**THÈSE DE DOCTORAT DE**
**l'UNIVERSITÉ PIERRE ET MARIE CURIE**

Spécialité

**Informatique**

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

**Alexandre CARBON**

Pour obtenir le grade de

**DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE**

Sujet de la thèse :

# Accélération matérielle de la compilation à la volée pour les systèmes embarqués

soutenue le 17 octobre 2013

devant le jury composé de :

| | |
|---|---|
| M. Bertil FOLLIOT, Professeur à l'Université Pierre et Marie Curie | Président de jury |
| M. Philippe CLAUSS, Professeur à l'Université de Strasbourg | Rapporteur |
| M. Gilles SASSATELLI, Directeur de recherche CNRS à l'Université de Montpellier 2 | Rapporteur |
| M. Fabrice RASTELLO, Chargé de recherche à l'INRIA | Examinateur |
| M. Henri-Pierre CHARLES, Ingénieur Chercheur au CEA LIST | Directeur de thèse |
| M. Yves LHUILLIER, Ingénieur Chercheur au CEA LIST | Encadrant de thèse |

# Remerciements

Ce manuscrit est le fruit de trois années de thèse passées au sein du Laboratoire Calcul Embarqué (LCE) du Département Architecture, Conception et Logiciels Embarqués (DACLE) du CEA LIST. C'est pourquoi, je tiens tout d'abord à remercier Messieurs Thierry Collette et Raphaël David, respectivement chefs du DACLE et du LCE, de la confiance qu'ils m'ont témoigné et de m'avoir accueilli au sein de leurs équipes, me permettant d'effectuer cette thèse dans d'excellentes conditions. Je remercie également Raphaël pour ses nombreuses suggestions au cours de ces trois années.

Je tiens également à remercier Monsieur Henri-Pierre Charles, Ingénieur chercheur au CEA LIST, d'avoir accepté la direction de cette thèse et pour les nombreux conseils avisés et retours d'expériences dont il a pu me faire bénéficier.

J'adresse mes remerciements les plus sincères à Monsieur Yves Lhuillier, Ingénieur chercheur au CEA LIST, pour avoir encadré cette thèse avec dévouement et patience. Son aide, ses conseils et son écoute durant ces trois années m'ont permis d'aboutir à la réalisation de ces travaux et ont contribué à faire de cette thèse une expérience extrêmement enrichissante, tant d'un point de vue professionnel que personnel.

Je remercie Monsieur Bertil Folliot, Professeur à l'Université Pierre et Marie Curie, de m'avoir fait l'honneur de présider mon jury de thèse.

Je remercie Monsieur Philippe Clauss, Professeur à l'Université de Strasbourg, et Monsieur Gilles Sassatelli, Directeur de recherche CNRS au LIRMM, d'avoir accepté d'évaluer mon travail en qualité de rapporteurs.

Je tiens à remercier Monsieur Fabrice Rastello, Chargé de recherche à l'INRIA, pour sa participation à ce jury de thèse comme examinateur.

Je remercie l'ensemble de l'équipe du LCE (et ses anciens membres), pour m'avoir permis de réaliser ces travaux au sein d'une équipe extrêmement compétente, dynamique et pleine de bonne humeur. Nos nombreux échanges et les très bons moments passés avec eux durant ces trois années m'ont été d'une aide précieuse. C'est un très grand plaisir pour moi de continuer à travailler avec eux.

Je tiens plus particulièrement à remercier mes collègues et amis thésards du bureau 2060, David, Julien, Marie, Olivier et Phuc, pour leur aide et leurs encouragements, en particulier durant ma période de rédaction. L'entraide et la bonne ambiance au sein de ce bureau m'ont beaucoup aidé, et je garderai un très bon souvenir de toutes les soirées que nous avons pu passer ensemble. Je remercie tout particulièrement Julien pour nos nombreuses discussions et son soutien dans les moments de doute, et lui souhaite bon courage pour la fin de sa thèse.

J'ai également une pensée pour Karim et Maroun, avec qui j'ai pu vivre d'excellents moments durant ces trois années, et notamment un séjour inoubliable au Liban.

Enfin, je souhaiterais terminer ces remerciements par un immense merci à mes parents et à mes amis, Christelle et Morgan. Leur présence et leur soutien sans faille durant ces trois années, dans les bons comme dans les mauvais moments, m'ont été indispensables pour arriver au terme de ces travaux. C'est pourquoi je souhaiterais leur dédier cette thèse.

# Table des matières

# Table des figures

# Liste des tableaux

# Introduction

Apparue au début des années soixante, **la compilation dynamique (ou compilation à la volée)** connaît un essor considérable depuis une quinzaine d'années. Initialement développée avec succès pour les architectures de type station de travail ou serveur, elle est aujourd'hui largement transférée dans le domaine des systèmes embarqués.

Consistant au transfert des phases de compilation à l'exécution de l'application, son développement s'explique par l'émergence de deux phénomènes dans la conception des applications et des architectures.

Le premier phénomènes concerne **le dynamisme croissant des applications**, avec une sensibilité grandissante de ces dernières à leur jeu de données. Ce phénomène est illustré par une modification profonde des applications, notamment de traitement d'image. A l'origine basées sur l'utilisation de codes réguliers (imbrication de boucles, tâches séquentielles, etc.) pour des traitements bas-niveau de filtrage, elles utilisent aujourd'hui des codes de nature plus complexe comme ceux dédiés à l'analyse de contenu. Ces codes peuvent évoluer de manière importante au niveau comportemental en fonction de leur environnement d'exécution. D'importantes optimisations ne peuvent donc plus être réalisées statiquement par les compilateurs classiques. C'est à ce niveau que l'intérêt de la compilation dynamique prend tout son sens. Cette dernière permet en effet de compiler l'application et d'en générer le code machine tout en profitant des informations disponibles à l'exécution sur le jeu de données. Ces applications, initialement réservées aux architectures de type station de travail ou serveur, sont aujourd'hui largement transférées dans l'embarqué, avec l'émergence par exemple de systèmes de vision toujours plus intégrés. Un autre phénomène illustrant le dynamisme croissant des applications est l'augmentation des interactions entre elles. Ce phénomène est parfaitement visible dans le cadre de l'émergence des smartphones, sur lesquels les applications vont évoluer en fonction des autres, comme illustré sur la figure 1, et où l'ensemble des données des différentes applications interagissent entre elles : les réseaux sociaux, le cloud, les données mutlimédia, etc. De manière générale donc, l'importance grandissante de l'environnement d'exécution sur l'évolution comportementale d'une application a fortement poussé à l'émergence de la compilation dynamique dans ce domaine.

Le second phénomène concerne **l'évolution des architectures** au cours de ces quinze dernières années. Les limites technologiques ne permettant plus l'augmentation des fréquences de fonctionnement, l'apparition d'architectures multi-coeurs s'est généralisée, aussi bien dans le domaine des architectures de type station de travail ou serveur que dans le domaine embarqué. Ce développement du parallélisme s'est accompagné par l'apparition d'une approche hétérogène dans la conception des architectures. Elle consiste à associer des ressources de nature différente sur un même circuit, afin de fournir aux applications les ressources adéquates pour leur exécution. Dans le domaine des architectures de type station de travail ou serveur, cette hétérogénéité apparaît aujourd'hui comme étant relativement simple. Elle consiste le plus souvent en l'association de coeurs généralistes avec un coeur graphique. Concernant les architectures

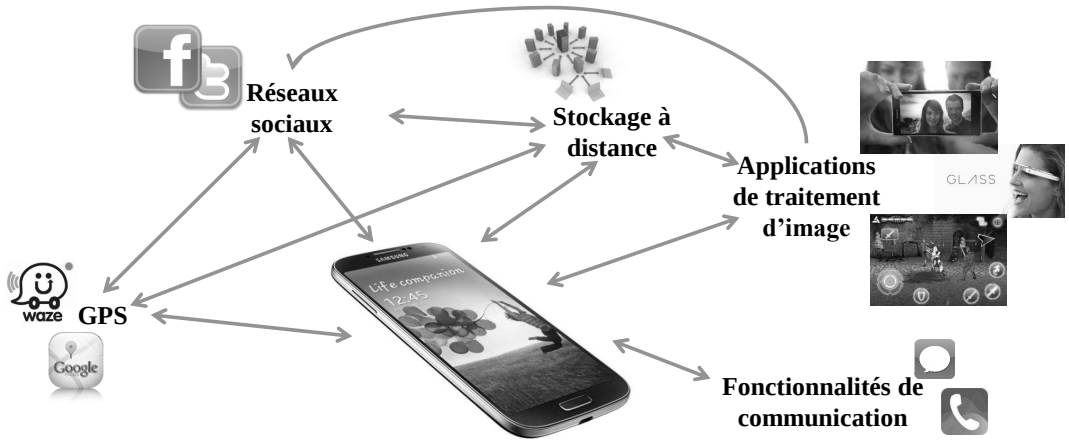



FIGURE 1 – Evolution des interactions entre les applications : illustration avec le cas des smartphones.

embarquées, les fortes contraintes au niveau des efficacités énergétique et surfacique nécessitent l'utilisation de ressources spécifiques pour les applications. Ces contraintes ont conduit à l'émergence d'une hétérogénéité très forte dans ces architectures, avec l'association de coeurs généralistes, de coeurs graphiques, mais aussi de coeurs de traitement de signal, de coeurs à faible consommation (approche ARM BigLITTLE [1]) et même d'accélérateurs matériels spécifiques au domaine applicatif auquel se destine l'architecture. Ces nouvelles architectures asymétriques sont largement développées et utilisées dans l'industrie (TI OMAP [2], Qualcomm [3], ST, etc.). La figure 2 illustre cette évolution des architectures.

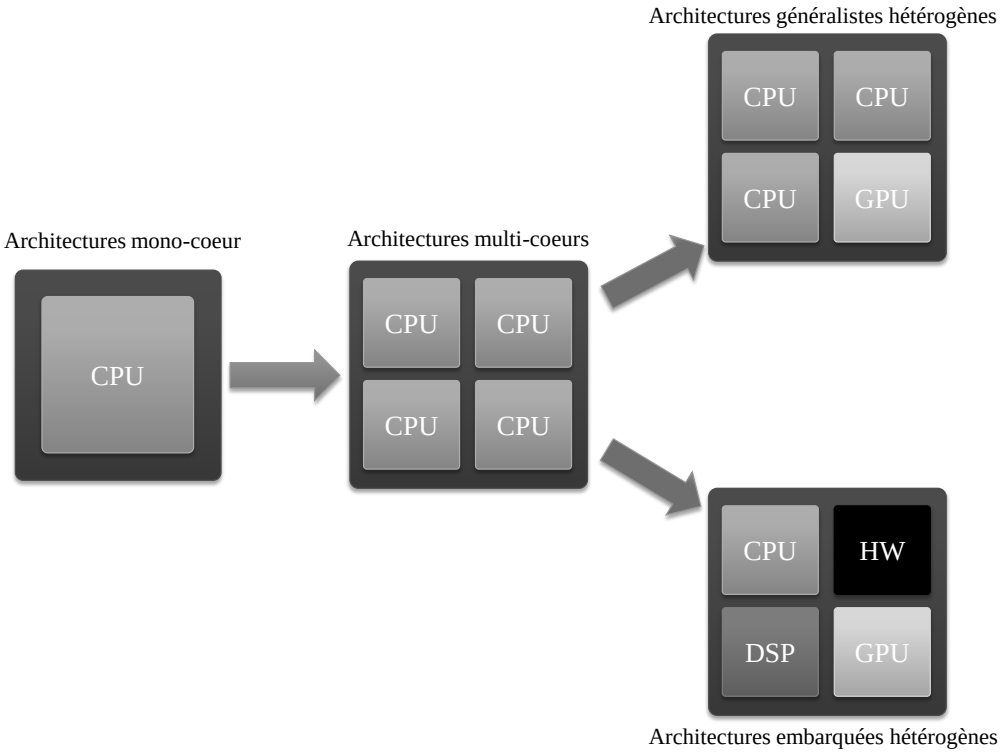



FIGURE 2 – Evolution des architectures lors de ces quinze dernières années : des architectures mono-coeur aux architectures multi-coeurs hétérogènes.

**Cette hétérogénéité croissante** a conduit à l'apparition d'une problématique majeure sur le **déploiement des applications** sur ces architectures pour deux raisons principales : **la difficulté de cibler ces architectures** et **la segmentation croissante du marché** entre les développeurs d'applications et les développeurs d'architectures, conduisant à l'émergence de nouveaux besoins en portabilité.

Concernant le ciblage de ces architectures, la multiplication des jeux d'instructions et des fonctionnalités devant être gérés simultanément et l'optimisation de l'utilisation des ressources complexifient ce déploiement. Ce phénomène est accentué par le dynamisme des applications préalablement mentionné. Il devient difficile d'en optimiser statiquement le déploiement, celui-ci dépendant fortement de l'évolution de son exécution, tout en garantissant une utilisation optimale des ressources. La solution parfois adoptée de conserver en mémoire l'ensemble des codes natifs de l'application pour les différentes ressources de l'architecture n'est également pas envisageable dans l'embarqué pour des questions d'empreinte mémoire.

Concernant la segmentation croissante du marché, cet aspect engendre l'émergence d'architectures toujours plus spécialisées. Les applications doivent aujourd'hui être facilement transférables d'une architecture à une autre. Ces mêmes architectures ont également vu leur durée de vie considérablement diminuer ces dernières années face à celle des applications. Les développeurs font, de plus, largement appel à de nouveaux langages de programmation à haut niveau d'abstraction afin de s'affranchir de ces problématiques (Java, C#, JavaScript, etc.). Ces éléments ont entraîné l'apparition de nouveaux besoins en portabilité.

Cette évolution des architectures et des applications a conduit à l'augmentation de l'utilisation **de solutions de virtualisation** visant à offrir une couche d'abstraction entre le logiciel et le matériel. Les premiers travaux sur le sujet datent du début des années 60 avec l'apparition de la première machine virtuelle développée par IBM, IBM 360/67 [4]. Aujourd'hui, les machines virtuelles sont largement utilisées, aussi bien dans le domaine des architectures de type station de travail ou serveur que dans les systèmes embarqués : machines virtuelles Java (HotSpot VM [5], Google Dalvik [6], etc.), CIL [7], LLVM [8], moteurs d'exécution JavaScript (Google V8 [9], Microsoft SPUR [10], etc.), Python [11] ou Matlab [12]. Ces solutions de virtualisation, basées initialement sur l'interprétation, ont vu l'apparition de la compilation dynamique au sein de leur système afin de satisfaire les besoins croissants en performances, comme illustré sur la figure 3.

En conséquence, **la compilation dynamique est aujourd'hui largement utilisée**, aussi bien dans le domaine des architectures généralistes que dans celui des architectures embarquées. Toutefois, les codes de compilation ont la particularité établie d'être **des codes complexes** à gérer, avec des spécificités très hétéroclites, mêlant phases complexes de contrôle et gestion de données abstraites. L'introduction des flots de compilation au sein de l'exécution du programme a mis en évidence l'importance de gérer au mieux cette complexité afin d'en limiter l'impact sur les temps globaux d'exécution, incluant temps d'exécution et de compilation. Les résultats obtenus sur les architectures généralistes (type station de travail ou serveur), ont été rapidement prometteurs (même si des perspectives d'amélioration subsistent) et ont poussé au développement de solutions basées sur la compilation dynamique, sa complexité étant tout à fait gérable par ces architectures et son incorporation permettant d'obtenir des gains significatifs.

Cependant, il est difficile pour **des processeurs embarqués** à fortes contraintes en efficacités énergétique et surfacique **de gérer la complexité inhérente aux codes de compilation**, ceux-ci ne disposant pas des mécanismes à disposition des architectures généralistes (exécution dans l'ordre, peu ou pas de spéculation, hiérarchies mémoires

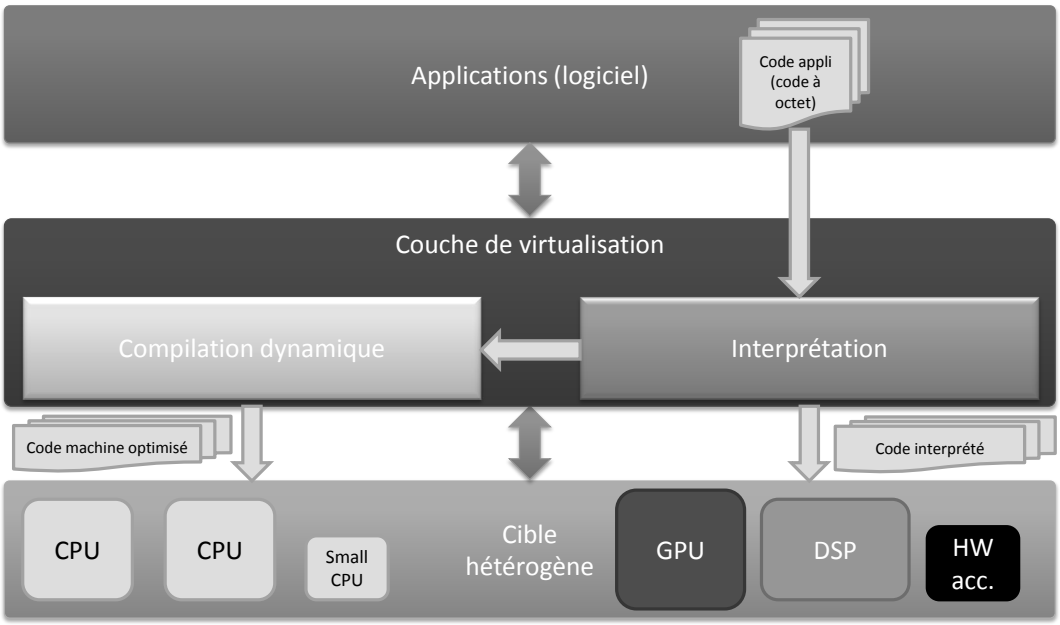



FIGURE 3 – Incorporation de la couche de virtualisation entre le logiciel et le matériel, basée sur l'utilisation de l'interprétation et de la compilation dynamique pour les portions de code les plus critiques.

et tailles de caches limitées). Il en résulte un problème de passage à l'échelle pour les technologies basées sur la compilation dynamique. Les temps d'exécution globaux s'en trouvent considérablement rallongés par rapport à l'exécution d'une application compilée au préalable, hors-ligne, et peu réduits par rapport à une virtualisation basée sur de l'interprétation, limitant ainsi fortement l'attractivité de la compilation dynamique, en dépit d'un gain potentiel élevé.

Afin d'optimiser les codes de compilation dynamique et d'en réduire l'impact sur les temps d'exécution, deux approches ont aujourd'hui essentiellement été envisagées : **les optimisations logicielles** des cadriciels (*frameworks*) de compilation dynamique et **l'optimisation des systèmes** sur lesquels ces cadriciels s'exécutent.

## Optimisations logicielles

Concernant ces optimisations, trois approches majeures peuvent aujourd'hui être mentionnées.

La première se focalise sur **la mise en place de systèmes de caches sophistiqués** afin d'augmenter les capacités de stockage des portions de code générées et de limiter ainsi au maximum les phases de compilation dynamique. C'est notamment le cas des travaux menés par Baiocchi et al. [13] qui proposent un système de cache intelligent permettant d'accroître les possibilités de stockage du code généré, par un système d'analyse des portions les plus fréquemment utilisées, directement accessibles, et une compression des autres. Les gains obtenus par cette approche sont en moyenne de 1.5× en fonction des paramètres spécifiés (et notamment de la taille du cache), en comparaison avec une solution classique de gestion mémoire dans le cadre d'une machine virtuelle (Strata VM [14]). Toutefois, les auteurs cherchent au cours de cette étude à profiter de cette approche afin de limiter au maximum les nouvelles générations de code, pour limiter leur impact sur les performances et accroître les gains obtenus. Cela réduit considérablement l'utilisation des solutions de compilation dynamique et donc leur intérêt.

L'approche suivante se focalise sur **les optimisations logicielles des algorithmes des cadriciels (*frameworks*) de compilation dynamique**, afin de limiter l'impact des phases les plus coûteuses en temps d'exécution. C'est le cas du compilateur du cadriciel LLVM où les développeurs se sont affranchis de l'utilisation des librairies standards afin d'utiliser leurs propres librairies de gestion des structures de données, réduisant leur temps d'accès. Les développeurs ont également cherché à limiter au maximum les allocations dynamiques de la mémoire, très coûteuses en temps d'exécution. Les résultats obtenus montrent que LLVM permet, à performances de code généré équivalentes, de générer ce code aussi rapidement que GCC [15, 16], la portabilité et la faible empreinte mémoire en plus afin de permettre son utilisation dans un contexte dynamique et embarqué. La portabilité offerte par LLVM, sa réutilisation facilitée et ses performances lui permettent d'être utilisé aujourd'hui aussi bien pour des solutions de virtualisation que pour des solutions de performances pures. Côté virtualisation, nous pouvons citer par exemple VMKIT pour lequel LLVM permet d'obtenir des performances équivalentes et même parfois meilleures que les machines virtuelles concurrentes, la portabilité en plus [17]. Côté performances, il est possible de mentionner les travaux menés pour la parallélisation d'applications OpenCL par Jääskeläinen et al. [18] montrant des gains potentiels permettant d'exploiter au mieux le parallélisme (gains de 3.76×). Le compilateur de LLVM est également aujourd'hui utilisé comme brique de base pour le déploiement d'applications via OpenCL sous ARM [19], Apple [20] et NVidia PTX [21]. Toutefois, le problème majeur avec cette spécialisation des librairies réside dans l'augmentation significative du volume de code au sein du compilateur, dépassant le million de lignes de code, et une difficulté supplémentaire pour les développeurs qui doivent s'adapter à l'utilisation de ces librairies.

La troisième approche, défendue par Erven Rohou et Albert Cohen dans [22], propose **une interaction plus forte entre les différentes phases de compilation** intervenant durant la vie d'un programme, via une approche appelée compilation scindée, ou *split compilation*. L'étude réalisée s'inscrit dans le cadre de l'accroissement de la demande en virtualisation dans les systèmes embarqués, poussée par l'hétérogénéité grandissante de ces systèmes. L'idée, derrière cette notion de compilation scindée, est de mettre en place une variante de la compilation dynamique permettant de réaliser les étapes d'optimisation aux différentes phases de l'existence d'un programme, en coordonnant les opérations réalisées entre ces phases (allant de la compilation hors-ligne aux étapes inter-exécution, en passant par l'étape de raccordement mémoire et l'état d'exécution). Erven Rohou et Albert Cohen proposent, à travers cette approche, d'accroître les performances de la virtualisation pour des architectures embarquées hétérogènes grâce à cette notion de compilation scindée. Pour cela, ils proposent une solution en trois étapes :

- étendre cette compilation scindée aux programmes écrit en C/C++, facilitant la portabilité des applications pour les développeurs de ces dernières ;
- se placer dans une optique de performances, en permettant la génération de code natif optimisé hors-ligne ;
- étendre cette compilation scindée à l'ensemble des ressources des systèmes hétérogènes, incluant par exemple les processeurs de traitement de signal et les accélérateurs matériels.

Comme pour les approches classiques, le code initial de l'application est transformé dans une représentation intermédiaire interne, afin notamment d'y réaliser les différentes optimisations, avant d'être traduit en code machine. Mais, contrairement à ces approches classiques de virtualisation et de compilation dynamique, les optimisations statiques ne sont pas complètement remplacées par leur version dynamique, elles sont au contraire complémentaires. Ainsi, les programmes optimisés statiquement sont à

nouveau optimisés à l'exécution en tirant profit des informations disponibles à ce moment là. La complexité des optimisations est donc en grande partie cloisonnée aux étapes hors-lignes, limitant l'impact de leur réalisation sur l'exécution, où ne seront opérées que les dernières étapes relatives à ces optimisations. Le champ concerné au niveau de ces dernières est grand, allant de la vectorisation automatique, notamment pour les processeurs SIMD (*Single Instruction, Multiple Data*), aux optimisations plus classiques, comme l'allocation de registres ou la parallélisation de boucle.

### Optimisation des systèmes

Elle consiste à fournir à la compilation dynamique, et plus globalement à l'ensemble des services de virtualisations (incluant notamment ramasse-miettes et interprétation), leurs propres ressources, afin que ceux-ci s'exécutent en parallèle et sans interférer avec les ressources allouées à l'exécution de l'application. Deux approches existent à ce niveau : l'utilisation de **ressources dédiées spécialisées** ou de **ressources dédiées existantes** (type processeur embarqué).

Au niveau **des ressources spécialisées**, il est possible de citer l'ensemble des processeurs et extensions matérielles Java développés au début des années 2000 dans l'optique de permettre d'optimiser la gestion et l'exécution des applications Java. C'est le cas de l'étude effectuée par ARM dans le développement de l'extension de processeur Jazelle [23] et qui compose aujourd'hui l'ensemble de ses processeurs. Cette extension vise à fournir un jeu d'instructions permettant d'optimiser l'exécution des machines virtuelles Java et d'accélérer l'exécution de leur code à octet. Les résultats présentés par ARM mettent en avant un gain significatif de 6× par rapport à une exécution de machine virtuelle logicielle avec interprétation seule, et des performances équivalentes à celles obtenues par l'inclusion de la compilation dynamique dans une machine virtuelle logicielle, avec toutefois de meilleurs résultats au niveau des efficacités énergétique et surfacique et sans le surcoût induit à l'exécution par les phases de compilation. Par rapport à un co-processeur ou à un processeur Java, comme JOP [24], ARM Jazelle annonce des gains de l'ordre de 2×. ARM Jazelle permet l'accélération de 95 % des codes à octet exécutés avec un encombrement silicium pour l'extension de l'ordre de 12 Kgates. Toutefois, l'utilisation de cette approche reste marginale. En effet, elle manque de flexibilité, est difficilement accessible et ne peut s'employer que pour des applications Java, dont l'utilisation reste très spécifique dans l'embarqué.

