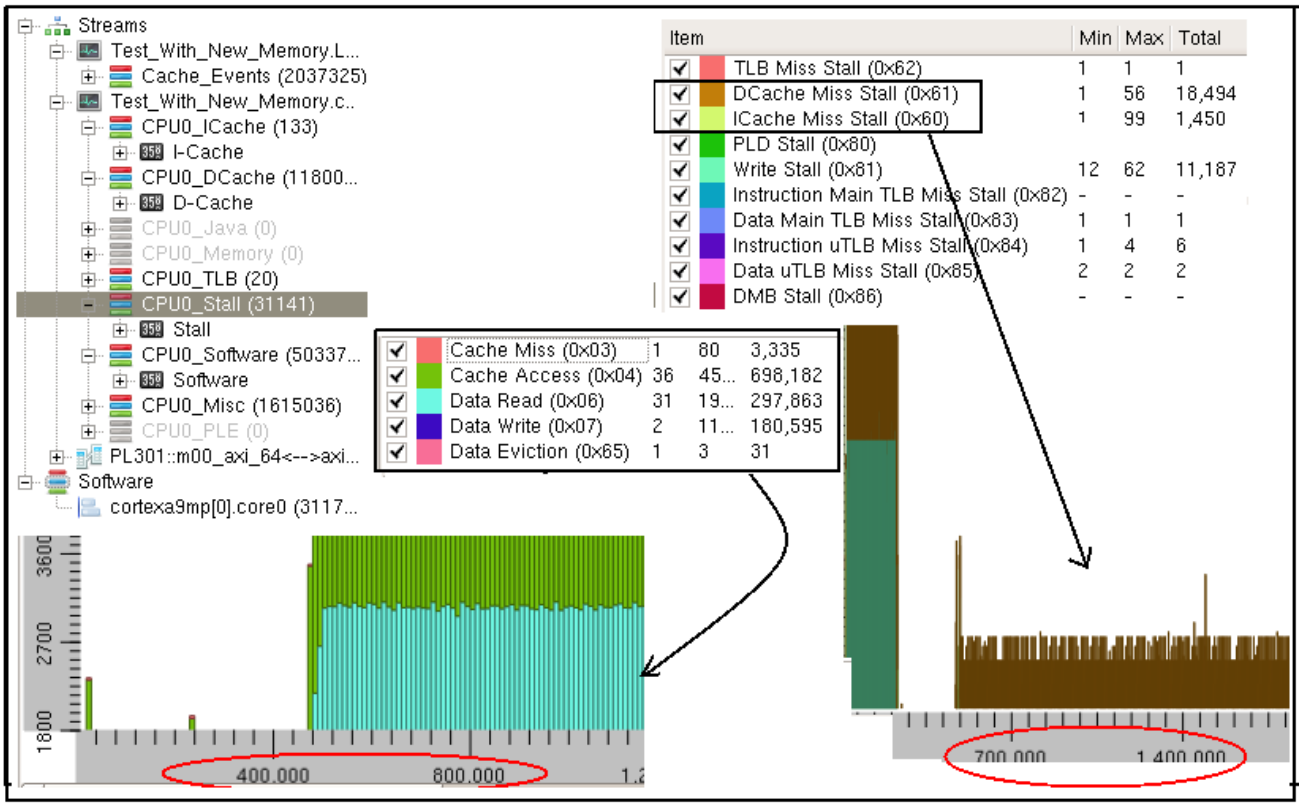

FIGURE 4.2: Exemple d'informations extraites des registres de monitoring du processeur

produire pour visualiser l'évolution de chacune des informations collectées dans le temps (cycles d'exécution du programme testé). Il est ainsi possible d'avoir une répartition dans le temps de l'occurrence de chacun des évènements monitorés.

4.3 Analyse des accès de lecture de données en mémoire

4.3.1 Détection des cycles de gels du processeur

Les informations de profilage de la microarchitecture permettent de savoir ce qui s'est passé à chaque cycle d'exécution du programme, grâce à l'exploitation, l'analyse et la corrélation des différentes informations extraites de manière non-intrusive des registres de PMU du processeur. À partir de ces informations, il est possible de détecter les moments durant lesquels le processeur a été gelé à cause de l'indisponibilité de données dans les mémoires caches, et d'estimer le nombre total de cycles consommés par les différents gels du processeur pendant la simulation.

Par ailleurs pour chaque gel du processeur une étude du comportement du bus situé entre le dernier niveau de cache et la mémoire principale est effectuée. Cette étude permet

de savoir si le bus était occupé ou libre au moment où le gel s'est produit. Pour cela, nous nous appuyons sur la trace des différents signaux qui constituent le bus. A partir de cette trace, les transactions émises par le CPU en direction du contrôleur DRAM sont reconstituées pour extraire les différentes transactions stockées dans la trace, et les statistiques sur l'utilisation du bus, notamment la bande passante, la latence moyenne associée à chaque transaction, le nombre de transactions de lecture, ...

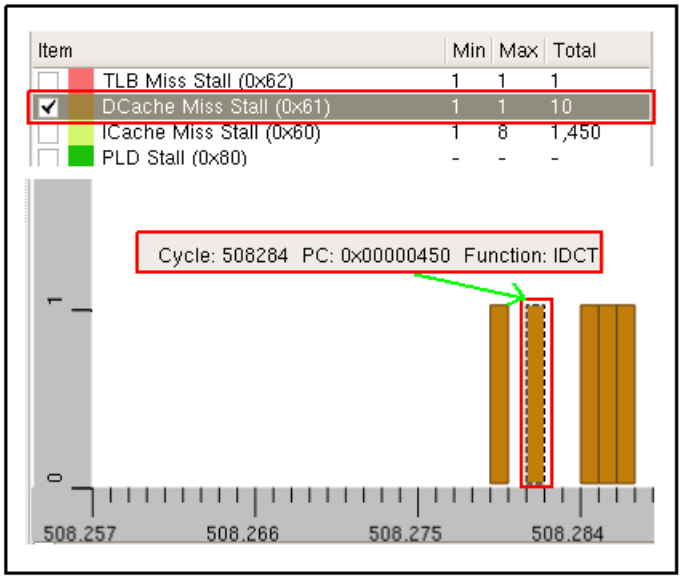

FIGURE 4.3: Zoom sur les détails fournis par les registres de PMU sur un gel du processeur

La figure 4.3 présente quelques détails que l'on peut obtenir sur l'occurrence d'un gel de processeur à partir des informations extraites des registres du PMU du processeur. On observe que pour le gel qui survient au cycle 508284, la fonction en cours d'exécution est la fonction « IDCT () », qui correspond au PC $0x00000450$. Il est donc possible d'associer à un gel du processeur, le cycle auquel il est survenu et le PC (correspondant à la ligne du code désassemblé) de la fonction en cours d'exécution au moment où le gel survient.

4.3.2 Identification des fonctions et/ou instructions produisant les gels du processeur

Pour identifier les cycles de gel du processeur, nous identifions dans un premier temps à quels cycles les échecs de lecture de données dans le cache L_1 se produisent. Ensuite, nous corrélons ces cycles identifiés aux cycles de gels du processeur détectés. Les cycles qui correspondent aux échecs de lecture dans le cache de données L_1 sont associés au diagramme d'exécution du programme. Un exemple de diagramme d'exécution du programme sur processeur ARM Cortex-A9 est présenté sur la figure 4.4. On observe dans ce diagramme que les

fonctions *IDCT, idct_1d, SUB, ADD, rot, ...* sont exécutées. Pour chaque cycle d'exécution, il est possible d'identifier précisément la fonction exécutée correspondante. Au début du diagramme, la fonction *idct_1d()* est exécutée. Ensuite, on observe des aller-retours entre les fonctions *idct_1d(), SUB() et ADD()*. Puis la fonction *IDCT ()* est exécutée. On peut remarquer que cette fonction est bien en cours d'exécution lorsque le gel du processeur survient au cycle 508284 tel que présenté dans la figure 4.3.

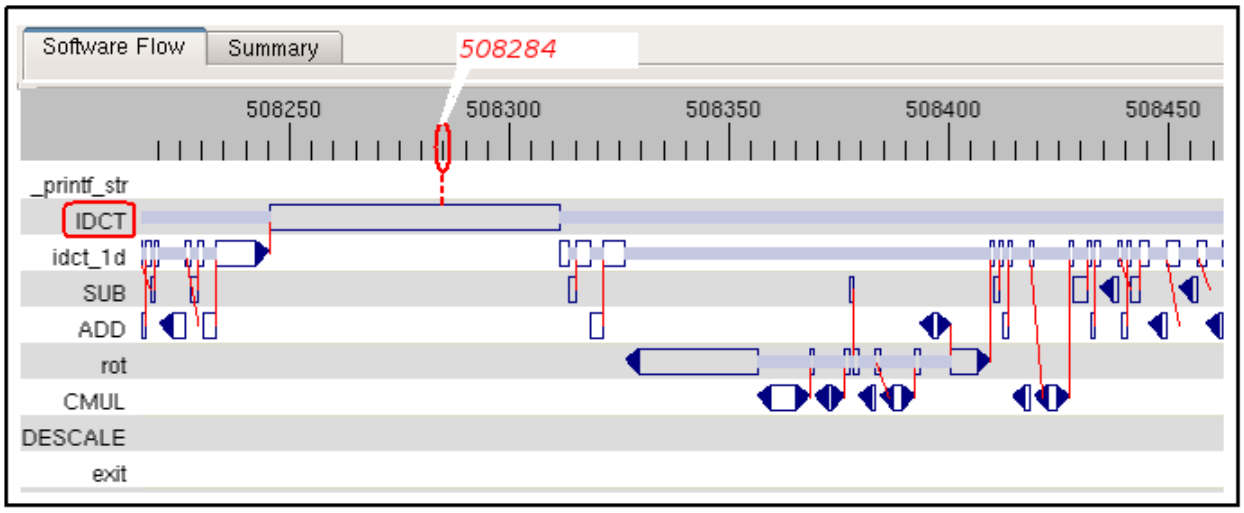

FIGURE 4.4: Exemple de diagramme d'exécution d'un programme produit grâce à l'observabilité fournie par les plateformes virtuelles précises au niveau du cycle

Chaque cycle d'échec dans le cache L_1 de données est associé à une valeur du registre PC qui correspond au moment où l'échec est survenu. Cette valeur du registre PC correspond à la première instruction de la fonction dans laquelle l'échec est survenu. Ceci permet dans une premier temps d'associer une fonction à chaque échec généré dans le cache L_1 de données. Cette association est faite en s'appuyant sur l'algorithme 1. Il est possible à partir des informations extraites des registres PMU du processeur de construire une liste des différents échecs survenus dans le cache L_1 de données, ainsi que la liste des différentes occurrences d'appels de chacune des fonctions du programme testé. Pour chacun des échecs, si le cycle auquel un $echec_i$ survient est compris entre le cycle de début et le cycle de fin de l'appel d'une fonction ($Occurr_{k\_}fct_j \rightarrow cycle_{debut} \leq echec_i \rightarrow cycle_{occur} \geq Occurr_{k\_}fct_j \rightarrow cycle_{fin}$), $l'occurence_k$ de la $fonction_j$ appelée était en cours d'exécution au moment où l'$echec_i$ est survenu. Ainsi, les fonctions du programme (typiquement en C) associées à chacun des échecs sont identifiées. Il est alors possible de localiser les fonctions du programme qui produisent statistiquement le plus d'échecs dans le cache L_1 de données. Plus une fonction génère des échecs, plus la contribution de cette fonction au rallongement du temps d'exécution du programme testé est important.

Il est à noter qu'un gel de processeur dû à l'indisponibilité de la donnée dans les registres cibles de l'instruction qui utilisent les résultats d'une lecture de données en DRAM est lié à un échec dans le cache L_1 de données du processeur. Les fonctions qui produisent statistiquement beaucoup plus d'échecs dans le cache de données sont donc celles susceptibles de produire le plus de gel du processeur.

---

1 **struct** {
2 | $Cycle_{occur}$: cycle auquel l'échec est survenu ;
3 | $Valeur_{PC\_fct}$: Valeur du PC de la fonction exécutée ;
4 } *Echec*;
5 **struct** {
6 | L_{temps_appel} : Liste des temps d'appel de la fonction ;
7 | PC_{fct} : Valeur du PC de la premiere instruction de la fonction exécutée ;
8 } *fct*;
9 **struct** {
10 | $cycle_{debut}$: Cycle de début d'exécution d'une fonction ;
11 | $cycle_{fin}$: Cycle de fin d'exécution d'une fonction;
12 } *temps*;

Rappelons qu'une transaction de lecture d'une ligne de cache est émise sur le bus suite à l'indisponibilité de la ligne de cache lue dans les différents niveaux de cache de la hiérarchie mémoire. Pendant le traitement de la transaction, le processeur (superscalaire) poursuit l'exécution du programme testé jusqu'à l'instruction qui utilise les registres cibles dans lesquelles la ou les données lues doivent être chargées. Si cette instruction qui utilise la ou les données lues est exécutée quand la transaction n'est par encore terminée, le processeur est gelé sur cette instruction jusqu'à la disponibilité de la ou les données dans les registres utilisés par cette instruction, c'est–à–dire jusqu'à la fin de la transaction de lecture. L'instruction gelée est alors celle exécutée juste à la fin du traitement de la transaction. Il est alors possible en utilisant des outils dédiés, de retrouver facilement la ligne correspondante dans la fonction (C par exemple) du programme à partir du PC de l'instruction gelée.

Dans cette partie du document, nous n'identifions pas précisément l'instruction qui a été gelée car nous ne disposons pas des informations sur toutes les valeurs de PC des instructions du programme exécuté au cours de la simulation. Les outils que nous utilisons ne génèrent pas des informations fiables sur ces valeurs de PC. Mais dans la suite du document, nous proposons un algorithme qui permet d'identifier précisément chacune des

Algorithm 1: Association entre les échecs dans le cache de données et les fonctions dans lesquelles ces échecs sont survenus

Input: $\mathrm{L}_{Echec}(echec_1, echec_2, ..., echec_{n-1})$: Liste des différents échecs

Input: L_{fct} : Liste des fonctions du programme

Output: $\mathrm{L}_{Echec}(echec_1, echec_2, ..., echec_{n-1})$: Liste des différents échecs à jour avec les fonctions correspondantes

```
1  for echec_i ∈ L_Echec do
2      if echec_i est dû à une lecture de donnees then
3          fct_j ← (L_fct → debut());
4          repeat
5              Occurr_k_fct_j ← (L_temps_appel → debut()) ;
6              repeat
7                  if (Occurr_k_fct_j → cycle_debut ≤ echec_i → cycle_occur ≥ Occurr_k_-
                     fct_j → cycle_fin)
                   then
8                      (echec_i → Valeur_PC_fct) ← (fct_j → PC_fct);
9                      stop ← vrai;
10                 else
11                     Occurr_(k←k+1)_fct_j;
12             until ((stop == vrai) ou (k ≥ L_temps_appel → size()));
13             fct_(j←j+1);
14         until ((stop == vrai) ou (j ≥ L_fct → size()));
15 for fct_j ∈ L_fct do
16     CalculerStat_Echecs(L_Echec, fct_j);
```

instructions gelées pendant la simulation.

On peut ainsi déduire qu'au cours du traitement d'une transaction, 0 ou n gels peuvent survenir en fonction du nombre de registres utilisés par l'instruction gelée et qui nécessitent d'être chargés avec les données copiées par la transaction en cours de traitement. La figure 4.5 représente un récapitulatif de cette situation. Après un échec dans le cache de données au cycle t, la transaction est visible sur le bus à $t + x$ du fait par exemple de la charge du bus et de la latence entre les différents niveau de cache.

Cependant, il est difficile d'associer précisément un échec dans le cache de données à une transaction sur le bus, car le cache L_1 est adressé virtuellement (au moins en partie)

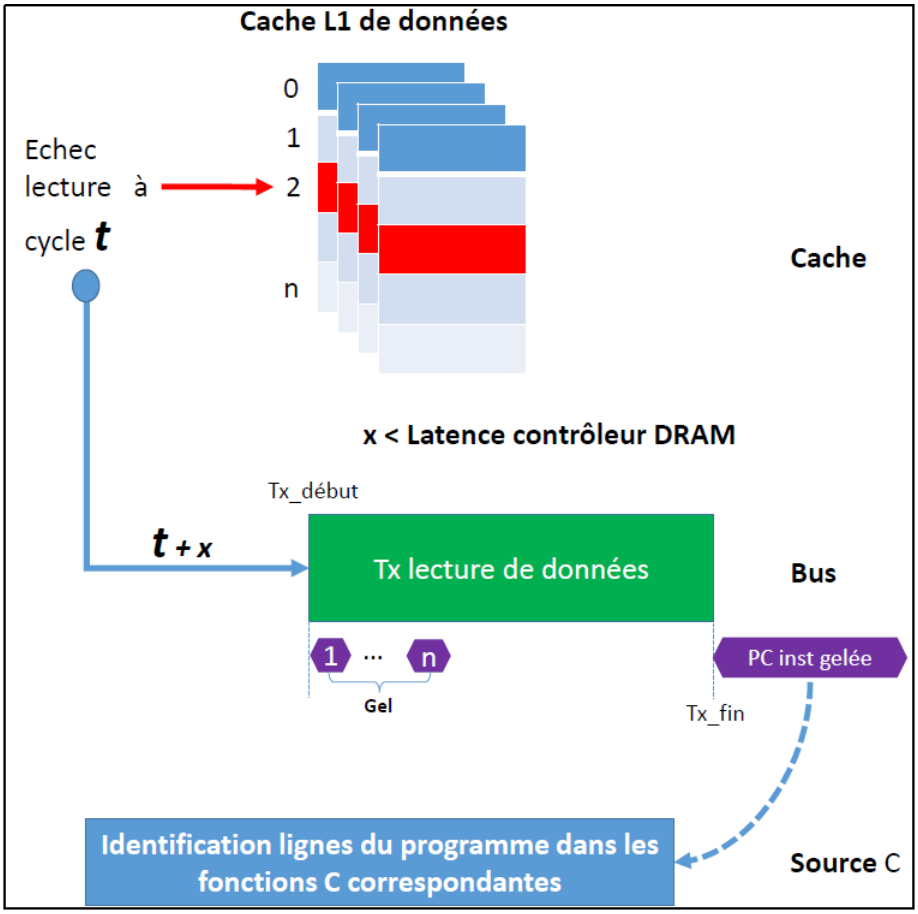

FIGURE 4.5: Récapitulatif sur l'association entre un cycle d'échec de lecture dans le cache L_1 et les valeurs de PC des fonctions du programme afin d'identifier les fonctions et/ou les instructions du programme (typiquement en C) qui ont produit les échecs lors des accès au cache L_1.

et les adresses manipulées par les transactions sont des adresses physiques. Une solution pourrait être de tracer l'activité du Translation Lookaside Buffer (TLB) et extraire de la trace le moment où la transaction passe pour retrouver la correspondance adresses physiques <—> adresses virtuelles. Cependant, si plusieurs adresses virtuelles sont mappées sur une même adresse physique (rare mais possible), il est alors compliqué de retrouver la bonne adresse virtuelle qui permet de retrouver l'instruction ayant engendrée la génération de la transaction.

Un exemple de code ad-hoc permettant d'illustrer cette situation est donné dans le listing 4.1. On suppose qu'au moins une des deux instructions de lecture de la fonction *addition* fait un échec à tous les niveaux de cache de la hiérarchie mémoire et qu'une transaction est émise sur l'interconnexion. Le processeur poursuit son traitement et le gel du processeur peut être observé sur l'instruction *ligne 4 (add)* car elle attendra la disponibilité des données lues dans les registres $0 et $3. En remontant la simulation cycle après cycle à

partir du début de la transaction jusqu'à l'apparition d'un premier échec, il est difficile de préciser exactement laquelle des deux instructions de lecture en mémoire est associée à cet échec et a généré la transaction sur le bus. Il n'y a aucun moyen de savoir laquelle des deux instructions de lecture a fait ou n'a pas fait un « hit » dans les caches. Par exemple, c'est difficile de savoir si la donnée lue en mémoire par l'instruction (*ligne 3*) se trouvait dans les caches ou se trouvait dans la même ligne de cache que la donnée lue par l'instruction (*ligne 2*). Dans ces situations, aucune transaction ne serait émise sur le bus lors de l'exécution de l'instruction (*ligne 3*) et il serait moins difficile de déduire que c'est la première lecture qui aurait engendré la génération de la transaction sur le bus. Mais les informations sur le comportement des instructions vis–à–vis des caches ne sont pas faciles à obtenir.

```
1        addition :
2        lw   $0, ($s2)
3        lw   $3, ($s4)
4        add $s5, $3, $0
```

Listing 4.1: Code assembleur MIPS ad–hoc pour montrer la difficulté à associer un échec à une transaction

L'algorithme 1 permet d'associer à chacune des occurrences d'échec dans le cache L_1 de données la fonction du programme testé en cours d'exécution. Il est alors possible d'identifier statistiquement les fonctions qui produisent le plus d'échecs et par conséquent susceptibles d'être à l'origine de l'apparition de plus de gels du processeur. Le développeur peut ainsi concentrer son analyse de performance sur les fonctions qui produisent le plus d'échecs. Dans le cas particulier de l'exemple présenté dans la figure 4.6 , la fonction *addition* est celle qui produit le plus d'échecs, puis la fonction *modulo*. Ainsi, dans ces différentes fonctions le développeur peut identifier manuellement les différentes instructions de lecture de données en mémoire et se focaliser dessus, plutôt que de passer du temps dans les fonctions qui produiraient moins de gain possible.

Les fonctions qui génèrent le plus d'échecs dans le cache L_1 de données du processeur et susceptibles de produire le plus de gels du processeur sont associées aux différentes transactions sur la base du temps. C'est–à–dire au moment où ces fonctions sont en cours d'exécution, quelles sont les transactions qui circulent sur le bus suite aux échecs dans le cache L_1 de données? Dans l'exemple de diagramme d'exécution d'un programme fourni dans la figure 4.4, au cycle *508284* la fonction *IDCT* est en cours d'exécution. Si cette fonction fait partie des fonctions qui produisent le plus d'échecs, on cherche par la suite à identifier les transactions présentent sur le bus au moment de l'exécution de cette fonction. Il est possible d'obtenir l'ensemble des adresses physiques accédées par chacune de ces transactions à partir des informations extraites des registres de performance du PMU du processeur. Une

analyse de ces adresses est faite pour identifier celles qui sont prédictibles.

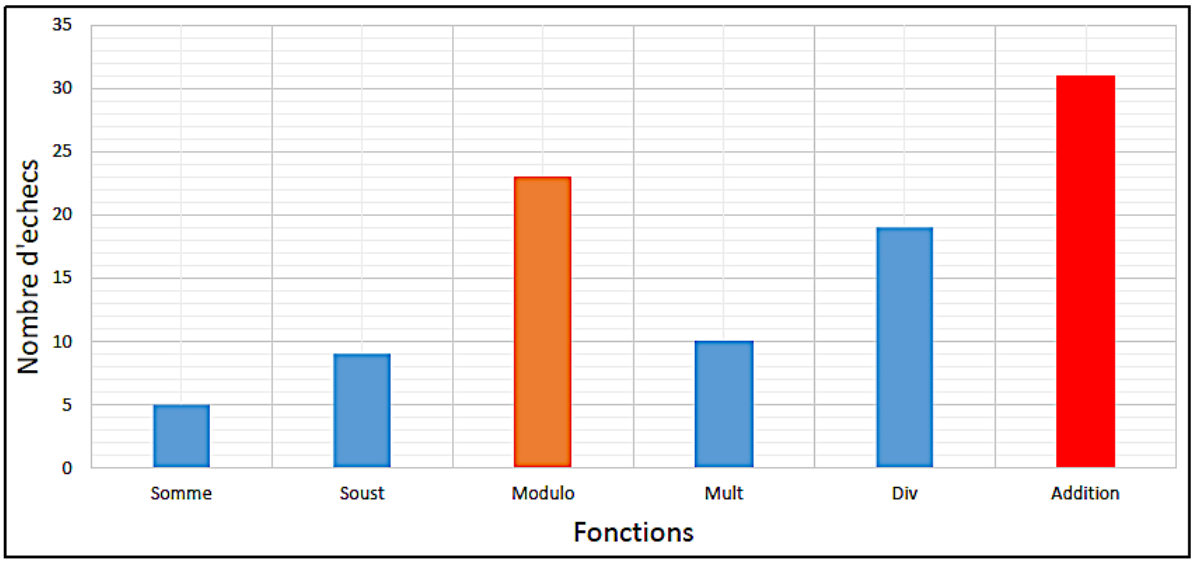

FIGURE 4.6: Exemple de statistique extraite grâce à l'algorithme 1 sur les échecs associés à chacune des fonctions du programme.

4.3.3 Identification des adresses accédées prédictibles

L'identification des adresses accédées prédictibles est dirigée par l'analyse des références régulièrement accédées. Les adresses mémoires accédées peuvent se présenter sous plusieurs schémas. Nous donnons un premier exemple (4.2) dans lequel les accès s'effectuent à des adresses régulièrement espacées, qui typiquement sont des fonctions affines des indices de boucles. Le code 4.2 effectue une addition entre des valeurs stockées à des adresses adjacentes à l'adresse à laquelle le résultat sera stocké. Un gabarit (pas) uniforme est calculé et se répète pour chacune des adresses accédées.

Le code 4.3 présente les accès mémoires qui s'effectuent de manière indexée. Dans ce cas de figure, la majorité des adresses sont accédées au travers d'un tableau d'index. Les éléments du tableau d'index (*index*) sont accédés de manière affine. En revanche, les adresses du tableau *A* sont accédées de manière irrégulière parce qu'elles dépendent de la valeur stockée dans la case d'index i du tableau *index*. Ces accès réduisent très souvent l'efficacité des caches du fait de la mauvaise localité qu'ils peuvent engendrer. De plus, le fait que le tableau d'index soit également susceptible de produire des échecs dans le cache L_1 de données ajoute davantage non seulement de trafic sur l'interconnexion pour copier dans le cache les valeurs stockées dans le tableau d'index, mais aussi de pression sur le cache L_1 de données. JAIN et LIN [2013] et YU et collab. [2015], proposent le développement de modules matériels

qui permettent de construire et créer une localité temporelle et spatiale entre les différents accès de ce type. Ceci facilite l'analyse de la ré–utilisabilité des adresses, car on se retrouve dans une situation où il est possible d'évaluer plus simplement la régularité spatiale et temporelle des accès. Dans notre cas, les adresses mémoires sont traitées telles qu'elles sont accédées par le processeur, c'est–à–dire qu'aucunes transformations, ni de modules matériels (car matériel figé) ne sont utilisés pour construire une localité entre les différents accès.

```
1 //
2 for (i = 1 ; i < N ; i++){
3     for (j = 1; j < N ; j++){
4         A[j] += A[j-1] * A[j+1] ;
5     }
6 }
```

Listing 4.2: Code C ad–hoc mettant en exergue la régularité d'accès aux références mémoires

```
1 index[N]
2 for (i = 1 ; i < N ; i++){
3     for (j = 1; j < N ; j++){
4         A[index[j]] += A[index[j-1]] * A[
5             index[j+1]] ;
6     }
7 }
```

Listing 4.3: Code C ad–hoc mettant en exergue l'accès indexé aux références mémoires

Enfin, les accès mémoires peuvent être effectués au travers de pointeurs. On retrouve généralement ce type d'accès dans les structures de données par exemple de type liste et graphe. Ce type d'accès n'est pas traité dans cette thèse, car en pratique dans les applications exécutées par les SoC, ils sont minoritaires.

Ces différents types d'accès peuvent mener à une mauvaise performance à cause des échecs dus aux conflits sur le cache L_1 ou encore des échecs dus à la capacité du cache L1 (2.3). Ces échecs proviennent de la grande quantité de données susceptibles d'être accédées par les applications dont le schéma d'accès est similaire à ceux sus–présentés.

Les informations collectées des registres internes du PMU contiennent toutes les adresses accédées en lecture tout comme celles accédées en écriture. Nous nous intéressons principalement aux adresses accédées en lecture. Il est à noter que nous parlons de l'accès aux adresses physiques en mémoire. En effet, les adresses qui circulent sur le bus sont obtenues suite à la traduction des adresses virtuelles par la table de translation d'adresses(TLB). Ainsi, les différentes transactions recueillies sur le bus contiennent les adresses physiques en mémoire.

Lorsqu'une donnée est accédée, une transaction de rechargement d'un niveau de cache (cache exclusif) ou de plusieurs niveaux de cache (cache inclusif) est émise en direction de la DRAM afin de copier la ligne de cache qui contient cette donnée s'il y a préalablement eu un échec dans le dernier niveau de cache.

Une mémoire cache inclusive stocke une copie des données à tous les niveaux de cache dans la hiérarchie mémoire ($L_1, ..., L_n$). Par exemple, dans une hiérarchie mémoire à deux niveaux de cache, une donnée qui se trouve dans le cache L_1 se trouvera forcément dans le cache L_2, ce qui conduit à une utilisation sous optimale de l'espace de stockage.

En revanche, une mémoire cache exclusive évite le problème de l'espace de stockage en ne stockant les données qu'à un seul niveau de cache dans la hiérarchie mémoire. Ainsi, pour une hiérarchie mémoire à deux niveaux de cache, une donnée sera stockée soit dans le cache L_1 de données, soit dans le cache L_2. Dans le processeur ARM Cortex–A53 par exemple, une ligne de cache provenant de la DRAM est copiée directement dans le cache L_1 de données sans passer par les niveaux de cache supérieurs ($L_2, ..., L_n$).

Il serait intéressant d'évaluer l'influence de chacun de ces deux types de cache sur la méthodologie proposée. Dans notre cas, nous nous intéressons aux caches exclusifs, car ils présentent de bonnes propriétés pour le domaine de l'embarqué. Nous faisons donc une analyse manuelle de la réutilisation des adresses physiques au cours de l'exécution du programme (BODÍK et collab. [1999a]). L'analyse consiste à identifier :

— La réutilisation temporelle : Les mêmes données sont réutilisées de manière périodique. Le listing de code 4.2 la référence A[j] a une réutilisation auto-temporelle dans la boucle i.

— La réutilisation spatiale : Les données d'une même ligne de cache sont utilisées dans des itérations distinctes. Dans le code la référence A[j] a une réutilisation spatiale dans la boucle j.

— La réutilisation de groupe (temporelle + spatiale) : Les mêmes données sont utilisées par des références distinctes. Dans le code 4.2 tous les opérandes de l'expression *A[j] += A[j-1] * A[j+1]* ont une réutilisation de groupe dans la boucle j.

On peut remarquer qu'il est plus évident de faire l'analyse manuelle de la ré–utilisabilité sur le code 4.2. En revanche, sur le code 4.3, cela est moins évident. L'algorithme 2 apporte une aide à l'analyse de la ré–utilisabilité, indépendamment du schéma des accès aux données. Pendant l'analyse, les adresses de lecture de données en mémoire qui sont référencées et utilisées plusieurs fois pendant la simulation sont identifiées. À partir de la liste des différentes adresses physiques accédées en mémoire principale, obtenues à partir des différentes transactions sur le bus, la structure de données $addr$ est construite. Elle contient le cycle d'accès à l'adresse (nécessaire pour l'évaluation de la localité temporelle de l'adresse), la valeur de l'adresse, et le type d'accès effectué (donnée ou instruction). Une liste des différentes adresses mémoire accédées est ensuite créée (L_{addr}). Pour chacune des adresses $addr_i$ de

Algorithm 2: Identification et regroupement des occurrences des différentes adresses physiques accédées

```
1  struct {
2      Cycle_acces : cycle d'acces à l'adresse ;
3      Valeur_adresse : Valeur de l'adresse accedée;
4      Type_acces : Type d'acces (Instruction ou donnee);
5      L_TempsOccurrence : Liste des cycles des différentes occurrences de l'adresse;
6  } addr;
   Input: L_addr(addr_1, addr_2, ..., addr_{n-1}) : Liste des adresses physiques accédées
   Output: L_addrPoids : Liste des adresses regroupees avec leurs poids
7  // Poids : Nombre d'accès à l'adresse;
8  for addr_i ∈ L_addr do
9      if addr_i est une adresse de lecture de donnees then
10         if addr_i est accedee premiere fois then
11             Ajouter(L_addrPoids, addr_i);
12         end
13         else
14             ptr_index ← Index(L_addrPoids, addr_i);
15             AugmenterPoids(addr_i, ptr_index);
16             SauvegarderTempsAccesNouvelle_occurrence(addr_i, addr_i->
                L_tempsOccurence);
17             // Chaque element de la liste L_TempsOccurrence correspond au temps
                d'acces a chacune des différentes occcurrences de addr_i dans l'ordre
                d'apparition;
18         end
19     end
20 end
```

cette liste, si l'adresse est accédée pour la première fois et est une référence de lecture de données, ladite adresse est ajoutée dans la liste des adresses référencées de manière unique avec son nombre d'occurrences ($L_{addrPoids}$). Si l'adresse est déjà dans la liste $L_{addrPoids}$, un pointeur sur ladite adresse dans la liste est retourné afin de mettre à jour son poids ainsi que le cycle auquel cette nouvelle occurrence de l'adresse est accédée. Ainsi, il est possible d'identifier les boucles d'exécution en évaluant manuellement la régularité spatiale entre les différentes adresses de la liste $L_{addrPoids}$, mais aussi la régularité temporelle entre les diffé-

rentes occurrences d'une référence. Par exemple, considérons le code fourni dans le listing 4.3. Le code désassemblé équivalent pour une architecture ARM 32–bit est fourni dans la figure 4.7.

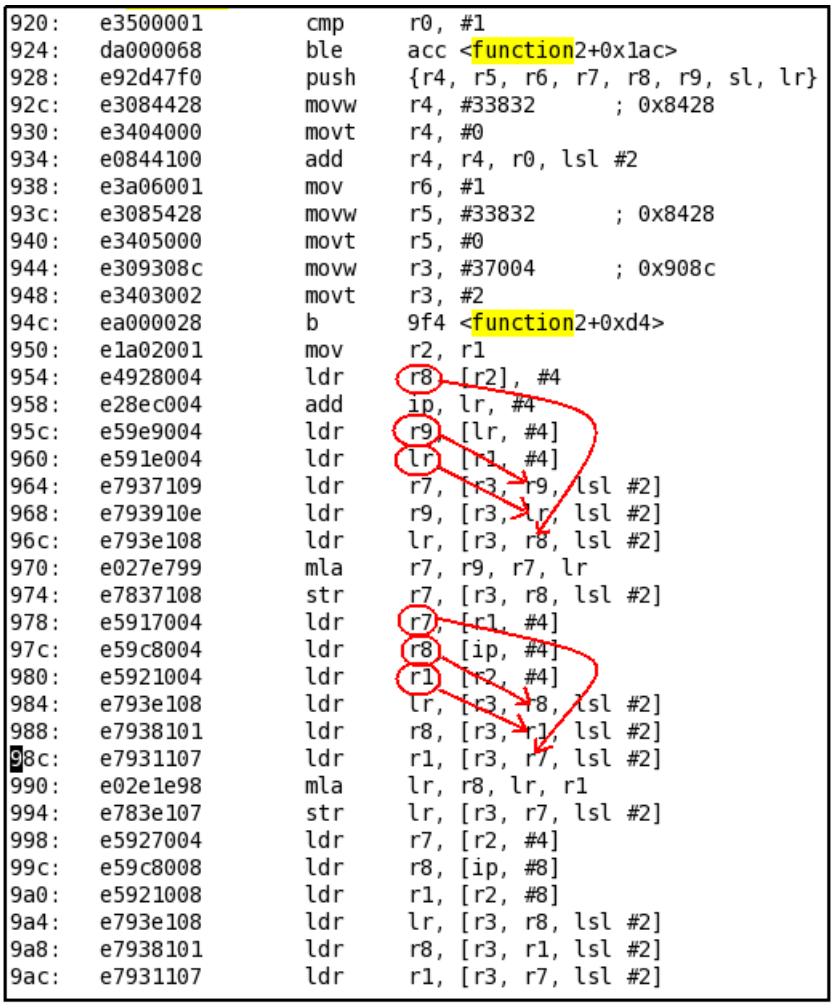

FIGURE 4.7: Code désassemblé du listing de code 4.3.

On observe qu'avant chaque évaluation de l'expression *A[index[j]] += A[index[j-1]] * A[index[j+1]]* les valeurs stockées dans le tableau d'index qui seront utilisées sont chargées (*ldr*) dans les registres en premier. Puis, les valeurs de ces registres sont utilisées pour charger les opérandes de chacune des instructions *mla* (Multiply Accumulate). On remarque ainsi que le code est susceptible d'effectuer plusieurs lecture des données (en cache ou en mémoire), non seulement pour les données du tableau *index*, mais aussi pour le tableau *A*. Si après le traitement d'une ligne de cache un échec doit survenir, il peut y avoir un grand trafic entre la mémoire et le cache L_1 de données. Le résultat est stocké à l'une des adresses dont la valeur est une opérande de l'instruction *mla*. En exécutant cet exemple, un échantillon d'adresses physiques (choisies aléatoirement) des données lues en mémoire est extrait des différentes

transactions. L'algorithme permet de produire un résultat tel que présenté dans la figure 4.8.

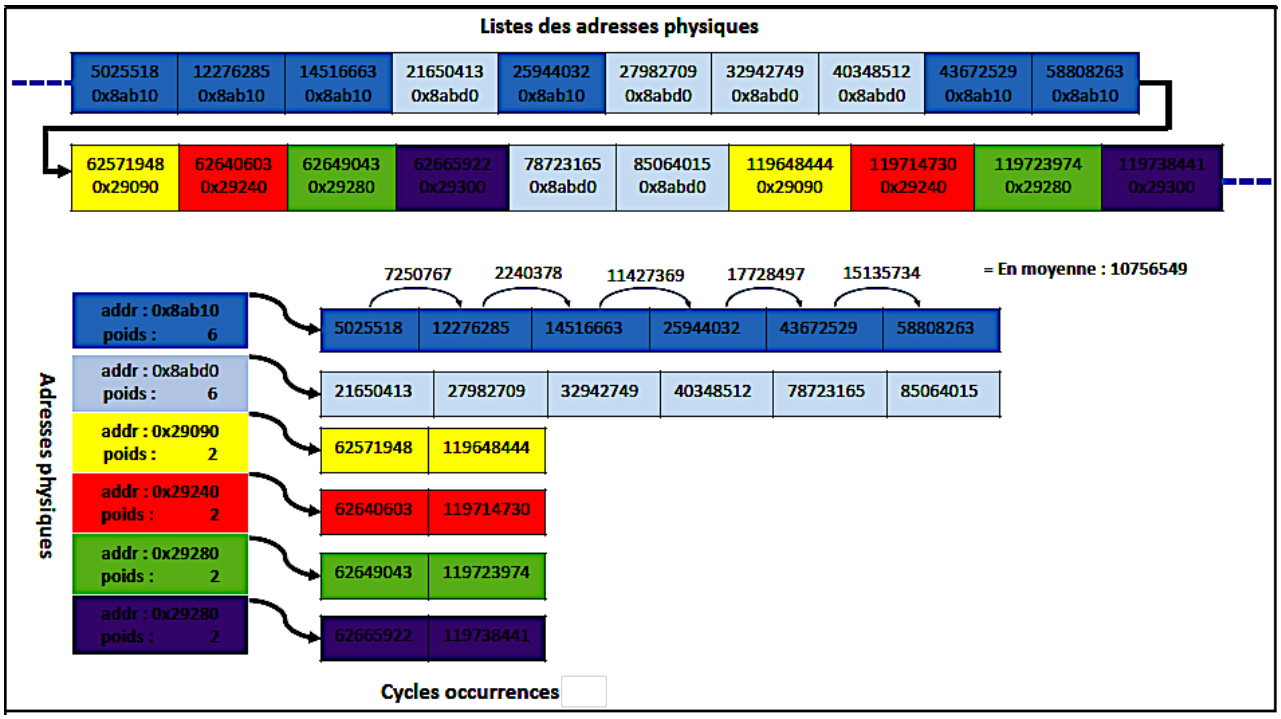

FIGURE 4.8: Exemple de résultat de regroupement d'occurrences d'adresses produit grâce à l'algorithme 2.

L'adresse initiale du tableau *A* est *0x8ab10* et celle du tableau des index est *0x29090*. Pour N = 1000000 et la latence du contrôleur mémoire fixée à 804$cycles$, à partir de la liste des adresses physiques, chacune des occurrences des adresses *0x8ab10, 0x8abd0, 0x29090, 0x29140, 0x29280,* et *0x29300* est comptabilisé et le cycle auquel chacune de ces occurrences survient est ajouté à la liste des moments d'occurrences de chacune d'elles. Il est possible de calculer le nombre moyen de cycles nécessaires pour la lecture d'une nouvelle occurrence de chacune de ces adresses. Pour l'adresse *0x8ab10* qui a un poids égale à 6, le nombre de cycles moyen pour un nouvel accès de cette adresse est de 10 756 549 cycles. On peut en déduire qu'environ tous les 10 756 549 cycles, l'adresse *0x8ab10* est accédée et dans ce cas à *6 reprises*. Ce qui correspond à 6 lignes de cache copiées par chacune des transactions générées lors de l'accès de cette adresse suite aux échecs dans le cache L_1 de données du CPU.

Ensuite, il est également possible en regardant les cycles d'occurrences des échecs de lecture de données en cache L_1 qui ont été identifiés préalablement et grâce aux adresses identifiées qui sont fréquemment réutilisées par le programme, de déduire les données ou les blocs de données qui devraient être en mémoire cache à un cycle donné. Une base de connaissances sur les échecs possibles dans le programme est construite à partir des infor-

mations sur les échecs de données précédemment extraites des registres de PMU, ainsi qu'à partir des références qui correspondent aux données qui devront être copiées en mémoire cache (YAZDANBAKHSH et collab. [2016]).

es adresses physiques qui sont fréquemment accédées peuvent être pré–chargées. Mais il faut préalablement les traduire en adresses virtuelles car les adresses que le processeur utilise sont uniquement des adresses virtuelles. Dans notre situation puisque l'exécution du programme s'effectue à nu « bare », il n'y a pas de Memory Management Unit (MMU) qui fait la translation d'adresses. Donc les adresses virtuelles sont équivalentes aux adresses physiques.

Il faut maintenant trouver à quels endroits insérer les instructions de pré–chargement dans le code source du programme afin de précharger les lignes de cache au moment opportun.

4.4 Insertion des instructions de pré–chargement de données en mémoire cache

La combinaison entre les différentes valeurs prises par le registre PC du processeur, le graphe d'exécution des différentes fonctions du programme, et les transactions d'utilisation du bus durant l'exécution de l'application, ont permis de connaître précisément les fonctions et les instructions en assembleur exécutées pour chacun des échecs générés dans le cache L_1 de données. Ainsi, il est possible d'insérer dans le code source de l'application les instructions de pré–chargement de données en cache, en s'appuyant sur les adresses prédictibles identifiées et susceptibles de produire des échecs de données. Cette insertion d'instructions de pré–chargement dans le code source nécessite préalablement une analyse conjointe de la localité temporelle et spatiale des différentes adresses physiques. Ceci permet d'éviter les effets négatifs engendrés par ces instructions si elles ne sont pas insérées dans le code source du programme à des emplacements idoines.

Pour que les instructions insérées dans le code soient efficaces, il est nécessaire de prendre en compte la latence du contrôleur mémoire tel que présenté dans la figure 4.9 sur laquelle nous considérons dans le *(a)* que la demande d'accès à la donnée génère un échec dans le cache L_1 de données du processeur. Le processeur est donc contraint de patienter L cycles (latence du contrôleur mémoire) avant de voir la donnée requise disponible. Dans figure 4.9–*(b)*, le pré–chargement est effectué en amont de la lecture effective de la donnée à une distance $D > L$. On peut remarquer que la latence du contrôleur mémoire est masquée et que lorsque la demande d'accès est effectuée, la donnée est disponible toute suite (en pra-

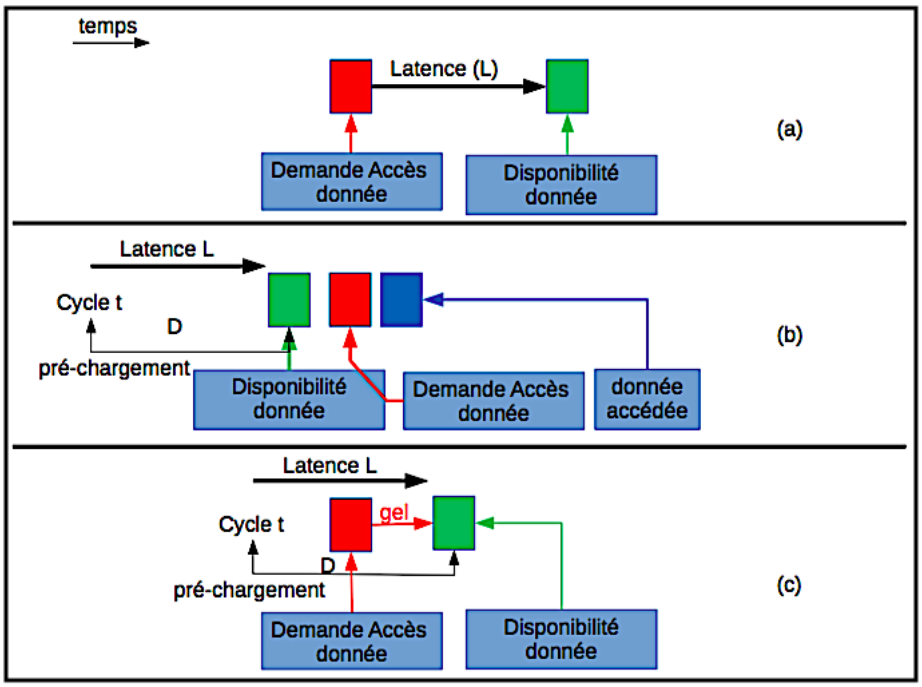

FIGURE 4.9: Relation entre la distance de pré–chargement et la latence du contrôleur mémoire

tique quelques cycles équivalent à la latence du cache L_1 de données). Dans la figure 4.9–*(c)* l'instruction de pré–chargement est émise en amont de la lecture effective de la donnée à une distance D < L. Le processeur est donc contraint d'être gelé de (D – L) cycle(s). Dans cette situation le pré–chargement est effectué un peu tard. La distance de pré–chargement D d'une donnée en cache est définie ainsi (SAAVEDRA et collab. [1994]) :

Distance de pré–chargement : nombre de cycles d'exécution de l'application nécessaires pour qu'une ligne de cache pré–chargée puisse être disponible dans le cache avant qu'elle ne soit sollicitée par une instruction du programme en cours d'exécution.

Pour l'optimisation des boucles d'exécution dans lesquelles les données sont lues en DRAM et les traitements effectués sur ces données, nous estimons la distance d'itération que nous définissons ainsi :

Distance d'itération : nombre d'itérations nécessaires pour permettre qu'une ligne de cache pré–chargée puisse être disponible dans le cache avant qu'elle ne soit requise par les futures requêtes de lecture de ces données en DRAM.

Ce nombre d'itérations est estimé grâce à la relation (4.1) développée par MOWRY et collab. [1992] durant sa thèse de doctorat.

$$D \geq \left\lceil \frac{l}{s} \right\rceil \tag{4.1}$$

Avec :

— D : Distance d'itération,

— l : Latence moyenne (en cycle) du contrôleur mémoire ,

— s : Estimation du nombre de cycles nécessaires pour l'exécution du plus court chemin possible pris par une itération dans une boucle.

Cette distance d'itération est dépendante de deux principaux paramètres : la latence du contrôleur mémoire et le nombre de cycles nécessaires pour l'exécution du plus court chemin possible pris par l'exécution d'une itération dans une boucle. D $\in \mathbb{N}$. Le symbole $\lceil \rceil$ indique que la valeur de D est obtenue par arrondi à la plus petite valeur entière supérieure.

Dans le cas d'une boucle simple telle que présentée dans le listing 4.4, la distance d'itération s'utilise ainsi : dans le tableau *T*, l'adresse mémoire d'une ligne de cache à précharger sera calculée en ajoutant la valeur de la constante *D* à l'adresse de la case du tableau *T* en cours de traitement. Par exemple, si l'adresse de base du tableau *T* est *0x2B00* et D = 4 la distance d'itération, *T[i+D]* est l'adresse de la ligne de cache à précharger ;
soit $0x2B00 + (i + \mathrm{D}) \times taille(element_{\mathrm{T}})$. La figure 4.10 illustre l'utilisation de la distance d'itération. Lorsque la case *i* est en cours de traitement, la donnée de la case *i+4* est pré–chargée.

```
1        for(i=0; i< N; i++){
2            ...
3            T[i] = T[i+2]*5 + X;
4            ...
5        }
```

Listing 4.4: Code C avec boucle simple pour illustrer l'utilisation de la distance d'itération

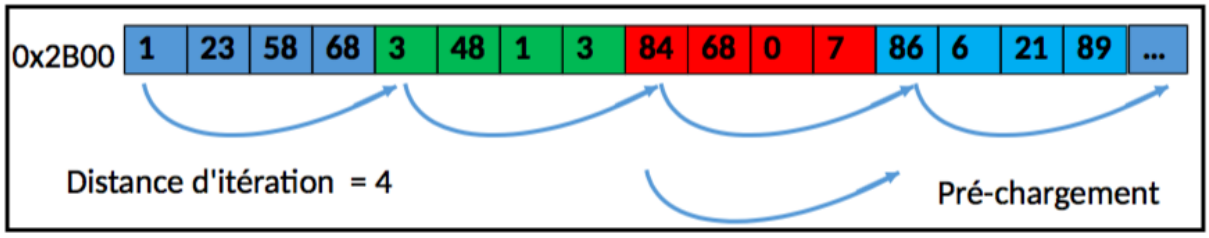

FIGURE 4.10: Exemple d'utilisation de la distance d'itération lorsque D = 4.

En utilisant les informations des registres de PMU du processeur, pour estimer le paramètre *s* de la relation 4.1, nous définition une configuration du programme de manière à ce qu'il

s'exécute en passant par le plus court chemin possible. La configuration du programme pour qu'il s'exécute en passant par le plus court chemin possible est effectuée dans les fonctions identifiées précédemment et qui génèrent le plus d'échecs de données. Les blocs de base sont identifiés dans ces fonctions et un graphe de contrôle est construit manuellement pour chacune des fonctions. Ce graphe permet de définir les blocs de bases qui constituent le plus court chemin possible de l'exécution de la fonction, mais aussi des boucles contenues dans ces fonctions.

Un bloc de base est constitué uniquement d'instructions qui a) ne sont pas des branchements, sauf la dernière, et b) n'intègrent aucune instruction qui est la cible d'un branchement. Par exemple, si on considère la fonction présentée dans le listing de code 4.5, le graphe obtenu en identifiant les blocs de base est fourni dans la figure 4.11. On observe qu'il y a 5 blocs de base et que le plus court chemin d'exécution de la fonction passe par l'exécution des blocs de base *B1* et *B5*. Mais en s'intéressant à la boucle while, le plus court chemin possible pour l'exécution d'une itération de cette boucle while, passe par les blocs de base *B1, B2, B4* .

```
1  void exemple (A){
2      while(C < 0){
3          x = y + 1;
4          A[0] = 10;
5          z = x % A[0];
6          if(z==0)
7              A[x] = y * A[z-1] ;
8          z = 1;
9      }
10     z = x ;
11 }
```

Listing 4.5: Code C ad–hoc pour montrer l'identification des blocs de base

La figure 4.12 présente le graphe équivalent avec blocs de base du code 4.3 utilisé comme exemple. Ce graphe permet d'identifier le plus court chemin pour l'exécution d'au moins une itération des boucles *i* et *j*. La partie hachurée du graphe montre le plus court chemin d'exécution d'une seule (N = 1) itération des deux boucles *i* et *j*. Le programme peut alors être modifié de manière à ce qu'une seule itération des boucles *i* et *j* soit exécutée sur la plateforme simulation précise au niveau du cycle. Les caches ne sont pas vides au début de l'exécution du programme. Grâce aux informations des registres de PMU, le nombre de cycles d'exécution d'une itération de chacune des deux boucles *i* et *j* est calculé. Pour le code 4.3 $s = 119 cycles$. La latence du contrôleur mémoire de la plateforme de simulation est fixée

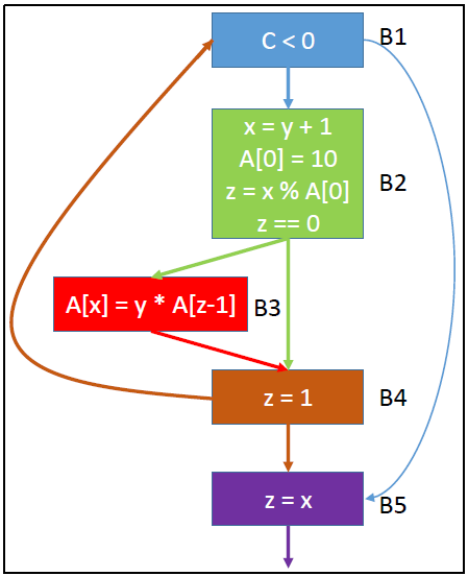

FIGURE 4.11: Exemple de graphe de contrôle obtenu à partir du code 4.5 fourni en exemple.

à 124 cycles. On en déduit alors que $D \geq \lceil 124/119 \rceil = 2$.

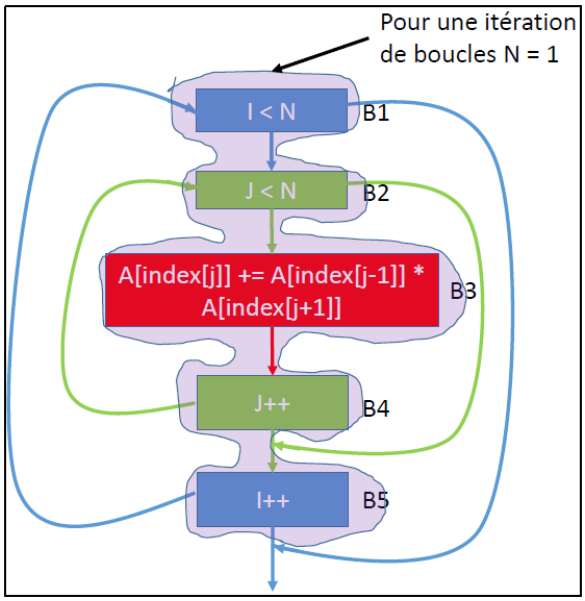

FIGURE 4.12: Graphe de contrôle du code 4.3.

Pour que le compilateur puisse générer au moment de la compilation les instructions de pré–chargement dans le code exécutable du programme, l'extension interne à GCC et indépendante du processeur *__builtin_prefetch(addr, r/w, locality)* est utilisée et insérée manuellement dans le code source du programme en langage de haut niveau (C par exemple). Par exemple pour un processeur ARM qui supporte cette extension, les instructions *PLD* seront générées pour les pré–chargements en lecture, et *PLDW* pour des pré–chargements

avec intention d'écriture. Les pré–chargements avec intention d'écriture sont effectués sur les données qui sont modifiées dans le cache et qui devront ensuite être écrites en mémoire principale. Deux cas de figure se présentent :

Le cache de données est « write–through » ; dans ce cas, une fois que la donnée est modifiée dans le cache, la mémoire principale est directement mise à jour car les caches et la mémoire doivent avoir à tout moment des données identiques. A chaque mise à jour d'une donnée en mémoire, il faut émettre un pré–chargement de ladite adresse. Ceci peut générer beaucoup de trafic en écriture sur le bus.

Dans le second cas de figure, le cache est « write–back » ; la donnée n'est écrite en mémoire qu'au moment où la ligne de cache contenant la donnée est invalidée ou éjectée. Chaque ligne de cache dispose d'un bit (« dirty bit ») indiquant si celle-ci a été modifiée depuis son chargement. L'adresse d'écriture de la donnée en mémoire ne doit pas être pré–chargée à chaque modification de la donnée mais uniquement lorsque la ligne de cache qui contient ladite donnée est éjectée ou invalidée. Les plateformes utilisées durant la thèse intègrent ce type de cache. Mais il serait intéressant de voir l'influence de chacune de ces deux politiques de mise à jour entre les caches de données et la mémoire principale sur la correction des cycles de gel du processeur.

L'extension *__builtin_prefetch(addr, r/w, locality)* permet donc de copier de manière asynchrone (en parallèle de l'exécution normale du programme) les données dans le cache avant qu'elles ne soient accédées. Plus de détails sur l'utilisation de cette extension sont présentés dans GCC [2017]. En cas d'échec, ou lorsqu'une donnée est préchargée, une ligne de cache est copiée dans le cache L_1 de données. Il faut également tenir compte de la taille d'une ligne de cache de données au moment de l'insertion de l'instruction de pré–chargement dans le code source du programme. La quantité de données à précharger (c'est–à–dire le nombre de lignes de cache) dépend du nombre de tampons de pré–chargement disponibles sur le processeur. Le listing de code 4.6 présente un exemple d'insertion intuitive et manuelle d'instructions de pré–chargement dans le code 4.3 décrit précédemment. Sur cet exemple, on suppose que les premiers accès aux données génèrent des échecs dans les caches de données. L'identification des adresses prédictibles grâce à l'algorithme 2 montre que le tableau *index* et le tableau de données *A* produisent des échecs dans le cache L_1 de données. Il faut alors appliquer le pré–chargement aux deux tableaux.

```
1            index[N];
2            __builtin_prefetch(&index[0], 0, 3);
3            __builtin_prefetch(&A[index[0]], 0, 3);
4            for (i = 1 ; i < N ; i++){
5            __builtin_prefetch(&A[index[i + D*CACHE_LINE]], 0, 3);
```

```
6                   for (j = 1; j < N ; j++){
7                       if(j % CACHE_LINE){
8                       __builtin_prefetch(&index[j + D*CACHE_LINE], 0, 3);
9                       __builtin_prefetch(&A[index[j + D*CACHE_LINE]], 0, 3);
10                      }
11                      A[index[j]] += A[index[j-1]] * A[index[j+1]] ;
12                  }
13              }
```

Listing 4.6: Code C ad–hoc mettant en exergue l'insertion manuelle des instructions de pré–chargement dans le code source du programme 4.3

On remarque dans ce listing que le pré–chargement logiciel augmente le nombre d'instructions (instructions de calcul d'adresses à précharger, instructions de pré–chargement, utilisation des registres, ...) à exécuter par le processeur. Comparé au temps d'exécution du programme d'origine, les instructions de pré–chargement ajoutées en plus au programme d'origine ne doivent pas augmenter le temps d'exécution de celui–ci. Ceci nécessite l'évaluation de la performance du pré–chargement logiciel.

4.5 Evaluation du pré–chargement logiciel

L'évaluation du pré–chargement logiciel peut se faire selon plusieurs métriques. Sans être exhaustif, nous pouvons citer l'accélération de l'exécution (« speedup »), la précision du pré–chargement, la couverture des échecs, la consommation de la bande passante sur le bus, ou encore la pollution de cache, ... Nous précisons dans la suite la définition précise de ces métriques.

En ce qui concerne l'augmentation de la vitesse d'exécution du programme testé, elle est donnée par la relation 4.2. Elle représente le rapport entre le temps d'exécution du programme d'origine et le programme avec instructions de pré–chargement insérées. Par exemple si le temps d'exécution du programme d'origine est 100µs et le temps d'exécution du programme après insertion manuelle des instructions de pré–chargement dans le code source est 90µs, $AugmentationVitesse_{Exe} = 1.11$. Dans ce cas, le temps d'exécution du programme d'origine est donc divisé par 1.11.

$$AugmentationVitesse_{Exe} = \frac{Temps\_Exécution_{Prog\_origine}}{Temps\_Exécution_{Prog\_avec\_pré-chargement}} \tag{4.2}$$

Pour ce qui est de la couverture des échecs, elle consiste à évaluer la réduction du nombre d'échecs dans le cache de données grâce au pré–chargement dirigé par le logi-

ciel. Cette métrique est importante pour estimer le pourcentage d'échecs qui ont été évités lorsque les instructions de pré–chargement logiciel sont insérées manuellement dans le code source du programme.