L'autre approche, plus récente, se focalise sur **l'utilisation d'une ressource standard (existante)**, de type processeur embarqué, avec transfert des services de virtualisation les plus génériques. C'est le cas de l'étude réalisée par Cao et al. qui propose de mettre en place un processeur embarqué de type RISC dédié à l'exécution des services de virtualisation. Cette solution offre une flexibilité accrue par rapport aux processeurs Java, lui permettant de s'adapter à différents types de machines virtuelles (et pas seulement les machines virtuelles Java). Les gains enregistrés montrent une réduction de 13 % en moyenne du temps d'exécution global en comparaison avec un partage des temps d'exécution entre virtualisation et exécution sur les ressources allouées à cette dernière. Les auteurs mentionnent toutefois que les gains sont essentiellement obtenus sur les services annexes à la compilation dynamique (ramasse-miettes et interprétation) et que cette dernière ne tire pas ou peu de bénéfice de la ressource dédiée, malgré un très fort potentiel souligné. Ils l'expliquent par la forte complexité des codes de compilation dynamique, que le processeur ne parvient pas à gérer efficacement et que seule une augmentation significative de la puissance de ce dernier permettrait d'améliorer.

## Bilan

En bilan de cette étude, deux approches sont majoritairement développées aujourd'hui dans l'optique de réduire les surcoûts induits par la compilation dynamique en temps d'exécution : **les optimisations logicielles** et **les optimisations au niveau système**. Elles permettent également de s'attaquer à un autre problème majeur, celui de l'empreinte mémoire. En étant capable de générer plus rapidement le code machine, il est possible de s'affranchir de la nécessité de conserver en mémoire celui des portions de code les moins utilisées, ne conservant que la représentation utilisée en entrée du compilateur dynamique (souvent plus compacte) et en le générant à nouveau si besoin. Concernant l'empreinte mémoire des compilateurs en eux-mêmes, ce point ne représente plus un problème majeur, une machine virtuelle complète pouvant aujourd'hui ne pas dépasser le mégaoctet. Les deux approches, bien qu'offrant des gains substantiels permettant de limiter le surcoût induit, mettent toutes en avant **la complexité des codes de compilation dynamique**. Les optimisations de la gestion mémoire limitent fortement l'intérêt de la compilation dynamique puisqu'elles cherchent au maximum à en réduire l'utilisation. Les optimisations algorithmiques, quant à elles, ont un impact non négligeable sur le volume de code et donc la complexité du compilateur, de par le développement de nouvelles librairies. L'utilisation de ressources dédiées permet une séparation des services de virtualisation, et notamment des phases de compilation, et d'exécution de l'application, limitant ainsi leur impact sur les ressources allouées à celle-ci, avec plus ou moins de flexibilité. Cette technique est basée sur l'utilisation de processeurs, spécialisés ou non, adaptés aux contraintes de l'embarqué. Toutefois, l'utilisation de ressources spécialisées telles que les processeurs Java est aujourd'hui largement abandonnée de par les problèmes de flexibilité qu'ils engendrent, la tendance étant plutôt à l'émergence de l'utilisation de ressources dédiées standardisées (processeur existant).

Pour toutes ces approches, les processeurs embarqués, qu'ils soient dédiés ou non à la compilation dynamique, ne permettent pas **de gérer la complexité de ces codes de compilation**. Il en résulte ainsi une forte limitation des gains et un surcoût induit restant significatif.

Basé sur ces conclusions, les travaux présentés dans cette thèse couvrent **la conception et le développement d'extensions matérielles aux processeurs embarqués pour la compilation dynamique** dans les systèmes embarqués, afin d'en gérer la complexité. L'objectif est d'obtenir un ensemble constitué d'un processeur et de ses extensions offrant à la fois **performances** (grâce aux extensions) et **flexibilité** (grâce au processeur). Les phases accélérées correspondent aux plus génériques et aux plus critiques des codes de compilation dynamique, les autres phases étant laissées au processeur afin de conserver cette flexibilité. Une étape préliminaire de profilage de ces codes de compilation dynamique est donc nécessaire afin de mettre en avant des signatures caractéristiques de la compilation dynamique dans l'optique de comprendre quel type d'accélérateurs sont envisageables. Nous souhaitons montrer à travers cette démarche que la complexité des codes de compilation dynamique peut être gérée et accélérée significativement à l'aide d'accélérateurs d'opérateurs simples, aux surcoûts en consommation et en surface silicium bien inférieurs à l'utilisation d'un processeur embarqué plus performant. Nous cherchons ainsi à maximiser les efficacités énergétique et surfacique des processeurs embarqués dans la gestion de ce type de codes.

## Plan du manuscrit

Cette étude s'articule autour de six grands axes présentés dans la suite de ce document. Dans un premier temps (Chapitre 1), nous proposons la réalisation d'un **état de**

**l'art de la compilation dynamique**, à travers la présentation des différentes technologies existantes. Nous nous focalisons pour cela sur les quinze dernières années, afin de couvrir l'essor important connu par la compilation dynamique durant ces années. Nous montrons, dans un premier temps, les grands principes, les principaux domaines applicatifs dans laquelle elle est utilisée et ses principales évolutions. Nous présentons ensuite une illustration des gains potentiels que permet d'obtenir la compilation dynamique à travers les différentes technologies, en se basant sur l'analyse des différents résultats de l'état de l'art sur la question. L'objectif est ainsi de réaliser un balayage des travaux existants sur le sujet, des différents domaines applicatifs visés, et de justifier l'attrait croissant pour cette approche de transfert des phases de compilation à l'exécution.

Le second axe (Chapitre 2) porte sur une analyse de **l'état de la technique de la compilation dynamique**. Pour cela nous proposons de réaliser une classification des technologies associées en fonction de leurs objectifs et des spécificités de leur flot de compilation, basés sur l'état de l'art réalisé dans le chapitre précédent. L'objectif est de mettre en avant une convergence entre toutes ces technologies dans l'optique de démontrer que la compilation dynamique peut être considérée comme un domaine algorithmique à part entière. Puisque notre étude se focalise sur la gestion de la compilation dynamique dans les systèmes embarqués, une ouverture vers ce domaine sera réalisée en fin de ce chapitre, mettant notamment en avant les optimisations existantes aujourd'hui pour accroître l'attractivité de ce domaine sur ces systèmes.

Le troisième axe (Chapitre 3) se focalise sur la présentation d'une **analyse des caractéristiques des codes de compilation dynamique**, dans l'optique de mettre en évidence des signatures caractéristiques de celle-ci à travers les différentes technologies. La complexité inhérente aux codes de compilation est aujourd'hui largement reconnue dans la littérature mais ses causes sont relativement peu explorées, ce que nous nous proposons de réaliser dans cette partie. L'objectif est de déterminer le type d'accélération envisageable dans le cadre de cette étude et de pré-identifier les principaux points potentiellement accélérables avant de passer à une analyse des portions critiques de ces codes. Pour cela, nous nous appuyons sur les conclusions du chapitre précédent ayant permis de mettre en avant la possibilité de considérer la compilation dynamique comme un domaine algorithmique à part entière. Nous réalisons au sein de ce chapitre une analyse détaillée à grain fin des différentes technologies dans l'optique d'en extraire des caractéristiques comportementales communes. Cet axe se focalise également sur l'évaluation de l'efficacité relative d'un processeur embarqué à gérer les codes de compilation dynamique, et notamment leur complexité, en comparaison avec des codes issus d'algorithmes conventionnels.

Le quatrième axe (Chapitre 4) de cette étude porte sur **l'identification des portions critiques des codes de compilation**, à partir des signatures caractéristiques (relatives aux performances) mises en avant dans l'axe précédent, et sur la mise en évidence et la justification de la mise en place d'accélérations matérielles pour l'embarqué. Cette étude se base pour cela sur le compilateur de LLVM, LLC, permettant la compilation de la représentation intermédiaire de LLVM en code machine. L'objectif de l'identification de ces portions critiques est de permettre ensuite la proposition d'optimisations adéquates de celles-ci. La deuxième partie de ce chapitre est consacrée à la justification du positionnement de cette thèse : pourquoi s'orienter vers la mise en place d'accélérations matérielles ? Nous nous basons pour cela sur l'état de l'art réalisé sur les optimisations existantes au niveau de la compilation dynamique dans l'optique d'en réduire l'impact sur les architectures embarquées. Nous en présentons les avantages et les inconvénients et nous focalisons plus spécifiquement sur leurs limites, avant de présenter notre positionnement par rapport à cette analyse.

Le cinquième axe (Chapitre 5) porte sur notre **première proposition d'accélé-**

**ration matérielle de la compilation dynamique**, concernant la **la gestion des tableaux associatifs et de l'allocation dynamique de la mémoire**. Ce choix d'accélération est la conséquence des résultats obtenus dans le quatrième axe, mettant en avant leur criticité en temps d'exécution. Pour cela, nous avons analysé les optimisations logicielles existantes sous LLC et qui sont présentées dans le quatrième axe. Nous avons ensuite proposé un affranchissement de ces optimisations logicielles en introduisant une version normalisée de LLC, ne faisant appel qu'aux librairies standards. Puis, après analyse de ces librairies, nous avons opté pour une uniformisation de l'implémentation des tableaux associatifs dans celles-ci, ces tableaux étant aussi employés pour l'allocation dynamique de la mémoire. Obtenant une base commune d'implémentation, basée sur l'utilisation des arbres rouges et noirs, nous proposons une accélération matérielle de leur gestion. Les résultats obtenus pour cet accélérateur, implémenté sous la forme d'une unité fonctionnelle dans notre cadre d'étude, en font un élément prometteur pour une éventuelle intégration réelle sur processeur, pouvant bénéficier non seulement aux codes de compilation dynamique, mais aussi à tous les codes utilisant massivement les tableaux associatifs.

Le sixième axe (Chapitre 6) présente **une proposition de concept pour une seconde accélération matérielle de la compilation dynamique**, portant sur **la gestion du graphe des instructions de l'application à compiler**, au niveau de sa représentation intermédiaire et de son code machine. Ici encore, ce choix d'accélération est la conséquence des résultats obtenus dans le quatrième axe, mettant en avant la criticité de la gestion de ce graphe quant à son temps d'exécution. Les travaux d'expérimentations sur le sujet, en cours au moment de la rédaction de ce manuscrit, font apparaître un gain potentiel limité de la solution mise en place. Toutefois, nous mettons également en avant une influence positive remarquable de cette solution sur la pollution du cache d'instructions engendrée par le codes de compilation dynamique.

Enfin, nous proposons de terminer cette étude par une **conclusion de l'ensemble des travaux réalisés**, récapitulant les résultats principaux mis en avant dans l'ensemble de la démarche. Nous ouvrons ensuite une discussion autour de ces résultats, afin de mettre en avant les forces et les faiblesses de l'étude face aux approches déjà existantes. Enfin, une analyse des perspectives potentielles est engagée, présentant les opportunités à court, moyen et long termes engendrées par ces travaux.

# Chapitre 1

# État de l'art de la compilation dynamique

**Sommaire**



## 1.1 Introduction

Nous nous proposons dans ce chapitre de réaliser **un état de l'art de la compilation dynamique**. Nous en montrerons dans un premier temps les grands principes, les principaux domaines applicatifs et les principales évolutions. Nous présenterons ensuite **une illustration des gains potentiels** que permet d'obtenir la compilation dynamique à travers les différentes technologies, en se basant sur l'analyse des différents résultats de l'état de l'art sur la question. L'objectif est ainsi de réaliser un balayage des travaux existants sur le sujet, des différents domaines applicatifs visés, et de justifier l'attrait croissant pour cette approche de transfert des phases de compilation à l'exécution.

## 1.2 Introduction à la compilation dynamique

### 1.2.1 Introduction à la notion de la compilation

**La définition simplifiée de la compilation** telle que posée par Aho et al. dans [25] est la suivante : *"Un compilateur est un programme qui lit un autre programme, rédigé dans un langage de programmation, le langage source, et qui le traduit en un programme équivalent rédigé dans un autre langage, le langage cible."*.

Poussée par l'émergence de nouveaux langages de programmation permettant de s'affranchir de l'écriture des programmes sous la forme de code assembleur (C, FORTRAN), la première apparition de la notion de compilation remonte au début des années 50 avec l'apparition du premier compilateur, le A-0 System, développé par Grace Hopper et ciblant l'UNIVAC 1 [26], considéré comme le premier ordinateur commercial. Les fonctionnalités de ce compilateur sont limitées à la gestion des étapes de chargement et de raccordement du code à exécuter. Il faut attendre 1957 et les travaux de John Backus et de l'équipe FORTRAN de IBM pour voir apparaître le premier compilateur complet, offrant la possibilité de compiler puis d'exécuter directement les programmes écrit sous la forme d'un langage de programmation (FORTRAN en l'occurrence) sans utiliser la représentation assembleur.

Par la suite, les compilateurs ont continué à se développer pour devenir toujours plus performants, aussi bien en temps de compilation, qu'en capacité d'optimisation ou qu'en capacité à supporter un éventail varié de cibles, avec l'apparition notamment de la notion de cross-compilation, où le processeur cible (celui pour lequel le programme est compilé), diffère du processeur hôte (celui sur lequel le compilateur est exécuté). Le plus connu d'entre eux, *GNU Compiler Collection* (GCC) [27], est apparu en 1987 et fait office de référence, permettant de compiler un large panel de langages de programmation (C/C++, Objective-C, etc.) et supportant la plupart des cibles et des systèmes d'exploitation existants. Les capacités d'optimisation de ce compilateur ont été fortement développées, aussi bien au niveau de sa compréhension des applications et des langages de programmation, qu'au niveau des spécificités propres à l'architecture cible pour laquelle le programme est compilé (pas moins de deux cent cinquante phases de compilation sont environ réalisées par GCC lors de la compilation d'un programme).

Qu'il s'agisse de GCC ou d'un autre compilateur, leur structure reste proche, telle que présentée sur la figure 1.1. Celle-ci se compose de trois principaux étages : **la partie frontale (*front-end*)**, **la partie intermédiaire (*middle-end*)** et **la partie arrière (*back-end*)**. La partie frontale est totalement indépendante de la cible et dépend uniquement du code source en entrée. La partie intermédiaire est indépendante au sens strict du code source et de la cible pour faciliter la portabilité du compilateur. Elle peut ainsi s'adapter par exemple à différents codes machines. Toutefois, nous verrons que les architectures ciblées par le compilateur ont une influence notable sur cette partie, justifiant les choix technologiques qui y sont faits (représentation type machine à registres ou machine à pile, etc.). Les transformations dépendantes de la cible, au niveau du code à compiler, sont cloisonnées dans la partie arrière du compilateur, seule celle-ci est donc théoriquement à modifier d'une cible à une autre.

Le rôle de **la partie frontale** est d'analyser le code source en entrée du compilateur, au niveau lexical, syntaxique et sémantique, afin d'en assurer la validité pour le compilateur, tel que présenté sur la figure 1.1. Cette partie n'a pas été considérée dans le cadre de la thèse, car nous nous focalisons sur les étapes plus proches de la cibles, propres aux parties intermédiaire et arrière.

**La partie intermédiaire** constitue, quant à elle, le coeur même du compilateur. C'est à ce niveau que les optimisations les plus importantes vont être réalisées sur le code (déroulage de boucle, mise en ligne des portions de code, réduction de boucles, etc.). Afin d'accélérer la réalisation de ces optimisations et leur efficacité, le code est au préalable traduit dans une forme appelée représentation intermédiaire. Cette représentation est optimisée pour en faciliter la manipulation et est conçue pour permettre d'y réaliser les optimisations les plus importantes. Elle offre un excellent compromis entre le code source et le code machine, liant les spécificités de chacune des représentations. Elle permet également d'assurer une indépendance entre les deux parties pour des questions de portabilité. Il n'y a ainsi aucune communication directe entre la partie frontale et

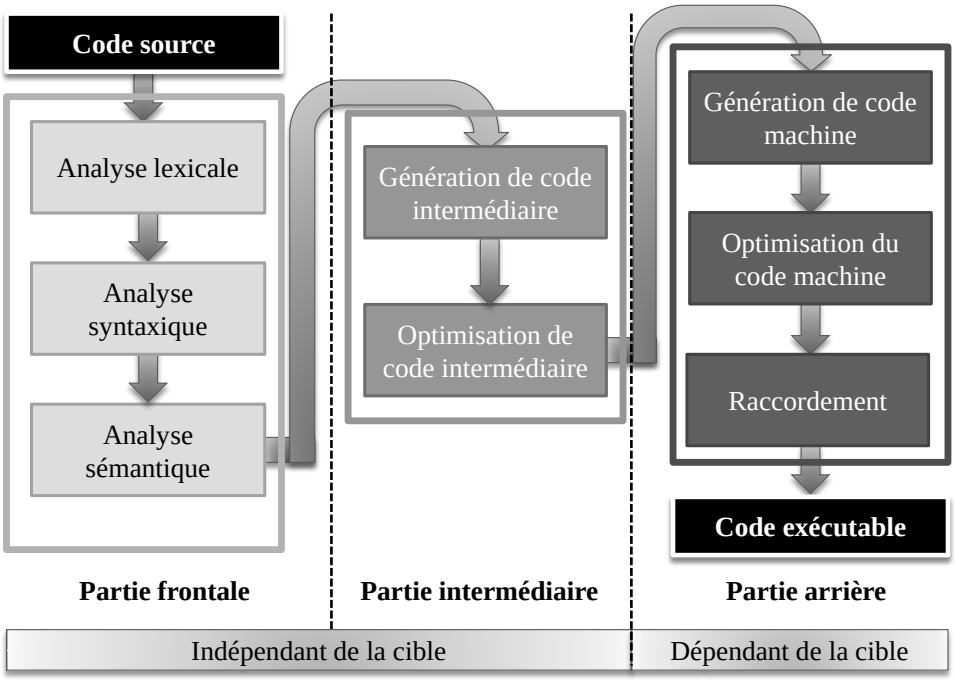



FIGURE 1.1 – Structure d'un compilateur avec mise en avant des trois principaux étages d'un compilateur et séparation des parties indépendantes et dépendantes de la cible.

la partie arrière, le tout passant systématiquement par la partie intermédiaire. Les caractéristiques des représentations intermédiaires sont plus amplement présentées dans la suite du manuscrit.

La représentation optimisée, obtenue en sortie de la partie intermédiaire, est transmise à **la partie arrière** du compilateur. Celle-ci est en charge de la traduction de cette représentation sous forme de code machine, compréhensible par la cible pour laquelle le programme est compilé. Ce code machine est ensuite optimisé en fonction des spécificités de la cible (allocation de registres, etc.), avant d'être raccordé en mémoire afin d'être exécuté par le processeur.

### 1.2.2 Introduction à la notion de compilation dynamique

Il convient de distinguer deux types de compilateurs : **les compilateurs statiques** et **les compilateurs dynamiques** (ou compilateurs à la volée). Ces premiers, à l'instar de GCC, compilent le programme avant son exécution, on parle alors de compilation hors-ligne. Les seconds, sur lesquels se focalise l'étude menée dans le cadre de cette thèse, permettent de compiler certaines portions d'une application durant son exécution, sans l'interrompre. La compilation dynamique peut donc être vue comme un transfert des phases de compilation au sein de l'exécution du programme.

Les premiers travaux publiés sur le sujet remontent au début des années 60 et ont porté sur le langage LISP [28]. L'idée est de compiler dynamiquement en langage machine les fonctions appelées récursivement, permettant notamment de les mettre en ligne et de bénéficier de la connaissance de leurs paramètres d'appels, tout en étant suffisamment rapide pour s'affranchir du besoin de garder en mémoire le code machine obtenu en sortie du compilateur. Les projets se sont succédés dans les années qui ont suivi avec par exemple le développement du langage de programmation Self (1987 [29]), de Erlang (1997 et 2000 [30, 31]), ou encore le développement de modèles de code (1999 [32]) et de solutions de spécialisation de code (1998 [33]). Ces développements ont atteint leur paroxysme durant les 15 dernières années, aussi bien dans l'industrie

que dans la recherche académique.

Cet essor considérable de la compilation dynamique résulte de trois principaux aspects, relatifs aux **performances**, au **déploiement des applications** sur les architectures, et à leur **la portabilité**.

L'aspect **performances** est notamment lié au dynamisme croissant des applications. Certaines d'entre elles, comme les applications de vision, sont de plus en plus sensibles au jeu de données comme évoqué en introduction. Les applications sont également de plus en plus interactives entre elles, échangeant toujours plus d'informations les unes avec les autres. Toutes ces raisons engendrent une importante dépendance au contexte d'exécution pour les applications. Les optimisations réalisées hors-ligne par les compilateurs dynamiques sur ces applications sont donc de plus en plus limitées. La compilation dynamique est vite apparue comme une solution efficace pour adresser cette dépendance croissante [34, 35, 36].

L'aspect **déploiement d'applications** est lié à la complexité croissante des architectures modernes. Celles-ci sont en effet composées de plus en plus de ressources de calcul (systèmes multi-processeurs) avec une hétérogénéité croissante (association de coeurs généralistes avec des coeurs graphiques ou de traitement de signal par exemple). Les applications doivent donc pouvoir être déployées sur ces architectures de manière optimale, afin d'exploiter au mieux les ressources et s'adapter à l'évolution de l'environnement dans lequel elles s'exécutent. Les optimisations statiques pour le déploiement des applications ne suffisent plus à satisfaire les contraintes grandissantes en performances, et la compilation dynamique s'est rapidement imposée comme un moyen efficace de gérer l'adéquation algorithme/architecture (avec par exemple le développement de nouvelles approches comme OpenCL [37, 38]).

L'aspect **portabilité** est lié au fait que les architectures et les logiciels bas-niveau (systèmes d'exploitation, librairies) évoluent aujourd'hui plus rapidement que les applications s'exécutant dessus. A cela s'ajoute aussi la forte diversité au niveau de ces architectures pour les raisons précédemment évoquées. De plus, la complexité croissante des applications a conduit à l'apparition de nouveaux langages de programmation haut-niveau, tels que Java ou les langages de scripts comme JavaScript, Python ou Matlab. Pour permettre de passer rapidement d'une architecture à une autre sans devoir adapter toute la chaîne de compilation et gérer ces langages à haut-niveau d'abstraction, les concepteurs de couches bas-niveau ont développé de nouvelles solutions de virtualisations, offrant un étage intermédiaire entre le logiciel et l'architecture afin d'en assurer l'indépendance. Basées initialement sur de l'interprétation de code, consistant en une traduction directe en code machine du code de l'application, ces solutions utilisent massivement aujourd'hui les compilateurs dynamique afin de satisfaire des contraintes en performances toujours plus importantes grâce aux optimisations réalisées.

## 1.3 Présentation des différentes technologies de compilation dynamique

L'état de l'art réalisé dans le cadre de cette thèse se focalise essentiellement sur l'essor de la compilation dynamique durant ces quinze dernières années. Une étude réalisée par John Aycock [39] en 2003 présente l'évolution de la compilation dynamique de ses débuts en 1960 jusqu'à la fin des années 90. Quatre grandes technologies de compilation dynamique ont été identifiées : les machines virtuelles, les traducteurs dynamiques binaires, les compilateurs dynamiques multi-étages et moteurs d'exécution pour les langages typés dynamiquement.

### 1.3.1 Machines virtuelles

**Les machines virtuelles**, et en particulier les **machines virtuelles Java** [40, 41, 42, 5, 6], ont été le premier déclencheur de l'essor de l'utilisation de la compilation dynamique à la fin des années 90. Basées initialement sur la seule interprétation de code, le développement d'applications aux contraintes grandissantes en performances et au potentiel croissant d'optimisation a conduit au développement de solutions permettant de compiler dynamiquement le code à octet en entrée en code machine optimisé [43, 44, 45, 46, 45, 47]. L'objectif d'une machine virtuelle est de fournir au développeur un environnement d'exécution transparent permettant de porter facilement son application d'une cible à l'autre. C'est la machine virtuelle qui se charge de gérer l'interfaçage avec la cible afin d'exécuter l'application. Cette approche a permis l'émergence de nouveaux langages à haut-niveau d'abstraction, tels que Java.

Les chercheurs ont focalisé leurs travaux sur l'amélioration de l'efficacité des machines virtuelles, en particulier les *Java Virtual Machines* (JVM), en raison des besoins croissants en performances pour les applications utilisant ces langages. Ceux-ci apparaissent aujourd'hui comme une alternative intéressante pour la virtualisation et le développement d'applications indépendantes de la cible.

La structure classique d'une machine virtuelle est présentée sur la figure 1.2. Le code à octet est statiquement généré à partir du code source de l'application. C'est ce code à octet qui constitue la représentation intermédiaire manipulée par la machine virtuelle. Dans les premières phases d'exécution, et comme indiqué sur la figure 1.2, le code est directement interprété par la machine virtuelle et exécuté à la volée sur la cible. Par profilage, les portions les plus critiques de ce code sont détectées. Si un seuil statiquement défini est franchi, la portion de code est compilée dynamiquement dans le code machine de la cible, bénéficiant des optimisations du compilateur et s'exécutant ainsi beaucoup plus rapidement que la version interprétée. L'objectif est ainsi de réduire le temps d'exécution global de l'application. Le ramasse-miettes figurant sur la figure 1.2 permet la gestion mémoire de l'application et du code généré, en lien avec la mémoire physique de la cible.

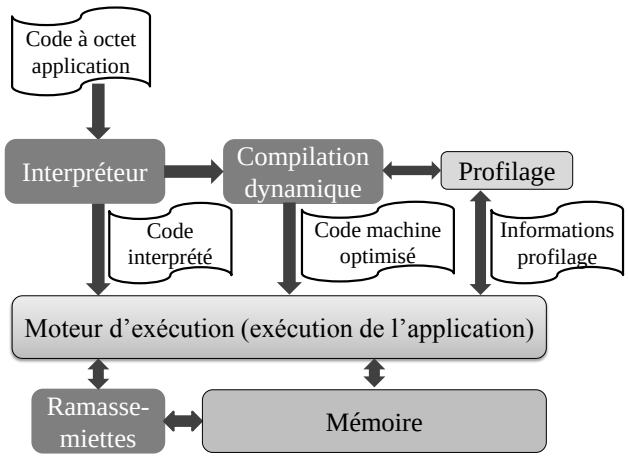



FIGURE 1.2 – Structure classique d'une machine virtuelle (exemple Java).