Cependant, la réduction du nombre d'échecs ne garantit pas toujours la réduction du nombre de cycles d'exécution du programme du fait du surcoût lié aux instructions de pré–chargement ajoutées dans le code source du programme. Dans cette situation l'exécution des instructions de pré–chargement pour éviter les échecs est plus coûteuse en temps que le temps que ces échecs ajoutent au temps d'exécution global du programme. Le calcul de la couverture des échecs est effectué grâce à la relation 4.3.

$$Couverture = \frac{Nombre\_Échec_{Prog\_avec\_pré-chargement}}{Nombre\_Échecs_{Prog\_origine}} \tag{4.3}$$

L'évaluation de la bande passante sur le bus consiste à évaluer la bande passante consommée par le programme avec instructions de pré–chargement et celle consommée par le programme d'origine et à en faire une comparaison. Si l'implémentation du processeur supporte les instructions de pré–chargement et que les caches sont actifs, l'instruction de pré–chargement génère donc une transaction sur le bus si la donnée préchargée n'est pas stockée pas dans le cache. La bande passante est quantifiée en regardant le nombre de transactions sur le bus sur un intervalle de temps correspondant au début et à la fin de l'exécution de la fonction dans laquelle les instructions de pré–chargement ont été insérées. Il est alors possible de comparer le nombre de transactions générées sur le bus par le programme d'origine à celui du programme avec instructions de pré–chargement. Trois cas de figure se présentent : Posons $Nb_{prog\_origine}$ et $Nb_{prog\_avec\_pré-chargement}$ le nombre de transactions générées par le programme d'origine respectivement par le programme avec instructions de pré–chargement.

— Si $Nb_{prog\_origine} < Nb_{prog\_avec\_pré-chargement}$: Les instructions de pré–chargement génèrent sur le bus une bande passante supérieure à celle du programme d'origine. Le bus est alors un peu plus occupé. A l'exécution du programme, plusieurs requêtes dans les caches de la hiérarchie mémoire ont produit des échecs. Il est également possible que les caches aient été victime de cache thrashing ou pollués car pour mémoire les instructions de pré–chargement logiciel sont censées anticiper les lectures de données qui seront requises dans le futur par le processeur.

— Si $Nb_{prog\_origine} = Nb_{prog\_avec\_pré-chargement}$: La bande passante utilisée par les deux versions du programme est identique. On pourrait alors évaluer la répartition des transactions sur le bus pour voir si cette métrique a de l'importance. Dans cette thèse nous n'étudions pas la métrique de répartition des transactions sur le bus.

— Si $Nb_{prog\_origine} > Nb_{prog\_avec\_pré-chargement}$: On pourrait émettre l'hypothèse que les données pré–chargées se trouvent dans le cache de données et donc les transactions ne sont pas émises sur le bus.

La pollution du cache L_1 de données du processeur peut survenir lorsque les instructions de pré–chargement sont insérées dans le code source du programme. En effet, les lignes de cache importantes ou dont les données seront utilisées dans un futur proche peuvent être remplacées par les lignes de caches pré–chargées. Cependant, il est difficile d'évaluer la pollution du cache à partir des informations extraites des registres de PMU. Par ailleurs, lorsque les instructions de pré–chargement sont insérées dans le code source du programme, les adresses des instructions du programme d'origine sont décalées. Ceci peut modifier la manière dont les données du programme sont stockées par le « linker » au moment de l'édition des liens. Par exemple, les données qui étaient stockées sur une même page dans la version d'origine du programme pourraient se retrouver être stockées sur des pages différentes dans la nouvelle version avec instructions de pré–chargement. Au moment du linkage, le « linker » n'a aucunes informations sur l'architecture du cache, la politique de remplacement, ... La nouvelle version du programme qui intègre les instructions de pré–chargement est donc un programme légèrement différent (adresses des instructions) du programme d'origine, mais conserve la même exactitude d'exécution que la version d'origine du programme. L'évaluation de la pollution peut par exemple consister à savoir si la nouvelle version du programme génère plus d'échecs que la version d'origine. Dans ce cas le pré–chargement logiciel serait juste considéré inefficace vis–à–vis de l'utilisation des caches.

4.6 Conclusion partielle

Dans ce chapitre, nous avons présenté un cas d'étude qui permet de montrer que l'observabilité fournie par des plateformes virtuelles précises au niveau du cycle est essentielle pour faciliter l'estimation et l'optimisation en amont de la performance du logiciel sur la puce (SoC). Bien qu'il soit possible d'obtenir des informations pertinentes sur l'utilisation des capacités du matériel directement sur un SoC physique, cela présente deux désavantages : (a) cela n'est faisable que très tard dans le flot de conception, (b) sur un SoC physique, il est difficile d'avoir accès aux différentes transactions sur le bus, sinon de manière intrusive, et la lecture du PMU ne peut se faire en permanence, ce qui réduit la visibilité sur l'intégration logiciel/matériel.

Nous avons donc tiré avantage de cette observabilité des modèles CA pour proposer une

stratégie de pré-chargement de données dirigée par le logiciel. Cette stratégie s'appuie sur l'analyse des accès en mémoire qui produisent des cycles de gel de données, informations difficiles à obtenir sur un SoC physique.

La simulation a donc joué un rôle prépondérant lors du développement de cette stratégie. Cette stratégie permet d'insérer manuellement les instructions de pré–chargement dans le code source du programme tout en conservant l'exactitude de l'exécution du programme. Ces instructions permettent de précharger en mémoire cache la bonne quantité de données, avec une bonne distance d'itération (boucle d'exécution), ceci pour une latence mémoire bien définie.

La ficgure 4.13 présente de manière synthétique l'approche développée dans ce chapitre. Le flot prend en entrée un programme source décrit en langage de haut niveau (C ou C++ par exemple). Ce code source est compilé en croisé (« cross-compilé ») avec des options de compilation qui permettent d'optimiser le code exécutable. Par exemple le temps d'exécution (–O_n avec GCC), ou encore l'insertion automatique des instructions de pré–chargement par le compilateur (–fprefetch-loop-array). Le code source est donc compilé pour une architecture cible (Cortex–A9 ou Cortex–A53 32–bit par exemple). Le fichier exécutable produit est ensuite exécuté sur une plateforme précise au niveau du cycle, telle que par exemple une plateforme de simulation, pour produire principalement deux traces d'exécution : la trace des informations issues des registres de PMU et la trace des différents signaux du bus. Les signaux du bus sont analysés pour reconstituer les différentes transactions de lecture ou d'écriture en mémoire principale.

Une analyse manuelle de la trace PMU est effectuée afin de détecter l'apparition des cycles de gel du processeur au cours de l'exécution du programme, identifier les fonctions du programme exécuté qui produisent le plus de gel, et identifier au moyen d'un algorithme (cf Algorithme 1), celles qui produisent le plus d'échecs dans le cache L_1 de données. Grâce à l'algorithme 2, l'analyse des différentes transactions de lecture de données sur le bus entre le dernier niveau de cache (LLC) et le contrôleur mémoire est effectuée afin de regrouper et identifier les adresses de données accédées prédictibles.

Les limitations liées à l'utilisation et aux informations fournies par les registres de PMU rendent la méthodologie proposée empirique (plusieurs processus manuels) et dépendante du processeur, c'est–à–dire de sa capacité à fournir les informations sur son fonctionnement interne à partir des registres de performance qui ne sont pas toujours disponibles ou accessibles sur certains processeurs. Ce qui rend l'automatisation de cette méthode nécessiteuse des traitements supplémentaires. L'idée serait donc de trouver une approche indépendante du processeur sur lequel le programme testé s'exécute pour pouvoir proposer une approche générale.

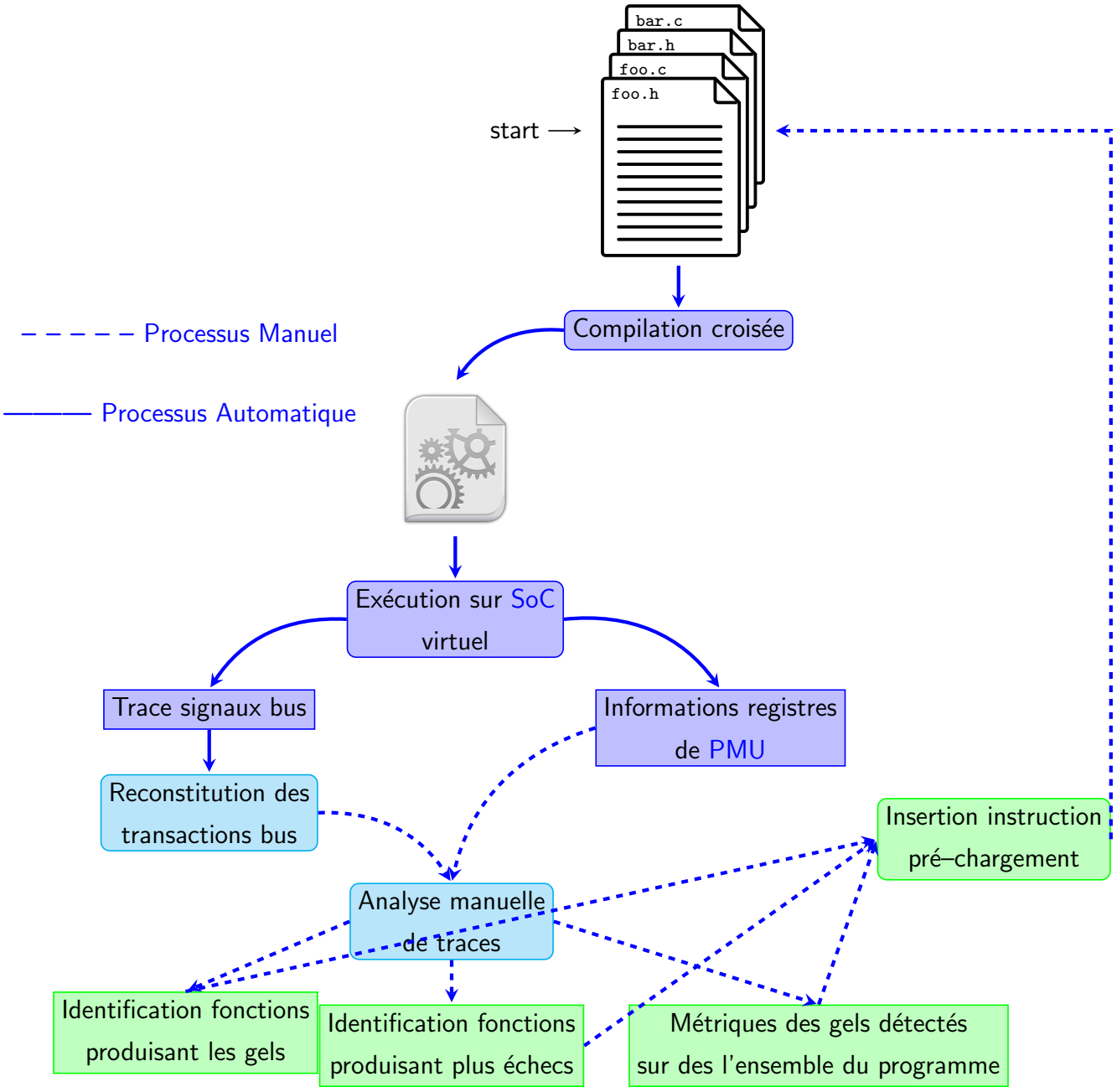

FIGURE 4.13: Flot itératif de détection et d'identification manuelle des gels du processeur et des fonctions qui produisent le plus d'échecs et de cycles de gel.

4.7 Références

BODÍK, R., R. GUPTA et M. L. SOFFA. 1999a, «Load-reuse analysis : Design and evaluation», dans *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, ACM, New York, NY, USA, ISBN 1-58113-094-5, p. 64–76, doi :10.1145/301618.301643. URL http://doi.acm.org/10.1145/301618.301643. 66

BODÍK, R., R. GUPTA et M. L. SOFFA. 1999b, «Load-reuse analysis : Design and evaluation», *SIGPLAN Not.*, vol. 34, nº 5, doi :10.1145/301631.301643, p. 64–76, ISSN 0362-1340. URL http://doi.acm.org/10.1145/301631.301643.

GCC. 2017, « Other Built-in Functions Provided by GCC», https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html. [En ligne ; dernier accès le 9 Août 2017]. 75

JAIN, A. et C. LIN. 2013, «Linearizing irregular memory accesses for improved correlated prefetching», dans *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, ACM, New York, NY, USA, ISBN 978-1-4503-2638-4, p. 247–259, doi :10.1145/2540708.2540730. URL http://doi.acm.org/10.1145/2540708.2540730. 64

MOWRY, T. C., M. S. LAM et A. GUPTA. 1992, «Design and evaluation of a compiler algorithm for prefetching», *SIGPLAN Not.*, vol. 27, nº 9, doi :10.1145/143371.143488, p. 62–73, ISSN 0362-1340. URL http://doi.acm.org/10.1145/143371.143488. 71

SAAVEDRA, R. H., W. MAO et K. HWANG. 1994, «Performance and optimization of data prefetching strategies in scalable multiprocessors», *J. Parallel Distrib. Comput.*, vol. 22, nº 3, doi :10.1006/jpdc.1994.1102, p. 427–448, ISSN 0743-7315. URL http://dx.doi.org/10.1006/jpdc.1994.1102. 71

YAZDANBAKHSH, A., B. THWAITES, H. ESMAEILZADEH, G. PEKHIMENKO, O. MUTLU et T. C. MOWRY. 2016, «Mitigating the memory bottleneck with approximate load value prediction», *IEEE Design & Test*, vol. 33, nº 1, doi :10.1109/MDAT.2015.2504899, p. 32–42. URL https://doi.org/10.1109/MDAT.2015.2504899. 70

YU, X., C. J. HUGHES, N. SATISH et S. DEVADAS. 2015, «Imp : Indirect memory prefetcher», dans *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, ACM, New York, NY, USA, ISBN 978-1-4503-4034-2, p. 178–190, doi :10.1145/2830772.2830807. URL http://doi.acm.org/10.1145/2830772.2830807. 64

Chapitre 5

Exploitation de traces d'exécution pour l'identification et la quantification des gels du processeur

Sommaire

5.1 Introduction

La performance du logiciel sur une architecture constituée d'un processeur et d'une hiérarchie mémoire pour l'accès aux données est le plus souvent recherchée par la conjugaison de deux facteurs. Le premier est le développement d'algorithmes sophistiqués, implémentés de manière à avoir la meilleure solution possible dans un langage de programmation particulier. Le second est l'utilisation des meilleures options d'optimisation du compilateur et des outils d'édition des liens qui permettent en particulier de préciser le placement de certaines données pour la génération de l'exécutable final. Par exemple, l'optimisation de la compilation du logiciel peut être réalisée en activant successivement (ou en parallèle sur plusieurs machines hôtes de compilation) toutes les combinaisons d'options d'optimisation du compilateur. Puis en exécutant les fichiers exécutables résultants sur un même jeu de données représentatif, en essayant ainsi de trouver la combinaison qui produit la meilleure performance d'exécution du logiciel (PAN et EIGENMANN [2006]).

Néanmoins, malgré ces optimisations, il subsiste très souvent des cycles de gel du processeur lors de l'exécution dus à la lecture d'une donnée indisponible dans les mémoires caches, ainsi que décrit dans la section 2.1.

Ce chapitre présente une méthodologie qui permet à un développeur logiciel d'identifier les instructions du programme à optimiser qui produisent des gels, sans instrumentations spécifiques du processeur autre que la trace des PCs, par une analyse qui combine cette trace avec la trace bus des accès aux données en mémoire par le cache. Cette méthodologie permet de fournir au développeur des indications qui permettent d'identifier les meilleurs emplacements possibles pour l'insertion dans le code source du programme des instructions de pré–chargement de données en mémoires caches.

Des outils de profilage de programme comme *gprof* permettent aussi d'identifier les différents endroits dans lesquelles celui-ci consomme le plus de cycles d'exécution. Mais les données collectées le sont par échantillonnage statistique de PC et par injection de code, ce qui rend la mesure intrusive. De plus, ces données ne fournissent pas à un développeur de programmes embarqués des informations à un niveau de détail suffisamment fin, qu'il pourrait par exemple relier à la microarchitecture. Les informations données par les outils de profilage se limitent à fournir les coûts (en termes de temps d'exécution) des différentes fonctions qui constituent le programme. Une telle approche peut être utilisée en complément de la méthode que nous proposons, car nous identifions pour une fonction donnée, les instructions qui contribuent au rallongement du temps d'exécution de ladite fonction du fait de la latence de la hiérarchie mémoire.

L'objectif est d'apporter au développeur des outils permettant de fournir des informa-

tions sur les instructions du programme qui contribuent le plus à la dégradation de la performance du logiciel. Ceci de manière à ce qu'elles soient utilisées comme indices pour faciliter l'insertion manuelle des instructions de pré–chargement de données et de favoriser ainsi la disponibilité de ces données au moment où le processeur en a besoin, pour une amélioration de la performance (réduction de la durée d'exécution du programme) par une augmentation de l'utilisation des capacités de traitement disponibles sur le matériel.

5.2 Exécution de l'application sur SoC virtuel

Pour aider les ingénieurs à prendre de bonnes décisions pour l'optimisation de l'utilisation du matériel par le logiciel, il est important de produire des informations pertinentes pour le développement d'une stratégie d'optimisation. Pour cela, le programme à optimiser est exécuté sur un SoC réel ou sur une plateforme virtuelle précise au niveau du cycle. Au cours de l'exécution du programme, des traces précises au niveau du cycle (CA) sont extraites. Ces traces peuvent servir à différents usages (HEDDE et PÉTROT [2011]) ; dans notre cas, elles visent l'élimination des gels du processeur. Nous extrayons deux traces qui servent d'entrées à la méthode développée. Elles sont générées lors de l'exécution du programme.

5.2.1 Production de la trace au niveau du bus

La trace du bus est générée par observation du trafic à la sortie du dernier niveau de cache (LLC ou cache de niveau final $n, n \geq 2$) en direction du contrôleur mémoire, à travers le bus ou l'interconnexion. Elle est extraite de manière à ne pas fausser les mesures, grâce à la capacité des plateformes virtuelles (en simulation ou émulation matérielle) à être instrumentées sans impact sur le déroulement fonctionnel et temporel de l'exécution. La trace bus est obtenue de manière non-intrusive à ce niveau grâce à la simulation ou à l'émulation matérielle, ou même, si le circuit silicium en dispose, par un bloc de génération de traces.

Ce point d'extraction est choisi parce que c'est là que la latence est celle qui est subie par le sous-système constitué du processeur et des caches. Le chemin d'accès à la mémoire y est le plus long, comparé à d'autres points d'observation qui seraient situés plus en amont c'est–à–dire plus proches du processeur. La figure 2.6 (Chapitre 2) donne une vue globale de la latence dans une hiérarchie mémoire simple dans un SoC. Ce sont les échecs qui se produisent sur le dernier niveau de cache qui sont les plus pénalisants. Ce sont ces échecs qui conduisent à ce que le cache aille réellement chercher la donnée requise en mémoire via l'interconnexion, et non simplement à la charger à partir du cache de niveau suivant. Aussi l'on dispose d'une marge de manœuvre (impact plus importante de la correction) plus

importante sur les échecs qui surviennent au LLC de la hiérarchie mémoire, comparé aux échecs qui surviennent dans le cache L_1, qui sont moins pénalisants (typiquement 10 fois moins pénalisant qu'un cache L_2).

La figure 5.1 présente quelques signaux qui constituent le bus AXI et le point d'observation à travers laquelle passe la trace qui contient ces signaux. La trace contient les signaux aux dates successives de lecture et d'écriture en DRAM.

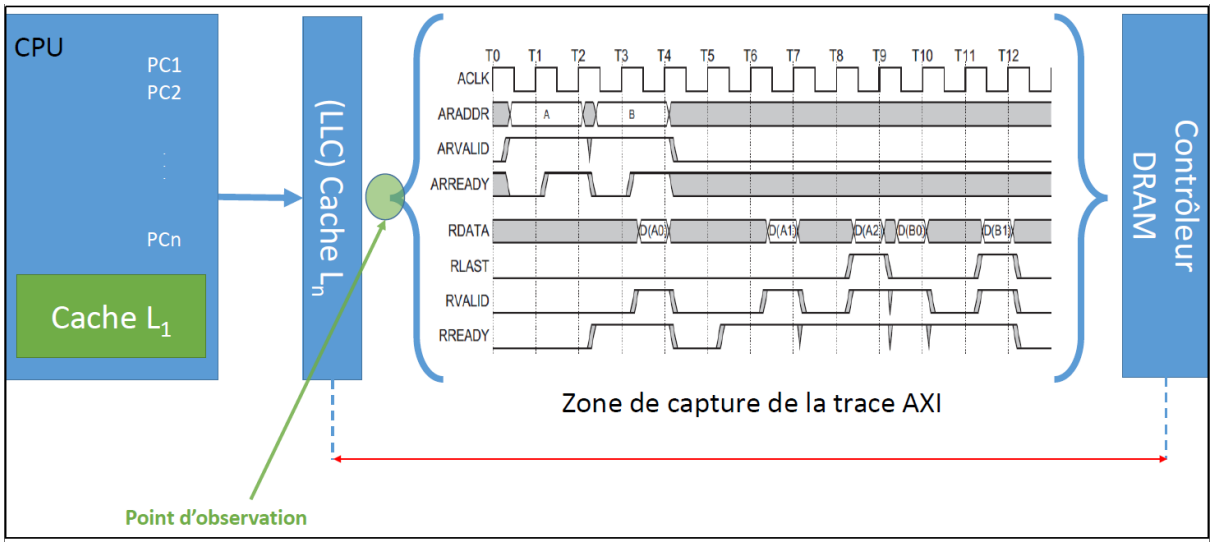

FIGURE 5.1: Zone de capture de la trace qui contient tous les signaux du bus AXI

Ces signaux proviennent de l'interface bus AXI qui effectue le transfert de données entre la mémoire accédée (DRAM ou autre, flash par exemple) et le CPU. Il est à noter que dans cette thèse le protocole d'interconnexion utilisé dans les expérimentations est le protocole Advanced Microcontroller Bus Architecture (AMBA)–AXI (ARM-AMBA-SPECIFICATION [2017]). AXI est un bus développé par ARM Ltd et dont le protocole est spécifié dans le standard de-facto AMBA. C'est une spécification d'interface et de protocole d'intercommunication des blocs dans un SoC permettant d'intégrer typiquement des sous-systèmes processeurs, des contrôleurs mémoires et des périphériques d'entrées–sorties. La figure 5.2 présente des points typiques (en vert sur figure) auxquelles les interfaces AMBA peuvent être retrouvées dans un SoC.

Dans le reste du document, ***rafale*** sera utilisé pour désigner « burst ».

La trace des signaux du bus de données est ensuite lue et analysée (automatiquement) par notre outil, afin de reconstituer les différentes transactions de lecture et d'écriture effectuées entre le processeur et le contrôleur mémoire. Ces transactions sont générées sous forme de structures de données telles que présentées dans la figure 5.3. On observe dans chacune des structures de données : l'adresse accédée par la transaction, le type de rafale,

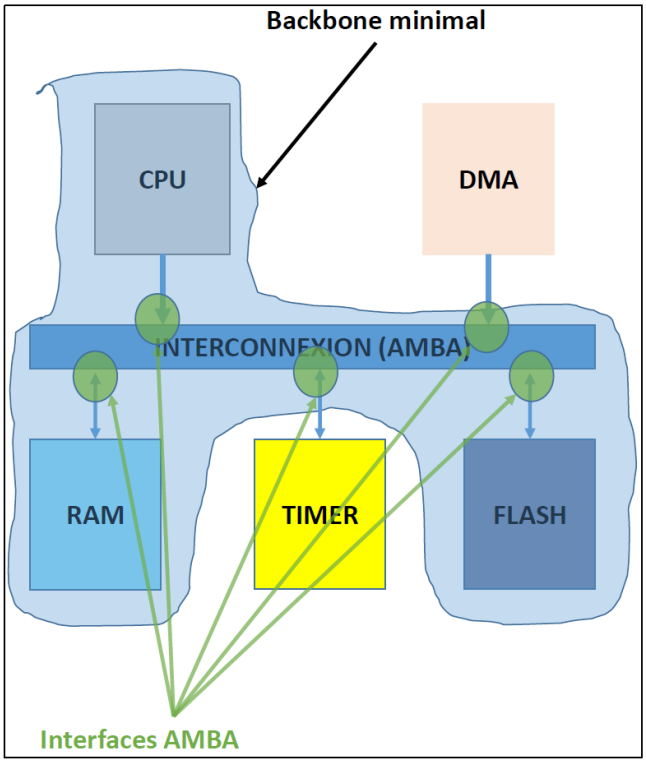

FIGURE 5.2: Exemple d'interfaces AMBA dans l'ossature d'un SoC

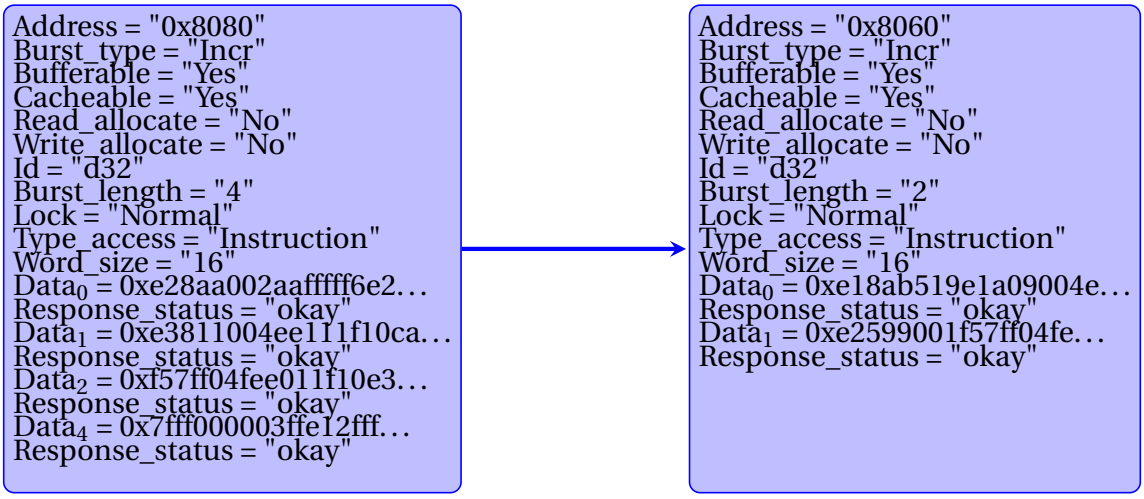

FIGURE 5.3: Exemple de transactions reconstituées à partir des signaux AXI capturés dans la trace au cours de l'exécution du programme testé

l'identifiant de la transaction, le type d'accès effectué qui a provoqué la génération de la transaction, la taille d'un mot de donnée transférée par chaque rafale, les données transférées par la transaction, ...

Il est à noter qu'une transaction est constituée de plusieurs rafales qui effectuent des transferts tel que présenté dans la figure 5.4. Cette figure présente une image de l'encapsula-

Address	L'adresse mémoire accédée
Burst type	Définit le type de transfert bloc de type de rafale, tel que (Incr, Wrap, ...) : une transaction peut être composée de plusieurs rafales
Burst length	Indique la longueur de la rafale
Cacheable	Indique que le contenu de la transaction peut être mis en cache. Si la transaction peut être pré–chargée ou chargée une seule fois pour plusieurs transactions de lecture. Pour l'écriture il indique que plusieurs écritures peuvent être mergées ensemble
ID	Indique l'identifiant de la transaction globale (chaque rafale peut aussi avoir son ID)
Type access	Définit le type d'accès effectué par la transaction (instructions ou données)
Response status	Définit l'acquittement effectué à la fin de chaque transfert effectué par une rafale/burst de la transaction
$Data_0 \ldots Data_n$	Représentent les octets de données transférées par chacune des rafales/bursts de la transaction

TABLEAU 5.1: Description des champs principaux qui font partie d'une transaction AXI

tion entre transaction, rafale et transferts de données entre un composant maître et un composant esclave, ceci pour chacune des transactions de lecture de données. Les adresses et les informations de contrôle sont envoyées en premier au port esclave qui fournit en réponse à la requête du maître, des transferts de données. Chaque transfert a un nombre d'octets fonction de la largeur du bus/interconnexion.

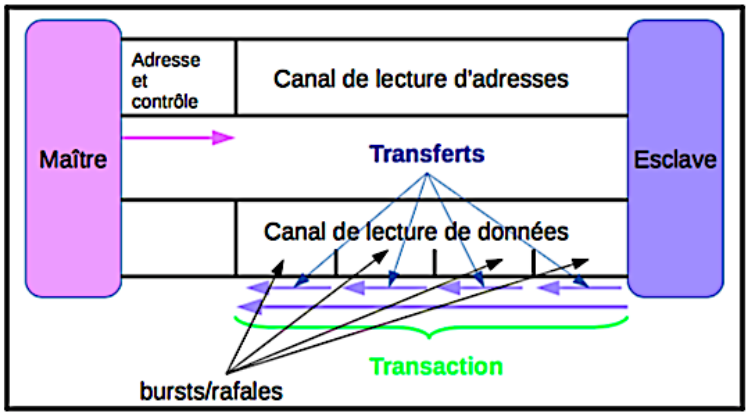

FIGURE 5.4: Exemple d'encapsulation entre transaction, transferts et rafale pour une transaction de lecture de données entre un composant maître et un composant esclave.

Le tableau 5.1 fournit une description de quelques champs qui constituent la transaction. Par exemple, on y trouve le type d'accès de la transaction, qui permet de savoir si c'est une transaction de lecture de données ou d'instructions, ou encore l'adresse physique de la plage d'adresses des données (ligne de cache) qui seront copiées en mémoire cache. Il est important de mentionner que la valeur de l'adresse contenue dans la transaction ne correspond pas toujours précisément à la valeur de l'adresse de la donnée présente dans la requête émise par le processeur et qui a engendré la génération de ladite transaction suite à des échecs dans les différents niveaux de cache. Cette adresse peut correspondre à la première adresse d'une plage d'adresses qui contiendra la donnée requise par le CPU suivant l'organisation de la hiérarchie mémoire (exemple : donnée située au milieu d'une ligne de cache). Dans d'autres cas, l'adresse contenue dans la transaction peut parfaitement correspondre à l'adresse de la donnée requise par le processeur : c'est le cas notamment si cette adresse est parfaitement alignée sur le début d'un bloc de données à transférer. Sans perte de généralité, nous ferons l'hypothèse que l'adresse requise et l'adresse de démarrage de la transaction émise par le CPU appartiennent au même bloc sans plus de contraintes.

Nous nous intéressons uniquement aux transactions de lecture de données en mémoire principale qui ont pour but de recharger une ligne de cache. Elles sont émises lorsque la lecture d'une donnée génère des échecs à travers les différents niveaux de cache de la hiérarchie mémoire. Ces types de transactions sont extraits en s'intéressant précisément aux paramètres qui permettent de désigner le type d'accès, lecture d'une instruction ou d'une donnée, et le type de rafale (« burst ») de la transaction. La figure 5.5 présente l'organisation des transferts dans les trois types de rafale identifiables dans le protocole AXI :

— Les rafales fixées, qui interviennent dans les transferts dans lesquels la prochaine adresse dans la rafale ne change pas. L'adresse reste la même pour tous les transferts au sein de la transaction. Une telle requête effectue des accès répétés à la même adresse, par exemple pour le chargement ou le vidage de la FIFO d'un périphérique. Dans la figure 5.5 on peut remarquer que l'adresse du transfert 0 reste inchangée.

— Les rafales avec incréments (INCR), qui sont des transferts dans lesquels l'adresse de chaque transfert est incrémentée de la taille de la donnée. L'adresse de chaque transfert dépend de l'adresse du transfert précédent. Par exemple, l'adresse de chaque transfert dans un transfert de quatre octets est égale à l'adresse du précédent transfert plus quatre. Dans la figure 5.5 les rafales de type incrément commencent par le transfert 0, puis 1, 2, ...

— Les rafales modulo (« wrap »), qui sont similaires aux rafales de type incrément. Cependant, dans le modulo, l'adresse retourne à la plus petite adresse du transfert lorsque la

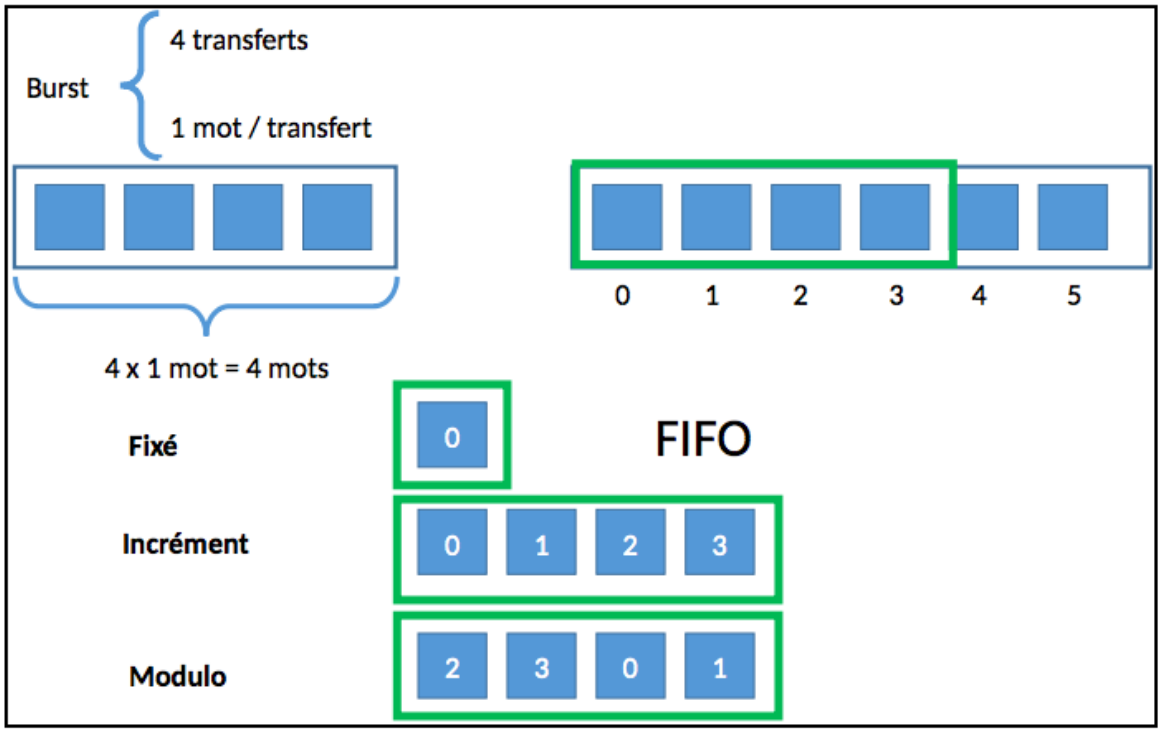

FIGURE 5.5: Exemple de rafales de longueur 4 qui traduit l'organisation des transferts dans les trois différents types de rafales identifiables dans les transactions.

limite du modulo est atteinte. La limite du modulo est égale à la taille de chaque transfert multipliée par le nombre total de transferts dans la rafale. En pratique, et pour des raisons de simplicité d'implantation, dans ce cas la longueur de la rafale b est typiquement une puissance de deux comprise entre deux et seize ($2 \leq b \leq 16$) . Dans la figure 5.5 on observe que le mode rafale modulo présenté commence par effectuer le transfert 2, puis 3, avant de revenir à l'adresse du transfert 0 dont l'adresse représente la plus petite adresse de la rafale.

Dans la spécification du protocole AXI ARM [2004], les transactions de type modulo sont exactement celles qui permettent de recharger une ligne de cache : dans l'architecture ARM, il existe une bijection entre les transactions qui rechargent les caches et celles qui sont de type modulo.

D'autres bus peuvent adopter d'autres stratégies mais, sans perte de généralité là encore, nous ferons l'hypothèse que nous sommes à même d'identifier les transactions permettant de recharger les lignes de caches parmi toutes les transactions présentes dans la trace d'exécution de l'application.

5.2.2 Production de la trace des différentes valeurs du pointeur d'instructions

Dans cette trace, les différentes valeurs prises par le registre PC interne au processeur sont consignées. Il est à noter que les valeurs du registre de PC sont des adresses virtuelles. C'est–à–dire qu'elles correspondent aux adresses dans le programme et non aux adresses en mémoire principale, car elles n'ont pas encore été traduites en adresse physique par l'unité de gestion mémoire (MMU). La production de cette trace est non-intrusive car elle est produite par des fonctionnalités additionnelles dans le modèle de processeur qui ne modifient pas les instructions exécutées dans la séquence de cycle d'horloge. Notamment, elle ne nécessite pas l'ajout d'instructions supplémentaires qui effectueraient le monitoring du modèle du processeur au cours de l'exécution (SARGUR et LYSECKY [2017]). KELLER et URQUHART [1994] ont déposé un brevet sur le profilage complet et non-intrusif des processeurs. La trace est donc produite par le modèle lui–même, ce qui évite que le processeur soit perturbé pendant l'exécution du programme.

Cette opération peut être définie en trois étapes telles que présentées dans la figure 5.6 :

— La détection de l'évènement : par exemple le changement de la valeur du registre PC. Le modèle scrute de manière continue un signal pour obtenir cette information.

— La correspondance : vérification du fait que l'évènement correspond bien à un évènement qui doit être monitoré. Ceci a pour but de savoir si l'évènement doit être considéré ou pas, et ne pas surcharger le fichier trace, dont la taille deviendrait sinon rapidement très importante (empreinte disque, temps de traitement, ...).

— La collecte : la donnée est stockée dans un fichier.

FIGURE 5.6: Production non-intrusive des valeurs de PC

Le PC est un registre interne du processeur qui stocke l'adresse de la prochaine instruction à exécuter. Néanmoins, dans les architectures pipeline, chaque étage du pipeline (fetch, decode, ...) peut avoir besoin de la valeur du PC de l'instruction en cours de traitement dans l'étage en question. Il y a donc en pratique possiblement n valeurs différentes de PC présentes dans le processeur s'il possède n étages de pipeline. Nous faisons le choix de collecter les valeurs de PC qui correspondent à l'étage d'exécution du pipeline du processeur. Dans un processeur superscalaire capable d'avoir plusieurs instructions en cours d'exécution à l'étage exécution du pipeline, les différentes informations sur les valeurs de PC collectées

sont enregistrées sous la forme d'un couple (*numéro de cycle, liste des différentes valeurs de PC dans l'étage d'exécution du pipeline*).

Cycle	x
	PC_1
	PC_2
	$\vdots$
	PC_n
Cycle	y
	PC_1
	PC_2
	$\vdots$
	PC_n
...	

TABLEAU 5.2: Format d'enregistrement des valeurs du registre PC

Par exemple dans le cas du processeur ARM Cortex–A53 avec un seul coeur, il y a deux unités de type arithmétique et logique (Arithmetic–Logic Unit (ALU)). C'est un processeur capable d'exécuter 2 instructions en parallèle. Ainsi, pour chaque cycle d'exécution du programme, la trace contiendra le numéro du cycle et les deux valeurs de PC dans les deux unités de traitement. Le pipeline peut être vu dans ce cas comme une file à deux entrées. Le listing 5.1 fournit un exemple de contenu de la trace des PC extraits d'un modèle du processeur ARM Cortex–A53 précis au niveau du cycle. C'est une trace textuelle avec un format précis, défini par la spécification ARM-TARMAC [2017] Ltd. On observe qu'à un même cycle d'exécution correspondent 2 valeurs de PC. On observe sur cette figure qu'au cycle *2696000*, les instructions ayant pour PC *0x00000168* et *0x0000016c* sont à l'étage exécution du pipeline. Il en est de même pour le cycle d'exécution 3517000 auquel correspondent les valeurs de PC *0x00000180* et *0x00000184*.

Cette trace (au format tarmac) est une trace générée de manière précise au niveau cycle par un module logiciel (« plug-in ») développé par ARM et intégré au modèle. Ce plug-in surveille (« monitoring ») les différentes activités de l'exécution qui affectent l'état interne du modèle. Plusieurs type de traces (*trace des instructions exécutées, trace du diagramme d'appel des fonctions du programme, trace des registres, ...*) peuvent être générées en fonction du besoin d'étude. Il est à préciser que la trace tarmac est utilisée ici pour présenter une illustration afin de rester concret, car le format de la trace générée n'influence pas la méthodologie. Seule compte la disponibilité dans la trace des informations de PC et de transactions bus à chaque cycle (et ceci, que la trace soit composée d'un fichier ou de plusieurs, avec toutefois des temps cohérents pour les cycles dans ce dernier cas).

Dans notre cas, nous nous intéressons uniquement à la trace des différentes instructions du programme qui ont été exécutées. Elle se présente sous un format particulier qui est le suivant (ARM-TARMAC [2017]) :

<temps> <unité> [code condition] <adresse> <code opération> ...

L'élément « temps » représente l'instant du cycle d'exécution suivi de l'unité du temps. Le nombre de « tic » d'horloge est utilisé. Les instructions ARM sont prédiquées par un code condition qui représente les modifications sur l'état interne du processeur. Ce code condition indique si l'instruction a été exécutée ou transformée en « nop » (échec à un code condition). L'adresse indique l'adresse virtuelle accédée dans le code du programme exécuté, et le code opération fournit une représentation hexadécimale de l'instruction du programme exécuté. Nous précisons que la méthode utilise uniquement les instants de cycle (*temps*) et les valeurs de PC c'est–à–dire l'adresse virtuelle de l'instruction que pointe le PC dans l'étage d'exécution du pipeline à ces instants.

```
1  Temps  ...  adresse instruction, instruction decodee
2  2696000 tic ES (00000168 : e3a06000)
3  2696000 tic ES (0000016c : e3a07000)
4  2699000 tic ES (00000170 : e3a08000)
5  2699000 tic ES (00000174 : e3a09000)
6  2700000 tic ES (00000178 : e3a08000)
7  2700000 tic ES (0000017c : e3a09000)
8  3517000 tic ES (00000180 : e3a0c000)
9  3517000 tic ES (00000184 : e3a0e000)
```

Listing 5.1: Exemple d'une trace des valeurs du registre PC extraite du modèle du processeur ARM Cortex–A53 précis au niveau du cycle

5.3 Première approche intuitive d'identification des instructions gelées

Une approche pour l'identification des instructions du programme qui contribuent au rallongement du temps d'exécution serait de se focaliser sur le processeur comme le ferait un compilateur lors de la génération du code exécutable dudit programme. En effet, lorsque le compilateur génère l'exécutable d'un programme, il s'intéresse principalement à l'architecture du processeur. En particulier à l'interblocage dans le pipeline pour ordonnancer les instructions du programme de manière à ce qu'elles puissent s'exécuter de la meilleure des manières possibles sur le processeur, meilleure au sens des informations que connait le compilateur au moment de la compilation.

Un développeur de logiciels embarqués qui s'intéresse de la même manière d'abord aux instructions, pourrait alors dans un premier temps regarder dans la trace les différents PCs, et identifier ceux dont les instructions correspondantes prennent plus de cycles d'exécution

comparés à leurs CPIs idéaux lorsqu'il n'y a aucune latence d'accès à la mémoire c'est–à dire que l'accès à la donnée est instantané. Ensuite, pour chacune des instructions qui prennent plus de cycles d'exécution, il pourrait vérifier, grâce à la trace des différentes transactions, qu'une seule transaction a été émise préalablement juste avant le début d'exécution de l'instruction et s'est achevée avant la fin d'exécution de l'instruction concernée. Dans cette approche, le développeur procède par élimination progressive des PCs (instructions) et s'intéresse à l'instruction qui effectue une lecture en mémoire.

Par ailleurs, une même instruction peut s'exécuter en un nombre de cycles variables. Cette variabilité provient de plusieurs raisons. Par exemple, dans un système avec translation d'adresses, s'il y a un échec dans la table de translation d'adresses, l'instruction en cours d'exécution est obligée de patienter jusqu'à la fin de la translation avant de poursuivre son exécution.

Dans tous les cas, avec cette approche qui part des instructions, il est nécessaire de regarder s'il y a eu une transaction sur le bus entre le début d'exécution de l'instruction et la fin d'exécution de celle-ci. Dans le cas de la translation d'adresses par exemple, il n'y aurait pas de transaction de rechargement d'une ligne de cache entre le LLC et la DRAM, montrant ainsi que le long temps d'exécution de l'instruction n'est malgré tout pas causé par un gel.

Un cache étant un sous–ensemble de la mémoire principale, lors de l'exécution d'un programme, si l'on suppose que les données ne tiennent pas toutes en mémoire cache, la probabilité d'accéder à la mémoire principale est plus élevée que celle d'accès à la mémoire cache (car la DRAM stocke plus de données). De ce fait, on peut supposer que de nombreuses instructions du programme vont s'exécuter en un nombre de cycles variables.

De plus, l'exécution d'une instruction en davantage de cycles que prévus par le CPI, peut être due à l'attente de la lecture d'une donnée en provenance du cache L_2 et/ou L_n en fonction du type de cache (inclusif ou exclusif) et de la hiérarchie mémoire. La latence entre le cache L_2 et le cache L_n est peu intéressante à essayer de traiter pour la masquer, car elle est faible et de plus l'on dispose de moins de marge de manœuvre (mécanismes matériels internes au processeur, peu accessibles au programmeur). Le programmeur n'a pas la possibilité de spécifier le niveau de cache dans lequel il veut précharger.

Pour mémoire, nous nous intéressons à la latence entre le dernier niveau de cache (LLC) et le contrôleur mémoire car c'est ce chemin de la hiérarchie mémoire qui peut influencer de manière considérable le temps d'exécution d'une instruction.

On observe donc que de nombreuses instructions du programme ont des durées d'exécution supérieures au CPI idéal, et donc que pour l'analyse de trace, cette approche devient rapidement fastidieuse car nécessite l'analyse de nombreux PCs, même ceux qui ne sont pas en fait des PCs avec gel. Il devient alors important, pour rendre l'analyse de trace efficace et

donc la méthode applicable, de trouver une approche qui identifie directement les instructions qui génèrent les cycles de gel du processeur dus à l'indisponibilité des données dans les caches. Cette méthode se base sur une vision autre que celle centrée sur le point de vue du processeur.

5.4 Méthode d'identification et de quantification automatiques des cycles de gel du processeur

Le but de la méthodologie est d'identifier les instructions du programme à optimiser qui génèrent les cycles de gel du processeur et de quantifier l'influence de chacune d'elles sur le rallongement du temps d'exécution global du programme. Les informations nécessaires sont stockées dans les traces et sont fournies en entrée au flot. Grâce aux deux traces (transactions du LLC vers le bus ou interconnexion, et les valeurs de PC de l'étage exécution du pipeline) et à la méthode, la détection des cycles durant lesquels le processeur est gelé se fait du point de vue du bus plutôt que du point de vue du processeur, ce qui permet d'avoir une vision sur le comportement de l'ensemble du système plutôt qu'une vision centrée sur le comportement du processeur. Dans ce cas, l'identification des cycles de gel du processeur est indépendante des capacités du processeur à fournir une observabilité sur son comportement interne lors de l'exécution d'un programme. En effet, hormis le PC qui est une information de base nécessaire aussi à d'autres besoins tels que le débogage du logiciel, la méthode décrite ci–après ne nécessite pas d'instrumentation spécifique du processeur ou de son modèle.

Toutes les instructions de chargement de registres depuis une mémoire extérieure peuvent potentiellement être source d'un gel, puisque c'est lors de l'utilisation d'un registre cible d'un chargement depuis une mémoire extérieure que les gels se produisent. La méthode que nous définissons conduit l'outil d'analyse de traces à assigner à chacune des instructions gelées un poids qui représente un niveau potentiel de réduction des cycles de gel du processeur survenus du fait du gel de ladite instruction. Nous définissons ce niveau potentiel comme la capacité de l'instruction, dans l'hypothèse d'une correction totale des gels de cette instruction, à réduire le nombre correspondant total de cycles de gel du processeur. Plus le niveau potentiel de réduction des cycles de gel est grand, meilleure sera la réduction effective du temps d'exécution si les instructions de pré-chargement de données en mémoires caches sont ensuite insérées au bon endroit dans le code source du programme pour approcher voire atteindre une correction totale. Cet indice est calculé par l'analyseur de traces en s'appuyant sur :

➢ L'identification de l'instruction produisant les cycles de gel du processeur. Pour identifier l'instruction qui est gelée à cause d'une transaction de lecture de données en mémoire, une fusion temporelle entre la trace des valeurs des PCs et les différentes transactions, reconstituées grâce à la trace du bus, est effectuée par l'analyseur.

En posant $List_{\mathrm{PC}}$ comme étant l'ensemble des valeurs prises par le registre interne PC du processeur lors de l'exécution d'un programme, $List_{\mathrm{TX}}$ l'ensemble des différentes transactions sur le bus permettant de recharger une ligne de cache de données et $List_{\mathrm{FT}}$ l'ensemble obtenu par fusion temporelle des deux ensembles sus-cités, la fusion temporelle est faite de la manière suivante :
$\forall tx_j \in List_{\mathrm{TX}}, \forall pc_i \in List_{\mathrm{PC}}$,

$$List_{\mathrm{FT}} = \begin{cases} List_{\mathrm{PC}} \cup List_{\mathrm{TX}} & \text{si } CycleDebut(tx_j) \leq CycleDebut(pc_i) < CycleFin(tx_j) \\ \{\} & \text{sinon} \end{cases} \tag{5.1}$$

Le symbole $\cup$ dénote l'union entre deux ensembles. Ainsi, l'ensemble $List_{\mathrm{FT}}$ contiendra les éléments des deux ensembles pour lesquels la condition $CycleDebut(tx_j) \leq CycleDebut(pc_i) < CycleFin(tx_j)$ est respectée. L'ensemble $List_{\mathrm{FT}}$ sera vide $\{\}$ dans le cas contraire.

Ceci permet d'associer à chaque transaction de lecture de données en mémoire l'ensemble des instructions du programme qui ont été exécutées entre le début et la fin de la transaction.

Ensuite, chacune des transactions de lecture de données qui permet de recharger une ligne de cache est analysée, en partant du cycle marquant la fin de la transaction et en remontant jusqu'au cycle marquant son début. On suppose que les traces ont été générées lors de l'exécution du programme sur un processeur superscalaire pipeliné (cf. figure 2.3 chapitre 2). En partant du cycle de fin de la transaction, si la dernière valeur du registre PC reste inchangée pendant plusieurs cycles, alors de deux choses l'une : (a) soit le PC pointe sur une instruction dont le temps d'exécution est long, mais le processeur n'est pas gelé (par exemple translation d'adresses en cours) ; (b) soit le PC pointe sur une instruction sur laquelle le processeur attend le résultat (données) d'une transaction mémoire qui a pu avoir lieu bien avant, pour continuer son exécution, et ce dernier est donc gelé. Comme le temps d'exécution des instructions est traditionnellement court sur les processeurs choisis, nous émettrons l'hypothèse que le PC qui suit la fin de la transaction de lecture de données a été gelé jusqu'à ce que le processeur puisse procéder à l'exécution de l'instruction désignée par ce PC (c'est–à–dire que l'on néglige les instructions de temps d'exécution « long »). Ce gel de l'instruction

aura une durée maximale équivalente au nombre de cycles nécessaires pour que les données de la ligne de cache lue soient disponibles. L'instruction suivante étant gelée du fait qu'elle doit utiliser une donnée requise par le processeur pour poursuivre son traitement. Cette dernière instruction est exécutée lorsque cette donnée requise par le processeur est disponible dans le(s) registre(s) cible(s) de celui-ci, c'est–à–dire après la fin de la transaction de lecture de la donnée en mémoire d'accès long (typiquement DRAM) et le chargement effectif de la ligne de cache cible dans le cache de niveau 1 et le(s) registre(s). L'instruction identifiée est donc gelée du nombre de cycles k pendant lesquels la dernière valeur de PC associée à la transaction est restée inchangée, tel que présenté schématiquement dans la figure 5.7 dans laquelle on suppose que les traces sont obtenues suite à l'exécution du programme à optimiser sur un processeur superscalaire. Sur cette figure, étant donné l'ensemble des transactions $(tx_1, tx_2, \ldots, tx_{n-1})$ possédant le type d'accès, le type de rafale, ... et l'ensemble des PCs $(1,2,3,\ldots,m-1)$ associés temporellement à chacune de ces transactions, on identifie par exemple pour la transaction tx_{n-1} que l'instruction de PC = 27 a été gelée de q cycles pendant l'exécution du programme. Sur l'architecture superscalaire, l'instruction de PC = 27 reste bloquée pendant q cycles parce qu'elle attend la disponibilité de la donnée transportée par la transaction tx_{n-1}.

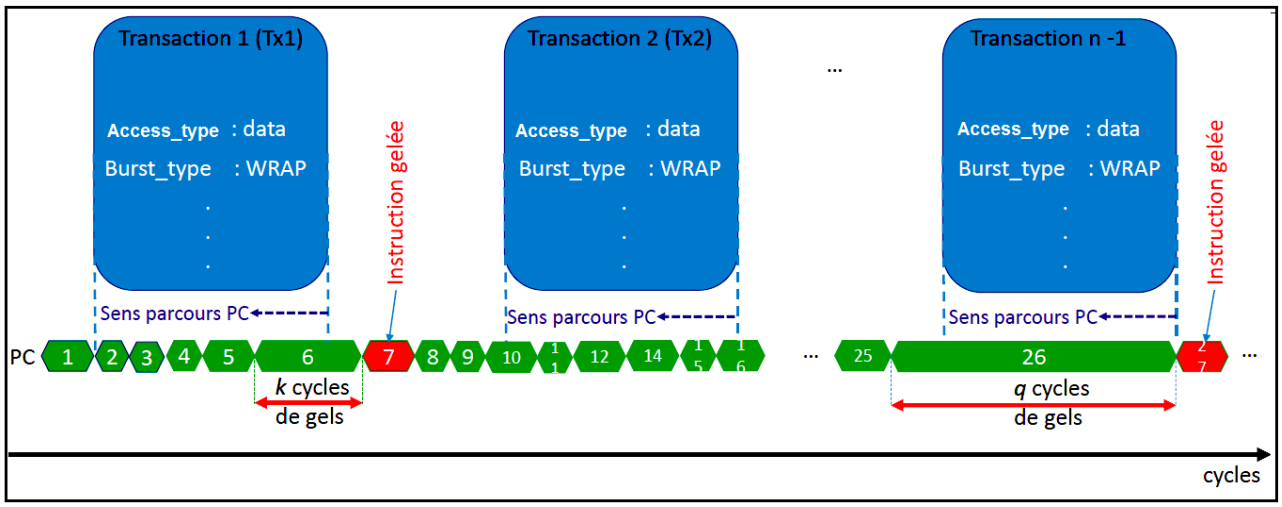

FIGURE 5.7: Identification d'une instruction gelée à partir de la trace contenant les valeurs de PC et les transactions dans le bus qui permettent de recharger des lignes de caches. Le sens de parcours définit la manière dont les PC associés à une transaction sont parcourus (du dernier au premier).

Pour le type d'architecture étudiée dans nos expérimentations, une transaction de rechargement d'une ligne de cache est caractérisée par le type rafale modulo. Sur

certaines architectures, il est parfois moins facile d'identifier les transactions qui permettent de rapprocher les données proche du processeur. Si le programme fait directement des accès aux données via des transactions en rafale qui sont non–cachées, la transaction peut ressembler sur le bus à un rechargement de cache, mais la requête de lecture de données est directement adressée au contrôleur mémoire sans passer par les mémoires caches. Dans ce cas, l'instruction du programme qui est gelée est directement l'instruction de lecture en mémoire (*load*) qui est bloquée, en supposant que le programme en cours d'analyse ne cherche pas délibérément à faire des accès non–cachés quand les accès cachés sont disponibles. Ainsi, l'optimisation par le pré–chargement dirigé par le logiciel devient moins facile. Evidemment la méthode proposée ne peut s'appliquer qu'aux parties du programme qui effectuent des accès cachés car l'un des objectifs de la méthode est d'optimiser les rechargements de caches qui sont responsables des cycles de gel de processeur.

Les accès cachés peuvent être générés en tant qu'effet de bord de l'exécution du programme, mais ne sont pas forcément voulus par le programme, comme les accès non-cachés. Par définition, ces accès non–cachés ne sont pas faits en parallèle de l'exécution normale d'un programme ; ils en font partie. Cependant une interface bus évoluée peut lancer successivement plusieurs accès de lecture sans attendre que le premier accès soit terminé. Si l'on voulait appliquer sur ces accès non–cachés une technique du même genre que l'approche proposée, on pourrait imaginer que le développeur déplace vers l'amont dans le code source du programne, les instructions de lecture de données en mémoire ; en garantissant que cela n'engendre la cassure de certaines dépendances de données entre instructions (par exemple remise en ordre par l'architecture bus ou Réseau sur puce – ou Network–on–Chip (NoC) des retours de transactions de lecture avant présentation des données au processeur).

Par ailleurs, si le chargement de données est effectué directement dans un ou plusieurs registres du processeur, il faut que le chargement possède un mécanisme qui permet d'alerter le processeur que la donnée chargée est désormais disponible dans le(s) registre(s) cible(s). Ce mécanisme est contrôlé en général par un ensemble de registres qui contient l'état des échecs d'accès (Miss Status Holding Register (MSHR)) comme expliqué dans l'article de KROFT [1981]. Ainsi, le processeur peut poursuivre l'exécution du programme pendant que l'échec est en cours de traitement. Ce mécanisme permet au processeur d'effectuer de multiples accès aux caches même si un échec de lecture d'un bloc de données est en cours de traitement. Dans ce cas, pour un accès au cache L_1, le MSHR est consulté pour vérifier s'il y a un échec en cours déjà survenu sur le même bloc de données qui est en cours de lecture. Plusieurs cas se

présentent :

— Un échec est en cours de traitement et le bloc de données requis par la lecture n'est pas encore disponible. Dans ce cas, aucune requête n'est émise. Aucune allocation d'une instance du registre n'est faite dans le MSHR.

— Un échec est en cours de traitement et le bloc de données requis par la lecture est disponible. Dans ce cas, la donnée est transférée directement à la dernière instruction de lecture correspondante.

— Aucun échec en cours de traitement. Une nouvelle instance de registre du banc de registres MSHR est allouée. Un processeur peut disposer d'un nombre fini de registres de gestion des échecs.

— S'il n'est plus possible d'allouer une instance du MSHR, il y a génération d'un gel en attendant la disponibilité d'une instance du registre MSHR.

Tant qu'il existe une instance disponible dans le MSHR, le processeur peut accéder à la mémoire cache même lorsqu'un échec est en cours de traitement. Il est à noter que le mécanisme contrôlé par le MSHR ne change pas la méthode proposée dans ce chapitre mais permet d'apporter plus de détails afin d'aider à la compréhension.

➢ Intéressons–nous maintenant aux poids respectifs des différentes instructions : le nombre de cycles de gel du processeur produit par chacune d'elles. Rappelons tel que présenté dans la figure 2.9 qu'au cours de l'exécution d'un programme, une même instruction peut être gelée plusieurs fois, avec des durées de gels différentes. Ainsi, au cours du temps, la fréquence d'apparition d'une instruction augmentera son influence si la latence cumulée générée par celle–ci est importante. La latence cumulée pour une instruction gelée au cours de l'exécution est donc la somme de la latence de chacune des occurrences de cette même l'instruction gelée.