La compilation dynamique induit toutefois un surcoût en performances, lié à l'exécution des phases de compilation, ainsi qu'au niveau de l'empreinte mémoire nécessaire à la compilation dynamique, incluant les codes à octet conservés en mémoire auxquels s'ajoutent les informations nécessaires au compilateur pour réaliser la compilation (la taille des compilateurs étant aujourd'hui quasiment négligeable). Nous verrons par la suite que le code à octet Java est extrêmement compact, mais, ayant été développé

initialement pour l'interprétation, celui-ci est extrêmement implicite et nécessite donc l'ajout d'informations nécessaires à la compilation, augmentant cette empreinte mémoire.

Même si aujourd'hui les études concernent majoritairement les machines virtuelles Java, d'autres types de machines virtuelles ont été développés, comme la machine virtuelle Smalltalk [48] pour les langages SELF ou Erlang dans les années 80/90. La première notion de machine virtuelle est apparue avec les expérimentations d'IBM au milieu des années 60 avec le développement du projet IBM 360/64 [4].

Les **machines virtuelles basées sur CLI** [49, 50], utilisant la représentation intermédiaire standardisée CIL (code à octet), sont un autre exemple de machines virtuelles actuellement développées. Le principal avantage de l'environnement CLI réside dans sa capacité à supporter différents langages de programmation en entrée (C#, Python, Java), contrairement aux JVM qui ne supportent que les applications Java.

Enfin, il est également possible de citer **le développement de nombreux cadriciels** (*framework*) permettant de faciliter le développement de nouvelles machines virtuelles, comme Open Runtime Platform [51] ou encore VMKit [17]. Cette dernière, basée sur le cadriciel de compilation LLVM [8], permet de cibler un plus large panel de langages que CLI ou les JVM, allant des langages à haut-niveau d'abstraction (Python, JavaScript, etc.) aux langages bas-niveau (C, C++) grâce notamment aux caractéristiques de la représentation intermédiaire utilisée dans LLVM, sur laquelle nous nous étendrons plus longuement dans la suite de ce manuscrit.

### 1.3.2 Traduction dynamique binaire

Bien que les solutions de compilation dynamique pour les machines virtuelles constituent aujourd'hui son point principal de développement, ce n'est pas son seul domaine d'utilisation. **La traduction dynamique binaire** est elle aussi un domaine majeur de recherche sur la compilation dynamique. Consistant à traduire un code machine pour une cible en un code machine pour une autre cible, le développement de cette technologie s'est déroulé en deux étapes.

La **première étape**, et la plus ancienne, a consisté en la traduction directe dans un nouveau code machine de l'ancien code machine, sans représentation intermédiaire et sans optimisation, ce qui la rend assimilable à **une méta-interprétation**, incluant les phases d'interprétation au milieu de l'exécution du programme (phases mises en ligne dans le code applicatif). De nombreux projets ont été développés sur la base de cette solution, avec notamment la mise en place de solutions matérielles dédiées.

Au niveau académique on peut notamment citer FX!32 [52], qui utilise la traduction binaire pour exécuter des programmes x86 win32 sur des cibles Alpha, ou encore Daisy [53], qui utilise la traduction binaire dans un but d'interopérabilité entre processeurs en convertissant du code machine PowerPC en code machine VLIW. Côté industriel on peut citer Transmeta Crusoe [54], qui présente les mêmes objectifs que Daisy mais pour une traduction de code machine x86 vers VLIW. Une présentation et une comparaison des deux projets ont été réalisées par Brian Remick [55].

La **seconde étape**, dont l'apparition est liée à l'essor de la compilation dynamique durant les 15 dernières années, consiste en l'utilisation d'une représentation intermédiaire dans l'optique **d'optimiser le code** avant de le traduire dans le nouveau code machine. Les exemples les plus connus sont Qemu [56] et Apple Rosetta [57].

Qemu permet l'émulation de différents processeurs sur plusieurs processeurs cibles (par exemple l'émulation d'un processeur ARM sur un processeur x86). Développé dans un premier temps comme un simple interpréteur, les contraintes grandissantes en performances et en portabilité ont conduit à l'ajout d'une représentation intermédiaire interne permettant d'effectuer des optimisations durant la traduction, en utilisant la

compilation dynamique. Outre le gain en performances, cette représentation intermédiaire permet également de standardiser les étapes intermédiaires d'optimisation et de traduction et de les rendre indépendantes (pas de lien direct entre le code machine en entrée et en sortie), permettant d'offrir une plus grande portabilité de l'outil.

Rosetta, développé par Transitive pour Apple, a permis la réutilisation des applications compilées pour PowerPC sur les nouvelles architectures Intel adoptées par Apple au début des années 2000. L'objectif a été d'éviter aux développeurs de logiciel Apple d'avoir à redévelopper et recompiler leurs applications, tout en offrant transparence et performances aux utilisateurs. Utilisé jusqu'en 2011, Rosetta s'est basé là encore sur l'utilisation d'une représentation intermédiaire pour pouvoir optimiser le code à traduire.

Une approche alternative développée en parallèle de celle-ci est l'approche dite **d'optimisation dynamique binaire**. L'objectif est de traduire un code machine pour une cible donnée en un nouveau code machine pour cette même cible, permettant de réaliser un certain nombre d'optimisations sur le code grâce aux informations disponibles à l'exécution. Cette approche est en fait, dans son principe, un mélange entre la vision de la traduction dynamique binaire et celle de la compilation dynamique multi-étages. L'objectif est d'utiliser cette première dans une optique de performances propre à la compilation dynamique multi-étages, que nous détaillerons dans la prochaine partie, mais à beaucoup plus bas niveau puisque se situant au niveau du code machine.

C'est sur ce principe que fonctionnent par exemple Dynamo [58], et plus récemment DynamoRIO [59, 60], dont l'objectif est de profiler, instrumenter, optimiser et traduire les portions critiques de code d'une application, le tout dynamiquement et en optimisant le profilage afin de rendre celui-ci transparent au niveau de son impact sur les performances (modifications ou non des instructions de l'application par exemple pour empêcher d'éventuels sauts dans le programme introduits par le profilage). Il est également possible de citer Valgrind [61] dont le compilateur dynamique permet de compiler le code machine en l'instrumentant pour le profiler, tout en limitant le surcoût induit par le profilage qui se trouvera optimisé par le compilateur car introduit au niveau de sa représentation intermédiaire interne.

Une étude importante réalisée dans cette optique d'optimisation dynamique binaire concerne les travaux réalisés par Henri-Pierre Charles et Damien Couroussé sur l'outil deGoal [62], introduisant la notion de "compilettes". Cette étude part du constat **d'un goulot d'étranglement sémantique croissant entre le domaine applicatif et les architectures sur lesquelles tournent ces applications**. Un nouveau domaine applicatif se traduit en effet le plus souvent par la nécessité de nouvelles modifications architecturales, comme par exemple le développement d'unités de vectorisation flottante pour l'algèbre linaire (MMX, SSE), ou encore pour les instructions spécialisées SAD (*Sum of Absolute Differences*) pour la compression vidéo. Le problème majeur réside dans les difficultés d'évolution des compilateurs pour cibler ces nouveaux domaines applicatifs sur ces modifications architecturales, de par les forts coûts engendrés en développement et la faible attractivité pour les développeurs à modifier ces compilateurs. Cela oblige les développeurs d'applications à incorporer manuellement les optimisations potentielles directement au sein de leur code (à un niveau d'abstraction plus élevé que le code machine, comme par exemple au niveau du code C). Ils procèdent pour cela par exemple par utilisation d'outils de versionnage de code, de techniques de métaprogrammation ou encore de code auto-adaptatif.

C'est pourquoi Henri-Pierre Charles propose dans [62] la mise en place d'une nouvelle façon d'optimiser le code en mettant directement en lien le niveau algorithmique de l'application et les spécificités matérielles de la cible considérée. Cela se fait par l'intermédiaire de "compilettes" générées à partir de l'outil deGoal. Une compilette est

un petit générateur de code directement inclu au sein de l'application permettant de générer dynamiquement, à l'exécution, des portions de code machine optimisées en tirant profit des informations disponibles à ce moment là sur leur jeu de données ou sur l'environnement d'exécution. Ces générateurs de code sont suffisamment performants pour générer le code de manière transparente, juste avant un appel à une fonction par exemple. Les compilettes se présentent, comme la représentation intermédiaire LLVM, sous la forme d'un jeu d'instructions neutre de processeur RISC, mais, contrairement à LLVM, elles ciblent également des instructions plus spécifiques comme les instructions vectorielles ou multimédia.

Une autre application de plus en plus répandue de la traduction dynamique binaire avec compilation dynamique est son utilisation dans le cadre de **l'amélioration des performances pour la simulation**. Initialement basée sur l'interprétation, les besoins croissants en performances ont conduit les développeurs de simulateurs à introduire des solutions de compilation dynamique, dans l'optique d'accroître les performances à l'exécution des codes à simuler. Ces besoins croissants s'expliquent notamment par la complexité grandissante des architectures à simuler, poussée par l'émergence de nouvelles architectures massivement parallèles [63, 64, 65].

En résumé, il est possible de voir le lien étroit entre l'évolution de la traduction dynamique binaire et celle de la compilation dynamique. L'approche initiale, permettant d'interpréter directement le code machine et de le traduire à la volée dans le nouveau code machine, permet une génération rapide du nouveau code machine au dépend de la qualité de celui-ci et par conséquent de son temps d'exécution. Cela complique également la portabilité du traducteur puisqu'il n'existe aucune indépendance entre le code machine en entrée et le code en sortie, nécessitant ainsi de redévelopper l'ensemble de la chaîne. Le développement de nouvelles solutions basées sur l'utilisation de la compilation dynamique a permis l'utilisation d'une représentation intermédiaire interne au compilateur sur laquelle sont réalisées les optimisations du code à traduire. Outre l'accroissement de la portabilité, cela permet également, grâce à ces optimisations, d'accroître la qualité du code généré.

### 1.3.3 Compilation dynamique multi-étages

Comme précédemment mis en avant, un flot de compilation standard (code source vers code machine) peut être le plus souvent décomposé en cinq phases principales si l'on omet la partie frontale : la génération du code intermédiaire, l'optimisation de ce code, la génération du code machine, son optimisation et son raccordement en mémoire, réalisées hors-ligne par les compilateurs statiques. L'objectif de la **compilation dynamique multi-étages** est de proposer des solutions permettant de déporter ces phases durant l'exécution dans l'optique d'en accroître l'efficacité.

L'exemple le plus connu est sans doute **le cadriciel LLVM** [8], pour *Low-Level Virtual Machine*, développé par Chris Lattner depuis 2002. Il est notamment utilisé par Apple sur MAC OS X pour les applications OpenGL et dans Nitro, le moteur d'exécution Apple pour JavaScript. Le compilateur de LLVM est également aujourd'hui utilisé comme brique de base pour le déploiement d'applications via OpenCL sous ARM [19], Apple [20] et NVidia PTX [21]. Un nombre important de projets, aussi bien académiques (VMKit, VaporSIMD, etc.) qu'industriels, utilisent aujourd'hui ce cadriciel, dont le **générateur de code LLC** (*Low-Level Compiler*) et la représentation intermédiaire sont reconnus pour leurs performances [66].

Un point important sur lequel nous reviendront dans la suite de ce manuscrit est l'importance de distinguer le générateur de code LLC, du moteur de compilation dynamique mis en place expérimentalement dans le cadriciel. Ce dernier est en effet à l'état de prototype aux performances relativement peu attractives et reste indisponible

sur de nombreuses cibles (il n'est notamment pas complètement fonctionnel sur ARM). Son rôle est essentiellement de proposer une illustration de la possibilité d'utiliser dynamiquement le compilateur LLC de LLVM. Il s'agit tout simplement d'une surcouche faisant appel aux différentes routines du générateur de code LLC. C'est pourquoi les utilisateurs de LLVM développent aujourd'hui leur propre moteur de compilation dynamique optimisé en se basant sur ce générateur de code.

Un autre exemple de compilateur de ce type est le compilateur du cadriciel Ildjit [50], développé depuis 2008 par Campanoni et al., se basant sur la représentation intermédiaire CIL et sur la librairie de compilation dynamique LibJIT [67]. Il propose notamment l'exploitation des ressources inutilisées sur les architectures multi-coeurs afin de compiler dynamiquement les portions de code. Pour cela, il analyse en avance de phase l'appel des différentes fonctions dans l'application et leur criticité, ce qui permet de jouer sur le niveau d'optimisation désiré.

Il est également intéressant de noter l'existence d'un projet plus récent, fruit d'une collaboration européenne entre membres du réseau HIPEAC, nommé VaporSIMD [68]. Ce projet, basé sur LLVM, propose la possibilité de déployer automatiquement, dynamiquement et avec des performances proches de celles d'une compilation statique, des applications sur différentes architectures SIMD. De grandes opportunités d'exploitation des informations de vectorisation, disponibles à l'exécution, sont envisageables par l'intermédiaire de LLVM, et par conséquent un fort potentiel d'optimisation existe à ce niveau. Ainsi, alors que seul l'aspect portabilité/virtualisation a été exploité jusqu'à présent pour ce projet, nous pensons que des gains significatifs en performances sont possibles grâce aux choix technologiques effectués. Il est par exemple envisageable de vectoriser des portions de code que le compilateur n'a pas pu déceler statiquement, en fonction de l'évolution du jeu de données, permettant de tirer parti des extensions vectorielles des processeurs. Les cibles concernées sont les suivantes, avec leurs extensions vectorielles respectives : x86 + SSE/AVX, ARM + NEON, PPC + AltiVec.

Enfin, il est également possible de citer le projet de développement de 'C et tcc par Poletto et al. [69], proposant la mise en place d'un langage et d'un compilateur dans l'optique d'une génération dynamique de code pour des questions de performances. L'objectif est d'accroître la qualité du code généré en lui faisant bénéficier des informations disponibles à l'exécution.

Le déport des phases de compilation à l'exécution se réalise le plus souvent de la manière suivante : la génération de la représentation intermédiaire à partir du code source est réalisée hors-ligne. Puis, les optimisations au niveau de cette représentation, la génération du code machine et les optimisations au niveau de ce code sont réalisées durant l'exécution, prenant en considération les informations supplémentaires disponibles à ce moment là. Les optimisations possibles sont ainsi plus importantes que dans le cadre d'une optimisation hors-ligne (déroulage de boucle, évaluation de conditions de branchement, allocation de registres optimisée, etc.).

Utilisant un principe similaire à celui de la compilation dynamique multi-étages, il est également possible de noter l'existence d'approches basées sur la génération statique de motifs complétés à l'exécution. L'objectif est ainsi de tirer bénéfice des informations disponibles pour optimiser l'application considérée en utilisant des représentations intermédiaires et des générateurs de code spécialisés pour une série d'optimisations ciblées.

Parmi ces approches, Jimborean et al. [70] propose la mise en place d'un analyseur dynamique de dépendances. Celui-ci permet de réaliser la parallélisation spéculative de boucles. Cet analyseur est intégré dans le cadriciel VMAD [71], lui-même basé sur LLVM, qui permet la génération statique de squelettes de code binaire qui sont complétés à l'exécution en fonction de l'optimisation recherchée. VMAD est conçu pour

appliquer des transformations polyédriques à l'exécution afin de réaliser l'optimisation spéculative de boucles contenues au sein du code d'une application. Ces choix spéculatifs sont orientés par la réalisation de phases de profilage, basées sur un procédé d'instrumentation par échantillonnage. Ce choix permet l'exécution en alternance de boucles instrumentées et non-instrumentées, dans l'optique de limiter le surcoût introduit.

Il est possible de citer aussi l'approche retenue dans LiquidSIMD [72]. Afin de faciliter l'adaptation des applications aux différentes architectures SIMD existantes, les auteurs préconisent la mise en place d'une solution permettant d'identifier et de traduire rapidement à l'exécution les zones accélérables par ces extensions. Pour cela, les instructions SIMD initiales du code binaire sont transformées en une représentation intermédiaire basée sur des instructions scalaires. Cette transformation est effectuée selon une liste de règles permettant sa normalisation. Les instructions SIMD qui n'ont notamment pas d'équivalent direct en instructions scalaires sont transformées en un ensemble de ces instructions. Une attention toute particulière est portée aux instructions engendrant des permutations et des réorganisations au sein des vecteurs. Les instructions scalaires sont ensuite reconnues à l'exécution par un traducteur dynamique matériel (machine à états finis), grâce à l'application des règles précédemment établies, et transformées en fonction de l'unité SIMD de la cible.

Nous verrons dans le prochain chapitre que l'objectif de la compilation dynamique multi-étages est clairement de tirer bénéfice de la compilation dynamique pour accroître les performances en comparaison avec une compilation statique, contrairement aux autres approches qui tentent de l'utiliser pour compenser les pertes liées à la lenteur de l'interpréteur initialement introduit pour la virtualisation. Toutefois, une tendance apparaissant ces dernières années, liée à la demande grandissante en performances des solutions de virtualisation, est l'utilisation de cette compilation multi-étages dans ces solutions de virtualisation, afin d'en accroître les performances et l'attractivité (avec par exemple l'utilisation de LLVM dans des projets comme VMKit).

L'intérêt de ces **compilateurs optimisants** est leur forte capacité à analyser et à pouvoir faire bénéficier à l'application des optimisations disponibles à l'exécution dans le cadre d'une compilation dynamique. Ces compilateurs, comme Ildjit et LLVM, proposent également l'utilisation de solutions portables, facilitant leur déploiement sur de nouvelles cibles. C'est pourquoi ils peuvent bénéficier aujourd'hui à l'ensemble des technologies évoquées dans ce manuscrit. LLVM par exemple est aujourd'hui autant employé dans des projets relatifs à l'obtention de performances brutes, comme avec OpenCL pour le déploiement d'applications ou OpenGL sur MAC OS X, que dans des projets de virtualisation, comme par exemple les travaux réalisés par Ost et al. [73] dans le cadre de la gestion d'architectures hétérogènes ou encore ceux menés pour la réalisation de VMKit.

### 1.3.4 Moteurs d'exécution pour les langages typés dynamiquement

Cette dernière catégorie de technologie de compilation dynamique identifiée a connu un essor majeur ces 10 dernières années. C'est la conséquence d'une **demande croissante en vitesse d'exécution pour la gestion des langages typés dynamiquement à haut niveau d'abstraction** comme les langages de script tels que JavaScript [10, 9, 74, 75], Python [11] ou encore Matlab [12].

Ces langages, de part leur dynamisme, notamment au niveau des types des variables manipulées, sont impossibles à compiler et à optimiser statiquement. Les informations nécessaires ne sont en effet connues que durant l'exécution du programme. Alors que ces langages ont été essentiellement interprétés jusqu'au début des années 2000, l'essor de la compilation dynamique et l'explosion du développement des applications faisant appel à ces langages (les applications web utilisent aujourd'hui massivement le JavaScript par

exemple), ont conduit au développement d'environnements d'exécution, basés sur des techniques de virtualisation, compilant dynamiquement les applications en code machine optimisé dans l'optique d'en réduire les temps d'exécution. Les quatre principaux navigateurs (Chrome, Firefox, Safari et Internet Explorer) sont en forte concurrence sur le développement de solutions permettant d'accélérer l'exécution des applications JavaScript.

Pour mieux comprendre le fonctionnement de ce type de solution, il est possible d'effectuer une analogie avec la solution présentée précédemment sur la compilation dynamique multi-étages. Au lieu de déporter les parties intermédiaire et arrière, c'est cette fois-ci la totalité de la chaîne de compilation considérée qui se trouve déportée durant l'exécution, incluant la partie génération de la représentation intermédiaire à partir du code source. La partie frontale est quant à elle toujours réalisée statiquement lors de l'écriture du programme par le développeur.

Un des principaux problèmes relatif à ces langages est la détermination des différents types des variables manipulées, connus lors de l'exécution. Cette étape est effectuée dynamiquement en analysant les précédents types utilisés par les variables et en réalisant une prédiction grâce à ce profilage. Les études réalisées montrent que cette approche permet de prédire les types avec une forte efficacité, engendrant peu d'erreurs à ce niveau. Les pertes en performances induites par ces rares erreurs sont compensées par les gains obtenus. Un autre problème se pose également concernant la sécurité des moteurs d'exécution JavaScript utilisant la compilation dynamique. Ce langage n'étant pas prévu pour la compilation, il ne permet pas de se protéger efficacement de codes malicieux, essayant par exemple d'accéder à des zones mémoire non autorisées. Le compilateur doit donc prendre en compte cet aspect lors de sa conception [76].

Contrairement à Google V8 et jusqu'à il y a peu Firefox SpiderMonkey (respectivement les moteurs JavaScript de Google Chrome et de Firefox), incluant un compilateur dynamique traduisant directement le JavaScript en code machine sans utiliser explicitement une représentation intermédiaire réutilisable (au moment de la rédaction de ce manuscrit), Microsoft a développé SPUR, un moteur JavaScript utilisant le CIL comme représentation intermédiaire. Les développeurs montrent dans [10] l'impact négligeable de cette phase intermédiaire pour laquelle le surcoût induit par la traduction est compensé par les optimisations réalisées sur le CIL et par la facilité de traduction de celui-ci en code machine. Un des atouts majeur de ce projet est la possibilité de réutiliser les parties intermédiaires et arrières de ce compilateur pour l'utiliser avec d'autres codes sources que le JavaScript.

L'intérêt croissant pour l'utilisation des représentations intermédiaires a également poussé la communauté Firefox à remplacer SpiderMonkey par IonMonkey [77], utilisant lui aussi une représentation intermédiaire (NanoJIT [78]) au sein de laquelle un certain nombre d'optimisations, comme l'élimination de code mort ou la réduction de boucle, sont effectuées. Les résultats présentés montrent un gain de 26 % entre Firefox 17 et Firefox 18, incluant IonMonkey, sur la suite de tests Kraken. Concernant le moteur JavaScript utilisé par Apple sous Safari, nommé Nitro, il semble que celui-ci utilise le cadriciel LLVM comme support, signifiant là aussi l'utilisation d'une représentation intermédiaire.

### 1.3.5 La question du profilage

Dans toutes ces technologies, **l'intérêt du profilage** est clairement mis en avant pour la compilation dynamique. Instrumenter et analyser le code, détecter et sélectionner les portions critiques sont des points cruciaux pour déterminer **quand**, **quelle portion** et à **quelle granularité** il convient de compiler et d'optimiser dynamiquement l'application exécutée. Les solutions sans profilage obligent à compiler dynamiquement

des portions complètes de codes de manière périodique et sans analyse. Cela peut conduire à la compilation de portions de code ne tirant aucun bénéfice de la compilation dynamique et n'offrant pas le gain nécessaire à la compensation du surcoût induit. De nombreuses études ont été réalisées pour justifier de l'intérêt du profilage dans le cadre de la compilation dynamique [79, 80, 81]. D'autres ont permis la proposition de solutions toujours plus performantes pour permettre d'instrumenter et de profiler dynamiquement l'application dans l'optique d'optimiser le code de l'application.

C'est le cas des travaux menés par Jimborean et al. [71] proposant la mise en place d'une machine virtuelle pour l'analyse dynamique avancée (VMAD), se basant notamment sur une extension du cadriciel LLVM et de son compilateur en travaillant sur la représentation intermédiaire de celui-ci. VMAD n'utilise pas la traduction dynamique binaire pour corriger le code en incluant l'instrumentation mais se base sur l'utilisation de différentes versions, créées et instrumentées statiquement, au niveau de cette représentation intermédiaire. Il fait notamment appel à ce procédé d'instrumentation par échantillonnage pour la réalisation de choix d'optimisation spéculatifs, en particulier au niveau des boucles contenues au sein de l'application considérée. D'autres études, comme Pin [82] (simulateur instrumenté de Intel), Valgrind ou DynamoRIO, travaillent directement sur l'instrumentation à l'exécution du code binaire. Dans ces outils, la traduction dynamique binaire est utilisée pour mettre à jour le code avec l'instrumentation en ligne, à l'aide de la compilation dynamique.

Côté profilage, un certain nombre d'études ont également été réalisées afin d'assurer le suivi du code tout au long de son optimisation à partir du code source d'entrée. Ce point est très important en ce qui concerne la compilation dynamique afin d'assurer notamment le lien entre la représentation intermédiaire à optimiser et le code machine profilé. Parmi les travaux notables, il est possible de mentionner ceux réalisés par Kumar et al. [83] visant à proposer une solution transparente de débogage pour le code optimisé dynamiquement. LLVM propose également l'implémentation d'une instrumentation dans sa représentation intermédiaire permettant d'inclure un certain nombre de méta-informations utiles au débogage du programme tout au long de sa durée de vie (aussi bien lors des compilations statique et dynamique que pendant les phases d'analyse et d'optimisation entre les phases d'exécution). D'autres études portant sur l'instrumentation de code ont également été réalisées et cela à différents niveaux allant de la représentation intermédiaire [80] au code binaire [84].

Concernant la compilation dynamique, un aspect crucial au niveau du profilage et des manipulations effectuées concerne **la granularité de traitement** du code de l'application. Deux approches sont aujourd'hui envisagées dans l'état de l'art.

La première, au niveau méthode (ou fonction), consiste à **considérer l'ensemble d'une fonction** comme granularité de traitement par le compilateur dynamique. Chaque méthode sera donc profilée et compilée dynamiquement si sa fréquence d'exécution dépasse le seuil fixé. Cette approche présente l'avantage de limiter le profilage et la complexité de celui-ci, puisqu'il suffit de se limiter à l'observation des différentes méthodes appelées durant l'exécution de l'application. Toutefois, elle comporte deux défauts majeurs. Le premier est le manque de visibilité sur le contexte d'exécution pour d'éventuelles optimisations. Le compilateur n'a en effet aucune visibilité sur l'appelant de la méthode et se prive ainsi de certaines optimisations comme la mise en ligne de fonctions appelantes et appelées. Le second est relatif à l'impossibilité pour ce type d'approche d'avoir un profilage régulier quant au volume de code profilé, surtout lorsque ce dernier varie beaucoup d'une méthode à l'autre dans un programme. Cette variation impacte directement le niveau d'optimisation réalisable (visibilité sur le contexte d'exécution) et le temps de génération du code machine. Cette granularité est notamment celle retenue par Ildjit et la machine virtuelle Dalvik de Google [6].

La seconde approche consiste à **considérer les traces d'exécution** comme granularité de traitement. L'intérêt est d'avoir une meilleure visibilité sur le contexte d'exécution en considérant une suite d'instructions qui seront exécutées les unes derrière les autres, avec un impact direct sur les capacités d'optimisation du compilateur. Il est ainsi possible de traiter une suite d'instructions à compiler s'étalant sur différentes méthodes pour lesquelles il sera par exemple possible d'effectuer une mise en ligne. Le défaut majeur de cette approche est le surcoût induit par le profilage sur les performances et sur le volume de code, de part l'ajout et l'exécution de phases d'instrumentation. Une solution largement adoptée aujourd'hui dans le domaine de la compilation dynamique est le découpage de l'application sous la forme de **blocs de base**. Cette approche est notamment celle utilisée par LLVM pour la manipulation et la compilation des portions de représentation intermédiaire sous forme de code machine. Un bloc de base est une portion de code constituée d'un point d'entrée et d'un point de sortie. Ainsi, chacun de ces blocs peut être considéré comme une suite autonome d'instructions à profiler. Ce profilage peut se faire tout simplement à l'aide de compteurs, placés par exemple en début de boucle. C'est grâce à ces compteurs que la criticité de la portion est déterminée, justifiant ainsi l'utilisation de la compilation dynamique ou non. Ce découpage en bloc de base offre un bon compromis au niveau de la taille de la portion de code manipulée. Elle contient en effet suffisamment d'instructions pour permettre au compilateur d'effectuer différentes optimisations, tout en garantissant un volume de code suffisamment limité pour permettre une complexité et un temps de génération raisonnable du code machine.

L'ensemble des études réalisées sur le profilage montre qu'il s'agit d'un domaine déjà largement exploré et optimisé. Même si de nombreuses pistes d'optimisation sont encore envisageables, un arbitrage a été réalisé dans le cadre de cette thèse afin de ne pas se focaliser sur cet aspect, malgré son importance, dans l'optique de proposer une solution d'accélération de la compilation dynamique. Les profilages qui ont notamment été effectués sous LLVM appuient cette décision en mettant en avant que les phases de profilage, déjà largement optimisées, ne représentent pas un aspect critique en temps d'exécution et que d'autres postes s'avèrent être de sérieux candidats à une accélération potentielle. De plus, nous avons vu que l'instrumentation pour le profilage fait appel à la traduction dynamique binaire, et peut par conséquent bénéficier des solutions mises en place dans le cadre de ces travaux. Seul le traitement des informations de profilage a donc été réellement écarté du cadre de cette thèse.

## 1.4 Gains potentiels offerts par la compilation dynamique

Il est possible de voir à travers la présentation de ces quatre technologies que la compilation dynamique est aujourd'hui largement adoptée dans différents domaines applicatifs. L'objectif de cette partie est de présenter **un panel de résultats** obtenus par son utilisation en termes **de gains dans les différentes technologies**, afin de mettre en avant son intérêt.

Concernant **les machines virtuelles**, et en particulier les machines virtuelles Java, l'adoption de la compilation dynamique a permis de réduire de manière conséquente les temps d'exécution des applications Java en comparaison avec une interprétation de celles-ci. Ainsi, les machines virtuelles Dalvik et EBunny [6, 85] annoncent une division par 4 des temps d'exécution de ces applications grâce à l'inclusion de ces compilateurs dynamiques au sein des machines virtuelles, et cela en prenant en compte le surcoût induit par leur inclusion. La machine virtuelle HotSpot annonce quant à elle un gain d'un ordre de grandeur par l'utilisation de la compilation dynamique, en comparaison avec une machine virtuelle ne l'incluant pas, en l'occurrence KVM [86]. Les auteurs

avancent aussi un rapport 50 entre les temps d'exécution d'une portion de code compilée nativement et ceux de cette même portion interprétée (temps d'exécution bruts, sans les services de la machine virtuelle).

Au niveau de la **traduction dynamique binaire**, les gains avancés mettent en avant le faible impact de cette traduction en comparaison avec une exécution native de l'application directement dans le jeu d'instructions de la cible. Ainsi, les chiffres avancés pour Apple Rosetta, développé par Transitive, font état d'une perte en performances limitée à 20 % par rapport à l'exécution de ce même code compilé au préalable statiquement [87]. VaporSIMD avance quant à lui des résultats du même ordre de grandeur que ceux obtenus pour une compilation statique avec le compilateur propre au processeur ciblé (pertes limitées entre 0 et 20 %), tout en offrant la portabilité de la solution mise en place. Enfin, l'utilisation de solutions de traduction dynamique binaire basées sur la compilation dynamique dans le cadre de la simulation de jeu d'instructions a permis d'afficher une multiplication par 3 des performances en comparaison avec une solution basée sur l'interprétation [64].

Pour **l'optimisation dynamique binaire**, l'inclusion de la compilation dynamique a permis d'obtenir des performances supérieures à celle obtenues pour une exécution native de l'application compilée statiquement, grâce à l'utilisation des informations disponibles à l'exécution. Ainsi, Dynamo permet l'obtention de gains allant jusqu'à 20 % alors que DynamoRIO avance des gains allant jusqu'à 50 % par rapport à une exécution native. Les études réalisées sur deGoal par Lhuillier et al. [36] dans le cadre de l'optimisation dynamique de l'allocation et de la désallocation de la mémoire par l'allocateur mémoire montrent des gains pouvant atteindre 56 % par rapport à une compilation statique de la même application.

Dans le cadre de **la compilation dynamique des langages typés dynamiquement**, les résultats mis en avant dans le papier détaillant le projet SPUR de Microsoft [10] permettent l'obtention de gains de l'ordre de 10× pour Google V8 (Chrome) et de 6× pour TraceMonkey (Firefox) et SPUR (moyennant la suppression de certains tests de la suite de tests SunSpider mentionnés par les auteurs de l'étude) par rapport à une simple interprétation des applications JavaScript comme c'est le cas sous Internet Explorer 8 (Internet Explorer ayant inclus son premier compilateur dynamique pour JavaScript à partir de la version 9).

Concernant **LLVM**, et plus spécifiquement son compilateur LLC, nous avons évoqué en introduction les travaux réalisés par Jääskeläinen [18] qui avancent l'opportunité de paralléliser des applications OpenCL en se basant sur le compilateur LLVM, conduisant à des gains de l'ordre de 3.76×. L'efficacité du compilateur de LLVM permet d'envisager la génération d'un code avec des performances équivalentes à celles de GCC pour un même niveau d'optimisation [15, 16], offrant de plus **une plus grande portabilité et une plus faible empreinte mémoire** (quelques méga-octets). Enfin, LLVM a permis la mise en place de solutions de virtualisation à forte portabilité, offrant des performances équivalentes voir supérieures à celles obtenues pour des machines virtuelles spécifiques, grâce notamment aux travaux réalisés sur VMKIT.

Au niveau du **langage 'C et du compilateur tcc** mis en place, le surcoût induit par la compilation dynamique est estimé entre 100 et 600 cycles par instruction générée selon le niveau d'optimisation. Les résultats en performances brutes montrent des gains possibles sur les applications allant jusqu'à un ordre de grandeur grâce à l'utilisation de la compilation dynamique, avec des gains moyens situés entre 2× et 4×. Le surcoût induit par la compilation dynamique est ainsi compensé dans la plupart des cas après une centaine d'utilisations du code généré dynamiquement. Cette rentabilité peut même être atteinte dès la première utilisation pour certains cas où la compilation dynamique apporte un gain très important, tant l'application peut bénéficier d'optimisations à ce

niveau.

Enfin, une dernière illustration des gains potentiels qu'il est possible d'obtenir par la compilation dynamique, dans un tout autre contexte que ceux évoqués jusqu'à présent, est l'étude réalisée par Wu et al. dans [88], visant à mettre en place une stratégie de compilation dynamique dans **une optique de contrôle de l'énergie et des performances** sur un microprocesseur. L'objectif est ici de proposer un cadriciel d'optimisation de l'utilisation des variations de fréquences et de tensions (DVFS, *Dynamic Voltage Frequency Scaling*) à l'exécution, inclus dans un système de compilation dynamique. Les résultats obtenus mettent en avant une perte en performances de l'ordre de 2.5 % pour une économie de 20 % en énergie en comparaison avec une exécution native sans DVFS, contre une perte en performances de 8 % pour une économie d'énergie de seulement 12 % dans le cadre d'une planification statique, avant l'exécution, du DVFS.

## 1.5 Conclusion

Ce chapitre a permis de présenter le concept de compilation dynamique ainsi que l'état de l'art associé. Celui-ci a volontairement été focalisé sur les quinze dernières années durant lesquelles la compilation dynamique a connu un véritable essor pour les raisons évoquées en introduction : la demande croissante des besoins en virtualisation et l'exploitation du dynamisme et de la sensibilité aux jeux de données des applications. L'état de l'art réalisé a permis de mettre en avant le large panel de domaines applicatifs visé par la compilation dynamique à travers la présentation des différentes technologies y faisant appel. La partie consacrée à la présentation des gains obtenus à travers les différentes technologies par son utilisation a également permis de justifier de son attrait croissant. Nous nous proposons, dans le prochain chapitre, de conduire une analyse détaillée de l'ensemble des technologies présentées, en réalisant un état de la technique de la compilation dynamique.

# Chapitre 2

# État de la technique de la compilation dynamique

**Sommaire**



## 2.1 Introduction

Nous nous proposons dans ce chapitre de réaliser **une classification des technologies de compilation dynamique** en fonction de leurs objectifs et des spécificités de leur flot de compilation, basée sur l'état de l'art réalisé dans le chapitre précédent. L'objectif est de mettre en avant la convergence entre toutes ces technologies dans l'optique de démontrer que la compilation dynamique peut être considérée comme un domaine algorithmique à part entière. Puisque notre étude se focalise sur la gestion de la compilation dynamique dans les systèmes embarqués, une ouverture vers ce dernier sera réalisée en fin de ce chapitre, en mettant notamment en avant les optimisations existantes aujourd'hui pour accroître l'attractivité de ce domaine sur ces systèmes.

## 2.2 Classification des technologies de compilation dynamique

Le chapitre précédent a permis de passer en revue l'état de l'art de la compilation dynamique sous la forme d'une classification en différentes technologies de celle-ci. Nous avons pu voir que ce domaine vieux de 50 ans est majoritairement utilisé depuis une quinzaine d'années dans des projets aussi bien académiques qu'industriels. A travers l'état de l'art réalisé, un certain nombre de points essentiels ont été mis en avant, en montrant les spécificités liées à chacune de ces technologies. Parmi ces points, il est possible de noter l'importance du compromis à trouver entre temps alloué à l'exécution de l'application et temps alloué à la compilation, ceux-ci se retrouvant réalisés au même instant, afin de minimiser le temps global d'exécution. L'objectif de cette section est de réaliser une classification des technologies présentées en fonction de leurs objectifs, des spécificités de leur flot de compilation et de l'importance qu'occupe la représentation intermédiaire au sein de ces technologies.

### 2.2.1 Divergence entre les objectifs des différentes technologies

En analysant l'état de l'art précédemment réalisé, il a été mis en avant **deux objectifs principaux distincts** concernant la compilation dynamique :

- **l'interopérabilité**, conséquence de l'accroissement de la complexité, de la flexibilité et des évolutions rapides du matériel face au logiciel, avec l'émergence notamment des solutions de virtualisation et des langages à haut-niveau d'abstraction ;
- **la compilation multi-étages** (ou optimisation continue), utilisée dans une optique d'accroissement des performances grâce aux optimisations supplémentaires réalisables durant l'exécution.

**L'interopérabilité** consiste en la capacité des systèmes à interagir entre eux. Un objectif majeur du développement de la compilation dynamique est de permettre un interfaçage efficace entre le code source et la plateforme avec une indépendance totale entre les deux et avec un surcoût raisonnable en performances. L'idée, pour les machines virtuelles, la traduction dynamique binaire et les moteurs d'exécution des langages typés dynamiquement, est de compenser la perte en performances induite par leur utilisation. Ces technologies cherchent à bénéficier du gain offert par la compilation dynamique par rapport à une approche basée sur l'interprétation.

L'intérêt de ces technologies est de permettre de changer rapidement de cible dans l'optique d'un portage d'application : aucune modification n'est à apporter sur cette dernière, les efforts se concentrant sur la couche d'interfaçage que constituent ces technologies. Elles permettent également l'utilisation de langages à haut niveau d'abstraction. Toutefois, l'incorporation de ces phases de traduction et d'interprétation au cours de l'exécution de l'application limite les performances offertes par ces solutions. La compilation dynamique tend à réduire les pertes induites en tentant d'avoisiner les performances qu'offre une application compilée statiquement au niveau de son temps d'exécution. Pour y arriver, elle tire profit des gains obtenus par l'optimisation à l'exécution de l'application, compensant en grande partie le surcoût induit par la compilation dynamique et surtout par l'étage de virtualisation ou de traduction.

**La compilation multi-étages** est utilisée dans un objectif différent. Comme déjà mentionné, le dynamisme des applications et leur sensibilité au jeu de données sont grandissants et la complexité des architectures devient telle qu'il est difficile d'optimiser statiquement ces applications en vue de leur exécution. L'incorporation de la compilation dynamique est donc ici réalisée dans un unique but d'accroissement des performances par rapport à une compilation statique.

Le fait de déporter les phases de compilation à l'exécution permet de bénéficier de la connaissance de l'environnement d'exécution afin d'optimiser l'application, d'accroître la qualité du code généré et par conséquent d'en réduire le temps d'exécution. Le surcoût induit par la compilation dynamique est ici compensé par les gains obtenus sur l'application, permettant de réduire les temps d'exécution globaux, incluant temps de compilation et d'exécution, par comparaison avec une exécution du code compilé statiquement. Cette approche est notamment celle visée par les technologies de compilation multi-étages et d'optimisation dynamique binaire.

Dans les deux cas, schématisés sur la figure 2.1, l'enjeu essentiel est de trouver **le compromis optimal entre le temps alloué à l'exécution et celui alloué à la compilation**. Plus le temps accordé à la compilation sera long, plus la portion de code compilée pourra bénéficier de différentes phases d'optimisation et plus son temps d'exécution s'en verra réduit. Toutefois, l'accroissement de ce temps de compilation entraînera une monopolisation des ressources normalement allouées à l'exécution et/ou pourra entraîner des temps d'attente en cours d'exécution en cas de besoin d'une portion de code non compilée. A contrario, en règle générale, plus le temps accordé à la compilation sera court, moins la portion de code compilée sera optimisée et plus son temps d'exécution sera long, mais le temps disponible pour l'exécution s'en trouvera rallongé (ceci ne se vérifie toutefois pas pour certaines optimisations très courtes permettant d'obtenir de forts gains au niveau des temps d'exécution, comme la parallélisation conditionnelle de boucles [70]).

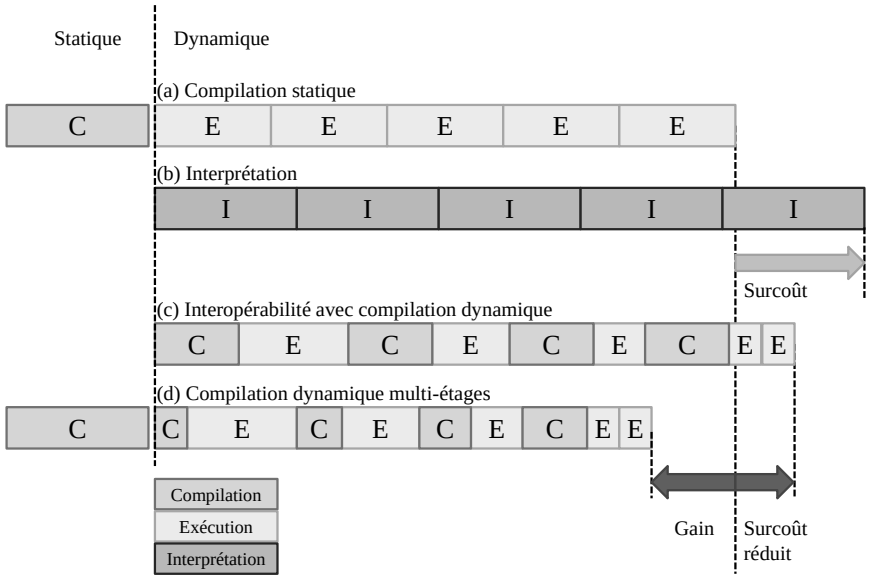



FIGURE 2.1 – Diagramme de présentation des notions d'interopérabilité et de compilation multi-étages.

Différentes études cherchent à optimiser la répartition entre les phases de compilation et phases d'exécution. Les compilateurs adoptent pour la plupart une **méthodologie d'optimisation variable**, en fonction du temps accordé, allant du niveau le moins optimisé, assimilable à de l'interprétation, au plus optimisé. Le code peut ainsi être simplement interprété au démarrage, s'il n'a pas été compilé statiquement au préalable, et optimisé par itération (réalisation de différentes phases de compilation avec un niveau croissant d'optimisation) en fonction de sa criticité et du temps alloué à sa compilation.

Certaines études se focalisent sur la détermination **du ratio optimal entre temps alloué à la compilation et temps alloué à l'exécution** et sur la proposition de

nouveaux ordonnanceurs pour optimiser cette répartition, comme celle réalisée par Kulkarni et al. [89]. Les auteurs se focalisent sur l'analyse de l'évolution des temps globaux d'exécution en fonction de la répartition et de l'allocation des tâches de compilation et d'exécution, du nombre de tâches disponibles et de l'agressivité du compilateur. Ils se concentrent pour cela sur les performances des ordonnanceurs de tâches du système en fonction de leur agressivité : un contrôleur optimisant de manière agressive dès le départ mettra plus de temps à compiler lors des premières itérations mais convergera vers la solution optimale plus rapidement, alors qu'un contrôleur moins agressif mettra plus de temps à converger vers cette solution. Ces ordonnanceurs visent à remplacer l'approche classique dite *round-robin* (tourniquet), ne tenant pas compte des spécificités des tâches de compilation face à celles d'exécution. Ils proposent également l'exploitation des temps d'inactivité du processeur afin de maximiser le nombre de tâches disponibles à la compilation. Ils en arrivent à la conclusion que pour une sollicitation des tâches de compilation disponibles à 100 %, l'ordonnanceur le plus agressif permet d'obtenir des gains de l'ordre de 18.2 % sur le temps total d'exécution alors que l'ordonnanceur le moins agressif permet des gains de l'ordre de 9.3 %, compensant largement le surcoût induit par l'utilisation de ce type d'ordonnanceurs.

D'autres études se focalisent sur les systèmes multi-coeurs [90, 91], parmi lesquelles Ildjit, proposant l'utilisation des ressources non utilisées par l'application pour y exécuter les tâches de compilation. Ildjit assigne ainsi des tâches de compilation aux différents processeurs en veille de l'architecture à l'aide d'un mécanisme de prédiction spéculatif appelé *Dynamic Look Ahead* (DLA). Ce mécanisme analyse pour cela le graphe d'appel des différentes méthodes composant l'application. En déterminant la criticité de ces méthodes, la distance à laquelle elles se trouvent de la méthode en cours d'exécution et leur probabilité d'exécution, ce mécanisme les insère dans une file de compilation en leur assignant un ordre de priorité. Le compilateur vient ensuite chercher les méthodes à compiler au sein de cette file en fonction du nombre de ressources disponibles pour la compilation (ressources en mode veille ou *IDLE*).

Une étude de Kulkarni et al. [92] se penche, quant à elle, sur les deux approches, simple-coeur et multi-coeurs, en expliquant les différences fondamentales entre les deux systèmes. Cette étude est en quelque sorte une synthèse des deux approches présentées, expliquant la répartition optimale des tâches de compilation et d'exécution sur un coeur unique et sur une architecture multi-coeurs. Elle insiste sur le fait que, dans tous les cas de figure, l'anticipation des portions de code à compiler est impérative pour permettre d'en limiter l'impact sur l'exécution de l'application et pour atteindre des niveaux d'optimisation élevés sur les portions de code compilées. Les résultats obtenus montrent les grandes opportunités offertes par l'utilisation de ressources multi-coeurs, non pas par une éventuelle parallélisation des compilateurs, pratiquement inenvisageable de par la complexité de leurs codes, mais grâce à la parallélisation des tâches de compilation entre les différentes portions de code à compiler, et notamment l'utilisation des ressources inutilisées comme proposé dans Ildjit.

Toutefois, toutes ces études n'abordent pas l'impact non négligeable qu'elles engendrent sur **la consommation globale du système** puisqu'utilisant des ressources normalement en sommeil ou des temps d'inactivité du processeur. Si cela n'est pas un problème sur les architectures de type station de travail, cela peut vite devenir problématique sur des architectures de type serveur ou embarquées, pour lesquelles les contraintes en consommation sont beaucoup plus fortes. Nous verrons d'ailleurs le cas particulier des systèmes embarqués plus en détails dans la suite de ce chapitre.

### 2.2.2 Évolution des flots de compilation entre les différentes technologies

Nous cherchons ici à montrer **une convergence au niveau de l'évolution des flots de compilation** des différentes technologies de compilation dynamique. Nous nous appuyons pour cela sur la figure 2.2, dont les grands principes ont déjà été présentés dans le chapitre 1, et sur laquelle figurent les différents états du code aux différentes phases de compilation.

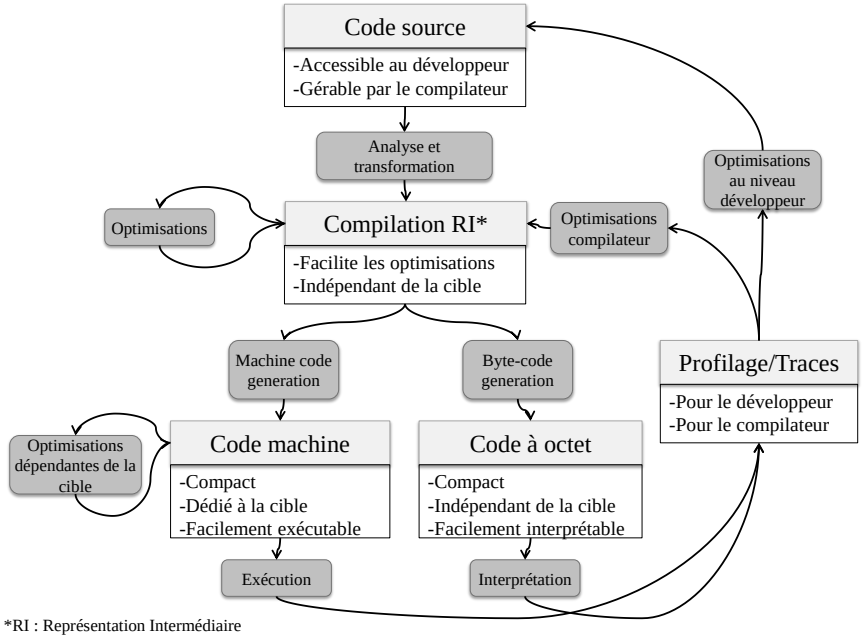



FIGURE 2.2 – Schéma classique d'évolution du flot de compilation.

Les propriétés des différents niveaux de représentation mis en avant sur la figure 2.2 varient en fonction de leurs objectifs. Le **code source** a ainsi pour propriété d'être compréhensible par le développeur. Il lui offre un compromis concernant le niveau d'abstraction afin de s'affranchir des spécificités de la cible tout en gardant une maîtrise totale sur l'algorithme qu'il développe. La principale opération réalisée au niveau de la partie frontale, outre les différentes analyses syntaxiques, sémantiques et lexicales, est la traduction du code source en un arbre syntaxique abstrait puis en représentation intermédiaire interne au compilateur.

Cette **représentation intermédiaire** permet de faciliter la réalisation des étapes d'optimisation et de génération de code. Elle est, pour la plupart des compilateurs, choisie de manière à rester indépendante du code machine de la cible pour laquelle le programme est compilé, cela permettant de faciliter la portabilité du compilateur. Pour les machines virtuelles, elle se présente souvent sous la forme de code à octet. Ce code à octet, dans son approche historique (code à octet Java), permet, en plus de sa compacité, de faciliter son interprétation et sa gestion par la machine virtuelle. L'émergence de la compilation dynamique a également poussé à l'apparition de codes à octet tout aussi compact mais beaucoup plus explicites pour en faciliter la compilation (code à octet CIL), contrairement aux premiers codes à octet. Dans le cadre de la compilation dynamique cette question de la compacité pour la représentation intermédiaire est centrale, afin de limiter son empreinte mémoire.

Cette représentation intermédiaire, une fois optimisée, est traduite en **code machine**. Ce code machine a la propriété d'être également compact et d'être compréhensible par la cible. A ce niveau sont réalisées certaines optimisations dépendantes de la

cible, comme l'allocation des registres. Le raccordement en mémoire du code compilé est ensuite réalisé, avant que celui-ci ne soit exécuté.

Comme nous l'avons introduit au chapitre 1, une chaîne de compilation conventionnelle permet de réaliser statiquement l'ensemble des étapes allant de la gestion du code source à la génération du code machine optimisé pour la cible. Nous nous proposons maintenant de la comparer avec l'ensemble des flots des différentes technologies de compilation dynamique, faisant notamment apparaître les phases réalisées hors-ligne et durant l'exécution. Les résultats de cette analyse sont présentés dans le tableau 2.1 et se basent sur la présentation des flots de la figure 2.2.