Il est à noter que la latence moyenne lors de l'exécution d'un programme est donnée par la relation (5.2). Logiquement, si l'on arrive à réduire la latence d'un échec, et a fortiori si l'on arrive à faire disparaître certains échecs (latence d'échec nulle vue de l'instruction en exécution), la latence moyenne s'en trouve réduite.

$$\mathrm{L}atence_{moyenne} = \mathrm{L}atence_{hit} + \mathrm{F}réquence_{échec}(\%) \times \mathrm{L}atence_{échec} \tag{5.2}$$

Le nombre de cycles de gel du processeur est calculé pour chacune des instructions gelées identifiées, en additionnant les nombres de cycles de gel de chacune des occurrences de gel de cette instruction. Nous appellerons « poids » cette somme. Plus le nombre total de cycles de gel associés à une instruction est grand, plus grand est son poids c'est–à–dire son influence sur le temps d'exécution du programme. Ainsi,

en comparant le poids respectif des différentes instructions gelées, il est possible de quantifier la contribution respective, que l'on peut également qualifier d'influence, de chacune des instructions gelées, au rallongement du nombre total de cycles d'exécution du programme par rapport à une exécution idéale sans cycles de gel. Il est à noter que le nombre de cycles de gel du processeur permet de savoir s'il est encore possible d'accélérer l'exécution du programme, en réduisant ces cycles de gel, et ceci en traitant d'abord les gels de poids le plus élevé afin de rationaliser l'effort d'optimisation.
LEE et SHRIVASTAVA [2008] proposent dans leur article de traiter les différentes occurrences de gel d'une même instruction de manière statique et en les regroupant pour créer un grand gel, puis de faire basculer le processeur en mode faible consommation d'énergie pour traiter chacun de ces grands gels. Dans notre cas, nous cherchons à réduire les cycles de gel, pour que le programme arrive plus rapidement au résultat. Par ailleurs, nous ne jouons pas sur la modification de l'état du processeur pour corriger les cycles. Aussi, nous n'effectuons pas le traitement dynamique des cycles de gel c'est–à–dire l'identification et la correction des cycles de gel du processeur au cours de l'exécution du programme à optimiser. Notre analyse de l'exécution conduit à des possibilités d'optimisation du code source, prises en compte ensuite avant la compilation et donc à l'exécution. Il pourrait être intéressant de rechercher si les techniques mises en œuvre dans les deux cas (celui de l'article précité, et le cas présent) pour des buts en apparence contradictoires (augmenter la performance instantanée dans notre cas, réduire la consommation d'énergie dans le cas de l'article), pourraient éventuellement être combinées dans une certaine mesure, pour un objectif global commun (réduction de la consommation en considérant la totalité de l'application, qui est de fait réduite par les deux techniques).

L'algorithme 3 décrit en pseudo–langage l'identification des cycles de gel du processeur et produit la liste des instructions qui ont été gelées au cours de l'exécution du programme, ainsi que leurs poids de gels respectifs. Il permet de calculer la contribution des cycles de gel de chacune de ces instructions gelées identifiées par rapport au nombre total de cycles d'exécution du programme à optimiser. L'algorithme prend en entrée la liste des transactions qui passent sur le bus entre le LLC et la DRAM extraites de la trace bus, et la liste des valeurs de PC construite à partir de la trace de PCs ; traces issues de l'exécution du modèle instrumenté et simulé de l'ossature du SoC. Ces deux listes comportent les informations d'instant dans le temps pour chaque évènement (transaction ou instruction). L'algorithme produit en sortie la liste des instructions qui ont été gelées au cours de l'exécution du programme à optimiser, et le poids de chacune (nombre total de cycles de gel de chacune des instructions

Algorithm 3: Identification instructions gelées + cycles de gels du processeur + pertinence du gel

Input: T($tx_1, tx_2, ..., tx_{n-1}$) : Liste des transactions

Input: P($pc_1, pc_2, ..., pc_{n-1}$) : Liste des valeurs du registre PC

Output: PC_{geles} : Liste des instructions gelées avec $PC_{geles} \subset P$

```
1  TP ← T ∪ P : TP liste obtenue par fusion temporelle en respectant la relation (5.1);
2  ListePC : Liste des PC associés à chacune des transactions tx_i;
3  for tx_i ∈ TP do
4      if tx_i est une transaction de lecture de donnees then
5          if tx_i est une transaction de type rafale modulo then
6              for PC_j ∈ tx_i → ListePC(PC_j, ..., PC_n, PC_{n+1}) do
7                  //PC_{n+1} représente PC instruction exécutée juste à la fin de tx_i;
8                  if
                     PC_j inchangé sur plusieurs cycles entre CycleDebut(tx_i) & CycleFin(tx_i)
                   then
9                      if PC_{n+1} ∈ (tx_i → ListePC) déja gelé then
10                         AugmenterPoids(PC_{n+1}, PC_{geles});
11                         AugmenterNombreCycleGel(PC_{n+1}, NombreDeCyclePC_j Inchange);
12                     end
13                     else
14                         PC_gele(PC_{n+1}, Vrai)//PC_{n+1} gelé première fois;
15                         NombreCycleGel(PC_{n+1}) ← NombreDeCyclePC_j Inchange;
16                         PC_{geles} ← PC_{geles} ∪ PC_{n+1};
17                     end
18                 end
19             end
20         end
21     end
22 end
```

identifiées).

Après avoir défini cet algorithme, nous l'avons implémenté sous forme d'un outil d'analyse automatique de traces. Son expérimentation sur des traces est décrite au chapitre expérimentation (6), ainsi que l'aide qu'apporte son résultat automatique au développeur de logiciel, pour l'insertion manuelle des instructions de pré–chargement dans le code source de son programme.

5.5 Conclusion partielle

La figure 5.8 présente globalement le flot méthodologique que nous proposons, ainsi que son niveau d'automatisation obtenu. Ce flot permet d'apporter aux développeurs des informations nécessaires pour caractériser, d'un point de vue système, la façon dont le logiciel utilise les capacités du matériel (cache, bus/interconnexion, contrôleur mémoire, mémoire) pour l'accès efficace aux données cachées. Le flot prend en entrée un programme source décrit en langage de haut niveau (C ou C++ par exemple). Ce code source est compilé en croisé (« cross-compilation ») avec des options de compilation qui permettent d'optimiser le code exécutable. Par exemple le temps d'exécution (–O_n avec GCC), ou encore l'insertion automatique des instructions de pré–chargement par le compilateur (–fprefetch-loop-array). Le code source est donc compilé pour une architecture cible (Cortex–A9 ou Cortex–A53 32–bit par exemple). Le fichier exécutable produit est ensuite exécuté sur une plateforme précise au niveau du cycle, telle que par exemple une plateforme de simulation ou d'émulation matérielle, pour produire les deux traces d'exécution à exploiter par notre méthode décrite dans ce chapitre : la trace des valeurs du registre PC au cours de l'exécution et la trace des différents signaux du bus. Les signaux du bus sont analysés pour reconstituer les différentes transactions de lecture ou d'écriture en mémoire principale.

Les deux traces sont fournies en entrée à l'analyseur de traces, algorithme (cf algorithme 3) que nous avons implémenté, qui en effectue une fusion temporelle (5.4) dans le but de 1) quantifier le gain potentiel maximum qu'il est possible d'obtenir si tous les cycles de gel du processeur sont corrigés, 2) produire l'ensemble des instructions (valeurs de PC) qui ont été gelées au cours de la simulation, avec pour chaque instruction gelée le poids de gel.

Pour chacune des instructions gelées identifiées, nous avons calculé sa contribution au rallongement du temps d'exécution global du programme : la somme des cycles de gel des occurrences de cette instruction au cours de l'exécution du programme. L'information de cette influence a pour but d'aider un développeur de programme embarqué 1) à prendre des décisions sur l'utilité de continuer à chercher à gagner des cycles pour une exécution plus rapide de son programme, 2) à choisir alors les gels du processeur les plus pertinents

à corriger, qui apporteront une réduction principale du temps d'exécution global du programme.

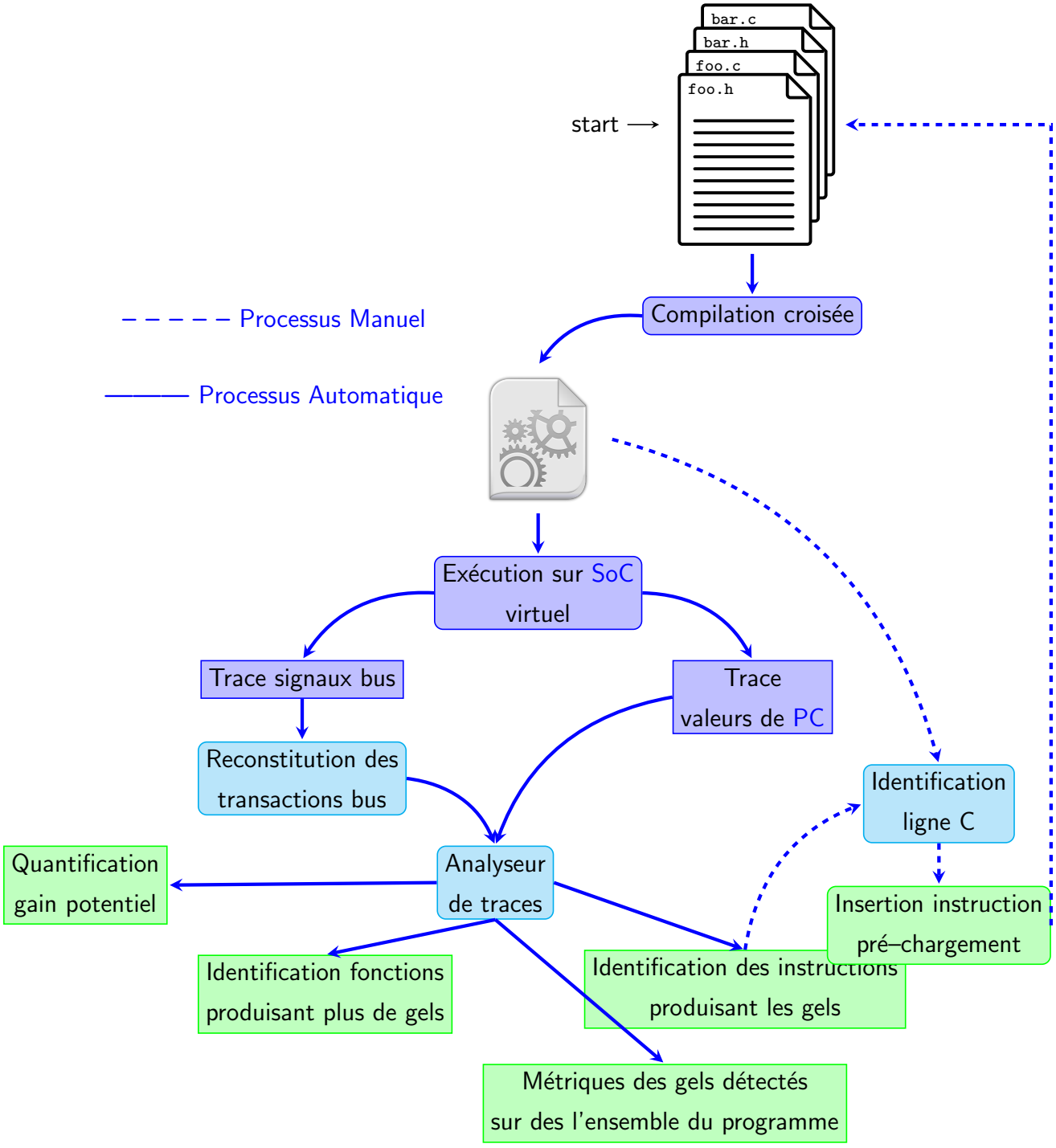

FIGURE 5.8: Flot itératif d'identification des instructions gelées et de quantification de l'influence de chacune des instructions gelées identifiées.

Grâce au fichier exécutable du programme et à la liste des PCs des instructions gelées, il est alors possible d'identifier les différentes lignes correspondantes dans le programme

source écrit en langage de haut niveau (typiquement en C dans le domaine des systèmes embarqués ou enfouis), comme par exemple par utilisation classique d'un débogueur source muni d'une vue du code assembleur. Ceci permet de connaître précisément les endroits du code source (lignes) qui produisent les gels du processeur du fait de l'absence des données en mémoire cache.

Dans l'état actuel de la méthodologie, l'insertion ensuite d'instructions de pré–chargement de données dirigé par le logiciel est réalisée manuellement par le programmeur. Il est ensuite possible d'itérer cette méthode (re–exécution de l'analyse automatique sur la nouvelle trace d'exécution du code modifié manuellement) afin d'augmenter par étapes la performance (réduction du temps d'exécution du programme). Sachant que l'insertion des instructions de pré–chargement de données dans le code source du programme peut causer l'éviction de lignes de cache qui auraient été utiles pour d'autres instructions peu après l'instruction gelée traitée, dans certains cas la pénalité peut être plus élevée que le gain ; il est donc conseillé de suivre une campagne d'optimisation organisée recensant les gains et potentiellement les pertes lors de chaque insertion d'instructions de pré–chargement de données dirigé par le logiciel dans le code source du programme.

La méthodologie proposée fait des hypothèses relativement faibles concernant les informations nécessaires à sa mise en œuvre : uniquement la trace PC avec cycles, et la trace des signaux bus avec cycles. Ces traces sont disponibles sur différents types de plateformes d'exécution (simulation précise au niveau du cycle, émulation matérielle, voire FPGA instrumenté). Ce qui permet de rester indépendant de l'architecture interne du processeur, de la topologie de l'interconnexion (bus, NoC), de la technologie du noyau de simulation, et de la manière dont les traces analysées par la méthode sont extraites de l'exécution.

5.6 Références

ANDERSON, J. M., L. M. BERC, J. DEAN, S. GHEMAWAT, M. R. HENZINGER, S.-T. A. LEUNG, R. L. SITES, M. T. VANDEVOORDE, C. A. WALDSPURGER et W. E. WEIHL. 1997, «Continuous profiling : Where have all the cycles gone ?», *ACM Trans. Comput. Syst.*, vol. 15, n° 4, doi :10.1145/265924.265925, p. 357–390, ISSN 0734-2071. URL http://doi.acm.org/10.1145/265924.265925.

ARM. 2017, «Introducing 2017's extensions to the arm architecture», https://community.arm.com/processors/b/blog/posts/introducing-2017s-extensions-to-the-arm-architecture. [En ligne ; dernier accès le 07 Novembre 2017].

ARM, L. 2004, «AMBA AXI protocol specification», http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022b/index.html. [En ligne; dernier accès le 27 juillet 2017]. 90

ARM-AMBA-SPECIFICATION. 2017, «AMBA AXI protocol specification», https://www.arm.com/products/system-ip/amba-specifications. [En ligne; dernier accès le 30 Octobre 2017]. 86

ARM-TARMAC. 2017, «Tarmac trace file format», https://developer.arm.com/docs/dui0845/f/tarmac-trace-file-format/instruction-trace. [En ligne; dernier accès le 13 Novembre 2017]. 92

HEDDE, D. et F. PÉTROT. 2011, «A non intrusive simulation-based trace system to analyse multiprocessor systems-on-chip software», dans *22nd IEEE International Symposium on Rapid System Prototyping*, IEEE, p. 106–112. 85

KELLER, T. et R. URQUHART. 1994, «Non-invasive trace-driven system and method for computer system profiling», URL https://www.google.com/patents/US5355487, uS Patent 5,355,487. 91

KROFT, D. 1981, «Lockup-free instruction fetch/prefetch cache organization», dans *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, IEEE Computer Society Press, Los Alamitos, CA, USA, p. 81–87. URL http://dl.acm.org/citation.cfm?id=800052.801868. 98

LEE, J. et A. SHRIVASTAVA. 2008, «Static analysis of processor stall cycle aggregation», dans *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '08, ACM, New York, NY, USA, ISBN 978-1-60558-470-6, p. 25–30, doi :10.1145/1450135.1450143. URL http://doi.acm.org/10.1145/1450135.1450143. 100

NAIR, A. et R. LYSECKY. 2008, «Non-intrusive dynamic application profiler for detailed loop execution characterization», dans *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, ACM, New York, NY, USA, ISBN 978-1-60558-469-0, p. 23–30, doi :10.1145/1450095.1450102. URL http://doi.acm.org/10.1145/1450095.1450102.

PAN, Z. et R. EIGENMANN. 2006, «Fast and effective orchestration of compiler optimizations for automatic performance tuning», dans *International Symposium on Code Generation and Optimization*, IEEE, p. 12 pages. 84

SARGUR, S. et R. LYSECKY. 2017, «Non-intrusive dynamic profiler for multicore embedded systems», dans *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, p. 500–505, doi :10.1109/ASPDAC.2017.7858372. 91

Chapitre 6

Expérimentation

Sommaire

6.1 Méthodologie expérimentale sur plateforme ARM Cortex-A9

6.1.1 Description de l'environnement de simulation

L'expérimentation porte sur l'insertion manuelle des instructions de pré–chargement de données dans le code source du programme testé en exploitant l'observabilité et la flexibilité fournies par les plateformes virtuelles précises au niveau du cycle. L'outil utilisé pour effectuer les travaux est ARM SoC Designer(ARM [2017]). C'est un outil qui permet la création et la simulation des prototypes virtuels. SoC Designer est un simulateur de modèles (SoC dont processeur et blocs/IPs) ; les modèles des blocs/IPs sont développés en C++, et le modèle du processeur est capable d'exécuter du code machine ARM (Instruction Set Simulator (ISS)).

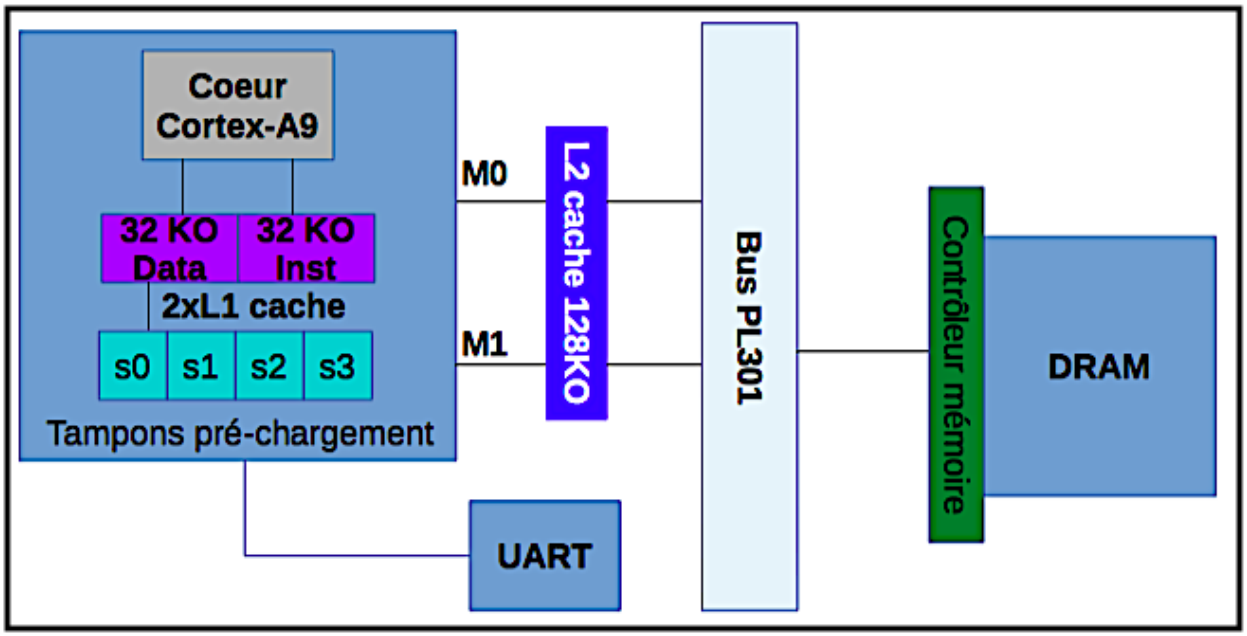

FIGURE 6.1: Diagramme de l'ossature de la plateforme architecturée autour du ARM Cortex–A9

L'expérimentation est effectuée sur une plateforme dont le diagramme en blocs est fourni dans la figure 6.1. Cette plateforme est architecturée autour du processeur ARM Cortex-A9 intégrant un seul cœur avec une fréquence d'horloge d'1GHz. Ce processeur intègre deux caches L_1 de type 4-way set associatif de 32 KO, un pour les instructions et l'autre pour les données avec une ligne de cache de 32 octets.

Il dispose également de quatre tampons (*s0, s1, s1, s2*) dédiés au pré–chargement de données. Ces tampons stockent temporairement les données préchargées avant de les copier ensuite dans le cache L_1 de données . Ces tampons de pré–chargement permettent d'exécuter simultanément plusieurs instructions de pré–chargement de données. La taille

de chaque tampon de pré–chargement est égale à celle d'une ligne de cache. Ces tampons de pré–chargement sont également disponibles sur d'autres architectures tels AMD, Scalable Processor Architecture (SPARC).

La plateforme intègre également un cache L_2 (PL310) de 128 KO de type set associatif partagé entre les données et les instructions. C'est le dernier niveau de cache. Ce cache fonctionne à la même fréquence que le CPU. Les communications entre les différents composants de la plateforme sont gérées par le modèle d'interconnexion PL301 au protocole de type AXI.

La plateforme intègre également un contrôleur mémoire directement relié à la DRAM. Ce contrôleur mémoire traite toutes les requêtes (lectures et écritures) adressées en direction de la mémoire principale. Le contrôleur mémoire utilisé est configurable, et il est possible de faire varier sa latence (en cycles) d'accès à la mémoire. Nous utilisons ceci afin de tester différentes configurations et visualiser les effets de la latence sur le gain potentiel (en terme de cycles d'exécution) qu'il est possible d'obtenir si les cycles de gel du processeur sont évités.

Cette plateforme est assez simple, mais représentative d'un sous–système matériel que l'on peut rencontrer dans divers SoCs. Elle permet d'émettre des hypothèses et faire des analyses réalistes sur le comportement de l'ensemble du système. Par exemple, dans un environnement sophistiqué, si le bus est relaxé en anticipant les échecs qui peuvent survenir dans les caches de données en se basant sur l'approche méthodologique proposée dans le chapitre 4, les performances globales du SoC peuvent être améliorées. Ainsi, le programme s'exécutera en utilisant mieux les capacités du matériel.

6.1.2 Méthodologie expérimentale et cas simple du tri à bulles

Le premier programme utilisé dans cette partie expérimentale est le tri à bulles. C'est un exemple de programme simple à contrôle dynamique qui permet d'illustrer comment les capacités des plateformes virtuelles à fournir des détails sur l'exécution d'un programme, peuvent aider à insérer convenablement et non-intuitivement des instructions de pré–chargement de données dans le code source d'un programme. La figure 6.2 présente un exemple d'utilisation du cache L_1 de données lors de l'exécution du tri à bulles (listing 6.1) sur une plateforme présentée plus haut dans la figure 6.1.

En utilisant les informations de profilage recueillies lors de la simulation, nous vérifions que le processeur est gelé pendant un certain nombre de cycles pour chacun des échecs de lecture de données (*flèches grises*) dans le cache L_1. Une fois la ligne de cache (de huit éléments parce que dans ce cas d'utilisation, la ligne de cache est de 32 octets et chaque élément est représenté sur quatre octets) qui contient la donnée lue est copiée dans le cache L_1,

aucun autre échec de lecture de données n'est observé (*flèches noires*). Ceci jusqu'à ce que le CPU ait fini le traitement des sept valeurs précédemment préchargées en supposant que l'on se trouve ni au début, ni à la fin du tableau traité. Cette analyse simple aide à prendre la décision de copier en amont certaines données dans la mémoire cache dans le but d'éviter les échecs de lecture de données.

```
1  void bubbleSort (int32_t *array, const uint32_t length){
2  bool swap = true ; uint32_t n = length-1;
3  while ((n != 0 ) && swap){
4      swap = false ;
5      for (uint32_t j = 0 ; j < n ; j++){
6          if (array[j] > array[j+1]){
7              swapElement (array, j, j+1);
8              swap = true ;
9          }
10     }
11     n--;
12     }
13 }
```

Listing 6.1: Code C du tri à bulles

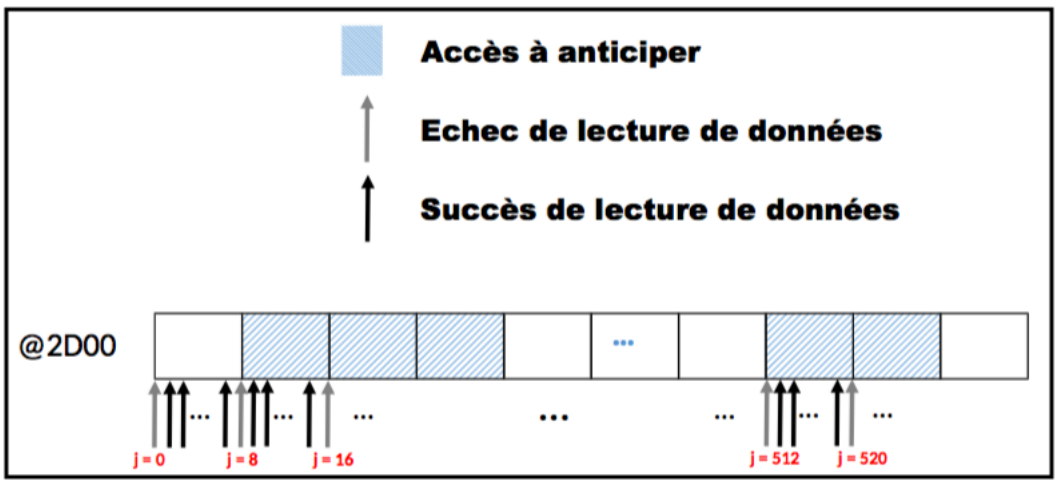

FIGURE 6.2: Comportement du cache L_1 de données du processeur au cours d'une première exécution du tri à bulles.

Pour copier les données en avance en mémoire cache et au moment opportun, une première approche intuitive consiste à précharger les données en respectant le prédicat suivant :

Précharger si (J modulo (ligne_cache / sizeof(int32_t)) == 0)

Puisque les opérandes de l'opération modulo sont des constantes et des puissances entières de deux, ce prédicat est remplacé à la compilation par *(j & ((ligne_cache » 2) -1))* pour que le code exécutable soit plus efficace. Ce qui signifie que le prochain bloc de données de 8 éléments est préchargé si l'actuelle adresse accédée est un multiple de *(ligne_cache / sizeof(int32_t))*. En appliquant cette approche intuitive, la performance de l'application n'est pas améliorée. Le temps d'exécution (en terme de cycles) du programme est plus long que la durée d'exécution de la version originale du code (sans instructions de pré–chargement).

Cependant, grâce aux informations de PMU, on observe une légère réduction du nombre de cycles de gel du processeur et du nombre d'échecs de lecture de données dans le cache L_1. En analysant le code source de la fonction, combinés à l'analyse des informations de PMU sur le nombre de branchements, on observe que cette mauvaise performance du programme avec instructions de pré–chargement est principalement due aux multiples comparaisons et branchements créés par le prédicat inséré. En effet, le tri à bulles est un programme à contrôle dynamique, et chaque contrôle dépend des données calculées au cours de l'exécution. Ainsi, les instructions générées en plus par l'opération modulo (même lorsque celle–ci se réduit à un simple et logique) et surtout son test nécessitent plusieurs cycles d'exécutions. Elles sont alors coûteuses en terme de nombre de cycles d'exécution.

Une seconde approche, alternative, consiste à ordonner au processeur d'effectuer un pré–chargement de données à chaque itération de boucle. Ceci implique l'émission de requêtes de pré–chargement qui sont plus nombreuses que celles réellement nécessaires. Cependant, ceci n'affecte pas le comportement du programme, puisque ce sont uniquement les requêtes de pré–chargement utiles qui seront effectivement émises lorsqu'un échec de lecture de données en cache apparaîtra. En appliquant cette approche, le tri à bulles continue à produire de mauvaises performances comparées à celles de la version originale du code. Ceci est dû au fait que les instructions de comparaison et la fonction de « swap » ne consomment pas beaucoup de cycles de calcul. Elles s'exécutent trop rapidement pour permettre aux données préchargées d'être copiées en mémoire cache avant qu'elles ne soient utilisées. Dans cette situation, en calculant la couverture des échecs réalisée lors du pré–chargement de données, en utilisant la relation (6.1), et l'efficacité du pré–chargement de données grâce à la relation (6.2), pour la latence du contrôleur mémoire fixée à $124 cycles$, la couverture est de 8,8% et l'efficacité de 8,08%. Nous réalisons que beaucoup de requêtes de pré–chargement de données inutiles sont émises, et que les données ne sont pas toujours disponibles en mémoire cache avant le début d'exécution de la prochaine itération qui les utilisera. Sinon il y aurait une meilleure couverture des échecs.

$$Couverture_{pré-chargement} = \frac{\text{N}_{\text{échecs–évités}}}{\text{Nombre Total d'Échecs}} \tag{6.1}$$

$$Efficaté_{pré-chargement} = \frac{N_{échecs-évités}}{\text{Nombre de pré-chargements Inutiles} + N_{échecs-évités}} \quad (6.2)$$

$N_{échecs-évités}$ définit le nombre d'échecs en cache évités grâce au pré–chargement de données.