TABLE 2.1 – Évolution des flots de compilation dynamique en fonction des technologies, en mettant en avant les portions réalisées hors-lignes et durant l'exécution ; RI : Représentation Intermédiaire, CM : Code Machine, CS : Code Source.

| | Phases hors-ligne | Phases à l'exécution |
|---|---|---|
| Compilation statique | gestion CS, génération RI, optimisation RI, génération CM, optimisation CM | |
| Machine virtuelle | gestion CS, génération RI (code à octet) | optimisation RI, génération CM, optimisation CM |
| Traduction dynamique binaire | | gestion CM, génération RI, optimisation RI, génération CM, optimisation CM |
| Compilation multi-étages | gestion CS, génération RI | optimisation RI, génération CM, optimisation CM |
| Moteur d'exécution | | gestion CS, génération RI, optimisation RI, génération CM, optimisation CM |

Ce tableau permet de mettre en avant **l'aspect central de la représentation intermédiaire** et des optimisations qui sont faites à son niveau dans la compilation dynamique. En effet, outre la génération et l'optimisation du code machine qui sont des phases redondantes mais dépendantes de la cible pour laquelle le code est compilé (et donc pour lesquelles la généricité est limitée, représentant un goulot d'étranglement sémantique), les opérations de manipulation et d'optimisation de la représentation intermédiaire sont réalisées dans chacune des technologies. Bien que ces représentations diffèrent d'une technologie à une autre (nous verrons qu'il en existe trois principales), les opérations réalisées présentent de grandes similarités entre les technologies (manipulation de blocs de base, optimisations itératives, etc.).

### 2.2.3 Importance de la représentation intermédiaire au sein des différents flots

Nous venons de mettre en avant l'aspect central de la représentation intermédiaire au niveau du flot de compilation, en particulier dans le cadre de la compilation dynamique et des différentes technologies mises en avant. Il convient de comprendre tout d'abord l'intérêt pour les concepteurs de réaliser l'essentiel des opérations du compilateur au niveau de cette représentation. Comme nous l'avons déjà mentionné, elle permet

de découpler le code source de la cible pour laquelle le code est compilé. Elle permet autrement dit de cloisonner les parties frontales (*front-end*), intermédiaires (*middle-end*) et arrières (*back-end*). Nous verrons également qu'elle permet d'optimiser les manipulations des portions de code par le compilateur, offrant les structures et les informations nécessaires aux différentes phases de compilation, et de réaliser une grande série d'optimisations indépendamment du langage source et de la cible.

Cet aspect central de la représentation intermédiaire impose de **nombreuses contraintes** à son niveau. Développée initialement dans le cadre d'une compilation hors-ligne, son utilisation dans le cadre de la compilation dynamique a conduit à l'apparition de ces contraintes qui, nous allons le voir, dépendent de l'objectif dans lequel la compilation dynamique est utilisée.

#### Cas de l'interopérabilité

Les contraintes des systèmes imposent aux représentations intermédiaires de s'adapter **à des points de démarrage divergents** de ceux rencontrés dans les chaînes de compilation conventionnelles. Par exemple, les machines virtuelles utilisent le code à octet comme représentation intermédiaire pour les compilateurs dynamiques qui sont mis en oeuvre au sein de celles-ci (machines virtuelles Java utilisant le code à octet Java). Les traducteurs dynamiques binaires peuvent utiliser une représentation intermédiaire quant à elle très proche du code machine (par exemple Apple Rosetta). Les moteurs d'exécution pour les langages typés dynamiquement peuvent utiliser directement le code source comme point de démarrage (JavaScript pour les applications web avec, par exemple, le compilateur SPUR de Microsoft où le JavaScript est dynamiquement traduit en CIL avant d'être converti en code machine, ou encore dans Google V8 où le JavaScript est directement traité comme représentation intermédiaire).

Dans cette optique, un ratio optimal doit être trouvé dans **le niveau de représentativité** entre le haut-niveau du code source et le bas-niveau du code machine. L'objectif est de conserver suffisamment d'informations pour permettre une compilation avec de fortes optimisations tout en étant suffisamment bas-niveau pour permettre une traduction rapide vers le code machine. La représentation intermédiaire doit également permettre d'assurer la portabilité des applications, en permettant de changer facilement les cibles pour lesquelles elles sont compilées.

La plupart des études s'accordent à dire que la mise en place d'une représentation intermédiaire indépendante du code source et de la cible en entrée du compilateur dynamique est essentielle dans l'optique d'une adaptation rapide du compilateur à un nouveau langage source et à une nouvelle cible. Cela permet d'assurer ainsi la portabilité des applications compilées mais aussi du compilateur en lui-même, facilement réutilisable avec un surcoût en termes de développement se concentrant uniquement sur les parties frontales et arrières (et non pas sur le coeur même du compilateur).

#### Cas de la compilation multi-étages

Cette approche consiste à déporter **les phases de compilation à l'exécution, dans une optique de performances**. Par conséquent, la représentation intermédiaire fait ici aussi l'objet d'importantes contraintes. Elle doit en effet conserver un niveau suffisant d'informations pour bénéficier des éléments connus à l'exécution pour l'optimisation du code, tout en garantissant une certaine proximité avec le code machine de la cible afin d'en assurer une génération rapide. Le choix de la représentation intermédiaire se fait donc sur deux plans :

– **le niveau de sémantique requis** par les passes d'optimisation du compilateur ;

– **les contraintes du système** au niveau du point d'entrée de la représentation intermédiaire et du code machine à générer dynamiquement.

Dans le cas où le second point n'impose aucune contrainte sur la représentation intermédiaire, c'est le niveau requis de sémantique qui détermine le choix de cette représentation. Si le code machine de la cible pour laquelle le code doit être généré est connu, la représentation intermédiaire est choisie en fonction de celui-ci. Si les optimisations imposées par le compilateur nécessitent de réaliser des analyses bas-niveau sur le code à compiler, la représentation choisie sera plus proche de la représentation intermédiaire utilisée dans les chaînes de compilation statiques conventionnelles (comme la représentation intermédiaire utilisée pour LLVM). Enfin, si les optimisations à réaliser nécessitent une connaissance algorithmique de l'application à compiler, la représentation intermédiaire combinera à la fois des éléments de représentation intermédiaire et de code source à utiliser dynamiquement, comme par exemple en utilisant des modèles de spécialisation de code, des stratégies explicites de compilation dynamique (explicitement mentionnées dans le code source), ou encore l'utilisation de codes à octet tels que CIL ou ses versions allégées développées pour réduire l'empreinte mémoire (Ildjit).

## 2.3 Convergence des technologies de compilation dynamique

L'analyse des différentes technologies de compilation dynamique nous a permis de mettre en avant deux grandes approches pour ces technologies, ainsi qu'une certaine convergence dans les flots de compilation avec la gestion de la représentation intermédiaire en point central. Ces éléments constituent une première étape dans l'optique de démontrer une convergence entre les différentes technologies, bien que se destinant à des domaines applicatifs extrêmement différents. Nous allons dans cette partie mettre en avant cette convergence à granularité plus fine, au niveau **des algorithmes des compilateurs dynamiques**, **des spécificités de leur représentation intermédiaire** et de **la représentation des données** pour la manipulation du code au sein de ceux-ci, en se basant sur l'analyse préalablement réalisée des flots de compilation.

### 2.3.1 Convergences sur les algorithmes

L'analyse des différentes technologies de compilation dynamique a permis de mettre en avant trois grands algorithmes partagés entre les différentes technologies : les **phases d'optimisation au niveau de la représentation intermédiaire**, **la génération du code machine** et **son optimisation**.

#### Phases d'optimisation de la représentation intermédiaire

C'est à ce niveau que sont réalisées **les optimisations indépendantes de la cible**, comprenant quelques unes des optimisations standards comme le déroulage de boucles, la mise en ligne de code, la prédiction de branchement, la réduction de boucle, etc. La plupart des études s'accordent à dire que cette étape est essentielle et que le surcoût induit par l'utilisation de la représentation intermédiaire est compensé par le gain obtenu sur le temps d'exécution du code à ce niveau. Pour chaque technologie de compilation dynamique, le même ratio optimal entre le niveau d'optimisation et le temps alloué à la compilation dynamique doit être trouvé.

Pour les chaînes de compilation statiques, ce ratio optimal n'est pas une contrainte puisque la compilation est réalisée en amont de l'exécution, sans contrainte forte en termes de temps. Mais concernant la compilation dynamique, cet aspect est critique

puisque les ressources allouées à la compilation ne sont plus disponibles pour l'exécution de l'application, qui peut de plus être retardée dans l'attente de la compilation d'une portion à exécuter. La solution majoritairement adoptée consiste en l'introduction **de plusieurs niveaux d'optimisation**, en fonction de l'urgence avec laquelle le code doit être compilé et du temps qui est accordé à cette compilation.

**Pour les cas urgents**, autrement dit quand l'exécution de l'application nécessite l'utilisation d'une portion de code machine non traduite, le compilateur ne réalise aucune optimisation sur cette portion, qui est **directement interprétée**, au détriment des performances du code généré. Cette réalisation est assimilable à une approche dite "trampoline", dans laquelle la tâche d'exécution est complètement interrompue pour laisser la place à la tâche de compilation. Pour éviter ces générations de portions de code non-optimisées, certains projets implémentent une première étape de compilation, statique ou dynamique, sur la totalité de l'application, avec un faible niveau d'optimisation. Ainsi, l'ensemble du code machine est dans tous les cas disponible, évitant cet effet "trampoline". L'inconvénient majeur de ce choix est la compilation de portions de code possiblement non-utilisées, constituant ainsi une perte de temps mais aussi d'espace mémoire. D'autres projets, comme Ildjit, utilisent un mécanisme de prédiction appelé Dynamic Look Ahead (DLA) pour déterminer en avance de phase quelles méthodes sont exécutées. Les méthodes avec la plus forte probabilité d'exécution sont alors compilées, avec un niveau d'optimisation dépendant de leur distance avec la méthode courante [91].

### Phase de génération du code machine

Le générateur de code est **le noyau principal** du compilateur dynamique. Après l'optimisation de la représentation intermédiaire, le compilateur dynamique appelle différentes méthodes permettant de traduire cette dernière en code machine. Certaines vérifications et optimisations sont réalisées durant cette traduction dans l'optique d'obtenir un code machine facilement manipulable par la partie arrière, en explicitant en particulier les différentes opérations et types utilisés. **Trois cas de figure** peuvent se présenter.

Le premier est le cas où il existe **une correspondance directe** entre l'instruction sous forme de représentation intermédiaire et une instruction du code machine cible. C'est le plus souvent le cas pour les opérations basiques comme les additions, les chargements, etc., qui existent aussi bien sous la forme de représentation intermédiaire que sous la forme de code machine.

Le second cas correspond à la situation dans laquelle une instruction issue de la représentation intermédiaire correspond à **une séquence d'instructions** du code machine. Ainsi, la traduction du code à octet Java (compact et implicite) nécessite l'utilisation d'une séquence d'instructions du code machine même pour les instructions les plus courantes. Par exemple, l'instruction code à octet Java `addi` est traduite en 5 instructions machines (deux `loads`, un `add`, deux `stores`).

Le troisième cas est celui où il n'existe **aucune correspondance directe** entre les instructions de la représentation intermédiaire et le code machine, par exemple lors de la traduction d'instructions flottantes sur une cible ne disposant pas d'unité permettant leur gestion. La solution consiste ici à trouver une séquence d'instructions du code machine permettant d'approximer le comportement des instructions de la représentation intermédiaire. L'impact sur les performances de ce mode de traduction est non-négligeable, de par le surcoût induit par l'utilisation d'instructions non-optimales et de par les vérifications imposées au compilateur sur la justesse des approximations. Ce cas est donc limité aux cas les moins fréquents, avec la possibilité d'utiliser un cache de traduction afin de conserver les séquences d'instructions machines générées en

fonction des instructions de la représentation intermédiaire.

#### Phase d'optimisations du code machine

Même si cette phase peut être considérée comme optionnelle pour certaines cibles (et notamment celles basées sur une représentation de type machine à pile dont nous parlerons dans la suite), les compilateurs dynamiques permettent le plus souvent la réalisation **d'un certain nombre d'optimisations** au niveau du code machine généré. Ces optimisations, dépendantes de la cible, sont réalisées afin d'accroître les performances du code machine généré et ainsi d'en réduire le temps d'exécution.

L'optimisation la plus récurrente à ce niveau est celle de **l'allocation de registres** [93], considérant les spécificités de la cible et en particulier le nombre de registres disponibles. Cette étape est fortement dépendante de la cible, celle-ci pouvant aussi bien disposer d'un faible nombre de registres comme les architectures x86 (basées sur un fonctionnement avec une représentation de type machine à pile), ou d'un nombre beaucoup plus conséquent comme c'est le cas pour les processeurs embarqués au jeu d'instructions réduit (*Reduced Instruction Set Computer*, RISC).

L'efficacité de l'allocateur de registres va dépendre notamment du temps accordé à cette étape d'optimisation. Certains algorithmes permettent de déterminer l'utilisation optimale des registres en considérant la durée de vie des variables, au détriment du temps nécessaire à la réalisation de cette optimisation (LLVM dispose notamment de trois ou quatre niveaux d'allocation de registres plus ou moins optimisants, avec des algorithmes à la complexité croissante pour réaliser les allocations).

### 2.3.2 Représentations intermédiaires émergentes

Nous nous intéressons ici plus spécifiquement aux propriétés des représentations intermédiaires, afin de mettre en avant une convergence entre elles. La criticité de leur gestion a déjà été largement mise en avant dans les sections précédentes. Nous nous intéressons dans cette partie à l'étude des trois représentations intermédiaires principalement utilisées pour la compilation dynamique : **le code à octet Java**, **CIL** et **la représentation intermédiaire LLVM (LLVA)**. Certains projets introduisent leur propre représentation, notamment pour des besoins très spécifiques à ce niveau. Parmi elles, il est possible de citer Tirex, développée par Pietrek et al. [94], MinIR, développée par Le Guen et al. [95], la représentation intermédiaire associée au générateur de code TCG (*Tiny Code Generator*), développée par Bellard et al. dans Qemu [56], ou encore celle utilisée par l'outil Valgrind[61] dans l'optique d'instrumenter le code à profiler.

#### Code à octet Java

Il s'agit de la plus ancienne des trois représentations, apparue dans les années 90 avec les premières machines virtuelles Java. Développée en parallèle de ces dernières, cette représentation permet une interprétation des applications Java. Le développement de techniques de compilation dynamique pour les machines virtuelles a conduit à l'élaboration de solutions permettant la génération de code machine à partir du code à octet Java. Cependant, cette représentation ayant été développée initialement **dans une optique d'interprétation**, elle souffre d'une forte compacité et d'une forte densité au niveau des informations qu'elle contient. Ces informations sont donc contenues de manière implicite dans le code à octet Java et par conséquent difficilement accessibles. Un certain nombre d'informations nécessaires à la compilation dynamique (et aux optimisations) doivent donc être explicitées afin de produire un code machine de qualité (explicitation des types, des accès mémoire, etc.).

A titre d'exemple, il est possible de faire référence à la situation déjà présentée préalablement pour le cas de l'instruction `addi` du code à octet Java, nécessitant en réalité cinq instructions en code machine pour être réalisée (suivant les cibles). La compilation de code à octet Java en code machine nécessite donc l'ajout d'informations supplémentaires par les développeurs nécessaires au compilateur [79]. Outre un accroissement de l'empreinte mémoire nécessaire au stockage de ces informations supplémentaires, leur traitement par le compilateur engendre un surcoût en performances pour la compilation dynamique de l'application.

### CIL

Développé depuis 2002 dans le cadre du développement de .NET [96] par Microsoft et notamment utilisé dans les projets Mono [96] et Portable.NET [97] par l'intermédiaire de l'utilisation de la spécification *Common Language Infrastructure* (CLI), le *Common Intermediate Language* (CIL) [7], est une représentation standardisée (ECMA-335) de type code à octet pour machine à pile, orientée objet, comme le code à octet Java. Cependant, contrairement à ce dernier, le CIL a été développé durant l'essor de la compilation dynamique et est donc prévu pour pouvoir **être aussi bien interprété que compilé**, notamment par les machines virtuelles [98]. Les informations nécessaires à la compilation sont directement explicitées, tout en garantissant la compacité de la représentation.

Certaines études se penchent sur l'évaluation de cette compacité en comparaison avec un code machine, en l'occurrence pour une cible Intel x86. Nous verrons plus tard qu'elles mettent en avant le bon compromis offert par le CIL entre empreinte mémoire et niveau d'informations disponibles. Cependant, certains projets comme Ildjit considèrent qu'un certain nombre d'informations contenues dans le CIL ne sont pas nécessaires à la compilation, et proposent donc une version allégée de cette représentation. A contrario, d'autres travaux estiment que des informations supplémentaires sont nécessaires et que des méta-informations doivent être ajoutées à la représentation afin de permettre une compilation optimale [99]. Ces avis dépendent surtout des compilateurs ciblés par ces différents projets, ceux-ci pouvant proposer plus ou moins d'étapes d'optimisation et se destinant à différentes cibles pour lesquelles les besoins peuvent différer.

### LLVA

La représentation intermédiaire LLVM, appelée LLVA [66], a été spécifiquement développée pour **le cadriciel LLVM**. Cette représentation est de type code à octet à trois adresses et de type machine à registres, et constitue un jeu d'instructions indépendant de tous langages ou systèmes. Les instructions utilisent l'assignation statique unique (*Static Single Assignement*, SSA), permettant de réaliser plus facilement et plus efficacement les différentes optimisations, par utilisation d'algorithmes plus performants d'allocation [100] et par une analyse facilitée de la durée de vie des variables manipulées [101]. Cette forme SSA est utilisée par la majorité des compilateurs qui traduisent les représentations intermédiaires sous cette forme lors de leurs optimisations. Un gain de temps est donc réalisé puisque cette étape est déjà effectuée avec LLVA. Pour utiliser cette forme SSA sous une représentation de type machine à registres, LLVM fournit à sa représentation intermédiaire un banc de registres infini, utilisant ensuite ses algorithmes d'allocation de registres pour s'adapter à la cible désirée lors de la génération du code machine. La représentation contient l'ensemble **des informations nécessaires à la compilation**, accessibles de manière explicite, au niveau de la gestion des types, du flot de contrôle ou du flot de données.

LLVA offre la particularité d'être plus bas-niveau et donc **plus proche du code**

**machine** que les deux représentations préalablement présentées, tout en restant indépendante du code machine de la cible. Grâce à ce choix, un large panel de langages peuvent être transformés dans cette représentation, allant des langages à haut-niveau d'abstraction (Java, C#, C++, C, etc.) aux codes à octet (code à octet Java, CIL). La représentation LLVA apparaît, semble-t-il, comme étant encore trop haut-niveau pour être intégrée dans des projets de traduction dynamique binaire. Dans l'article de référence de LLVM [8], Chris Lattner ne positionne pas LLVA comme étant un concurrent du code à octet Java et de CIL mais souligne plutôt leur complémentarité. LLVA et le générateur de code du cadriciel LLVM (réalisant à partir de cette représentation intermédiaire les phases d'optimisation, la génération du code machine et son optimisation), sont aujourd'hui au coeur de nombreux projets, allant de la machine virtuelle (VMKit) au moteur d'exécution pour les langages typés dynamiquement (Apple Nitro), en passant par la compilation multi-étages (OpenCL, VaporSIMD).

#### Compacité des représentations intermédiaires

Les études réalisées dans l'état de l'art dans l'optique d'évaluer **la compacité des différentes représentations intermédiaire** ont permis de mettre en avant une convergence des besoins au niveau de cette compacité, les représentations devant être le plus compact possible pour des questions d'espace mémoire. Nous nous sommes pour cela référés au code machine x86. Celui-ci est un des plus compacts existant avec sa forte densité et ses instructions de taille variable, ce qui n'est pas le cas, par exemple, pas pour un processeur 32 bits de type RISC tel que le SPARC.

Pour **le code à octet Java**, les études montrent, pour des applications équivalentes, un code en moyenne 15 % plus compact que sous la forme code machine x86. Toutefois, nous avons déjà mentionné à plusieurs reprises les besoins d'explicitation de ce code à octet pour une utilisation dans le cadre de la compilation dynamique.

Concernant **le CIL**, les études réalisées par STMicroelectronics [99] montrent que, même si celui-ci est en moyenne 30 à 40 % plus compact que sa représentation équivalente en code machine x86, un ajout de méta-informations est nécessaire à son utilisation dans le cadre de la compilation dynamique. Ces méta-informations contiennent notamment un certain nombre d'informations sur les différentes classes et sur les dépendances. Leur insertion a pour conséquence d'accroître l'empreinte mémoire total de la représentation CIL de l'application. Les chiffres mis en avant pour l'étude montrent ainsi que leur ajout rend la taille de l'ensemble équivalente à celle du code machine x86 de l'application. Les mesures réalisées à partir des tests de référence miBench [102] mettent en avant une taille d'en moyenne 2 KO pour ces informations alors que les plus petites applications font moins de 500 octets en représentation CIL. Pour des applications de taille plus importante, le surcoût induit par les méta-informations devient moins significatif, l'ensemble restant à peu près équivalent ou inférieur à l'équivalent en code machine x86. Si nous considérons la version allégée mise en avant dans le projet Ildjit, nous constatons une compacité équivalente à celle de LLVA, 30 % inférieure au code machine x86 d'une même application.

Les études réalisées sur **LLVA** montrent une importante réduction de l'empreinte mémoire en comparaison avec l'équivalent x86. Tout en conservant toutes les informations nécessaires au générateur de code pour compiler l'application et l'exécuter ensuite, les tests effectués sur la série de test SPEC CPU2000 [103] montrent une réduction de 30 % du volume de code en comparaison avec le code machine x86 équivalent.

### Approches machine à pile et machine à registres

Le choix de la représentation intermédiaire **dépend essentiellement des choix technologiques** réalisés à la conception du compilateur et de la famille de cibles à laquelle l'ensemble se destine.

Si l'on considère une vision simplifiée du problème, il est possible de dire que **les représentations intermédiaires du type machine à pile** vont être préférées pour des processeurs cibles au banc de registres limité, comme les processeurs au jeu d'instructions complexe (*Complex Instruction Set Computer*, CISC), tels que les processeurs Intel x86. Le nombre de registres à disposition dans ces processeurs étant réduit, la pile va davantage être utilisée pour stocker les données les plus courantes (au sommet de celle-ci).

A l'opposé, **les approches du type machine à registres** vont être préférées pour des processeurs cibles disposant d'un grand nombre de registres, comme les processeurs de type RISC largement utilisés dans l'embarqué.

L'intérêt d'utiliser une représentation intermédiaire se basant directement sur l'approche désirée est **de faciliter la génération du code machine** à partir de celle-ci, évitant des phases supplémentaires de conversion pour s'adapter aux caractéristiques des cibles visées. C'est une des raisons expliquant notamment les problèmes de performances pour la gestion mémoire des applications Java par les JVM sur les systèmes embarqués, celles-ci devant optimiser l'utilisation du banc de registres du processeur à partir d'une représentation de type machine à pile. Des comparaisons entre les deux approches ont notamment été réalisées par Yunhe Shi et al. dans [104].

### Récapitulatif du comparatif entre les représentations intermédiaires

Le tableau 2.2 est **une comparaison des trois grandes représentations intermédiaires présentées**. Afin de mettre en avant les différences majeures à ce niveau entre la compilation dynamique et les chaînes de compilation statiques conventionnelles, une comparaison avec la représentation intermédiaire de GCC, Gimple, a été réalisée.

Table 2.2 – Comparaison des trois principales représentations intermédiaires utilisées pour la compilation dynamique et comparaison avec Gimple, représentation intermédiaire utilisée par GCC.

| | Code à octet Java | CIL | LLVA | Gimple |
|---|---|---|---|---|
| Interprétation | ++ | ++ | + | – |
| Compilation | - | ++ | ++ | ++ |
| Représentation | machine à pile | machine à pile | machine à registres | machine à registres |
| SSA | non | non | oui | oui |
| Portabilité | + (JVM) | ++ (standardisée) | + (LLVM) | – (seulement pour GCC) |
| Compacité | + 15 % < x86 | = x86 avec méta-informations | ++ (30 % < x86) | – (10× x86) |
| Bas niveau | + | + | ++ | - |

Ce tableau montre **une convergence entre les points critiques** des différentes technologies. Bien que des stratégies différentes soient adoptées en fonction des applications et des cibles auxquelles elles se destinent, notamment au niveau de la représentativité et de la forme SSA, il est possible de noter un certain nombre d'efforts communs au

niveau de la portabilité, de la compacité et du niveau de représentation (suffisamment bas niveau pour faciliter la traduction vers le code machine, réalisée dynamiquement). Concernant CIL, celui-ci apparaît comme étant la solution la plus portable aujourd'hui, de par sa standardisation.

Ce tableau permet également de mettre en avant l'aspect polyvalent du CIL en comparaison avec le code à octet Java pour les machines virtuelles, acceptant aussi bien les phases d'interprétation que de compilation. La problématique majeure pour cette représentation intermédiaire réside surtout dans le manque de projets supports l'utilisant. Qu'il s'agisse de Mono ou de Portable.NET, ces projets sont aujourd'hui quelque peu délaissés au profit d'autres solutions. Ildjit, dont nous avons déjà mentionné plusieurs fois l'existence, souffre quant à lui d'un important manque de maturité.

La représentation LLVA a, quant à elle, pour défaut majeur son manque de portabilité. Il n'existe en effet qu'un seul compilateur capable d'en assurer la gestion : le générateur de code du cadriciel LLVM. Toutefois, ce générateur de code, de par sa rapidité, sa portabilité et son efficacité, est aujourd'hui largement utilisé dans de nombreux projets aussi bien industriels qu'académiques, comme nous l'avons déjà mentionné. Sa représentation sous forme SSA et de type machine à registres lui permet de très bien s'adapter aux cibles embarquées.

### 2.3.3 Représentation des données pour la manipulation du code

Dans l'ensemble des technologies de compilation dynamique présentées, il apparaît **une convergence dans la manipulation des données**. Celles-ci, contrairement aux codes conventionnels, sont abstraites et par conséquent le plus souvent manipulées sous la forme de structures de données complexes. Les différentes phases de compilation (traduction, optimisations, etc.), poussent à la réutilisation de ces structures, à leur modification, à leur annotation et surtout à leur tri, justifiant le choix d'une représentation sous la forme de graphes.

Il est intéressant de citer **la représentation sous la forme de graphes orientés acycliques** [105] (DAG, *Directed Acyclic Graph*), utilisée notamment par LLVM dans la manipulation des différents blocs de base au cours des phases de traduction et d'optimisation. Ces graphes orientés ont la particularité de ne pas présenter de cycles orientés entre les différents noeuds le composant. Ainsi, à partir d'un noeud de départ, il sera impossible de revenir sur ce noeud en suivant le graphe. Cette représentation permet notamment de simplifier les étapes d'analyse du chemin critique et d'évaluation de l'arbre. Elle est aujourd'hui largement utilisée dans le domaine de la compilation.

Ces DAGs permettent de représenter les blocs de base à traduire et l'explicitation des calculs communs, conduisant ainsi à un partitionnement plus aisé du graphe dans l'optique d'une association des instructions manipulées avec les instructions cibles (sous la même forme dans le cadre de phases d'optimisation ou sous la forme de nouvelles instructions dans le cadre de phases de traduction). Cette association constitue un problème NP-complet à résoudre. Ces blocs de base correspondent chacun à un sous-graphe du graphe de référence, associés entre eux afin de former le graphe orienté acyclique de l'application. L'intérêt de cette représentation sous forme de graphe est de permettre la réutilisabilité des blocs à manipuler.

## 2.4 Émergence de la compilation dynamique dans les systèmes embarqués

La compilation dynamique est aujourd'hui largement déployée pour les architectures de type station de travail ou serveur à travers l'ensemble des technologies préalablement

présentées. Les performances atteintes, aussi bien pour l'interopérabilité que pour la compilation multi-étages, en font un élément incontournable utilisé au quotidien dans les applications, pour les raisons évoquées en introduction de ce manuscrit. Un des éléments clés ayant permis l'obtention de telles performances sur ces cibles est l'importance de leurs ressources mémoires et l'optimisation de leur gestion (différents niveaux de caches, volume important de cache, etc.). Ces ressources ont permis le développement de systèmes intelligents visant à masquer les latences introduites par l'utilisation de la compilation dynamique, avec par exemple des systèmes de caches de traduction qui ont été mis en place dans l'optique de conserver en mémoire les portions de code fréquemment compilées dynamiquement. Les architectures embarquées ne disposent pas de ressources mémoires aussi développées et il devient par conséquent beaucoup plus difficile de cacher les latences introduites par l'utilisation de la compilation dynamique, latences accentuées par son utilisation sur des processeurs moins polyvalents et aux performances brutes inférieures.

Une tendance naturelle se dégageant ces dernières années, et déjà évoquée en introduction, est **la migration progressive de l'ensemble des applications initialement destinées aux processeurs généralistes vers l'embarqué**, poussée notamment par l'émergence des smartphones. Les applications web, les machines virtuelles ou encore les applications de vision très sensibles au jeu de données sont largement déployées sur ces architectures aux contraintes beaucoup plus fortes au niveau des efficacités énergétique et surfacique. **Cette migration a conduit à celle des technologies de compilation dynamique** avec pour optique d'obtenir les mêmes bénéfices de par son utilisation, majoritairement pour des besoins de virtualisation mais aussi pour des besoins de performances pour les applications fortement dynamiques et sensibles à leur jeu de données.

### 2.4.1 Développement de la virtualisation dans les systèmes embarqués

L'augmentation des fréquences de fonctionnement et le parallélisme au niveau instructions sont aujourd'hui peu à peu délaissés en faveur de l'accroissement du nombre de **ressources parallèles** sur les architectures. Le développement d'architectures multi-coeurs est largement engagé afin de répondre aux besoins croissants en performances des applications. Bien que le nombre de coeurs sur ces architectures soit amené à passer à plusieurs centaines voir milliers dans les dix prochaines années [106], il n'existe semble-t-il pas de consensus clair sur les choix architecturaux les concernant, en particulier au niveau de leur homogénéité. Les architectures généralistes ont historiquement conduit au développement d'architectures multi-processeurs symétriques, tandis que les architectures embarquées ont clairement adopté **une approche plus hétérogène pour la duplication des ressources** [107], dans l'optique d'un meilleur contrôle de la consommation électrique.

**L'hétérogénéité des ressources de calcul** est un moyen naturel de s'adapter aux différents besoins en ressources de calculs des applications. Alors que les architectures généralistes adoptent progressivement cette approche en combinant par exemple des processeurs généralistes et des processeurs graphiques, les architectures embarquées l'ont depuis longtemps adoptée, combinant par exemple ressources généralistes, ressources graphiques, ressources de traitement de signal ou encore accélérateurs matériels au sein d'une même puce. Les contraintes induites au niveau des efficacités énergétique et surfacique, très fortes dans le domaine embarqué, sont un argument fort en faveur de la duplication de ressources spécialisées en fonction de la charge de calcul considérée, dans l'optique d'offrir "les bonnes ressources à la bonne application".

Les applications aujourd'hui développées pour l'embarqué couvrent un spectre de plus en plus large de domaines de calculs, mêlant par exemple phases de contrôle et

de calculs intensifs. Nous avons ainsi vu l'émergence rapide **d'architectures hétérogènes parallèles asymétriques** (*Asymmetric Many-core Processors*, AMP) dans le domaine des systèmes embarqués [108, 109]. De nombreux industriels développent aujourd'hui ces AMPs, parmi lesquels il est possible de citer Texas Instruments, STMicroelectronics, Qualcomm ou encore NVidia. Leur nombre est aujourd'hui en constante augmentation, chaque constructeur proposant un large panel d'architectures différentes au cycle de vie toujours plus court. Ce phénomène est parfaitement illustré par la situation actuelle dans le domaine des Smartphones. Le développement de telles solutions requière le développement de chaînes de compilation spécifiques et/ou de nouveaux modèles de programmation. Par conséquent, **la portabilité et le déploiement des applications** sont devenus des enjeux majeurs sur ces architectures. Ces enjeux sont également accentués par l'accroissement de la durée de vie des applications, qui doivent donc pouvoir être facilement portées d'une cible à l'autre. Même si les opportunités en termes de performances offertes par ces architectures AMPs sont importantes, les coûts engendrés en développement sont considérables et rendent difficile une exploitation optimale de celles-ci.

La nécessité d'introduire **une couche d'abstraction** entre l'applicatif et l'architecture, permettant d'exploiter cette dernière de manière optimale, s'est donc rapidement présentée [110]. Suivant l'évolution des machines virtuelles préalablement présentées, ces couches de virtualisation ont tout d'abord fait appel à l'interprétation afin d'exécuter à la volée le code de l'application. Les besoins croissants en performances ont conduit à l'utilisation **de solutions de compilation dynamique**. Certaines d'entre elles, comme OpenCL [37], l'utilisent afin de s'adapter à l'architecture cible, optimisant le déploiement de l'application (versions de code en fonction du jeu d'instructions de la cible). D'autres utilisent la compilation dynamique afin de spécialiser et de compiler dynamiquement certaines portions de code [68, 111], en profitant des informations disponibles à l'exécution pour effectuer d'importantes optimisations sur le code. Enfin, certaines machines virtuelles comme les JVM proposent d'exécuter des applications dans des représentations indépendantes de la cible, en code à octet par exemple, utilisant la compilation dynamique pour en générer le code machine équivalent en fonction de la cible et du contexte [39]. Ces machines virtuelles, identiques à celles utilisées sur les architectures de type station de travail ou serveur, permettent la gestion de langages à haut-niveau d'abstraction comme le Java, C#, et même les langages de scripts comme Javascript ou Python. Un exemple de machine virtuelle très utilisée aujourd'hui dans le contexte des systèmes embarqués est la machine virtuelle Dalvik [6], utilisée par Google sous Android pour la gestion des applications, incluant un compilateur dynamique depuis 2010.

Aux contraintes préalablement présentées sur l'utilisation de la compilation dynamique, viennent s'ajouter **les contraintes inhérentes aux systèmes embarqués** pour une utilisation sur ces plateformes, notamment au niveau de la gestion de la consommation, des performances et de l'empreinte mémoire. L'impact de la compilation dynamique sur ces architectures est aujourd'hui un enjeu majeur en raison des surcoûts qu'elle engendre sur ces contraintes, bien plus significatifs sur ces architectures [13, 45] et limitant son attractivité. Cet impact est d'autant plus prononcé avec le fort accroissement de la complexité de ces architectures : multiplication des jeux d'instructions à gérer, manque d'unification des modèles de programmation, etc. De plus, la hiérarchisation limitée de la mémoire sur ces architectures ne suffit pas à compenser les latences introduites.

### 2.4.2 Optimisations existantes de la compilation dynamique dans l'embarqué

Dans l'optique de rendre l'utilisation de la compilation dynamique attractive sur les systèmes embarqués, différents projets ont porté sur la mise en place d'optimisations pour sa gestion par ces systèmes. Ces optimisations sont essentiellement de deux types : celles conduites **au niveau logiciel**, sur les cadriciels en eux-mêmes, et celles conduites **au niveau système**.

#### Optimisations logicielles des cadriciels

**LLVM** est probablement l'exemple le plus important de la mise en place d'optimisations logicielles au niveau des cadriciels. Afin de rendre son générateur de code aussi efficace que possible, les développeurs de LLVM ont concentré leur efforts sur l'optimisation du code au coeur même du compilateur. Ces optimisations, sur lesquelles nous reviendrons plus en détails dans la suite du manuscrit, se focalisent sur la gestion des phases d'optimisation et de traduction de la représentation intermédiaire en code machine, notamment au niveau de la manipulation même des données, de leur accès et de leur représentation, dans l'optique de les invoquer le plus rapidement possible et avec la complexité la plus faible. D'autres optimisations algorithmiques ont également été développées, notamment en ce qui concerne l'optimisation des services des machines virtuelles Java pour l'embarqué [85, 46, 47].

Une étude plus théorique, réalisée par Erven Rohou et Albert Cohen, consiste en la mise en place de la notion **de compilation scindée** (*split compilation*) [22]. L'objectif de ces travaux est de mettre en avant la nécessité d'un changement d'approche lors de la conception des compilateurs pour la virtualisation des processeurs et la gestion simultanée de plusieurs jeux d'instructions. Ce changement d'approche permet d'assurer une meilleure communication et surtout de profiter des avantages de chacune des différentes étapes de la vie d'un programme pour en optimiser continuellement les performances par compilation (statique, dynamique, etc.). Les phases d'optimisations sont virtuellement étendues à l'ensemble des représentations manipulées et aux différentes phases de compilation et d'existence par un système d'annotations permettant de conserver le flot d'information et la portabilité. L'idée est, par exemple, d'introduire un système d'annotations sur la représentation intermédiaire permettant de réaliser plus rapidement les optimisations à l'exécution en réalisant les parties d'analyses les plus coûteuses au préalable, hors ligne. Une des pistes explorées concerne notamment les optimisations relatives à la vectorisation automatique en fonction des ressources utilisées.

Alors que la vitesse des compilateurs dynamiques n'a cessé de croître ces dernières années, certaines études pour les machines virtuelles sur cibles embarquées tentent d'en limiter les invocations en conservant un nombre croissant de portions de code machine optimisé généré en mémoire. Cette approche se heurte toutefois aux capacités mémoires plus faibles de ces architectures. Afin d'y remédier, des propositions de nouvelles méthodes de gestion de la mémoire pour ces portions de code ont été introduites [112, 113, 13]. Ces propositions visent à mettre en place **des systèmes de blocs de caches sophistiqués** permettant de conserver les portions de code machine les plus fréquemment utilisées. Un système de compression permet également de stocker à moindre coût les portions moins fréquentes. Quant aux portions les moins utilisées, elles sont supprimées de ces blocs et sont générées à nouveau si nécessaire. L'intérêt de cette approche est de permettre un accès rapide aux portions critiques, tout en garantissant de conserver un nombre conséquent de portions en mémoire afin de limiter les besoins en compilation dynamique. Elle n'est utilisée que lorsqu'une portion de code n'est pas disponible ou que le gain potentiel en optimisant une portion existante

est suffisamment important pour justifier le lancement d'une nouvelle compilation de celle-ci.

Enfin, une dernière approche, explorée régulièrement dans l'optique de réaliser des optimisations dynamiques du code, est celle basée sur l'utilisation **de la spécialisation de code**, permettant de modifier le code en tirant notamment bénéfice de certaines information disponibles à l'exécution. Elle a notamment été adoptée dans [114] pour une utilisation dans le cadre d'une compilation dynamique de JavaScript, utilisant la spécialisation de code pour compléter les informations relatives au typage au sein du code en se basant sur les observations réalisées à l'exécution.

#### Optimisations des systèmes

L'objectif de ces optimisations est de permettre **d'affranchir les ressources normalement allouées à l'exécution de l'application des phases de compilation introduites**. Cette approche consiste à migrer les différentes technologies, sur leurs propres ressources, optimisées ou non pour leur gestion.

Pour les machines virtuelles en particulier, il s'agit de déporter leurs différents services sur **des ressources dédiées spécialisées**. Introduits initialement avec les premières machines LISP puis avec les processeurs Java au début des années 2000, comme ARM Jazelle [23] ou encore JOP [24], ces travaux ont très vite souffert de leur manque de flexibilité, ne pouvant cibler respectivement que les applications LISP et Java. De plus, la réalisation matérielle des différentes phases d'optimisation et de traduction n'a jamais été réellement abordée par ces études, se contentant essentiellement d'accélérer l'exécution de ces applications par un jeu d'instructions spécialisées (pour JOP et la première version de Jazelle), optimisé pour ces représentations, et d'accélérer les phases d'interprétation de ces mêmes applications (pour la seconde version de Jazelle), toujours grâce à un jeu d'instructions spécialisées. Aujourd'hui ne subsiste réellement pour ces travaux que Jazelle, toujours implémenté par ARM sur ces processeurs, et dont l'utilisation semble rester marginale, ou en tout cas sans réelle communication sur le sujet.

Il est également possible de citer l'approche adoptée par Ildjit dans une optique d'optimisation des systèmes. En effet, celui-ci propose d'utiliser **le parallélisme des architectures** afin de confier aux ressources non-allouées à l'exécution (processeur en mode de veille) les phases de compilation, limitant ainsi l'impact de celles-ci sur l'exécution de l'application. Toutefois, aucune étude de la gestion de l'impact de cette assignation de tâches de compilation aux ressources n'est réellement abordée par les développeurs du projet. Ainsi, la mise en indisposition de ressources normalement en veille pourrait avoir des conséquences sur l'exécution de l'application si celle-ci venait subitement à avoir besoin de ressources supplémentaires. De plus, cette approche peut également présenter un impact non négligeable sur la gestion des ressources mémoires de l'architecture, en particulier dans l'embarqué. Enfin, et toujours dans le cadre particulier d'une exécution sur cible embarquée, l'utilisation de ressources normalement en veille peut avoir d'importantes conséquences sur la consommation et sur l'évolution thermique de l'ensemble de la cible (utilisations d'importantes ressources de calcul sous-optimales pour gérer la compilation dynamique par exemple).

Transmeta Crusoe, à l'image des processeurs Java, fournit lui aussi **un jeu d'instructions optimisé** pour la traduction dynamique binaire des programmes x86 vers leur cible VLIW. Ce processeur a été développé par la société Transmeta à la fin des années 90 et au début des années 2000, sans connaître toutefois un réel succès commercial. Comme mentionné préalablement, ce processeur permet l'exécution transparente d'applications en code machine x86 sur leur processeur. Afin de masquer les latences introduites par les phases de traduction, le processeur dispose de ses propres instructions

permettant d'accélérer ces phases. Ces instructions bénéficient à la machine virtuelle Code Morphing Software (CMS) utilisée par Crusoe pour assurer l'émulation des applications x86 et ainsi la traduction dynamique binaire.

Une approche plus récente est celle présentée par Ting Cao et al. dans [115]. Elle propose la réalisation d'une solution similaire à celle des processeurs Java mais en introduisant une version plus flexible de celle-ci. Pour cela, il n'est plus question de déporter la totalité des services des machines virtuelles sur des ressources dédiées spécialisées, mais uniquement les plus génériques d'entre eux sur **des ressources dédiées standards** (processeurs existants). Ainsi, les services de ramasse-miettes, d'interprétation et de compilation dynamique sont déportés sur des ressources minimalistes, ce qui rend cette approche assimilable à celle de ARM *Big.Little* : un processeur principal pour les applications les plus importantes et consommant davantage et un processeur secondaire pour les services et applications les moins demandeuses en ressources. Le postulat fait par Ting Cao est que ce genre de ressource secondaire est suffisante pour les services déportés des machines virtuelles et que cette approche permet d'obtenir déjà un gain significatif par rapport à un partage avec l'application sur les ressources qui lui sont normalement allouées.

Les résultats obtenus montrent une opportunité de réduction du surcoût induit par les services de virtualisation de 13 % et une accentuation des efficacités énergétique et surfacique de 22 %. Les résultats mettent également en avant que la majorité des gains sont obtenus sur les services de ramasse-miettes et d'interprétation. La compilation dynamique ne tire qu'un bénéfice très limité de cette ressource dédiée, malgré un potentiel important de gains (plus de code optimisé, avec des optimisations plus agressives). Les raisons invoquées par Ting Cao sont la forte demande en performances des codes de compilation dynamique, liée à leur complexité. Leur transfert sur des ressources de calcul contraintes limite la rapidité de la réalisation des différentes phases d'optimisations et de traduction. Ting Cao avance l'explication de la forte irrégularité et de la complexité de ces codes, sans toutefois en révéler la teneur, ce que nous tentons de montrer dans le prochain chapitre. Les efficacités énergétique et surfacique de ces processeurs s'en trouvent ainsi réduites avec l'exécution des codes de compilation dynamique.

## 2.5 Conclusion

Bien que destinées à des domaines applicatifs extrêmement variés, nous avons pu mettre en avant un certain nombre de convergences entre les différentes technologies de compilation dynamique, notamment au niveau algorithmique. Cette étape nous permet de considérer la compilation dynamique comme **un domaine algorithmique à part entière**, avec des briques logicielles et des enjeux communs entre les différentes technologies. Les performances obtenues sur les architectures de type station de travail ou serveur ont fait de la compilation dynamique un choix technologique incontournable et très utilisé aujourd'hui. Malgré un besoin croissant dans les systèmes embarqués, notamment avec le transfert massif d'applications très sensibles au jeu de données et des besoins grandissants en virtualisation sur ces architectures, un problème important de passage à l'échelle se pose. Alors que les architectures de type station de travail ou serveur utilisent leurs importantes ressources mémoires pour cacher les latences introduites par la compilation dynamique, avec notamment des systèmes de cache de traduction, les architectures embarquées ne disposent pas de telles ressources. De plus, certaines études mettent en avant les fortes demandes en performances des codes de compilation dynamique. Les processeurs embarqués, aux ressources de calcul contraintes, ne permettent donc pas d'exécuter suffisamment rapidement et de manière optimale ces codes, engendrant une baisse de leurs efficacités énergétique et surfacique. Toutefois, aucune

étude ne se penche réellement sur l'origine de ces problèmes à travers une analyse poussée des codes de compilation dynamique. C'est ce que nous nous proposons de réaliser dans le chapitre suivant, dans l'optique de mettre ensuite en place une optimisation de ses performances, en particulier pour les cibles embarquées.

# Chapitre 3

# Mise en évidence d'une caractérisation commune des codes de compilation dynamique

**Sommaire**



## 3.1 Introduction

Il a été démontré dans la partie précédente que la compilation dynamique pouvait être considérée comme un domaine algorithmique à part entière. Bien que les diverses technologies l'utilisant aient des objectifs différents, il est possible d'observer des briques logicielles communes entre elles. Dans ce chapitre, nous conduisons une analyse plus fine afin de déterminer s'il existe des similitudes entre les caractéristiques calculatoires (régularité, localité, etc.) de ces différentes briques. Nous cherchons, à travers cette étude, à démontrer si ces similitudes peuvent éventuellement justifier une approche commune pour leur optimisation.

Dans ce chapitre, nous nous proposons de réaliser une analyse détaillée des différentes technologies dans l'optique de comprendre les problèmes architecturaux (par opposition aux problèmes algorithmiques) conduisant à une dégradation des performances de la compilation dynamique. Cette analyse vise à bâtir **une caractérisation (ou signature) commune aux différents codes de compilation dynamique**. Grâce à cette signature, nous cherchons à identifier le type et la nature des optimisations réalisables et applicables à l'ensemble des technologies de compilation dynamique. Elle a également pour objectif d'établir les meilleurs choix envisageables en termes d'architecture matérielle et d'optimisation logicielle pour l'exécution de codes de compilation dynamique, en particulier dans le cadre des systèmes embarqués, aux ressources de calculs contraintes.

Une analyse comportementale de ces codes a été effectuée à grain fin, au niveau instruction. Ces travaux ont fait l'objet d'une publication au Workshop international DCE '13 [116], organisé dans le cadre de la conférence HIPEAC '13.

A la fin de ce chapitre, nous évaluerons également l'efficacité relative d'un processeur embarqué (de faible empreinte) à exécuter les codes de compilation dynamique, en comparaison avec des codes dits conventionnels.

## 3.2 Mise en place des séries de tests pour l'analyse

Afin de réaliser l'analyse des codes de compilation dynamique des différentes technologies mises en évidence dans le chapitre précédent, **un cadre expérimental** commun a été mis en place. Ces codes étant en effet destinés à des domaines applicatifs très différents, une base commune de comparaison a été imaginée, aussi bien au niveau logiciel qu'au niveau matériel. Cette partie a pour but de présenter et de justifier les travaux préliminaires de mise en place de ce cadre expérimental.

### 3.2.1 Sélection d'un panel représentatif de technologies de compilation dynamique

Quatre grandes technologies de compilation dynamique ont été mises en évidence dans les chapitres précédents : les machines virtuelles, la traduction dynamique binaire, la compilation multi-étages et les moteurs d'exécution pour les langages typés dynamiquement. Afin d'évaluer le comportement à grain fin de ces quatre technologies, nous avons procédé à la sélection d'un panel de compilateurs dynamiques représentatifs de ces technologies. Pour cela, **trois critères de choix** ont été fixés.

- Support : afin d'effectuer une comparaison entre les différentes technologies, il est indispensable d'offrir une base matérielle commune à celles-ci. Il est en effet difficilement concevable de comparer des résultats entre deux applications s'exécutant sur des architectures différentes. Nous avons donc orienté notre choix sur une architecture commune à l'ensemble des technologies. Bien que notre étude se destine au domaine des architectures embarquées, le choix de l'architecture Intel x86 a été retenu, car elle représente la seule architecture cible supportée par l'ensemble des technologies visées. Aussi, ce choix nous a permis de disposer d'un ensemble complet de compilateurs dynamiques de l'état de l'art.
- Représentativité : afin d'évaluer un comportement représentatif de la technologie ciblée, le choix du compilateur doit également être guidé par les choix technologiques internes qui ont été effectués dans son développement. L'objectif est de disposer d'un compilateur conçu selon les grands principes de la technologie qu'il représente et qui ont été présentés dans les chapitres précédents.
- Accessibilité : il a été décidé de réaliser certaines modifications au sein des codes

de compilation dynamique afin d'en isoler certaines parties. Pour cela, il est indispensable de disposer de projets ouverts et surtout suffisamment modulaires, dans lesquels les choix d'implémentation effectués offrent une bonne flexibilité au niveau de l'utilisation des noyaux du code.

Basés sur ces caractéristiques, nous avons fait le choix des projets suivants, utilisant des technologies de compilation dynamique ou fortement apparentées :

- JikesRVM [42] pour les machines virtuelles. Cette JVM, développée sous licence Open Source, a été conçue dès 1997 par les ingénieurs d'IBM. Initialement utilisée en interne, elle est librement accessible depuis sa seconde version datant d'octobre 2001. Comme la plupart des JVMs, elle inclut à la fois un interpréteur et un compilateur dynamique pour les portions critiques de l'application Java à exécuter.
- Ildjit [50], LLVM [8] et LibJIT [67] pour la compilation multi-étages. LibJIT est une librairie proposant un ensemble de routines pour la compilation dynamique. Elle sert de bloc de base dans la conception de noyaux de compilation dynamique. Ildjit est un exemple de projet utilisant cette librairie. Elle a été intégrée dans les choix effectués car elle est assimilable à une version minimaliste de compilateur dynamique. Un certain nombre de petits noyaux sont fournis afin de pouvoir en tester le fonctionnement.
- Qemu [56], Valgrind [61] et un traducteur dynamique binaire pour la génération de fonctions de hachage réalisé dans notre laboratoire, nommé HashGenerator [36], ont été sélectionnés pour la traduction dynamique binaire.
- Le code de l'algorithme typeset, extrait des tests comportementaux miBench [102], pour l'interprétation, avec un comportement que nous estimons proche des moteurs d'exécution pour les langages typés dynamiquement, puisqu'il s'agit de générer à l'exécution des fichiers Postscript à partir d'une description à haut niveau d'abstraction [117].

Nous n'avons pu, pour des questions d'accessibilité et de temps de réalisation, nous focaliser sur les moteurs d'exécution JavaScript tels que Google V8, IonMonkey de Mozilla ou encore SPUR de Microsoft.

### 3.2.2 Isolation des noyaux de compilation au sein de ces technologies

Après la sélection du panel représentatif des technologies de compilation dynamique, il a été décidé de faire un choix différent de celui effectué par la majorité des études réalisées sur le sujet [118]. Contrairement à ces dernières, nous ne cherchons pas à analyser le comportement d'une application utilisant la compilation dynamique, en comparaison avec la même application compilée statiquement. Nous cherchons ici à analyser **le comportement du compilateur dynamique en lui-même**, l'application ne constituant qu'un ensemble de code permettant de le solliciter.