Finalement, nous proposons une troisième approche, qui est d'effectuer une légère modification du code source du programme, comme présentée dans le code 6.2, après analyse des versions précédentes du programme. Au regard des causes de mauvaises performances des versions précédentes du programme : (1) comparaison et modulo coûteux en nombre de cycles d'exécution, (2) fonction de « swap » qui s'exécute trop rapidement, alors dans l'option (3) la modification apportée consiste à augmenter légèrement le temps de calcul (nombre de cycles nécessaires) pour permettre aux directives de pré–chargement de données de terminer leur exécution avant le début d'exécution de la prochaine itération qui utilisera ces données préchargées. Pour atteindre cet objectif, les données sont traitées par bloc. Ici, un bloc est défini comme étant le nombre d'éléments à précharger dans le cache de données. L'observabilité fournie par les modèles précis au niveau du cycle a aidé à prendre cette décision d'apporter cette légère modification du code source du programme. Dans l'exemple présenté dans 6.2, le développeur ordonne au processeur de précharger quatre lignes de cache (*lignes 6, 7, 16 et 17*). C'est–à–dire 32 éléments ($4 \times ligne_$cache) en avance. Cette version modifiée de l'algorithme du tri à bulles fournit une meilleure performance comparée à celle de la version originale.

```
1  void bubbleSort (int32_t *array, const uint32_t length){
2      bool swap = true ; uint32_t n = length-1; uint32_t limit = 0;
3      while ((n not_eq 0 ) && swap){
4          swap = false ;
5          for (uint32_t j = 0 ; j < n ; j++){
6              __builtin_prefetch(&array[j + BLOCK_SIZE], 0, 3);
7              __builtin_prefetch(&array[j + BLOCK_SIZE + 8], 0, 3);
8              limit = MAX ((J + BLOCK_SIZE), n);
9              for (uint32_t k = j ; k < limit ; k++){
10                 if (array[k] > array[k+1]){
11                     swapElement (array, k, k+1);
12                     swap = true ;
13                 }
14             }
15         }
16         __builtin_prefetch(&array[0], 0, 3);
17         __builtin_prefetch(&array[8], 0, 3);
18         n--;
19     }
```

```
20 }
```

Listing 6.2: Tri à bulles qui traite les éléments

En se basant sur le flot méthodologique présenté en 4.1, nous avons alors étudié l'évolution des cycles de gel du processeur, et le gain en terme de nombre total de cycles d'exécution de l'application en faisant varier la latence du contrôleur mémoire entre 4 et 204 cycles. Les valeurs de latences qui apparaissent sur les courbes sont celles ressorties par le contrôleur mémoire à la fin de la simulation lorsque nous fixons la latence respectivement à 0, 28, 100, 120, 140, 160, et 200 cycles.

Ensuite, puisque le processeur utilisé dispose de quatre tampons dédiés au pré–chargement, une première stratégie consiste à utiliser uniquement deux des quatre tampons dédiés au pré–chargement de données. Ceci dans l'optique de voir par exemple ce qu'une architecture plus faible (seulement deux tampons) donnerait comme performance sur ce code. La seconde consiste à utiliser la totalité des tampons disponibles dédiés au pré–chargement pour émettre quatre requêtes de pré–chargement en parallèle de l'exécution normale du programme. La figure 6.3 présente quelques résultats de l'amélioration de la performance du logiciel (sous forme de gain obtenu) comparée à celle de la version originale du tri à bulles. Le gain est défini dans la relation 6.3 obtenue par simple règle de trois.

$$Gain(\%) = \frac{(Temps\_Execution_{prog\_origine} - Temps\_Execution_{prog\_optimise}) \times 100}{Temps\_Execution_{prog\_origine}} \quad (6.3)$$

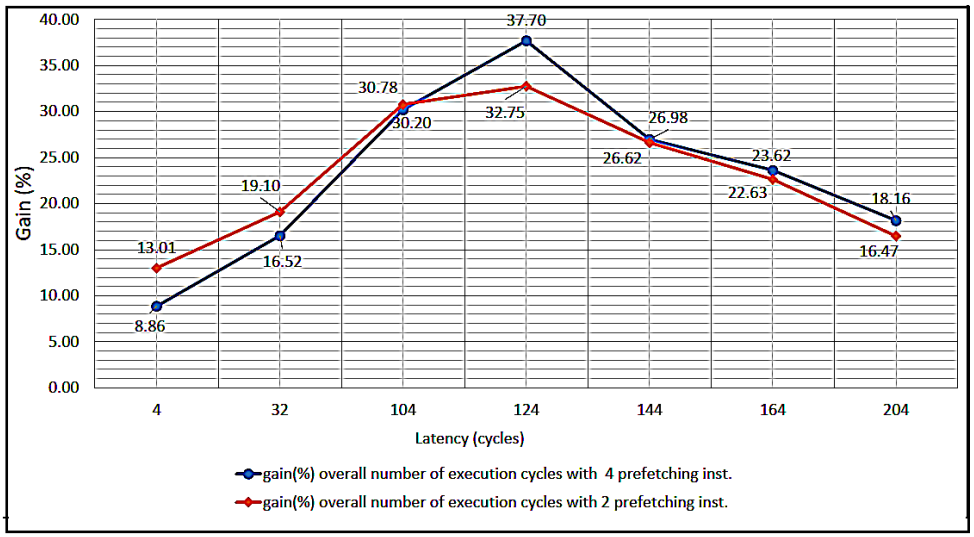

FIGURE 6.3: Impact du pré–chargement de données dirigé par le logiciel sur la performance du logiciel en termes de pourcentage (%) de nombre total de cycles d'exécution.

La version qui exploite simultanément les quatre tampons de pré–chargement présente logiquement une meilleure performance lorsque la latence du contrôleur mémoire augmente à partir de 124 cycles de latence du contrôleur mémoire. Le gain maximal est atteint lorsque la latence du contrôleur mémoire est fixée à 124 cycles. Ensuite, la courbe de gain décroît mais le gain reste quand même intéressant. Cette décroissance du gain est due au fait qu'à partir de 144 cycles de latence du contrôleur mémoire, une requête de pré–chargement de données met trop de temps avant que les données ne soient effectivement copiées en mémoire cache. Dans cette situation, il peut arriver que les instructions qui utilisent les données préchargées débutent leur exécution alors que les données sont encore entrain d'être transférées. On observe sur la figure 6.5 qu'à partir de cette valeur de latence du contrôleur mémoire, le nombre d'échecs évités dans le cache L_1 de données du processeur diminue. À partir de 144 cycles de latence, ce nombre d'échecs évités diminue davantage lorsque la latence du contrôleur mémoire augmente.

On observe également sur les figures 6.4 et 6.5 que lorsque la latence du contrôleur mémoire est inférieure ou égale à 104 cycles, tous les échecs dans le cache L_1 de données sont évités, et le nombre de cycles de gel du processeur est nul. Ainsi, pour favoriser l'utilisation des capacités du matériel par le logiciel, un architecte matériel pourrait par exemple effectuer des actions sur la latence du contrôleur mémoire de son système. Une action pourrait être de chercher à ramener la latence du contrôleur mémoire à 104 cycles. Car à 104 cycles de latence, lorsque les instructions de pré–chargement sont insérées dans le code source du tri à bulles, on observe un gain de performance (en terme de cycles d'exécution) de 30% environ pour zéro échec et zéro cycle de gel du processeur.

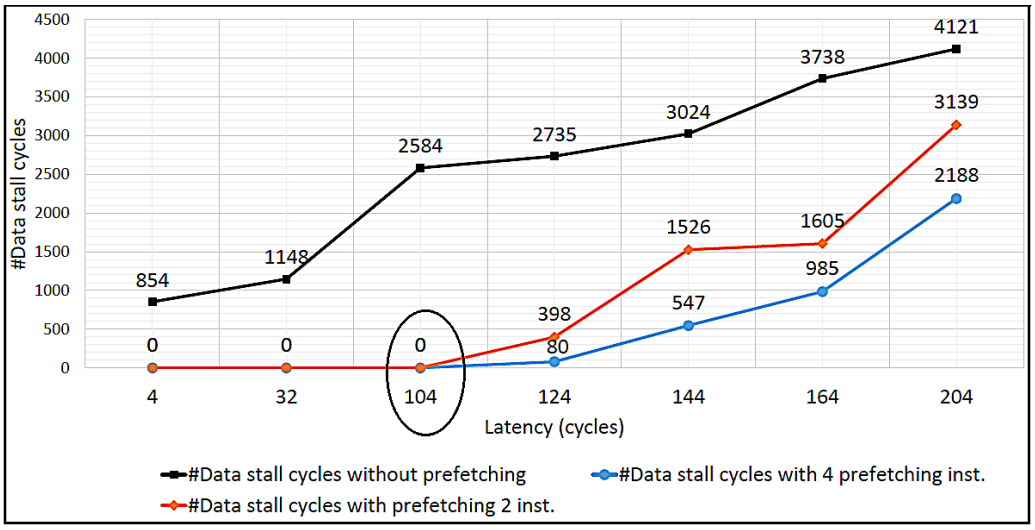

FIGURE 6.4: Impact du pré–chargement dirigé par le logiciel de données sur la performance du logiciel en termes de nombre de cycles de gel du processeur.

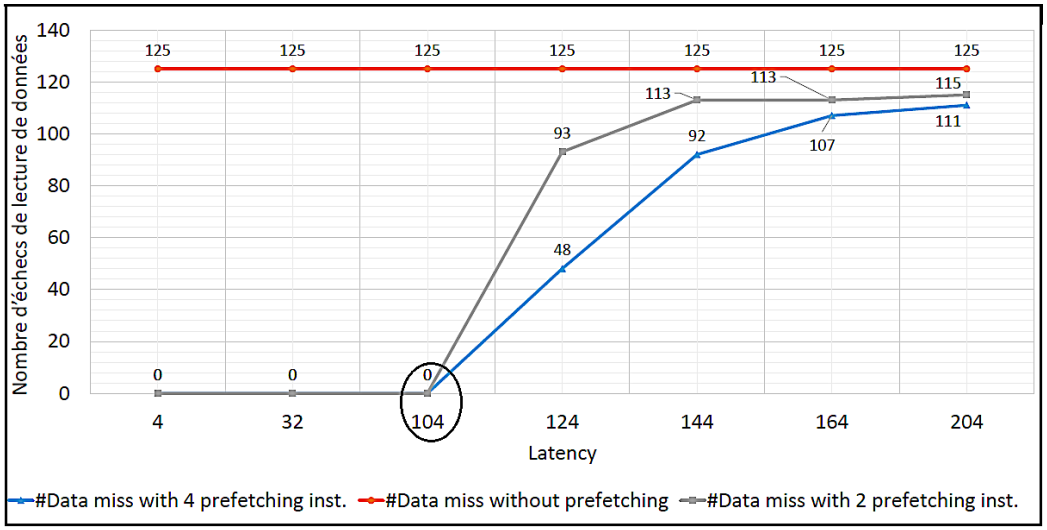

FIGURE 6.5: Impact du pré–chargement dirigé par le logiciel de données sur la performance du programme en termes de nombre d'échecs de lecture dans le cache L_1.

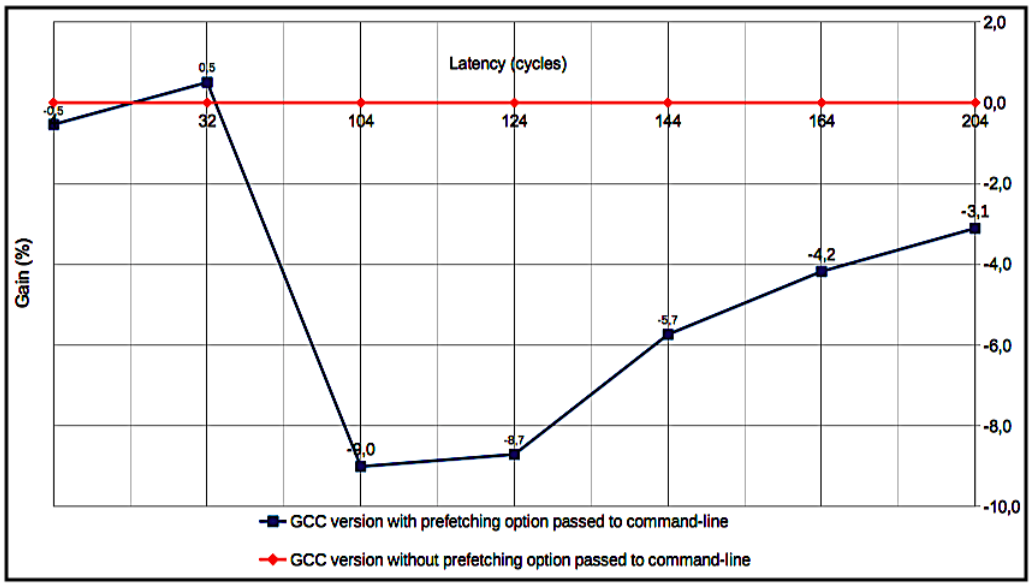

FIGURE 6.6: Impact de l'option *-fprefetch-loop-arrays* de GCC sur la performance en termes de nombre de cycles total d'exécution du tri à bulles.

Finalement, pour observer les optimisations qu'un compilateur pourrait effectuer sur le logiciel grâce à sa propre analyse du code, nous avons compilé le tri à bulles en fournissant l'option *–fprefetch-loop-arrays* en argument au compilateur GCC. Cette option de compilation permet de demander au compilateur d'insérer automatiquement des instructions de pré–chargement de données là où il estime que cela permettra d'améliorer les performances. En regardant le code désassemblé, on remarque que GCC n'a à aucun moment estimé qu'il serait rentable d'insérer de telles instructions. Un comportement similaire est aussi obser-

vable avec les autres compilateurs disponibles avec lesquels nous avons reproduit cette expérimentation.

La figure 6.6 présente une dégradation globale de la performance en terme de nombre total de cycles d'exécution de l'application lorsque le tri à bulles est compilé avec GCC et l'option *–fprefetch-loop-arrays* fournit en argument au compilateur. Le gain de performance du programme d'origine compilé sans l'option *–fprefetch-loop-arrays* est représenté par la courbe rouge située sur l'axe des abscisses. Cette dégradation de la performance du programme est probablement due au fait que l'option *–fprefetch-loop-arrays* active d'autres options de compilation. Ainsi telle que décrite dans la documentation ARM-OPTIONS [2017], cette option de compilation peut entraîner la génération d'un meilleur code exécutable ou d'un code exécutable moins bon ; les résultats dépendent fortement de la structure des boucles dans le code source. Ceci permet de comprendre que cette option de compilation ne garantirait pas toujours une meilleure utilisation des capacités du matériel par le programme exécutable, même pour des programmes aussi simples que l'est le tri à bulles.

6.1.3 Cas d'étude : Le programme IDCT

Le programme IDCT est un noyau logiciel utilisé dans le décodage d'images. Dans ce cas d'étude, nous utilisons l'algorithme de Loeffer LOEFFLER et collab. [1989], implémenté en langage C. Ce programme présente des propriétés assez différentes de celles du tri à bulles. L'IDCT présenté dans le listing 6.3 est un programme à contrôle statique, c'est–à–dire que les contrôles effectués sont indépendants des données calculées au cours de l'exécution. Les contrôles sont effectués majoritairement sur les index de parcours des boucles de la fonction. Par ailleurs, chaque bloc de 64 éléments fourni en entrée à la fonction est traité en parcourant le bloc horizontalement, puis inversé et parcouru verticalement. La fonction fait appel à d'autres fonctions telles *SCALE, idct_1d, DESCALE, ...* Ainsi, pour une itération de boucle, le programme IDCT effectue significativement plus de traitement que le tri à bulles.

```
1  /** Inverse 2-D Discrete Cosine Transform */
2  void IDCT(int32_t * input, uint8_t * output){
3      int32_t Y[64], k, l;
4      for (k = 0; k < 8; k++) {
5          for (l = 0; l < 8; l++)
6              Y(k, l) = SCALE(input[(k << 3) + l], S_BITS);
7          idct_1d(&Y(k, 0));
8      }
9      for (l = 0; l < 8; l++) {
10         int32_t Yc[8];
11         for (k = 0; k < 8; k++)
```

```
12              Yc[k] = Y(k, l);
13          idct_1d(Yc);
14          for (k = 0; k < 8; k++) {
15              int32_t r = 128 + DESCALE(Yc[k], S_BITS + 3);
16              r = r > 0 ? (r < 255 ? r : 255) : 0;
17              X(k, l) = r;
18          }
19      }
20  }
21  int main(){
22      for (i = 0; i < nblocks; i++) {  //nblocks = (sizeof(block) / sizeof(block[64]))
23          IDCT(block[i], out);
24      }
25      return 0;
26  }
```

Listing 6.3: Code C fonction IDCT

Pour ce programme, pour éviter les cycles de gel du processeur, les instructions de pré–chargement de données sont insérées manuellement à 2 emplacements différents dans le code source de l'IDCT.

Dans un premier temps, les instructions de pré–chargement de données en mémoire cache sont insérées de manière intuitive en se basant sur l'analyse du code source du programme. À l'exécution, le diagramme d'exécution du programme (obtenu á partir des informations fournie par le PMU) permet de remarquer que le programme complet passe plus de cycles d'exécution dans la fonction *IDCT ()*. Les données utilisées par cette fonction sont stockées et traitées par bloc de 64 *int32_t* soit 256 octets. Il nous apparait alors utile de précharger le prochain bloc de 256 octets de données qui sera utilisé par la fonction lorsque le bloc courant est en cours de traitement. C'est–à–dire précharger le bloc*[i+1]* lorsque le bloc*[i]* est en cours de traitement.

Dans cette première approche, les 4 tampons dédiés au pré–chargement disponibles dans le CPU du Cortex–A9 sont utilisés. Ce qui implique qu'uniquement 128 octets de données peuvent être préchargés en parallèle. En analysant les informations de profilage extraites des registres de performance PMU au cours de la simulation, nous observons que cette approche produit des cycles de gel du pré–chargement (PLD stall) pendant l'exécution du programme. Le listing de code 6.4 présente le pré–chargement du bloc*[i+1]* pendant que le bloc*[i]* est en cours de traitement. Elle présente également la fonction *prefetch_range()* qui permet d'effectuer le pré–chargement du prochain bloc de 256 octets de données.

```
__attribute__((always_inline)) inline  void prefetch_range(int32_t * addr) {
    PREFETCH(addr); // Un prefetch ramene une ligne de cache (32O = 8 int32_t)
    PREFETCH(addr + CACHE_LINE); // pour 256 O, il en faut 8 prefetch )
    PREFETCH(addr + CACHE_LINE * 2);
    PREFETCH(addr + CACHE_LINE * 3);
    PREFETCH(addr + CACHE_LINE * 4);
    PREFETCH(addr + CACHE_LINE * 5);
    PREFETCH(addr + CACHE_LINE * 6);
    PREFETCH(addr + CACHE_LINE * 7);
}
int main(){
    for (i = 0; i < nblocks; i++) {  //nblocks = (sizeof(block) / sizeof(block[64]))
#ifdef __PREFETCH__
        prefetch_range(block[i+1]);
#endif
        IDCT(block[i], out);
    }
    return 0;
}
```

Listing 6.4: Code C fonction IDCT avec insertion des instructions de pré–chargement intuitivement

Les cycles de gel du pré–chargement proviennent du fait que les tampons de pré–chargement disponibles ne peuvent stocker simultanément que 128 octets de données, or les instructions de pré–chargement insérées visent à envisager le stockage simultané de 256 octets. Par ailleurs, les données préchargées ne sont pas directement allouées dans le cache, mais plutôt dans ces petites mémoires tampons ayant une latence faible. Ceci en utilisant des stratégies de remplacement qui permettent de déterminer les données qui seront remplacées dans ces tampons lorsque de nouvelles données arrivent.

Pour réduire les cycles de gels générés par les instructions de pré–chargement de données, nous avons découpé les 256 octets préchargés en deux parties. Ainsi, les premiers 128 octets sont d'abord préchargés. Grâce à la méthodologie développée au chapitre 4 nous avons calculé la distance de pré–chargement D et nous nous sommes rendus compte qu'il ne faut pas précharger directement le prochain bloc de 256 octets mais plutôt le bloc *[i+2=D]*. Ensuite, pendant le traitement de la fonctions IDCT(), nous avons trouvé suite à plusieurs essais/erreurs l'endroit convenable dans le code source pour insérer d'autres instructions de pré–chargement qui permettraient de précharger les autres 128 octets restants. Ceci grâce aux capacités de profilage de la simulation avec modèles précis au niveau du cycle. Pour rendre assez précise l'identification de l'emplacement dans le code source du programme

qui conviendrait pour l'insertion des instructions de pré–chargement, nous avons envisagé une méthodologie qui permettrait de donner dans la fonction du programme des indications sur les lignes de code (typiquement en C) qui sont à l'origine de plusieurs échecs dans le cache L_1 de données du CPU. Ces lignes de code identifiées sont alors des indices pour faciliter l'insertion par le développeur, des instructions de pré–chargement dans le code source du programme. Cette approche a été développée dans le chapitre 5. Le listing de code 6.5 présente la version de l'IDCT qui permet d'obtenir un gain important, du même ordre que celui du tri à bulles : jusqu'à environ 30%.

```
1  void IDCT(int32_t * input, uint8_t * output) {
2      int32_t Y[64], k, l;
3      for (k = 0; k < 8; k++) {
4          for (l = 0; l < 8; l++)
5              Y(k, l) = SCALE(input[(k << 3) + l], S_BITS);
6          idct_1d(&Y(k, 0));
7      }
8  #ifdef __PREFETCH__
9      prefetch_range((input + D ) + (4 * CACHE_LINE));
10 #endif
11     ...
12 }
13 int main(){
14     for (i = 0; i < nblocks; i++) {  //nblocks = (sizeof(block) / sizeof(block[64]))
15 #ifdef __PREFETCH__
16         prefetch_range(block[i+D]);
17 #endif
18         IDCT(block[i], out);
19         }
20     return 0;
21 }
```

Listing 6.5: Code C fonction IDCT avec insertion des instructions de pré–chargement qui produisent un gain de performance

La figure 6.7 présente l'évolution du nombre de gels générés par les instructions de pré–chargement de données insérées dans le code source de l'IDCT. On observe que lorsque la latence du contrôleur mémoire varie entre 4 et 404 cycles, le nombre de cycles de gel générés par les instructions de pré–chargement insérées est nul comparé à la première approche où ce nombre de cycles est constant = 1468.

Dans la figure 6.8, On observe également que la seconde approche procure une amélioration de la performance d'un ordre de grandeur similaire à celui du tri à bulles. Dans la

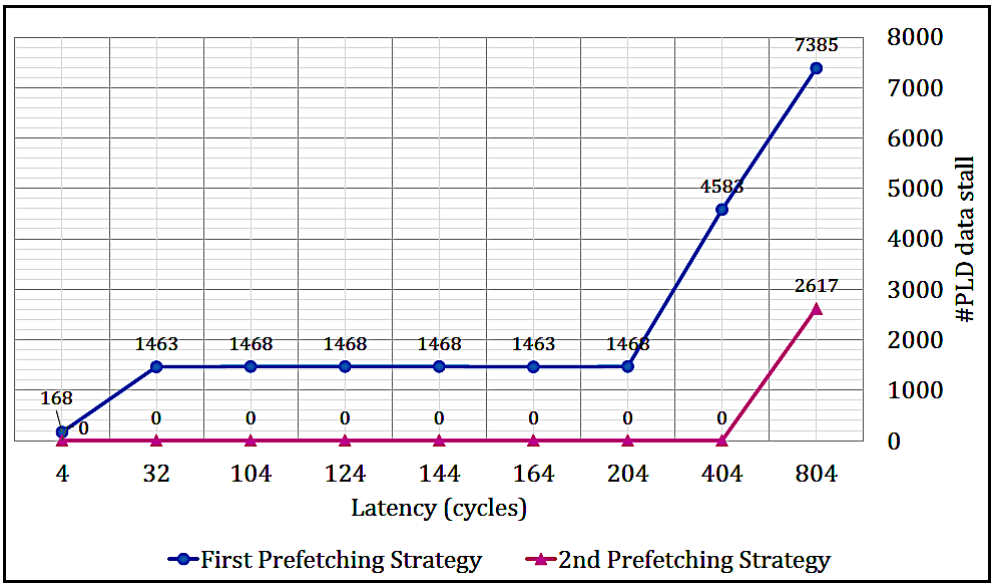

FIGURE 6.7: Evolution du nombre de gels générés par les instructions de pré–chargement de données insérées dans le code source de l'IDCT

première approche, le gain est limité malgré une réduction significative du nombre de cycles de gel du processeur. Cette limitation du gain provient des cycles de gel générés par les instructions de pré–chargement de données mal insérées dans le code source du programme.

Puisque l'IDCT nécessite plus de temps de calcul que le tri à bulles, on observe sur la

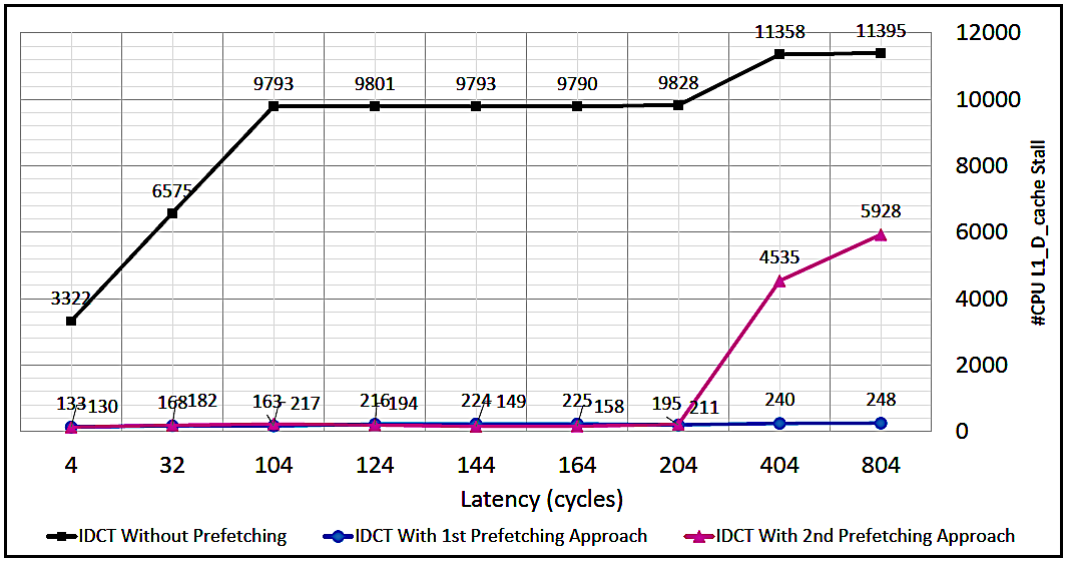

FIGURE 6.8: Résultat du pré–chargement logiciel de données sur la performance de l'IDCT en termes de nombre de cycles de gel du processeur.

figure 6.9 que la courbe de gain(%) en terme de nombre total de cycles d'exécution croît lentement. C'est la raison pour laquelle nous faisons varier la latence du contrôleur mémoire

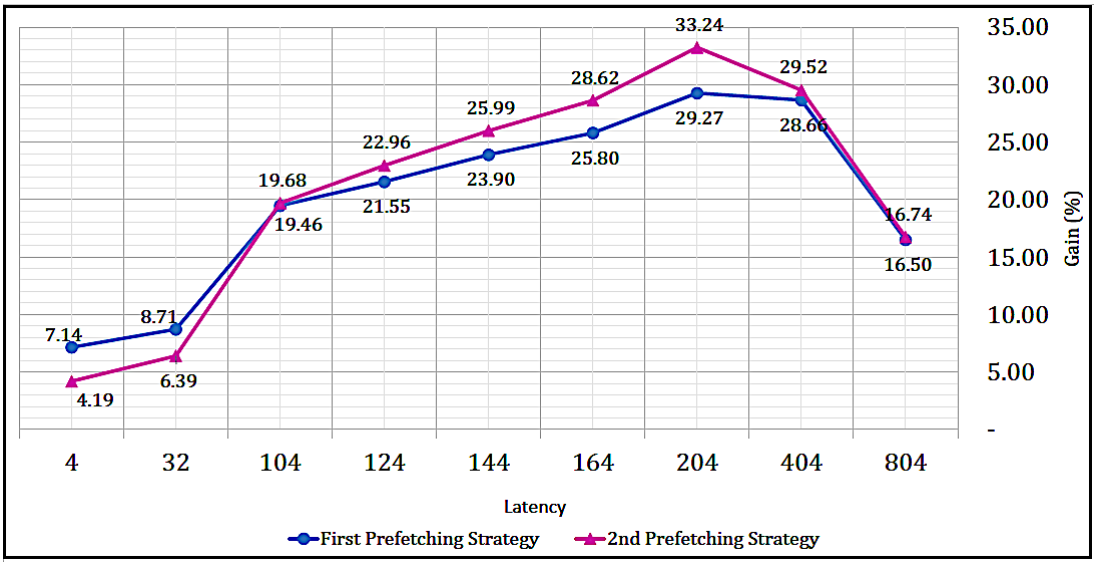

FIGURE 6.9: Résultat du pré–chargement logiciel de données sur la performance de l'IDCT en termes de nombre total de cycles d'exécution.

jusqu'à 804 cycles pour découvrir que l'on obtient une courbe en cloche se rapprochant de celle obtenue dans le cas du tri à bulles. Cette courbe montre un point (dans ce cas IDCT lorsque la latence du contrôleur mémoire est égale à 404 cycles), pour lequel le temps des calculs effectués par l'IDCT est inférieur à la latence du contrôleur mémoire. A partir de ce point, lorsque le paramètre de latence augmente, la stratégie de pré–chargement de données devient de moins en moins efficace, même si le gain reste observable.

6.2 Méthodologie sur plateforme ARM Cortex-A53

6.2.1 Description de l'environnement de simulation et du logiciel

Cette section présente l'expérimentation faite sur l'identification des instructions gelées après analyse des traces d'exécution du programme, PC et trace bus, sans utiliser d'instrumentation de type PMU ou autre. Elle présente également les résultats sur l'évaluation de la pertinence des gels de processeur à corriger, et le gain obtenu suite à la correction des cycles de gels du processeur identifiés, par l'insertion des instructions de pré–chargement de données dans le code source du programme testé, suite à cette analyse des traces PC et bus.