Le problème dans notre cadre d'étude est l'entrelacement entre les phases d'exécution et les phases de compilation. Nous avons par conséquent décidé de modifier les codes des compilateurs testés afin **d'isoler ces différentes phases**. L'objectif est ainsi de ne profiler que les phases relatives à la compilation. Les modifications nécessaires ont donc été apportées au niveau des compilateurs afin que ceux-ci ne réalisent pas les phases d'exécution après celles de compilation. Certains compilateurs, comme LLVM, propose déjà la possibilité de n'utiliser que leur générateur de code, sans raccordement en mémoire et sans exécution du code ensuite (utilisation de LLC, *Low-Level Compiler*). Pour d'autres, comme Ildjit, une modification du code a été nécessaire au niveau du moteur d'exécution afin d'en retirer les étapes d'exécution.

Afin "d'alimenter" ces compilateurs en code à compiler, nous avons choisi différents noyaux, extraits d'applications, dont le compilateur devra générer le code machine

équivalent. Puisque nous nous sommes affranchis des phases d'exécution, l'influence de ces codes sur le comportement du compilateur dynamique s'en trouve relativement limitée. Cet aspect est vérifié dans la suite de ce chapitre à travers l'analyse des résultats obtenus pour les compilateurs sur lesquels plusieurs portions de code ont été compilées. Ces applications ont été choisies selon les critères suivants : leur possibilité d'utilisation sur le projet (par exemple pour Ildjit et LLVM la possibilité d'en générer respectivement le code CIL et LLVA) et leur représentativité de codes potentiellement compilables dans le cadre de la compilation dynamique sur le projet ciblé. Les choix réalisés en fonction des projets sont présentés dans le tableau 3.1.

TABLE 3.1 – Projets de compilation dynamique utilisés pour l'analyse comportementale.

| Compilation dynamique | |
|---|---|
| Valgrind | fft |
| Qemu | Linux booting |
| Ildjit compilation | fft<br>adpcm |
| LLVM compilation | "Hello World"<br>loop algorithm<br>adpcm<br>fft |
| LibJIT | Test kernels |
| JikesRVM (JVM) | "Hello World"<br>Euclide<br>table sorter |
| HashGenerator | hash code |
| Typeset | lout file |

### 3.2.3 Sélection d'un panel représentatif de codes, dits "conventionnels"

Dans l'optique d'extraire des caractéristiques comportementales communes des codes de compilation dynamique, il est nécessaire **de comparer leur comportement à un référentiel commun**. Pour ce référentiel, nous avons choisi **un ensemble d'algorithmes, dits "conventionnels"**, largement utilisé comme ensemble de tests dans la communauté, et dont les spécificités des codes tranchent avec la complexité de ceux de compilation dynamique. Les codes sélectionnés sont ceux de l'ensemble miBench [102], dont les domaines applicatifs varient du traitement d'image aux télécommunications en passant par la sécurité et les codes de tri. Le tableau 3.2 présente les codes utilisés dans notre étude en fonction de leur domaine applicatif (selon la catégorisation miBench).

## 3.3 Mise en place des outils pour l'analyse

Cette section est dédiée à la réalisation **d'une caractérisation des codes de compilation dynamique**, basée sur l'observation de compteurs de performances. Les statistiques obtenues à l'aide de ces compteurs doivent nous permettre d'identifier les problèmes de performances communs aux différents codes de compilation dynamique. Notre objectif est ainsi de déterminer les meilleures stratégies d'optimisation pour ces codes au niveau de leurs performances et de leur portabilité.

Afin de réaliser les mesures relatives aux différents compteurs de performances, nous avons fait appel à l'utilisation de méthodes standards, basées par exemple sur le

TABLE 3.2 – Codes conventionnels des miBench utilisés pour l'analyse comportementale.

| miBench | |
|---|---|
| Automotive | bitcount<br>qsort<br>susan |
| Consumer | JPEG |
| Network | dijkstra<br>patricia |
| Office | stringsearch |
| Security | blowfish<br>pgp<br>sha |
| Telecom | adpcm<br>crc<br>fft |

**cadriciel Valgrind** et ses différents outils. Nous avons également mis spécifiquement en place un certain nombre d'outils, basés sur **des simulateurs d'instructions** existants. Ces simulateurs ont été instrumentés pour récupérer des informations relatives aux performances des codes de compilation dynamique. L'ensemble de ces outils et de cette méthodologie a pour but de nous permettre d'évaluer les caractéristiques importantes en termes de performances, comme **la régularité et la localité**.

### 3.3.1 Présentation de l'outil Valgrind

Valgrind [61] est un cadriciel d'instrumentation, basé sur l'utilisation de la traduction dynamique binaire et faisant appel à la compilation dynamique. Il réalise l'analyse dynamique du code d'une application au niveau instructions. Il permet notamment le débogage, le profilage et l'évaluation de la gestion mémoire du code d'une application. L'émulation réalisée par Valgrind est équivalente à une exécution mono-tâche de l'application, sans parallélisation. Sous licence Open Source, il est aujourd'hui utilisable sur un large panel d'architectures, allant de x86 à ARM en passant par AMD64 et PowerPC, avec les systèmes d'exploitation Linux, Android ou Mac OS X.

Ce cadriciel dispose d'un certain nombre d'outils parmi lesquels **Cachegrind**, dont la fonction est de simuler l'interaction du programme avec les **différents niveaux de mémoire cache** et **le prédicteur de branchement** de la cible. Le prédicteur de branchement simulé par Cachegrind est un prédicteur aux capacités équivalentes à ceux disponibles sur les architectures Intel x86, à mi-chemin entre celui du Pentium 4 et celui des processeurs Intel Core 2 duo.

Concernant les caches d'instructions et de données, Cachegrind s'adapte au processeur hôte sur lequel il est exécuté, décodant le `CPUID` afin de déterminer la taille des deux caches. Pour les niveaux de cache simulés, Cachegrind s'adapte ici aussi au processeur hôte en ne simulant que le premier niveau (les caches d'instructions et de données) et le dernier niveau, notamment pour les processeurs à trois niveaux de cache. Ce choix est justifié par la prédominance de l'impact de ces deux niveaux : le premier influant directement sur la localité du code et le dernier influant sur les communications avec la mémoire en cachant les latences des accès moins fréquents.

### 3.3.2 Présentation du simulateur instrumenté

Pour augmenter le nombre de compteurs de performances que nous analysons nous avons **instrumenté un simulateur de jeu d'instructions x86** (par soucis d'homogénéité avec l'outil Valgrind). Ce simulateur de jeu d'instructions x86, implémenté grâce au générateur de simulateur GenISSLib [119] a été instrumenté pour donner accès à des métriques supplémentaires, non disponibles dans les outils classiques tels que Valgrind, Qemu ou Pin (simulateur instrumenté de Intel). En particulier, les nouvelles instrumentations ont été mises en oeuvre pour **caractériser la complexité des chemins de données** : dépendances de données et profondeurs d'indirection des accès mémoires.

L'instrumentation du simulateur x86 a consisté à marquer chaque donnée échangée par le processeur depuis les registres vers la mémoire et inversement. Grâce à ce traçage des données, le simulateur peut, par exemple, déterminer la complexité des générations d'adresses lors d'un accès mémoire. Pour chaque adresse utilisée lors d'un de ces accès, le simulateur peut identifier si les sources à l'origine du calcul d'adresse proviennent de données constantes, ou si elles proviennent elles-mêmes d'accès mémoires antérieurs (indirection). La profondeur et la quantité d'indirections sont des informations cruciales pour déterminer si le chemin de données est statique ou fortement dépendant des données elles-mêmes. Cette information permet de notamment de conclure sur la prédictibilité des accès mémoires.

## 3.4 Caractérisation du flot de contrôle des codes de compilation dynamique

Dans cette section, nous nous focalisons sur **l'étude du flot de contrôle** des codes de compilation dynamique. La régularité (ou l'irrégularité) au niveau de ce flot est une information indispensable dans l'optique de déterminer le type d'optimisations applicables (parallélisation à grain fin, mise en place d'opérateurs matériels dédiés, etc.) ainsi que la géométrie des ressources de calcul idéales, ou en tout cas les plus en adéquations avec les spécificités de ces codes.

Nous nous sommes donc intéressés dans cette partie à l'évaluation de la gestion du contrôle pour les codes de compilation dynamique en comparaison avec celle des codes conventionnels. Pour cela, nous avons relevés des compteurs de performances standards dans l'optique de mettre en avant l'irrégularité du flot d'instructions sur ces codes de compilation. Ces compteurs sont relatifs aux branchements présents dans les codes. En particulier, nous nous sommes intéressés à leur nombre et aux taux d'erreurs au niveau du prédicteur de branchement.

Grâce à cette évaluation, nous pouvons juger de la prédictibilité des codes et par extension de leur régularité. Pour la réaliser, nous nous sommes basés sur l'outil Cachegrind de Valgrind.

### 3.4.1 Présentation des mesures réalisées et des résultats obtenus

Dans l'optique d'analyser la gestion du contrôle dans les codes de compilation dynamique, en comparaison avec les codes conventionnels, deux mesures ont été réalisées. Elles portent sur **la quantité de branchements** et sur **la prédictibilité du flot de contrôle** en lui-même, avec l'évaluation des taux d'erreurs du prédicteur. Elles concernent aussi bien les branchements directs que les branchements indirects.

Bien que sur l'ensemble des codes, les erreurs de prédiction de branchement et leur impact sur les performances soient relativement faibles pour un processeur super-

scalaire sophistiqué, ces erreurs peuvent devenir très significatives sur des processeurs embarqués dont les prédicteurs de branchement sont, en général, bien plus simples.

**Les erreurs de branchement sur les branchements indirects**, dont nous verrons par la suite qu'elles augmentent significativement sur des codes de compilation dynamique, révèlent une nature de code généralement peu sujette à des optimisations de type flot de données à grain fin. Ces erreurs sont caractéristiques des codes dit polymorphiques ; codes dont le flot de contrôle ne se limite plus à de simples tests de type `if-then-else`, mais à des constructions plus complexes, de types `switch-case`, pointeurs de fonction et polymorphisme de données dans le cas de codes orientés objet. En effet, les branchements indirects, contrairement aux branchements directs qui contiennent directement l'information de l'adresse de la prochaine instruction à exécuter, nécessitent la récupération de l'adresse avant de pouvoir procéder au branchement, par exemple en mémoire ou dans un registre.

L'adresse d'un branchement indirect n'est donc pas connue avant l'exécution de l'instruction et dépend fortement du contexte d'exécution. Une erreur sur ce type de branchement nécessite une évaluation de l'instruction cible qui, si elle varie souvent, peut rendre impossible le travail du prédicteur de branchement (en particulier du prédicteur de d'adresse cible ou BTB, *Branch Target Buffer*). Ces erreurs de branchement entraînent des latences pouvant être équivalentes, voir supérieures, à celles d'accès à la mémoire de données. Les processeurs les plus simples (de type processeurs embarqués) sont en général dotés de moyens limités pour gérer ce type de branchement, mais leur pipeline plus court limite la dégradation des performances.

Enfin, nous montrons dans les résultats que la proportion **de branchements conditionnels et indirects**, par rapport aux autres instructions exécutées, augmente significativement pour les codes de compilation dynamique. Ce constat révèle à nouveau la présence d'un flot de contrôle plus irrégulier que pour les codes conventionnels, rendant la parallélisation statique à grain fin plus complexe.

### 3.4.2 Évaluation des résultats

Les analyses réalisées sur les différentes technologies de compilation dynamique et sur les codes conventionnels préalablement sélectionnés ont permis de mettre en avant un certain nombre de résultats permettant de montrer le comportement commun recherché entre les différentes technologies. Les résultats concernant les ratios de branchements directs et indirects avec le nombre total d'instructions exécutées révèlent **une légère augmentation du nombre de ces branchements indirects** pour les codes de compilation dynamique, en comparaison avec les codes conventionnels. Ce fait est la conséquence de la forte utilisation par ces codes d'éléments de programmations tels que les `switch-case`, les structures conditionnelles et/ou le polymorphisme de données.

La figure 3.1, présente l'évolution des proportions de branchements indirects et conditionnels entre les codes conventionnels et les codes de compilation dynamique. En particulier, nous observons que la proportion de branchements indirects double (2.5×) dans les codes de compilation dynamique : 1 instruction sur 100 est un branchement indirect contre environ 1 sur 250 dans les codes dits conventionnels. Les branchements conditionnels augmentent eux aussi de 1.5× : 1 instruction sur 6 est un branchement conditionnel contre 1 instruction sur 10 dans les codes conventionnels.

Concernant ces branchements, la différence la plus significative réside dans **l'accroissement des erreurs de prédiction pour les branchements indirects**, comme illustré sur la figure 3.2. Le ratio moyen d'erreurs de prédiction passe de 1 pour 20000 instructions exécutées pour les codes conventionnels, à 1 pour 1000 pour les codes de compilation dynamique. Cela représente une multiplication par 20 des erreurs de prédiction pour ces branchements indirects. Nous pouvons observer sur cette figure que la

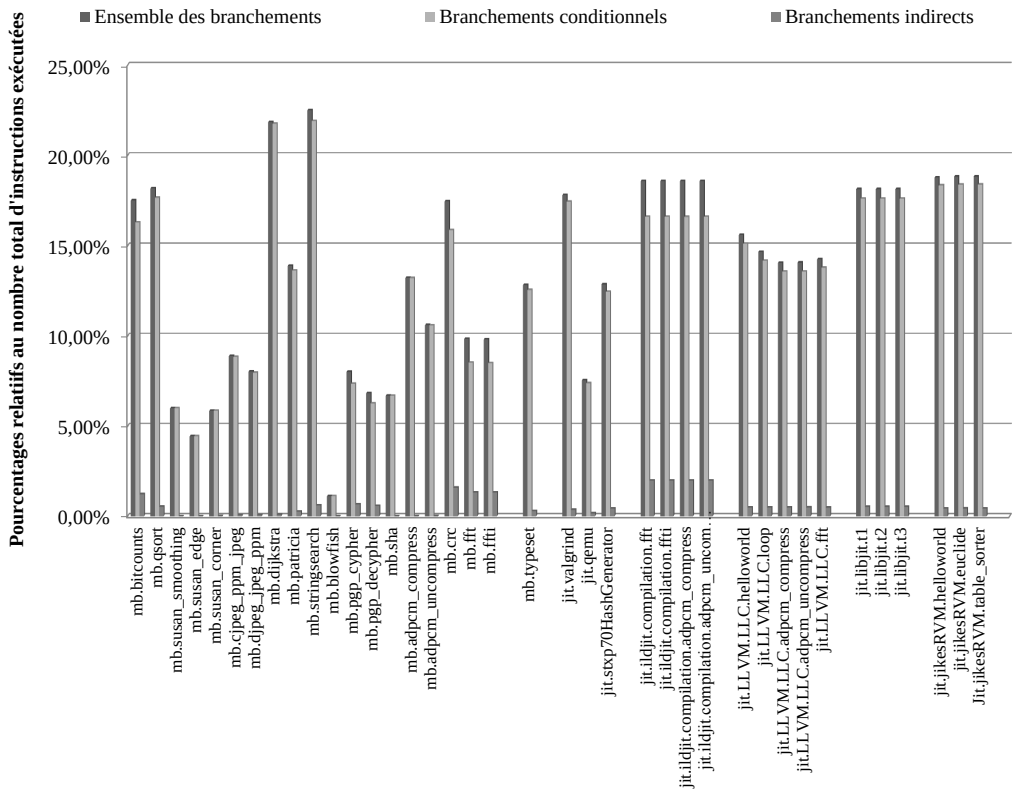



FIGURE 3.1 – Évolution des branchements directs et indirects dans les codes conventionnels et dans les codes de compilation dynamique.

tendance, bien que variant de par son ampleur d'une technologie à une autre, met en avant **un comportement commun des codes de compilation dynamique** à ce niveau, en comparaison avec les codes conventionnels.

Une exception est toutefois visible pour le code de l'algorithme patricia, qui présente lui aussi un fort ratio d'erreurs de prédiction pour les branchements indirects. S'agissant d'un code de gestion d'arbres pour structures de données, il manipule une forte quantité de données abstraites (par opposition à des données naturelles et/ou scalaires), expliquant ce phénomène.

Un autre point remarquable est **la stabilité des résultats** dans le cas d'une sollicitation des compilateurs dynamiques avec différentes applications à compiler. Cela montre les limites de la sensibilité des compilateurs aux applications dont ils sont chargés de la compilation, l'influence de celles-ci étant quasiment nulle sur leur évolution comportementale.

Il est important de souligner que même si les écarts entre codes conventionnels et codes de compilation sont importants, les pourcentages mis en avant restent faibles. Toutefois, ces erreurs de prédictions de branchement sont calculées à l'aide de l'outil Valgrind, modélisant un prédicteur de branchement très performant (prédicteur G-share 16 KO éléments avec 2 bits saturés pour les branchements conditionnels ; prédicteur d'adresse de branchement de 512 éléments, indexés selon les 9 bits de poids faible de l'adresse, pour les branchements indirects).

### 3.4.3 Conclusion sur l'analyse du flot de contrôle

Les résultats précédents mettent en avant le problème de la gestion des branchements indirects pour les codes de compilation dynamique. Ce problème a déjà été

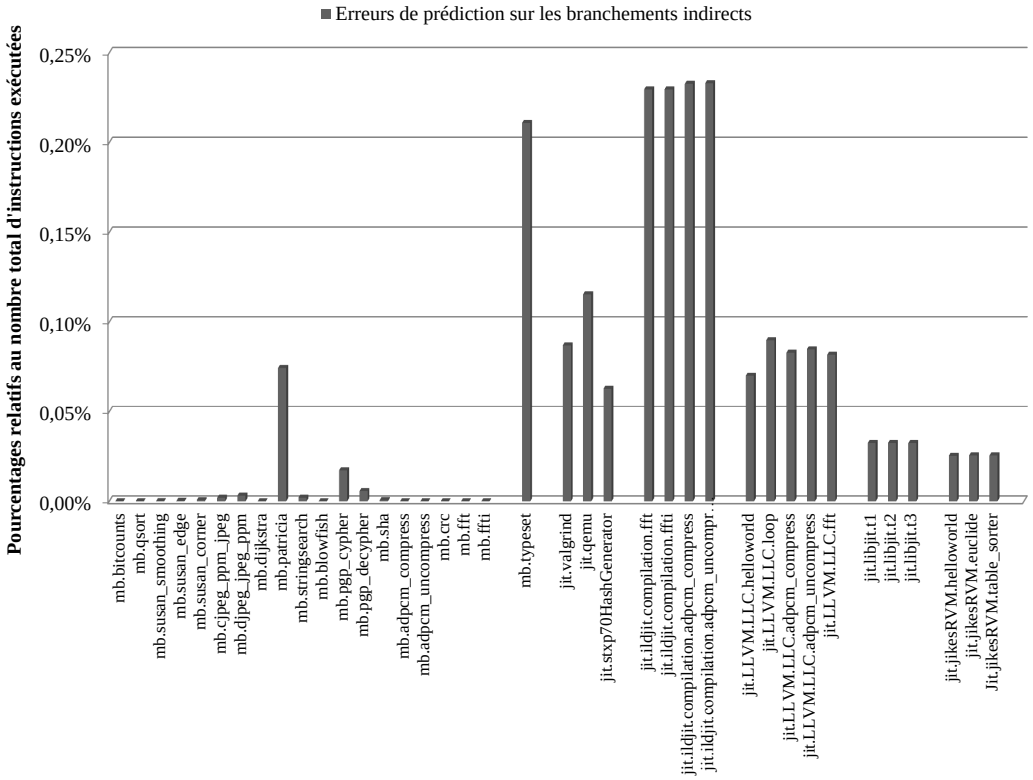



FIGURE 3.2 – Évolution des erreurs de prédiction pour les branchements indirects dans les codes conventionnels et dans les codes de compilation dynamique.

mentionné dans plusieurs études comme celles réalisées par Hiser et al. dans [120], par Casey et al. dans [121] et Li et al. dans [122], spécifiquement pour l'interprétation du code à octet pour les machines virtuelles.

Les branchements conditionnels, sont également plus nombreux et moins bien prédits. Ainsi, **l'ensemble du flot de contrôle est sensiblement plus complexe** sur les codes de compilation dynamique. Ce problème est d'autant plus critique que les architectures de processeurs généralement adoptées dans les systèmes embarqués sont plus simples et ne disposent pas de prédicteurs aussi sophistiqués que ceux des processeurs généralistes, pour des questions de surface et de consommation. Il y aura par conséquent une accentuation des effets mesurés, augmentant donc les pénalités induites en performances par l'utilisation de la compilation dynamique.

## 3.5 Caractérisation des manipulations de données dans les codes de compilation dynamique

Nous nous proposons maintenant de nous intéresser à **l'étude des accès mémoires** dans les codes de compilation dynamique. Nous cherchons dans cette section à caractériser ces accès, aussi bien du point de vue de la localité que de leur nature.

**La première mesure** réalisée porte sur **la localité des données** et la gestion des caches d'instructions et de données. Les données collectées portent notamment sur les erreurs de lectures et d'écritures sur ces caches (*cache miss*). Ces erreurs peuvent être de trois types :

- erreur de lecture du cache d'instructions ;
- erreur de lecture du cache de données ;

– erreur d'écriture du cache de données.

Les erreurs de lecture du cache d'instructions sont particulièrement problématiques puisqu'elles suspendent complètement l'exécution du programme le temps du chargement et du décodage de la bonne instruction à partir de la mémoire. Ces erreurs peuvent engendrer des latences proportionnelles à la longueur du pipeline. De plus, elles ne peuvent être recouvertes que difficilement, le processeur n'étant plus alimenté en instructions. Il est également possible de distinguer quatre catégories de défauts de cache : les défauts obligatoires (premières demandes), les défauts capacitifs (débordement de cache), les défauts conflictuels (conflits d'adresses) et les défauts de cohérence (pour les architectures multi-processeurs, ce qui n'est pas le cas de l'émulation réalisée par Valgrind).

**La seconde mesure** porte, quant à elle, sur **les profondeurs d'indirection**, traduisant la nature des accès aux données.

Le volume d'instructions nécessaire à la manipulation des données des codes de compilation dynamique étant bien plus important que pour les codes conventionnels, nous cherchons, dans cette partie, à mettre en avant une complexité commune dans la manipulation de ces données entre les différents codes de compilation. De plus, contrairement aux codes conventionnels qui manipulent des données, le plus souvent, naturelles et/ou scalaires, les codes de compilation gèrent, quant à eux, beaucoup plus **de structures de données dites abstraites** (arbres, graphes, énumérations, etc.). Ces structures contiennent notamment les portions de code à traduire et l'ensemble des informations nécessaires aux différentes phases de compilation. Elles représentent, par exemple, les différents blocs de base de l'application à compiler.

Ces structures de données sont manipulées au sein de graphes, le plus souvent des graphes orientés acycliques, comme mentionné dans le chapitre précédent (Chapitre 2. L'ensemble des opérations réalisées sur ces structures (annotations, tri, modifications, multiples accès, etc.) vont donc conduire à une multiplication des manipulations de ce graphe. Nous cherchons dans le cadre de cette étude à évaluer la complexité de la gestion de ces graphes qui représente le coeur de la manipulation des données au sein d'un compilateur dynamique.

### 3.5.1 Présentation des mesures réalisées et des résultats obtenus

#### Mesures relatives aux caches

Cette mesure porte **la localité du code de compilation et des données manipulées** au sein du processeur sur lequel celui-ci s'exécute. Nous nous proposons de réaliser la mesure des erreurs de lecture dans le cache d'instructions, ainsi que celles en lecture et en écriture au niveau du cache de données.

Les résultats relatifs au cache de données, présentés sur la figure 3.3, montrent une augmentation significative des erreurs d'accès en lecture. Ces dernières passent en effet de 6 pour 10000 pour les codes conventionnels à 30 pour 10000 pour les codes de compilation dynamique, soit une multiplication par 5 de leur occurrence. Les erreurs d'accès en écriture restent, quant à elles, relativement stable (passant de 4 pour 10000 à 6 pour 10000).

Concernant les erreurs de lecture du cache d'instructions, leur augmentation est fortement marquée pour les codes de compilation dynamique, en comparaison avec les codes conventionnels, comme présenté sur la figure 3.4.

Les résultats obtenus sur les erreurs d'accès au cache d'instructions montrent une augmentation significative, passant de 2 pour 10000 en moyenne pour les codes conventionnels à 2 pour 1000 pour les codes de compilation dynamique, soit un rapport 10 entre les deux. Ces résultats sont assez similaires à ceux obtenus dans la première étude

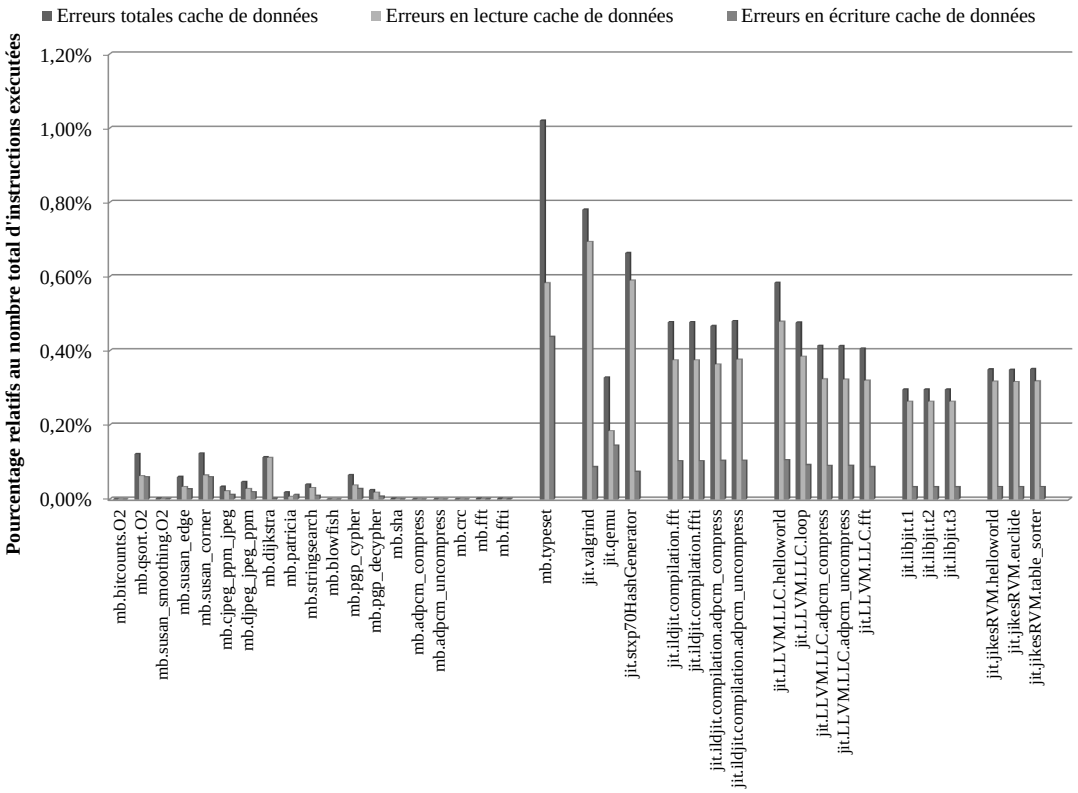



FIGURE 3.3 – Évolution des erreurs en lecture et en écriture dans le cache de données pour les codes conventionnels et pour les codes de compilation dynamique. Nombres d'accès manqués par instructions exécutées.

sur les erreurs de prédiction relatives aux branchements indirects, même si il n'existe pas de lien direct entre les deux. La quantité importante de données abstraites manipulées et les indirections rencontrées dans ces codes mettent en difficulté les éléments de prédiction de l'évolution du flot d'instructions exécuté, expliquant cette augmentation.