La figure 6.10 présente de manière simplifiée sous forme de diagramme de blocs la plateforme de simulation. Cette plateforme virtuelle précise au niveau du cycle est architecturée autour du processeur ARM Cortex-A53 intégrant un seul cœur avec une fréquence d'horloge de 1GHz. Ce processeur comporte 2 caches L_1 intégrés dans le cœur du processeur de

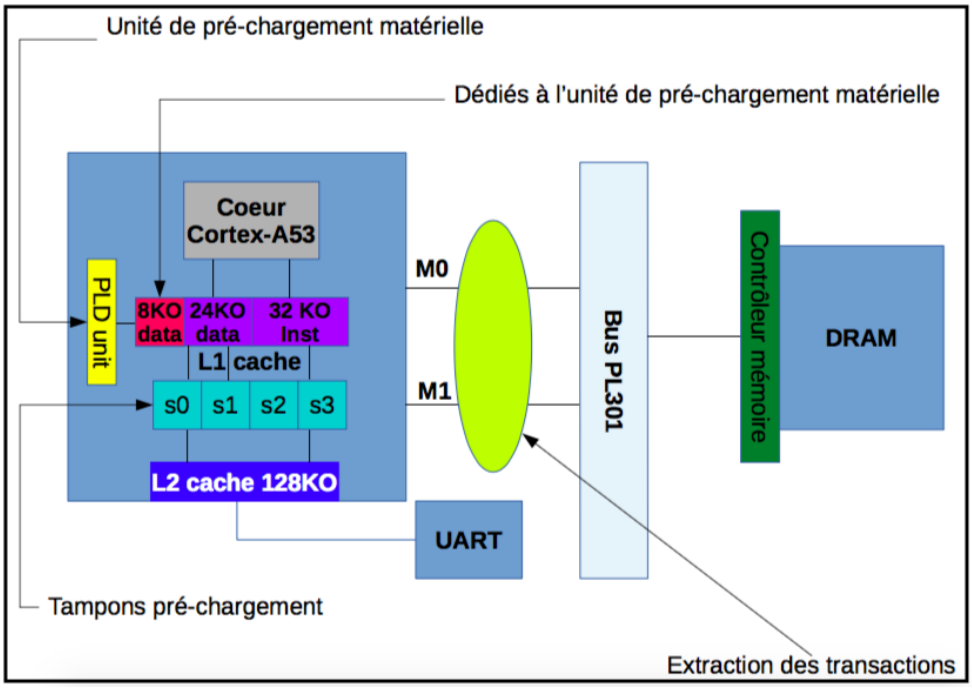

FIGURE 6.10: Diagramme simplifié de l'ossature de la plateforme architecturée autour du ARM Cortex–A53

type 4-way set associatif de 32 KO, l'un pour les instructions et l'autre pour les données. La taille d'une ligne de cache de données est de 64 octets. Le processeur dispose également d'un cache de niveau 2 de 128 KO. Quatre tampons dédiés au pré–chargement de données sont également disponibles sur ce processeur. Sur les 32 KO du cache L_1 de données, 8 KO de données sont dédiés au pré–chargement effectué par l'unité de pré–chargement matériel. Ce processeur supporte deux modes d'exécution : un mode 64–bit (AARCH64) et un mode 32–bit (AARCH32). Dans cette expérimentation, le mode 32–bit est utilisé.

La plateforme intègre également un contrôleur mémoire qui pilote la DRAM. Ce contrôleur mémoire traite toutes les requêtes de lectures et d'écritures adressées à la mémoire principale. Le contrôleur mémoire utilisé est configurable, et il est possible de faire varier sa latence (en cycles) d'accès à la mémoire. Ce qui nous permet de tester différentes configurations et de visualiser, par notre étude, les effets de la latence sur le gain potentiel (en terme de cycles d'exécution) qu'il est possible d'obtenir si les cycles de gel du processeur sont évités.

Comme la plateforme précédente, cette plateforme que nous avons développée est également assez simple, mais représentative du sous–système processeur que l'on peut typiquement rencontrer au sein de la gamme de SoC de STMicroelectronics; notamment les SoC destinés à l'internet des objets. Elle permet de représenter de manière réaliste le fonctionnement et le comportement de l'ensemble du système (processeur - mémoires caches

- interconnexion - mémoire principale). Les expérimentations ont été effectuées sur plusieurs programmes dont l'IDCT et un sous-ensemble de programmes du benchmark polybench/C(POLYBENCH/C [2017]).

6.2.2 Evaluation durée simulée (cycles) et durée de la simulation (Wall-clock, en minutes :secondes)

L'IDCT et le sous-ensemble de programmes d'algèbre linéaire du benchmark polybench/C ont été exécutés sur la plateforme de simulation précise au niveau du cycle avec la latence du contrôleur mémoire fixée arbitrairement à 164 cycles. Pour les programmes polybench/C, nous avons utilisé la plus petite taille de données des tableaux d'entrée : le MINI_DATASET. Nous avons essayé d'utiliser une taille d'entrée un peu plus grande (SMALL_DATASET) mais la fonction *posix_memelign* qui effectue les allocations mémoires alignées générait le message d'erreur : « Cannot allocate memory : not enough space ».

En essayant d'augmenter la taille du tas (« heap ») dans le but d'augmenter l'espace d'allocation mémoire dans une fonction, l'initialisation de la plateforme de simulation devenait trop longue (≈ 30 minutes). L'IDCT a été exécuté avec un tableau de 278 (choisi arbitrairement) éléments de type int32_t. Les programmes ont été exécutés dans un premier cas lorsque l'unité de pré–chargement matérielle est désactivée, et dans le second lorsque celle–ci est activée.

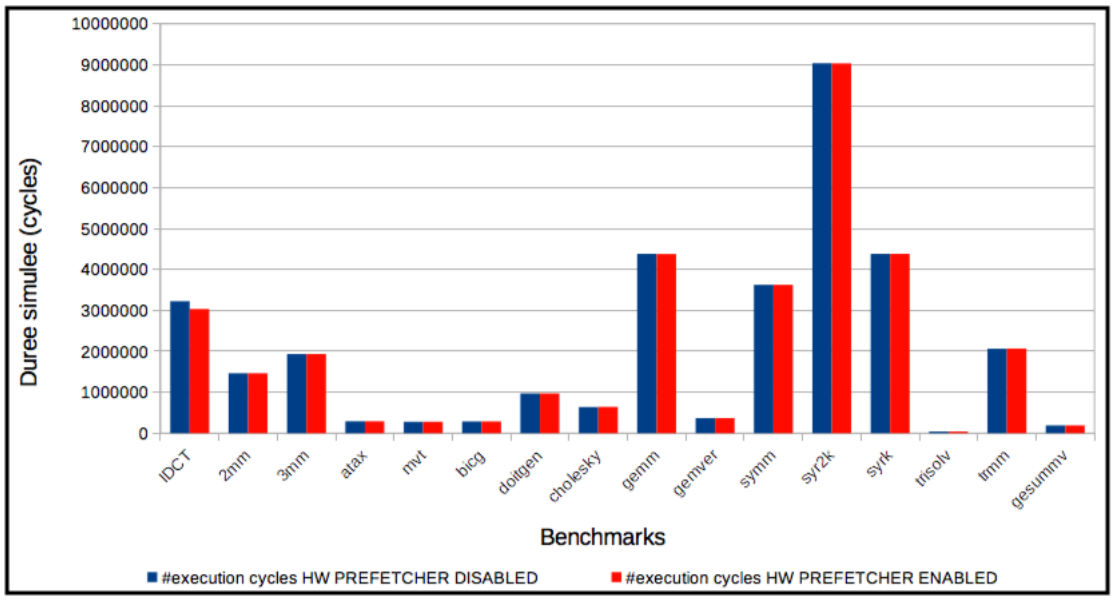

FIGURE 6.11: Temps simulée (cycles) benchmarks avec unité de pré–chargement matérielle désactivée, puis activée

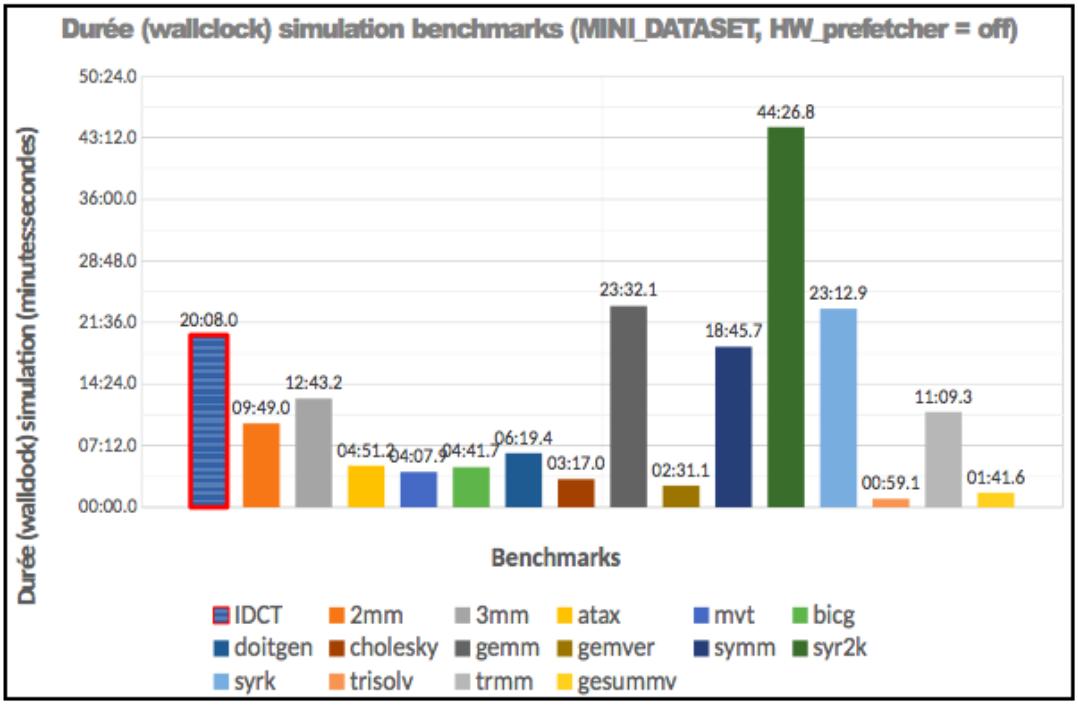

FIGURE 6.12: Temps de simulation (*minutes* : *secondes*) benchmarks lorsque unité de pré–chargement matérielle désactivée

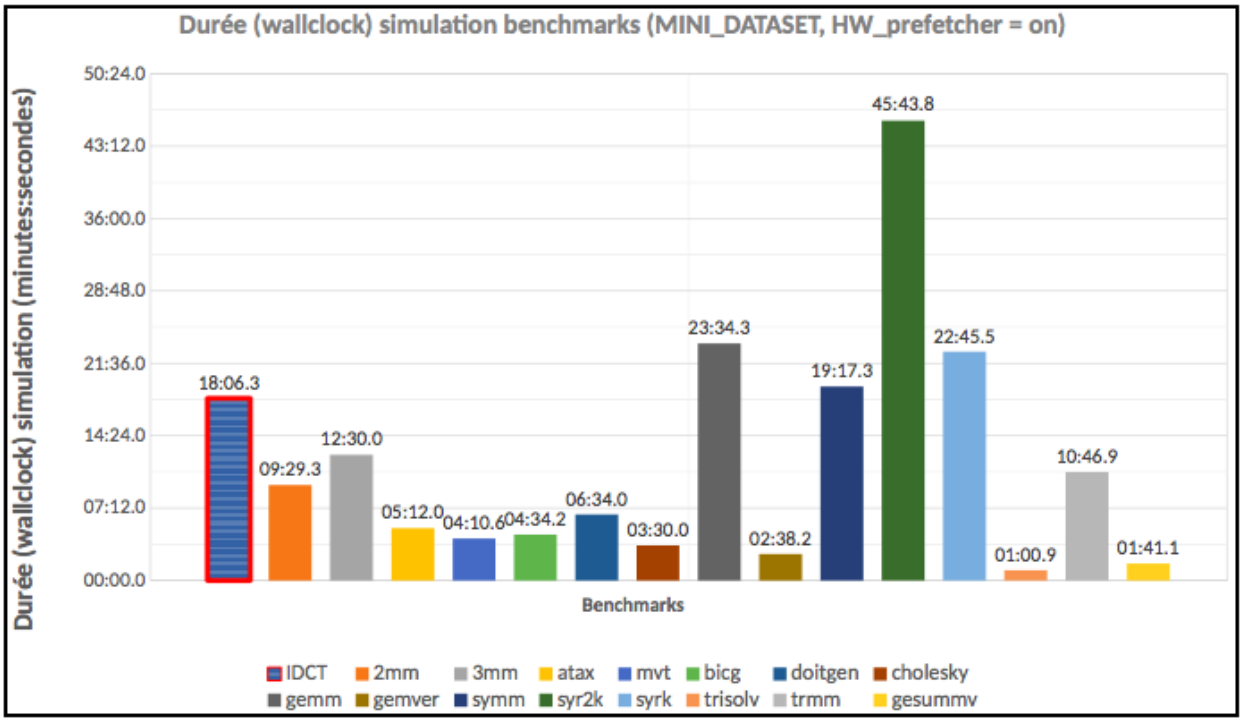

FIGURE 6.13: Temps de simulation (*minutes* : *secondes*) benchmarks lorsque unité de pré–chargement matérielle activée

La figure 6.11 présente les durées simulées en cycles obtenues. Dans les deux cas, les programmes *syr2k, syrk, gemm, symm et IDCT* sont ceux qui ont les plus longues durées d'exécution simulées. En revanche, on remarque que lorsque l'unité de pré–chargement est activée, la durée d'exécution de l'IDCT (en cycles) est réduite. Par contre, pour les programmes considérés du polybench/C (*2mm, bicg, gemm, syr2k, trisolv*), la réduction de la durée d'exécution n'est pas facilement observable car elle est infime. On pourrait alors émettre l'hypothèse que les programmes polybench/C sont suffisamment optimisés pour que l'unité de pré–chargement matérielle n'arrive pas à identifier les adresses des données à précharger. Nous avons donc ensuite évalué le gain potentiel maximal qu'il est possible d'obtenir grâce à l'approche développée dans le chapitre 5. Ce gain permet de savoir s'il est utile ou non de poursuivre l'optimisation des programmes polybench/C et IDCT. Ceci est décrit dans la section suivante.

Par ailleurs, les figures 6.12 et 6.13 présentent les durées de simulation en $minutes$: $secondes$. On observe que malgré l'utilisation du MINI_DATASET comme taille des tableaux de données fournis en entrées aux sous–ensemble de programme polybench/C, les durées de simulation sont relativement longues. Une vingtaine de minutes pour plusieurs programmes polybench/C et approximativement 45 minutes pour le programme IDCT pour seulement 278 éléments de type int32_t. Il est néanmoins possible d'envisager d'autres techniques de simulation précises au niveau du cycle, notamment l'émulation matérielle : la déployabilité de notre technique est évoquée dans une autre section (7.2.1).

6.2.3 Evaluation du gain potentiel maximal

La figure 6.14 présente les gains potentiels maximaux susceptibles d'être obtenus si tous les cycles de gel du processeur sont évités pour les programmes polybench/C et l'IDCT. Ces gains potentiels maximaux sont comparés à ceux obtenus en activant l'unité matérielle de pré–chargement. On observe que certains programmes (*IDCT, trisolv*) présentent un gain potentiel maximal élevé ($\approx$ 12%) comparé à ceux des autres programmes. On peut remarquer également que l'hypothèse selon laquelle les programmes polybench/C sont suffisamment optimisés pour cette architecture se vérifie pour les programmes *2mm, 3mm, doitgen, gemm, syr2k, syrk, trmm* car les gains sont très faibles ($\leq$ 0.7%). En revanche, pour les programmes *atax, mvt, cholesky, gemver, trisolv, gesummv*, il est encore possible d'améliorer la performance. Sur le programme IDCT par exemple, on observe que l'unité matérielle de pré–chargement réussit à améliorer la performance approximativement de 6% . Pour les autres programmes, les gains sont parfois nuls ou très faibles.

Dans le programme IDCT, il est encore possible d'optimiser la performance (en termes

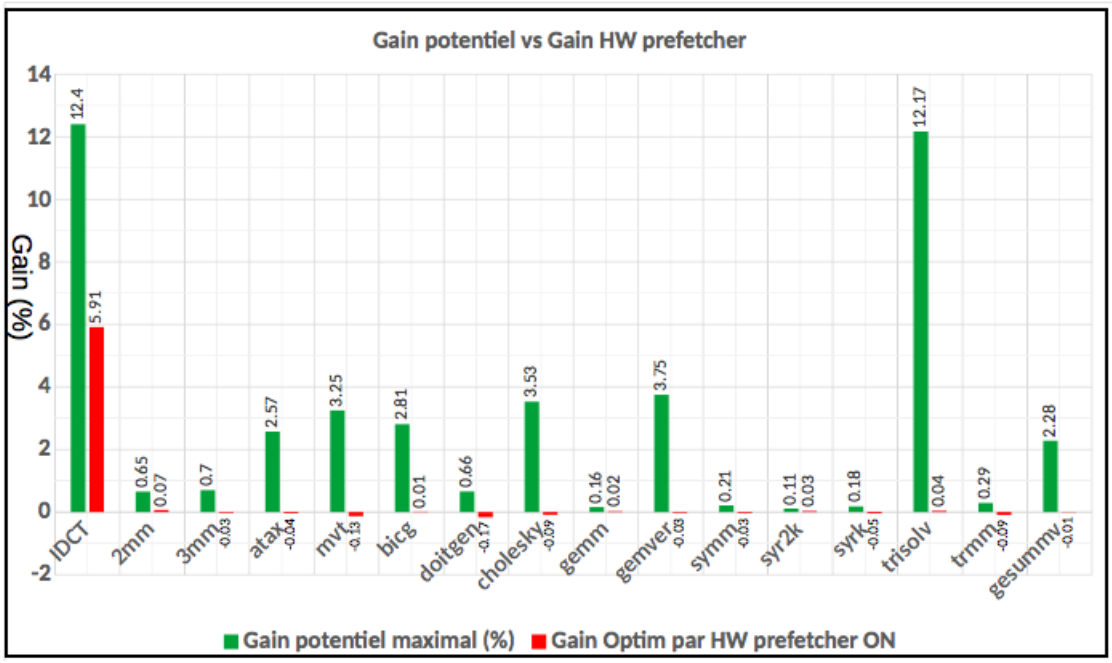

FIGURE 6.14: Gains potentiels maximaux estimés vs Gains obtenus par l'unité de pré–chargement matérielle

de temps d'exécution) d'environ 6% pour atteindre 12%. Il faut pour cela commencer par identifier les instructions du programme qui ont une contribution significative au rallongement du temps d'exécution du programme parce qu'elles sont à l'origine des cycles de gel du processeur.

6.2.4 Évaluation de la pertinence d'un gel

L'évaluation de la pertinence des gels est faite sur le programme IDCT. Le programme est compilé en fournissant en paramètre les options de compilation qui permettent au compilateur GCC d'optimiser le temps d'exécution. La latence du contrôleur mémoire est fixée arbitrairement à 164 cycles et l'unité de pré–chargement matérielle est désactivée.

La figure 6.15 présente pour chaque PC les cycles de gel de processeur qu'il génère à l'exécution du programme. On remarque alors que malgré les optimisations faites par le compilateur, des cycles de gel du processeur apparaissent. On observe que plusieurs instructions sont responsables des cycles de gel. Cependant, deux instructions dont les adresses sont PC = *0x990* et PC = *0x98c* cristallisent principalement l'attention puisqu'elles ont une contribution très forte au rallongement du temps d'exécution du programme. En utilisant l'outil *addr2line* (disponible dans le compilateur GCC utilisé), les lignes du programme écrit en langage C sont identifiées. Ces deux PCs correspondent à la ligne *Y(k, l) = SCALE(input[(k*

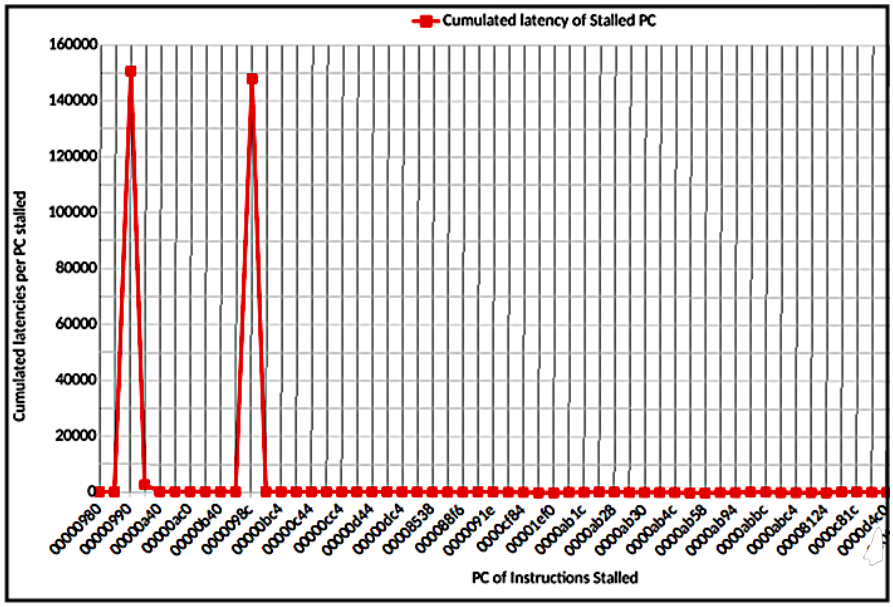

FIGURE 6.15: Listes des PCs (instructions) gelés identifiés à partir des traces d'exécution de l'IDCT.

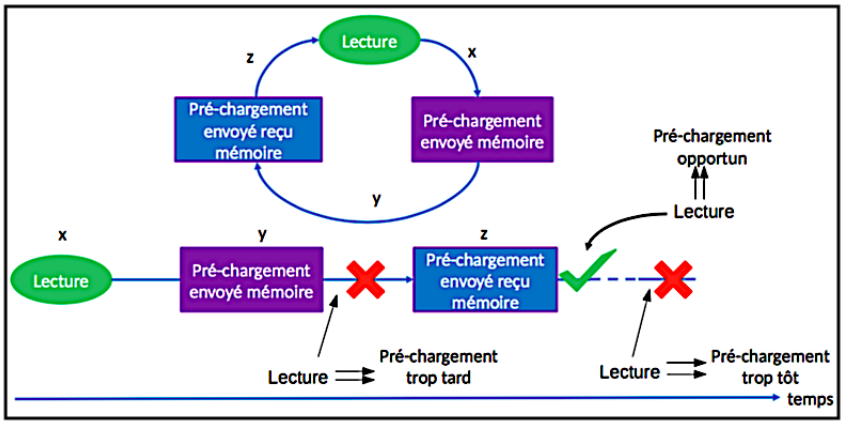

FIGURE 6.16: Chronologie de pré–chargement de données et requêtes de lecture associées.

« 3) + l], S_BITS); dans le code source de la fonction IDCT( ).

Avec cette information, un développeur pourrait avoir une vue suffisamment claire sur les principales instructions qui empêchent son programme de mieux utiliser les capacités du matériel. Le but est ensuite de trouver un moyen qui permettrait d'atténuer le nombre de cycles de gel du processeur produit principalement par ces 2 instructions. Dans notre cas, nous utilisons le pré–chargement de données dirigé par le logiciel pour qu'elles puissent être disponibles dès que nécessaire. L'insertion des instructions de pré–chargement est effectuée manuellement de façon à garantir l'effectivité du pré–chargement tel que décrit dans la figure 6.16.

Sur cette figure, est décrite la machine d'états et la chronologie des pré–chargements

de données et des requêtes de lecture associées. Trois cas se présentent. Si la prochaine lecture s'effectue entre l'instant y et l'instant z, le pré–chargement n'est pas bénéfique. Le pré–chargement de données a donc été effectué trop tard. En revanche, si la prochaine lecture survient juste après l'instant z, le pré–chargement sera efficace. Sinon, le pré–chargement aura été effectué trop tôt avec des risques d'éviction de la ligne de cache préchargée (SETHIA et collab. [2013],).

6.2.5 Réduction des cycles de gel identifiés et résultats

L'insertion des instructions de pré–chargement de données dans le code source de l'IDCT pour qu'elles s'exécutent au moment opportun, ceci grâce à la bonne distance de pré–chargement (4.1) permet d'obtenir la figure 6.17.

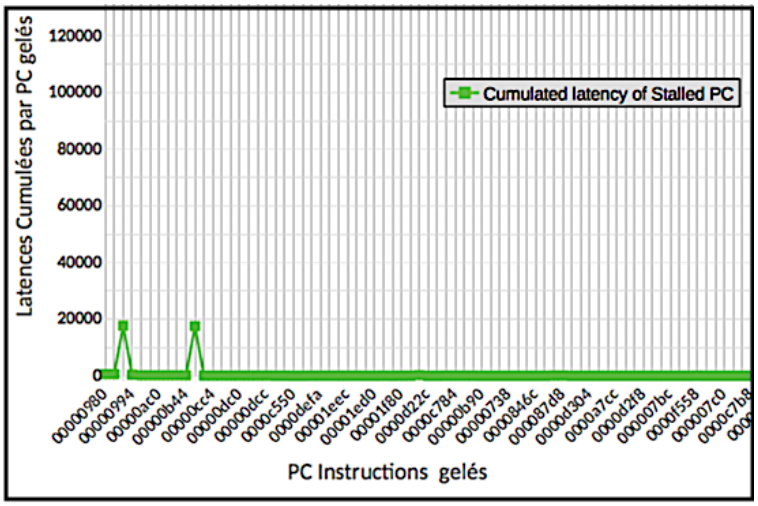

FIGURE 6.17: Réduction du nombre de cycles de gel du processeur à partir des traces d'exécution de l'IDCT

Cette figure présente la réduction du nombre de cycles de gel du processeur. On observe que les 2 pics produits par les instructions localisées aux adresses *0x990* et *0x98c* sont réduites par l'insertion effectuée. Elles chutent de 160K environ à 17K environ. Cette réduction très importante du nombre de cycles cumulés de gels des instructions qui ont une contribution forte au rallongement du temps d'exécution du programme, permet de réduire le temps d'exécution du programme sur le matériel. Le gain en terme de temps d'exécution et l'accélération (donnée par la relation 6.4) de l'exécution du programme sont visibles sur les figures 6.18 et 6.19. On remarque que l'accélération de l'exécution décroît lorsque la latence diminue. Cette décroissance est proportionnelle au gain qui lui croît lorsque la latence du contrôleur mémoire augmente. On peut constater que malgré les optimisations faites par le compilateur le gain de performance reste effectif sur le programme IDCT. L'unité matérielle

de pré–chargement réalise un gain de ≈ 6% environ. On peut remarquer sur la courbe de gain que lorsque la latence du contrôleur mémoire est égale à 164 cycles, le gain réalisé par les instructions de pré–chargement insérées dans le code source du programme est de 10% environ. Sur l'IDCT, la méthodologie permet de mieux faire que l'unité de pré–chargement matérielle, malgré les 8KO du cache L_1 qui lui sont dédiés.

$$Acceleration = \frac{Temps\_Execution_{prog\_origine}}{Temps\_Execution_{prog\_optimise}} \tag{6.4}$$

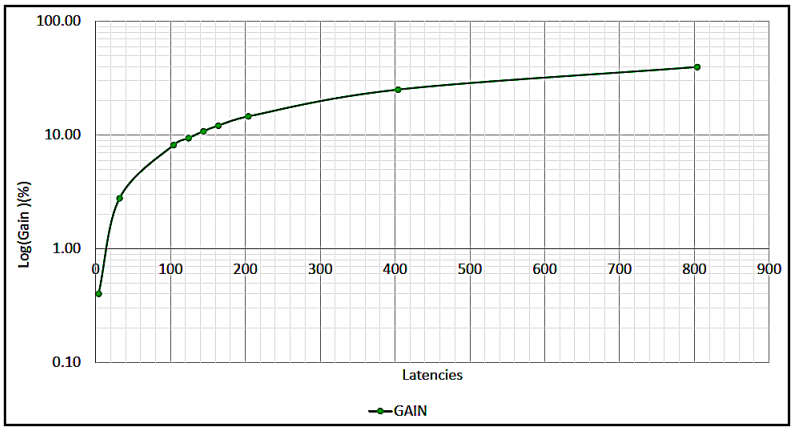

FIGURE 6.18: Gain de performance en termes de temps d'exécution sur l'IDCT grâce à l'analyse de traces et à l'insertioin des pré–chargement de données

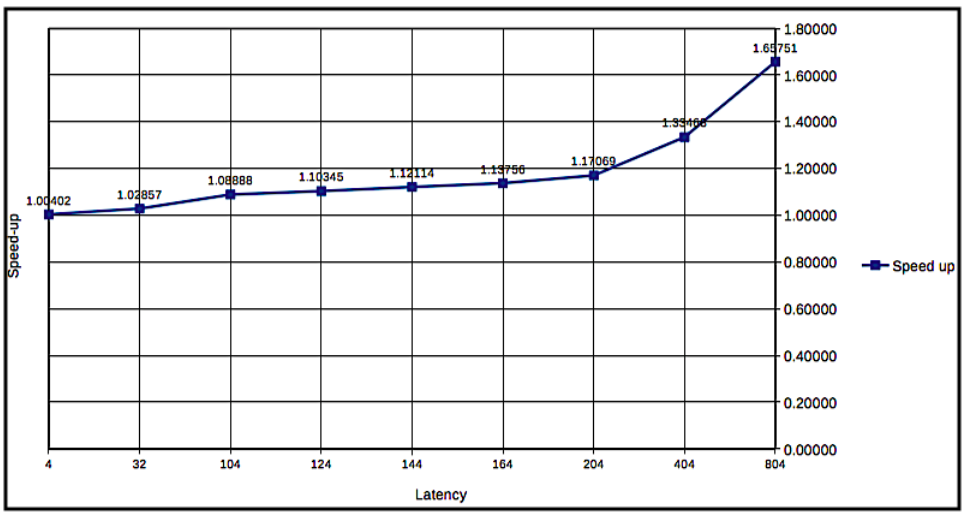

FIGURE 6.19: Accélération de l'exécution de l'IDCT après atténuation des pics de cycles de gel du processeur

6.3 Conclusion partielle

Dans ce chapitre, nous avons présenté les résultats obtenus en mettant en œuvre les méthodologies développées dans les chapitres 4 et 5.

Premièrement, nous avons mis en évidence que la réduction des cycles de gel de données du processeur pouvait causer l'apparition du gel des instructions de pré-chargement insérées avec un effet négatif sur la performance du logiciel. Ceci a nécessité en conséquence une analyse additionnelle que nous avons effectuée également en regardant les statistiques provenant des registres situés dans l'unité de mesure de performance (PMU) disponible dans le type de processeur considéré. Finalement, sur un cas d'étude simple (IDCT), nous avons obtenu une amélioration de la performance globale du logiciel de l'ordre de (25 %) environ.

Deuxièment, nous avons présenté les résultats concernant l'identification des instructions gelées du programme. Ceci en présentant l'importance d'évaluer la pertinence de chacune d'elles pour aider le développeur dans sa prise de décision : les mesures obtenues lui permettent de placer son effort sur les instructions à forte contribution de cycles de gel. Ensuite, toujours sur le programme IDCT, nous avons présenté le gain de performance en terme de temps d'exécution du programme, mais aussi en terme de réduction du nombre de cycles de gel du processeur produit par les instructions gelées les plus pertinentes. Ces instructions étant celles ayant une contribution significative au rallongement du temps d'exécution du programme.

Nous avons également évalué la durée (wallclock) de simulation pour les programmes polybench/C d'algèbre linéaire et de l'IDCT lorsque l'unité matérielle de pré–chargement de données est désactivée et lorsque celle–ci est activée. Ces durées de simulation sur plateforme précise au niveau du cycle permettent d'avoir une idée sur la nécessité d'envisager l'utilisation d'autres plateformes, telles que typiquement une plateforme d'émulation matérielle. Par ailleurs, grâce à l'approche proposée au chapitre 5, nous avons évalué pour chacun des programmes polybench/C le gain potentiel maximal qu'il est possible d'obtenir si tous les cycles de gel de processeur sont évités. Nous avons ensuite comparé le gain maximal pour chacun des programmes au gain réalisé par l'unité matérielle de pré–chargement de données disponible dans le processeur. Ces gains sont calculés à partir des temps d'exécution simulés (cycles) des programmes lorsque l'unité de pré–chargement matérielle est activée ou désactivée. Nous avons pu observer que certains programmes polybench/C sont déjà très optimisés (gain maximal $\leq 0.7\%$) mais qu'il est encore possible d'en optimiser certains autres (ceux qui présentent un gain potentiel maximal $\geq 2\%$).

Par ailleurs, au sein de STMicroelectronics, en parallèle de cette thèse, dans le cadre du

développement d'un circuit (SoC) pour boitier décodeur TV (Set-Top-Box), l'équipe développant le micrologiciel (firmware) du décodeur vidéo s'est retrouvée confrontée à un problème de performance : la durée (temps) de décodage de chaque image était de temps en temps légèrement supérieur au délai imparti pour obtenir un flux continu d'images décodées en Ultra–Haute–Définition, bien que les études d'architecture en amont de l'implémentation aient estimé au préalable que les performances étaient largement suffisantes pour le travail effectué par le processeur embarqué dans le décodeur vidéo.

Après étude des performances par l'équipe de conception sur la base des traces bus, il s'est avéré que le processeur était de temps en temps en attente trop longue sur la lecture des octets des images encodées, alors que celles-ci étaient censées passer par un cache local afin justement de réduire cette latence et d'anticiper les lectures séquentielles des octets d'image.

L'excellente connaissance des développeurs du code et de l'exécution du firmware de décodage a permis d'identifier un des endroits probables où la lecture montre des latences de mémoire excessives. Après étude de la documentation du processeur embarqué, il a été possible d'identifier que celui-ci embarquait une instruction assembleur permettant de précharger des données dans le cache. Ce qui a permis de suggérer l'insertion dans le programme, d'une instruction de pré–chargement de données, pour résoudre ce problème de temps de traitement de certaines images du flux vidéo.

Bien que cette plateforme de production n'était pas instrumentée pour l'analyse de latence décrite dans cette thèse, tout était néanmoins rassemblé pour appliquer « en aveugle » la méthodologie décrite dans ce document, et insérer à un emplacement du code source choisi par le développeur, une instruction assembleur de pré–chargement de données futures à lire par le programme.

Dès le premier essai, une amélioration de 6% des performances globales a été mesurée, largement suffisante pour éliminer les décodages trop longs constatés précédemment, et rentrer systématiquement dans le temps cyclique du traitement d'une image imparti par la spécification de l'architecture pour le standard vidéo concerné. Cette première expérimentation en vraie grandeur n'a pu que nous conforter dans la recherche d'une méthodologie permettant de mesurer les latences de lecture, et d'éliminer une recherche « aveugle » du placement des instructions de pré-chargement dans le code source du programme. C'est ce qui a été conduit et est décrit dans les chapitres 4 et 5.

6.4 Références

ANDERSON, J. M., L. M. BERC, J. DEAN, S. GHEMAWAT, M. R. HENZINGER, S.-T. A. LEUNG, R. L. SITES, M. T. VANDEVOORDE, C. A. WALDSPURGER et W. E. WEIHL. 1997, «Continuous profiling : Where have all the cycles gone ?», *ACM Trans. Comput. Syst.*, vol. 15, nº 4, doi :10.1145/265924.265925, p. 357–390, ISSN 0734-2071. URL http://doi.acm.org/10.1145/265924.265925.

ARM, L. 2017, «Arm soc designer», https://developer.arm.com/products/system-design/cycle-models/arm-soc-designer. [En ligne ; dernier accés le 28 Août 2017]. 108

ARM-OPTIONS. 2017, «Options that control optimization», https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. [En ligne ; dernier accès le 28 Novembre 2017]. 116

LOEFFLER, C., A. LIGTENBERG et M. GEORGE S. 1989, «Practical fast 1-d dct algorithms with 11 multiplications», *International Conference on Acoustics, Speech, and Signal*, p. 988–991. 116

POLYBENCH/C. 2017, «The polyhedral benchmarkssuite», http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/. [En ligne ; dernier accès le 28 Novembre 2017]. 123

SETHIA, A., G. DASIKA, M. SAMADI et S. MAHLKE. 2013, «Apogee : Adaptive prefetching on gpus for energy efficiency», dans *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, IEEE Press, Piscataway, NJ, USA, ISBN 978-1-4799-1021-2, p. 73–82. URL http://dl.acm.org/citation.cfm?id=2523721.2523735. 128

Chapitre 7

Conclusion générale et perspectives

Sommaire

7.1 Conclusion

Dans cette thèse nous nous sommes intéressés à la problématique d'identification et d'évitement des cycles de gel du processeur dus à l'indisponibilité d'une donnée dans les caches de la hiérarchie mémoire d'un SoC. Le type de SoCs visés est constitué de circuits de complexités variées pour différents segments de marché sur lesquels STMicroelectronics est acteur, dans les domaines des systèmes embarqués ou enfouis, un des cœurs de métier de la compagnie. L'identification et l'évitement de ces cycles de gel ayant pour objectif d'améliorer l'utilisation par le logiciel des capacités du matériel qui est contraint en termes de capacité mémoire (systèmes embarqués ou enfouis) et de fréquence du processeur (objectif de consommation et donc de fréquence basse). L'utilisation de plateformes d'exécution de type simulation ou émulation matérielle, permet de prendre des décisions d'optimisation tôt dans le flot de conception de la puce (une fois l'architecture de l'ossature du SoC figée), pour un meilleur temps de mise sur le marché du produit SoC complet matériel et logiciel.

Dans ce sillage, à la question « **Q1** : Comment identifier les instructions du programme qui génèrent des cycles de gel du processeur, et la latence produite lorsque l'on dispose d'une visibilité totale sur les registres internes concernés du processeur ? », nous avons dans un premier temps proposé une méthode itérative basée sur la simulation de plateformes virtuelles précises au niveau du cycle. Cette méthode se focalise sur l'observabilité que peuvent fournir ces plateformes afin de proposer une stratégie qui permet d'insérer manuellement à des endroits identifiés, des instructions de pré–chargement de données dirigé par le logiciel, dans le code source du programme, sans en modifier la fonctionnalité. Cette stratégie utilise l'analyse des accès en mémoire qui produisent des cycles de gel, au moyen des informations extraites des registres de performance (PMU) du processeur. Cette première méthode proposée est dépendante de la capacité du processeur et de son modèle à fournir les informations nécessaires sur son fonctionnement interne au cours de l'exécution du programme. Elle dépend aussi de l'analyse empirique des informations des registres PMU par le développeur. Néanmoins cette méthode nous a permis d'obtenir un gain de performance en termes de temps d'exécution du programme de l'ordre de 25% sur les deux programmes tri à bulles et IDCT, avec un gain mesuré qui dépend de la latence de l'accès à la mémoire.

Dans un second temps, aux questions : « **Q1** : Comment identifier les instructions du programme qui génèrent des cycles de gel du processeur, lorsque l'on ne dispose d'aucune visibilité sur les registres internes concernés du processeur ?, **Q2** : Comment mesurer la durée de chaque gel (en cycles), pour connaître l'importance de son impact ?, **Q3** : Comment utiliser au mieux l'effort d'optimisation, en le concentrant sur les gels ayant le plus d'impact ? » Nous avons proposé une méthode externe (sans besoin d'observabilité interne au proces-

seur) qui permet d'identifier automatiquement les instructions du programme qui contribuent au rallongement du temps global d'exécution de celui–ci parce qu'elles ont été gelées ; menant à la correction par insertion manuelle de pré–chargements dans le programme source. Les phases détection et mesures automatiques, et insertion manuelle, peuvent être itérées par étape (instructions à fort gels en premier), pour une amélioration incrémentale avec vérification de la performance, limitant les risques traditionnels de dégradation de performance du pré–chargement.

Cette seconde approche s'appuie sur l'analyse de deux traces : la trace précise au niveau du cycle des différentes transactions sur le point d'entrée à l'interconnexion, pour les transactions transférées par le dernier niveau de cache (LLC) en direction du contrôleur mémoire ; et la trace d'exécution, composée à chaque cycle des valeurs successives du Program Counter – PC, qui fournissent les adresses des instructions se trouvant à l'étage d'exécution du pipeline du processeur.

La méthodologie permet d'identifier précisément les instructions responsables des cycles de gel du processeur dus à l'absence de données en mémoire cache. Et pour chacune de ces instructions identifiées, la durée du gel c'est–à–dire la contribution en cycles de chacune de ces instructions au rallongement du temps global d'exécution du programme. Ceci apporte une aide au développeur de programme embarqué : (a) dans la connaissance de la possibilité ou pas, de continuer à chercher à gagner des cycles pour une exécution plus rapide de son programme, (b) sur le choix des gels du processeur les plus pertinents dont le traitement apportera rapidement (gels les plus influents en premier) une réduction de la durée d'exécution du programme. Cette approche fait des hypothèses réalistes pour sa mise en œuvre, indépendantes des choix d'implémentation du processeur, de l'interconnexion (hormis le protocole d'entrée qui doit être connu pour identifier les transactions), de l'outil de simulation, et de la manière (source, format) dont les traces sont produites.

En fixant la latence du contrôleur mémoire à 164 cycles, nous avons mesuré avec notre méthode, pour l'IDCT et pour chacun des programmes considérés du benchmark polybench/C d'algèbre linéaire, le gain potentiel maximal qu'il est possible d'obtenir si tous les cycles de gel de processeur sont évités. Nous avons ensuite comparé, pour chacun des programmes, le gain potentiel maximal au gain réalisé par l'unité matérielle de pré–chargement de données disponible dans le processeur. Nous avons observé que certains programmes polybench/C sont déjà suffisamment optimisés (gain maximal $\leq 0.7\%$) mais qu'il est encore possible d'en optimiser certains autres (ceux qui présentent un gain potentiel maximal $\geq 2\%$), notamment les programmes IDCT, trisolv ($\approx 12\%$).

Dans notre méthode, les données à lire par les instructions responsables du gel du processeur sont alors anticipées par pré–chargement en mémoire cache. Ces donnés sont pré-

chargées grâce aux instructions de pré–chargement dirigé par le logiciel, insérées par le développeur (manuellement dans les cas traités) dans le code source du programme; le développeur s'appuyant pour cela sur la connaissance du résultat de notre analyse automatique. Ceci en respectant pour l'insertion une distance d'itération (4.9) adéquate, et en chargeant à temps la bonne quantité de données (fonction de la taille de la ligne de cache et du nombre de tampons de pré–chargement), pour qu'elles soient disponibles dès qu'elles seront requises par l'instruction qui en aura besoin. Sur le programme IDCT, en réduisant les cycles de gel produits par certaines instructions identifiées grâce à l'approche du chapitre 5, et pour la latence du contrôleur mémoire fixée à 164 cycles, le gain en termes de temps d'exécution du programme est de 10% environ, ce qui est un gain significatif lorsque l'on parle de systèmes embarqués ou enfouis.

7.2 Perspectives

Les méthodes et outils décrits permettent l'analyse de traces pour faciliter l'amélioration, ou l'optimisation de la performance conjointe du matériel et du logiciel par pré–chargement de données dans le cache. Ceci grâce à l'identification dans le code source du programme et à la quantification des cycles de gel du processeur lors d'accès à la mémoire. Ceci peut être mis en œuvre à différentes étapes dans le flot de conception d'un SoC.

— (a) Lors de l'étude de l'architecture matérielle du SoC (étape numéro 2 du flot présenté dans la figure 3.2), pendant la phase d'estimation pour le dimensionnement de la hiérarchie mémoire : cache –> interconnexion (typiquement un bus ou un NoC) –> contrôleur(s) mémoire. La méthode apporte une aide dans le choix optimal d'architecture. En effet, les expérimentations effectuées ont permis d'observer par exemple sur le programme IDCT que la meilleure performance du logiciel était obtenue lorsque la latence du contrôleur mémoire était fixée à 204 cycles. Ainsi l'architecte matériel pourrait chercher à dimensionner l'interconnexion (NoC) pour réduire la latence de la hiérarchie mémoire plus près de 204 cycles, si elle était supérieure. Tout en sachant que le programme pourra utiliser au mieux les capacités du matériel lorsque les instructions de pré–chargement de données dirigé par le logiciel seront insérées dans le code source du programme. Lorsque du logiciel est disponible en début de phase d'architecture et conception du SoC, notamment les benchmarks classiques dans l'industrie ou du code existant provenant de l'utilisateur (client) du futur SoC, l'optimisation permise par notre méthode peut permettre pour certains cahiers des charges de démontrer que la performance maximale logiciel-sur-matériel qu'il est possible d'obte-

nir sur le benchmark satisfait le critère de l'utilisateur. La simulation permet à l'architecte SoC d'étudier facilement des variantes de son architecture (tailles et latences des mémoires, topologie et dimensionnement de l'interconnexion bus/NoC, ...).

— (b) Lors de la phase de vérification pré–PG/Tape–Out (avant création des masques) de la performance de la hiérarchie mémoire du SoC. L'architecture matérielle est figée et son implémentation RTL disponible. Davantage de logiciel est disponible qu'en début de conception SoC, grâce au développement logiciel sur plateforme virtuelle SystemC/TLM en parallèle du travail de conception du SoC; ce logiciel représentant les besoins du client ou du marché visé. Davantage de logiciel que de courts benchmarks, signifie un besoin d'exécution rapide de la vérification : l'émulation matérielle peut être envisagée.

— (c) Lors de la maximisation de la performance du logiciel réel sur le SoC final, par exemple les premiers échantillons de silicium, ou produit final pour les clients. (c.1) Lorsque le RTL est disponible, les modèles CA peuvent être générés automatiquement à l'aide d'outils de Conception Assistée par Ordinateur (CAO) électronique désormais matures. En pratique, ces modèles peuvent être utilisés efficacement si l'exécution du logiciel sur l'ossature du SoC simulée requiert au maximum une nuit, pour des raisons d'organisation du travail. Pour les logiciels qui requièrent davantage de cycles, l'émulation matérielle est envisageable; plus rapide, mais coûteuse en machine d'émulation. (c.2) Lorsque les échantillons de silicium sont disponibles et que les traces peuvent être obtenues de manière non–intrusive; ce qui signifie typiquement qu'un dispositif de trace a été prévu et s'il a pu être implémenté dans le SoC, avec une performance de génération de trace PC et bus à chaque cycle.

Dans cette thèse, nous avons expérimenté (a) et (c.1) en utilisant les modèles précis au niveau du cycle (CA). La démonstration de l'utilisation de la méthode (chapitre 5) sur émulateur matériel, pour $(b), (c.1), (c.2)$ est une perspective envisageable.

Les travaux effectués sont une base possible pour étudier également l'impact du pré–chargement dirigé par le logiciel sur le reste du système, notamment son impact sur la consommation d'énergie, ou encore son efficacité et ses effets sur le cache (pollution du cache, « cache thrashing », ...), même si dans le cas de nos expérimentations nous n'avons pas observé de pollution de cache grâce notamment à l'étude de la distance de pré–chargement : notre méthode a permis d'augmenter la performance.

7.2.1 L'émulation pour simuler plus rapidement

L'utilisation de modèles de simulation précis au niveau du cycle a permis par l'observabilité qu'ils fournissent, d'étudier la méthodologie et les outils associés d'analyse de traces. Ils sont utiles notamment en phase d'étude d'architecture SoC, car ils permettent d'itérer rapidement sur des options d'architecture de l'ossature du SoC, et le temps de simulation n'est pas un critère clé pour les architectes.

En revanche, les expérimentations montrent que même pour des programmes de petite taille (sous–ensemble d'algèbre linéaire du polybench/C, IDCT), les durées de simulation sont relativement élevées. Ainsi, pour des logiciels plus volumineux, lorsque le programme est développé en parallèle de la conception matérielle du SoC, une exécution plus rapide mais conservant la précision au niveau du cycle sera nécessaire pour pouvoir appliquer les méthodologies proposées dans cette thèse. Une modélisation au niveau transactionnelle (TLM) fournit une exécution rapide mais n'a pas la précision au niveau du cycle nécessaire au type d'analyse de la méthodologie proposée au chapitre 5. Une exécution de type VHDL RTL ou Verilog RTL est précise mais lente.

Une solution est l'exécution du RTL de l'ossature du SoC sur un émulateur matériel (MENTOR [2017]). Ceci est rapidement possible après la disponibilité du RTL au cours d'un projet. Ce RTL de l'ossature du SoC peut être exécuté pour certains projets sur une plateforme FPGA. Cependant cette plateforme FPGA nécessiterait d'être instrumentée pour pouvoir créer la trace PC et bus/NoC (à l'interface LLC) à chaque cycle.

Notre méthodologie reste utilisable sur émulateur, du fait qu'elle ne requiert que des données d'entrées faciles à obtenir sur un émulateur : trace des PCs, trace des transactions bus, avec une référence commune d'horloge. Le déploiement sur émulateur est utile aux équipes de développement logiciel en amont des échantillons silicium, ainsi qu'aux ingénieurs de validation qui mettent au point leurs logiciels de tests (fonctionnels et performance SoC) pour être prêts à recevoir les échantillons du SoC.

7.2.2 Evaluation de l'impact du pré–chargement de données sur la consommation énergétique

Avec l'augmentation de la fréquence de traitement des processeurs au sein des SoCs, la consommation énergétique est devenue un critère essentiel, car la puissance consommée à tout instant augmente avec la fréquence de fonctionnement. D'où par exemple l'ajout de modes de gestion des horloges et donc de la puissance (« power modes »), modes de plus en plus complexes dans les SoC récents. Or, quand un CPU est en attente active de données

afin de poursuivre l'exécution d'un programme, il consomme inutilement de l'énergie. Une partie de cette énergie est dissipée sous forme de chaleur. Ceci pouvant générer d'autres problèmes, notamment de refroidissement du circuit intégré sur la puce.

Dans les travaux de l'outil AVALANCHE HENKEL et YANBING [Aug. 2002], des modèles de consommation énergétique pour certains composants sont présentés, notamment le processeur, le cache et la mémoire. Concernant le processeur, on remarque dans l'équation (7.1) que le nombre de cycles de gel du processeur y figure et plus il croît, plus l'énergie totale finale consommée par le circuit pour l'exécution du programme augmente.

$$E_{proc} = E_0 \times \mathbf{N_0} + E_1 \times N_1 \tag{7.1}$$

Avec

— $\mathbf{N_0}$: le nombre de cycles de gel du processeur

— N_1 : le nombre de cycles durant lesquels le processeur est actif

Les énergies E_0 et E_1 sont respectivement données par relations suivantes :

$$E_0 = Tw_c \times V_{dd} \times I_{nop} \tag{7.2}$$

$$E_1 = Tw_c \times V_{dd} \times I_{inst} \tag{7.3}$$

Avec

— Tw_c : le temps nécessaire pour l'exécution du programme

— I_{nop} : le courant consommé pendant un cycle de gel du processeur

— I_{inst} : le courant consommé pendant un cycle actif du courant.

Pour la détermination des intensités I, les auteurs font référence aux travaux réalisés par TIWARI [1996].

La méthode d'identification automatique des instructions responsables de gels, et de leur évitement, présentée dans cette thèse, pourrait donc être utilisée par un développeur de logiciel pour diminuer la consommation énergétique de son programme sur une architecture matérielle pour laquelle il disposerait de la trace d'instructions et la trace de transactions du dernier niveau (LLC) de cache vers la mémoire.

7.2.3 Modèles encore plus amont : TLM–timed (TLM timés)

Pour réaliser ces travaux, la simulation sur des plateformes CA a joué un rôle prépondérant. Cependant, afin de permettre aux architectes matériels d'utiliser la méthodologie plus en amont dans le flot de conception du SoC((*a*)), il serait intéressant d'envisager la mise en

œuvre de celle–ci sur des plateformes intégrant des modèles TLM–timed (TLM timés) des blocs/IPs dont le RTL n'est pas encore disponible. En effet, lorsque dans un projet SoC certaines parties de la hiérarchie mémoire sont à développer (par exemple un nouveau contrôleur flash pour atteindre les objectifs de performance d'un marché nouvellement visé), on peut envisager l'utilisation initiale en (*a*) de modèles TLM–timed, par exemple pour le nouveau contrôleur flash (avec des transacteurs pour l'insertion dans le modèle CA de l'ossature du SoC). Puis, le modèle TLM–timed du contrôleur flash pourra être remplacé par son RTL quand il devient disponible au cours du projet. Ensuite, la performance des benchmarks sera vérifiée et comparée à celle de la phase initiale qui comportait un modèle initial TLM–timed moins précis que le modèle CA du (nouveau) RTL, afin de vérifier au cours des phases du projet SoC de la tenue des performances.

7.3 Références

ELABIDINE, K. Z. 2014, *Méthode de prototypage virtuel permettant l'évaluation précoce de la consommation énergétique dans les systèmes intégrés*, thèse de doctorat, université Pierre et Marie Curie - Paris VI.

HENKEL, J. et L. YANBING. Aug. 2002, «Avalanche : an environment for design space exploration and optimization of low-power embedded systems», *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, p. 454–468. 139

MENTOR, G. 2017, «Veloce emulation platform–mentor graphics», https://www.mentor.com/products/fv/emulation-systems/. [En ligne ; dernier accès le 09 Décembre 2017]. 138

TIWARI, V. 1996, «Logic and system design for low power consumption», *Princeton University*. 139

Annexe A

Liste des acronymes

ALU Arithmetic–Logic Unit. 92

AMBA Advanced Microcontroller Bus Architecture. 86, 103, 104

ASIC Application-Specific Integrated Circuit. 28

ASSP Application-Specific Standard Product. 28

AXI Advanced eXtensible Interface. vii, 86, 87, 89, 90, 103, 104, 107

CA Précise au niveau du cycle –ou *Cycle Accurate*. 2, 16, 25, 28–31, 55, 56, 78, 85, 135, 137, 138

CAO Conception Assistée par Ordinateur. 135

CPI Cycles Par Instruction –ou *Cycles Per Instruction*. 12, 14, 15, 94

CPU Central Processing Unit. 10, 12, 16, 36, 41, 56, 58, 69, 86, 89, 107, 108, 115, 117, 136

DMA Accès Direct à la mémoire –ou Direct Memory Access. 25, 37

DRAM Dynamic Random–Access Memory. 12, 14, 15, 26, 28, 56, 58, 60, 65, 66, 71, 86, 94, 96, 101, 107, 120

EDA Electronic Design Automation. 2

ESL Electronic System Level. 2

FPGA Field Programmable Gate Array. 29

GCC GNU Compiler Collection. 4, 40, 46, 74, 79, 101, 113, 114, 124

HDL Hardware Description Language. 30, 31

IA Instruction Accurate. 28

IDCT Inverse Discrete Cosine Transform. iii, 105, 114, 118, 121, 123, 128, 132

IoT Internet des objets –ou *Internet of Things*. 8

IP Intellectual Property. 8, 9, 25, 26, 29–31, 106, 138

ISS Instruction Set Simulator. 106

KO Kilo Octet. 34, 40, 106, 107, 120

LLC Last Level Cache. 55, 79, 85, 86, 94, 95, 101, 133, 137

MIPS Microprocessor Without Interlocked Pipeline Stages. 10, 32

MMU Memory Management Unit. 70, 91

MSHR Miss Status Holding Register. 98

NoC Réseau sur puce – ou Network–on–Chip. 134, 135

PC Program Counter. vii, ix, 17, 58–60, 70, 84, 91–97, 101, 102, 119, 124, 133, 136

PMU Performance Monitoring Unit. 16, 56, 57, 59, 63, 65, 70, 72, 73, 78–80, 109, 115, 119, 128, 132

RISC Reduced Instruction Set Computing. 9, 11, 14

ROB ReOrder Buffer. 32

RTL Register Transfert Level. 2, 25, 28–31, 56, 135, 136, 138

SMT Simultaneous Multithreading. i, 23, 34

SoC Système sur la puce –ou *System-on-a-Chip*. v, 2, 3, 8, 9, 12, 13, 17, 25, 26, 29, 31–34, 39, 44, 54, 65, 78–80, 85, 86, 102, 106, 107, 120, 132, 134–138

SPARC Scalable Processor Architecture. 107

TEI Temps d'Exécution Idéal. 15

TER Temps d'Exécution Réel. 15, 17–19

TLB Translation Lookaside Buffer. 62, 65

TLM Transaction Level Modeling. 28–30, 136–138

Annexe B

Résumé / Abstract

Résumé

L'un des objectifs de la microélectronique est de concevoir et fabriquer des SoCs de petites tailles, à moindre coût et visant des marchés tel que l'internet des objets. À matériel fixe sur lequel l'on ne dispose d'aucune marge de manœuvre, l'un des challenges pour un développeur de logiciels embarqués est d'écrire son programme de manière à ce qu'à l'exécution, le logiciel développé puisse utiliser au mieux les capacités de ces SoCs. Cependant, ces programmes n'utilisent pas toujours correctement les capacités de traitement disponibles sur le SoC. L'estimation et l'optimisation de la performance du logiciel devient donc une activité cruciale. A l'exécution, ces programmes sont très souvent victimes de l'apparition de cycles de gel de processeur dus à l'absence de données en mémoire cache. Il existe plusieurs approches permettant d'éviter ces cycles de gel de processeur. Par l'exemple l'utilisation des options de compilation adéquates pour la génération du meilleur code exécutable possible. Cependant les compilateurs n'ont qu'une idée abstraite (sous forme de formules analytiques) de l'architecture du matériel sur lequel le logiciel s'exécutera. Une alternative est l'utilisation des processeurs « Out–Of–Order ». Mais ces processeurs sont très couteux en terme de coût de fabrication car nécessites une surface de silicium importante pour l'implantation de ces mécanismes. Dans cette thèse, nous proposons une méthode itérative basée sur les plateformes virtuelles précises au niveau du cycle qui permet d'identifier les instructions du programme à optimiser responsables à l'exécution, de l'apparition des cycles de gel de processeur dus à l'absence de données dans le cache L_1. L'objectif est de fournir au développeur des indices sur les emplacements du code source de son programme en langage de haut niveau (C/C++ typiquement) qui sont responsables de ces gels. Pour chacune de ces instructions, nous fournissons leur contribution au rallongement du temps d'exécution totale du programme. Finalement nous estimons le gain potentiel maximal qu'il est possible

d'obtenir si tous les cycles de gel identifiés sont évités en insérant manuellement dans le code source du programme à optimiser, des instructions de pré–chargement de données dirigé par le logiciel.

Abstract

One of microelectronics purposes is to design and manufacture small-sized, low-cost SoCs targeting markets such as the Internet of Things. With fixed hardware on which there is no possible flexibility, one of the challenges for an embedded software developer is to write his program so that, at runtime, the software developed can make the best use of these SoC capabilities. However, these programs do not always properly use the available SoC processing capabilities. Software performance estimation and optimization is then a crucial activity. At runtime, these programs are very often victims of processor data stall cycles. There are several approaches to avoiding these processor data stall cycles. For example, using the appropriate compilation options to generate the best executable code. However, the compilers have only an abstract knowledge (as analytical formulas) of the hardware architecture on which the software will be executed. Another way of solving this issue is to use Out-Of-Order processors. But these processors are very expensive in terms of manufacturing cost because they require a large silicon surface for the implementation of the Out-Of-Order mechanism. In this thesis, we propose an iterative methodology based on cycle accurate virtual platforms, which helps identifying precisely instructions of the program which are responsible of the generation of processor data stall cycles. The goal is to provide the developer with clues on the source code lignes of his program's in high level language (C/C++ typically) which are responsible of these stalls. For each instructions, we provide their contribution to lengthening of the total program execution time. Finally, we estimate the maximum potential gain that can be achieved if all identified stall cycles are avoided by manually inserting software preloading instructions into the source code of the program to optimize.