Tout comme le prédicteur de branchement, les caches de données et d'instructions émulés par Valgrind sont surdimensionnés par rapport à ceux utilisés dans les architectures embarquées (caches du processeur hôte, un Intel Core 2 Duo dans notre cas). En conséquence, le phénomène observé sera lui aussi accentué dans le cadre d'une exécution des codes de compilation dynamique sur cibles embarquées, les tailles de caches mises en jeu étant bien inférieures.

### Mesures relatives aux profondeurs d'indirection

Ces mesures relatives **aux profondeurs d'indirection** portent sur un ensemble de codes extraits de ceux utilisés dans l'étude précédente. En effet, il semble naturel d'imaginer que les codes de compilation dynamique, de par leur manipulation de graphes et de structures de données chaînées, présentent beaucoup plus d'accès mémoires indirects (autrement dit d'accès mémoires dont l'adresse provient elle-même d'un accès mémoire précédant).

Pour étudier ces indirections, nous avons utilisé un simulateur d'instructions x86 instrumenté pour suivre l'origine des calculs d'adresses des accès mémoire et en déterminer la profondeur. Ce simulateur réalisant une émulation minimaliste d'un système d'exploitation et impliquant quelques contraintes sur la natures des binaires simulables, nous n'avons pu réaliser les expériences que sur un sous-ensemble des cas de tests pré-

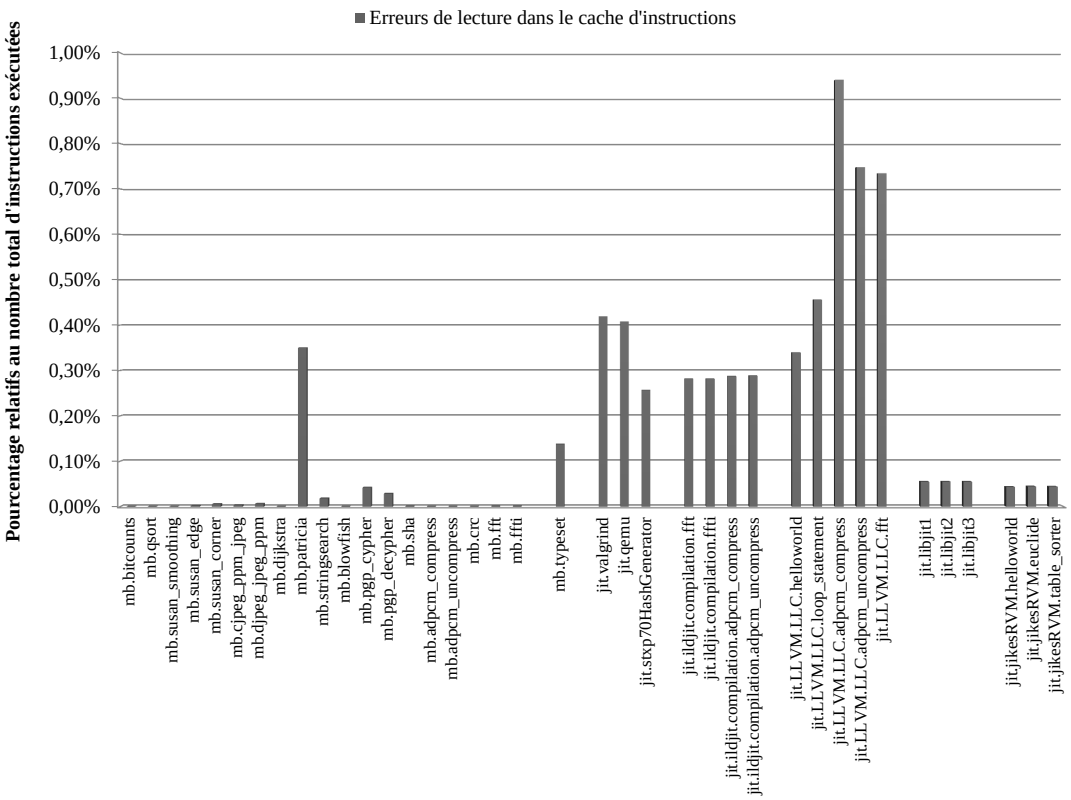



FIGURE 3.4 – Évolution des erreurs de lecture dans le cache d'instructions pour les codes conventionnels et pour les codes de compilation dynamique. Nombres d'accès manqués par instructions exécutées.

cédents, en générant des exécutables totalement autonomes (option `-static` au raccordement mémoire). Aussi, nous avons limité l'étude des codes de compilation dynamique au code le plus représentatif et le plus répandu du panel sélectionné : le compilateur LLC de LLVM.

Nous avons ensuite réalisée la même étude pour une série de codes conventionnels. Nous les avons classés en trois catégories différentes : les algorithmes standards (comme un simple programme "Hello World" ou fft), les algorithmes de compression (comme adpcm ou jpeg) et les algorithmes de gestion de graphes (comme dijkstra ou patricia). Les codes standards ne présentent pas ou peu d'indirections. En réalité, la majorité de ces accès indirects sont limités aux phases d'initialisation du programme avant l'exécution de son `main`. Les codes de compression nécessitent une manipulation de données plus complexe et entraînent une augmentation des profondeurs d'indirection, afin de réaliser les étapes de compression. Les codes de gestion de graphes, enfin, sont les algorithmes qui, de loin, présentent la plus forte utilisation de ces indirections, de par leur structure sous forme de graphe dans leur gestion des données.

Notre objectif, à travers cette étude, est **de positionner les codes de compilation dynamique**, avec l'évaluation de LLC, sur cette échelle représentative des différentes profondeurs d'indirection qu'il est possible de rencontrer au sein des codes.

Les résultats obtenus sont présentés dans le tableau 3.3. Ils mettent en avant comme attendu la nullité de cette profondeur pour les codes conventionnels (en ne considérant pas les imbrications liées à l'initialisation). Les codes de compression présentent quant à eux une profondeur d'indirection plus prononcée, de l'ordre de 200 en moyenne. Enfin, les codes de gestion de graphes présentent une profondeur d'indirection de l'ordre de 80000, montrant la forte imbrication des chargements de données au sein de ceux-ci.

Pour LLC, et plus globalement pour les codes de compilation, cette profondeur est de l'ordre de 8000, soit un rapport 1/10 avec les codes de gestion de graphes, mais surtout un rapport 4000 avec les codes standards, en considérant une profondeur de 2 les concernant (imbrications mesurées à l'initialisation). Cela met en avant le fait que même si les ordres de grandeurs obtenus diffèrent de ceux des codes de gestion de graphes, les codes de compilation dynamique présentent une forte imbrication de chargement au niveau de leurs données. Ces résultats traduisent donc la forte utilisation de pointeurs au sein de ces codes pour la manipulation de données abstraites sous la forme de structures. L'impact de ces profondeurs d'indirection sur les processeurs embarqués sera d'autant plus prononcé que ces processeurs ne disposent pas de mécanismes de prédiction aussi sophistiqués que ceux des processeurs généralistes pour gérer ce type de données.

Table 3.3 – Comparaison des profondeurs d'indirections moyennes selon les classes d'algorithmes.

| Algo. standards | Algo. de compression | Algo. de gestion de graphes | Algo. de compilation (LLC) |
|---|---|---|---|
| 2 | 200 | 80000 | 8000 |

### 3.5.2 Conclusion sur l'analyse des accès mémoires

Les résultats précédents mettent en avant plusieurs phénomènes essentiels relatifs aux codes de compilation dynamique :

- La proportion de code par rapport aux données manipulées est plus importante dans les codes de compilation dynamique.
- Les accès aux données sont moins prédictibles (moins de localités spatiale et temporelle)
- La génération d'adresse des accès mémoires est plus complexe (plus d'indirections dans les chargements mémoires)

Ces résultats renforcent les résultats précédents (sur le flot de contrôle). En particulier, les optimisations applicables pour les codes de compilation dynamique ne peuvent s'appuyer sur l'existence de flots de données réguliers à grain fin. Ainsi, l'utilisation d'optimisations relatives à ce type de flot n'apparaît pas comme étant une solution judicieuse pour les codes de compilation dynamique (par exemple l'utilisation de processeurs de type SIMD, *Single Instruction, Multiple Data*).

## 3.6 Efficacité relative d'un processeur embarqué pour la compilation dynamique

Nous venons de démontrer à travers les deux précédentes analyses **une irrégularité commune des codes de compilation dynamique au niveau de la gestion des flots de contrôle et de données**. Il a également été mis en avant l'accentuation de ces irrégularités sur les processeurs embarqués, liée à une sophistication réduite des mécanismes de prédiction et des tailles de caches bien inférieures.

Nous nous proposons dans cette partie **de quantifier la difficulté pour les architectures embarquées à gérer les spécificités de ces codes de compilation dynamique**. Pour cela, nous cherchons à évaluer l'évolution des temps d'exécution des codes de compilation dynamique, à travers l'étude de LLC, et d'un sous-ensemble des codes conventionnels considérés précédemment (bitcounts, convolution et pgcd) lors d'une migration vers un processeur embarqué.

Pour comparer l'évolution de ces temps d'exécution, nous nous basons sur un processeur x86 Intel Core 2 Duo (2.6 GHz) et sur un processeur embarqué ARM-Cortex A9 (400 MHz). Les codes testés ont été modifiés afin d'y introduire des boucles permettant d'augmenter la charge de calcul réalisée. Cette opération a pour but de limiter l'impact des différents appels systèmes et autres bruits de mesures. Les codes testés sur les deux architectures sont bien évidemment identiques et présentent les mêmes modifications.

Les fréquences de fonctionnement des deux cibles ne sont pas considérées dans les résultats. En effet, nous évaluons les temps d'exécution des codes à fréquence constante sur chacune des cibles. Les comparaisons d'évolution des temps d'exécution sont ensuite réalisées sur les ratios de ces évolutions, et sont donc indépendantes de la fréquence de fonctionnement.

Les temps d'exécution sont donnés en secondes et correspondent aux temps utilisateurs, obtenus par la commande `time`. Les résultats obtenus sur l'architecture x86 sont présentés dans le tableau 3.4 et ceux pour l'architecture ARM dans le tableau 3.5. Le tableau 3.6 présente les ratios de l'évolution des temps d'exécution entre ARM et sur x86 des codes de compilation dynamique et des codes conventionnels. Ces résultats expriment l'accentuation des temps d'exécution engendrée par le passage à l'échelle.

Alors que le passage à l'échelle des codes conventionnels d'Intel x86 à ARM provoque en moyenne un ralentissement d'un rapport 12, celui des codes de compilation dynamique est d'un rapport 17. Cela signifie, en d'autres termes, que les processeurs embarqués sont 40 % moins efficaces en moyenne pour la gestion des codes de compilation dynamique, en comparaison avec des codes conventionnels. Ces résultats sont présentés sur la figure 3.5.

Table 3.4 – Temps d'exécution mesurés sur x86 pour les codes conventionnels et pour les codes de compilation dynamique (LLC).

| | bitcounts | convolution | pgcd | llc (bit-counts) | llc (pgcd) | llc (hello) | llc (convo-lution) |
|---|---|---|---|---|---|---|---|
| user (s) | 0.980 | 0.608 | 1.096 | 12.629 | 3.440 | 2.936 | 5.284 |

Table 3.5 – Temps d'exécution mesurés sur ARM pour les codes conventionnels et pour les codes de compilation dynamique (LLC).

| | bitcounts | convolution | pgcd | llc (bit-counts) | llc (pgcd) | llc (hello) | llc (convo-lution) |
|---|---|---|---|---|---|---|---|
| user (s) | 11.030 | 10.330 | 9.140 | 207.491 | 61.740 | 49.000 | 94.050 |

Table 3.6 – Décélération induite par le passage à l'échelle x86 vers ARM pour les codes conventionnels et pour les codes de compilation dynamique (LLC).

| | bitcounts | convolution | pgcd | llc (bit-counts) | llc (pgcd) | llc (hello) | llc (convo-lution) |
|---|---|---|---|---|---|---|---|
| user (s) | 11.255 | 16.990 | 8.339 | 16.430 | 17.948 | 16.689 | 17.799 |

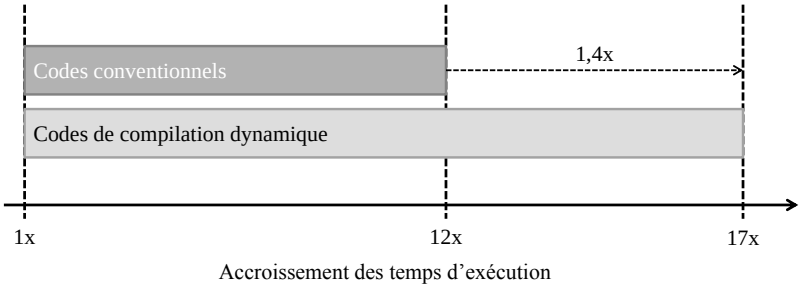



FIGURE 3.5 – Évolution des temps d'exécution pour un passage à l'échelle x86 (Intel Core 2 Duo) vers ARM (ARM Cortex-A9) des codes de compilation dynamique et des codes conventionnels.

Ces résultats mettent en évidence l'impact **des limites des mécanismes de prédiction et des capacités des mémoires caches** pour la gestion des codes de compilation dynamique sur les processeurs embarqués. Par conséquent, ces processeurs, en plus de leurs performances brutes inférieures à celles des processeurs généralistes, éprouvent de plus grandes difficultés à gérer efficacement les codes de compilations dynamique en comparaison avec les codes conventionnels, liées à leur manque de polyvalence.

Il est donc plus difficile de masquer les latences introduites par les codes de compilation dynamique sur ces architectures, en comparaison avec les architectures généralistes, type station de travail ou serveur. Ces problématiques, communes à l'ensemble des technologies de compilation dynamique, ont un impact direct sur l'attractivité de la mise en place de telles solutions sur ces architectures.

## 3.7 Conclusion

Nous avons réalisé, à travers ce chapitre, une caractérisation des codes de compilation dynamique, dans l'optique d'identifier le type et la nature des optimisations réalisables et applicables à l'ensemble de ces codes. Pour cela, nous avons conduit une analyse détaillée des différentes technologies de compilation dynamique afin d'en comprendre les problèmes architecturaux conduisant à une dégradation des performances sur les cibles embarquées. Nous nous sommes basés sur l'observation de compteurs de performances et avons réalisé une comparaison de leur évolution pour des codes dits conventionnels.

Au niveau de l'observation du flot de contrôle, les résultats mettent en avant un problème commun de gestion des branchements indirects par les codes de compilation. Les branchements conditionnels sont également plus nombreux et moins bien prédits. Ces résultats mettent en avant **une complexité accrue de ce flot de contrôle** dans les codes de compilation, en comparaison avec des codes dits conventionnels. La criticité de ce problème est accrue par la simplicité des prédicteurs sur les architectures embarquées, en comparaison avec ceux des architectures généralistes.

Au niveau de la manipulation des données, et en particulier des accès mémoires, les résultats obtenus mettent en lumière un accroissement significatif des erreurs de lecture au niveau du cache d'instructions (10× pour le cache d'instructions pour les codes de compilation dynamique, en comparaison avec les codes conventionnels testés). Ils montrent également la présence de profondeurs d'indirection largement supérieures à celles d'algorithmes standards (profondeur moyenne de 8000 contre 2 pour un algorithme standard). Ces résultats traduisent l'important volume d'instructions nécessaire à la manipulation des données au sein des codes de compilation dynamique, la difficulté de prédiction des accès aux données et la complexité accrue des accès mémoires.

Ces éléments sont notamment liés à **la manipulation de données abstraites au sein de ces codes**, par opposition aux données naturelles et/ou scalaires manipulées dans les codes conventionnels. Les tailles de caches limitées des processeurs embarqués engendreront un accroissement de ces phénomènes.

Les résultats obtenus à ce niveau laissent également envisager une forte irrégularité, et par conséquent une faible prédictibilité, des accès mémoires générés pour les codes de compilation dynamique. Ce phénomène est en opposition avec les résultats traditionnellement obtenus sur la localité temporelle des codes conventionnels. Afin d'évaluer plus précisément cet aspect, il est possible de se baser sur les travaux de Ketterlin et al. [123], portant sur la réalisation d'analyses statiques de codes binaires afin d'en évaluer la complexité des générations d'adresse mémoire.

Une quantification de la difficulté pour ces processeurs embarqués à gérer ces codes a été réalisée, par comparaison de l'évolution des temps d'exécution lors d'une migration de codes de compilation dynamique et de codes dits conventionnels sur ce type d'architecture.

Cette forte complexité des codes de compilation dynamique rend inenvisageable une optimisation globale (logicielle ou matérielle) des codes de compilation. Par exemple, une mise en circuit complète d'un compilateur reviendrait quasiment à créer un processeur superscalaire sophistiqué de type x86. De la même façon, l'irrégularité constatée, et notamment l'important volume d'instructions nécessaires à la manipulation des données, rend inenvisageable une optimisation similaire à ce qui est normalement mis en place pour les flots de données réguliers, comme par exemple l'utilisation de processeurs SIMD.

A contrario, ces résultats laissent envisager **l'opportunité de la mise en place d'opérateurs spécifiques communs** à l'ensemble des technologies de compilation dynamique dans l'optique d'en accroître les performances. Il est ainsi envisageable d'isoler les portions critiques de code à l'origine de l'irrégularité commune constatée au niveau du flot de contrôle et du flot de données (comme par exemple celles relatives à la gestion des structures récursives et chaînées). L'objectif est de proposer des optimisations simples focalisées sur ces portions afin de compenser la simplicité des mécanismes de prédiction et les faibles tailles des caches des processeurs embarqués.

Le prochain chapitre est dédié à l'identification de ces portions critiques des codes de compilation dynamique, ainsi qu'à la présentation de l'approche retenue dans le cadre de ces travaux, consistant à proposer des accélérations matérielles pour les opérateurs simples permettant de gérer ces portions.

# Chapitre 4

# Identification des portions critiques des codes de compilation et justification de la mise en place d'accélérations matérielles

**Sommaire**



## 4.1 Introduction

Le précédent chapitre nous a permis de mettre en évidence une complexité commune entre les codes de compilation dynamique, liée à la forte irrégularité de ces codes quant à la gestion des flots de contrôle et de données, en comparaison avec des codes dits conventionnels. Nous avons également pu mettre en évidence la forte difficulté pour les processeurs embarqués à gérer ces irrégularités, en comparant l'évolution des temps d'exécution pour un passage à l'échelle d'applications conventionnelles et de codes de compilation dynamique. Toutefois, cette étude a permis de démontrer l'opportunité

d'inclure des opérateurs simples d'optimisation sur ces processeurs dans l'optique d'accroître leur capacité à gérer ces codes de compilation. Ce chapitre est justement dédié à la détermination des opérateurs potentiellement envisageables.

La première partie de ce chapitre est dédiée à **l'identification des portions de code critiques en termes de temps d'exécution entraînant l'irrégularité commune constatée**, accentuée lors d'un transfert de la compilation dynamique dans l'embarqué. Nous nous proposons de nous focaliser sur un seul et unique compilateur dans la suite de l'étude, à savoir le compilateur de code à octet de LLVM (LLC). Nous justifierons ce choix au début de ce chapitre. L'objectif de l'identification de ces portions de code critiques est de permettre ensuite la proposition d'optimisations adéquates de ces portions.

La deuxième partie de ce chapitre est consacrée à **la justification du positionnement de cette thèse** : pourquoi s'orienter vers la mise en place d'accélérations matérielles ? Nous nous basons pour cela sur l'état de l'art réalisé sur les optimisations existantes au niveau de la compilation dynamique, mises en place dans l'optique d'en réduire l'impact sur les architectures embarquées. Nous présentons les avantages et les inconvénients de ces solutions et nous nous focalisons plus spécifiquement sur leurs limites, avant de justifier notre positionnement par rapport à cette analyse.

## 4.2 Identification des points critiques des codes de compilation dynamique

### 4.2.1 Présentation de l'étude et de ses objectifs

Nous cherchons dans le cadre de cette étude à mettre en évidence **les portions de code critiques** des codes de compilation dynamique. Ces points chauds, étroitement liés aux problèmes d'irrégularité soulevés dans le chapitre précédent, serviront ensuite de base à la proposition d'optimisation dans l'optique d'accroître l'attractivité de la compilation dynamique sur les architectures embarquées.

Afin de réaliser cette étude, nous nous basons sur un simulateur de jeu d'instructions réalisé dans le cadre de la thèse, de précision approximative au cycle. Il modélise un processeur embarqué, en l'occurrence un ARM Cortex-A5, dans sa version mono-coeur. Le choix de ce processeur résulte du fait qu'il s'agisse d'un des processeurs embarqués à exécution dans l'ordre (*in-order processor*) les plus minimalistes de chez ARM, s'intégrant donc parfaitement à notre contexte de transfert de la compilation dynamique dans le domaine embarqué.

Les caractéristiques de ce processeur sont les suivantes [124] :
- processeur 32 bits, support Arm32 et Thumb2 (Armv7) ;
- support unité flottante et extension vectorielle Neon ;
- caches d'instructions et de données de respectivement 32 KO chacun dans notre implémentation ;
- performances annoncées de 1.57 DMIPS/MHz par coeur (Dhrystone) ;
- fréquence de fonctionnement de 600 MHz (530-600 MHz) ;
- consommation de 0.12 mW/MHz en technologie TSMC 40 nm faible consommation ;
- efficacité énergétique de 13 DMIPS/mW en technologie TSMC 40 nm faible consommation ;
- surface de 0.28 $mm^2$ sans les caches, de 0.53 $mm^2$ avec respectivement 16 KO de cache d'instructions et de cache de données, 0.68 $mm^2$ en y ajoutant l'extension vectorielle Neon.

Le simulateur est basé sur la librairie GenISSLib [119] permettant la génération

de simulateurs de jeu d'instructions. Les jeux d'instructions Arm32 et Thumb2 ont été implémentés (jeu d'instructions Armv7). Thumb2 est une extension des jeux d'instructions Thumb (sous-ensemble d'instructions 16 bits des instructions ARM 32 bits) et Arm32 (32 bits). Il permet la manipulation simultanée d'instructions 16 et 32 bits, alliant à la fois l'économie mémoire offerte par ce jeu d'instructions 16 bits et les performances de celles du jeu d'instructions Arm32, offrant plus de possibilités que Thumb. Le code machine d'une application compilée en Thumb2 peut ainsi se composer aussi bien d'instructions 16 que 32 bits.

Ce simulateur a été instrumenté afin de permettre l'évaluation du nombre exact de cycles passés dans des portions précises du code préalablement définies au niveau du logiciel.

Le simulateur a également été instrumenté afin de relever les instructions les plus souvent exécutées au sein du simulateur. Les relevés ont ensuite été raffinés sur les zones concernées par ces instructions afin de déterminer leur pourcentage relatif exact en termes de temps d'exécution. A ce simulateur nous associons un simulateur de cache avec une taille de cache L1 de 32 KO.

Les applications s'exécutant sur ce simulateur ont été compilées avec le compilateur GNU GCC (`arm-none-linux-gnueabi-gcc-4.6.1`), avec l'option d'optimisation `-O2` et l'option `-static` afin de produire des binaires ELF autonomes puisque notre simulateur ne fournit qu'une émulation de système d'exploitation minimaliste. Concernant LLC, nous utilisons une version autonome similaire à celle mise en place au chapitre précédent, permettant la génération de code machine ARM et s'exécutant cette fois-ci sur architecture ARM.

Afin de solliciter le compilateur, **une série de tests a été sélectionnée** parmi le panel d'algorithmes conventionnels du chapitre 3. Ces codes de test ont été choisis pour être représentatifs au niveau de leur taille et de leur complexité des portions de code que les compilateurs dynamiques peuvent être amenés à recompiler dynamiquement. Une exception est à noter toutefois concernant susan, dont il sera possible d'observer par la suite un comportement légèrement différent des autres en raison de sa grande taille et de sa complexité relative par rapport aux autres codes. Voici la liste de la série de tests considérée : bitcount, convolution (produit de convolution), crc32, dijkstra, fft, hello, neon intrinsincs (tests de vectorisation pour processeur ARM), patricia, gcd (calcul de pgcd), quicksort, adpcm (compression et décompression), stringsearch, sha et susan.

### 4.2.2 Justification du choix du compilateur de LLVM, LLC, différenciation avec LLI

Comme mentionné en début de ce chapitre, les résultats obtenus dans les chapitres précédents ont permis de mettre en avant non seulement que la compilation dynamique peut être considérée comme un domaine algorithmique spécifique, mais également qu'il existe une complexité commune entre les différentes technologies présentées, notamment au niveau de l'irrégularité des flots de données et de contrôle.

Nous avons donc choisi de nous focaliser sur **l'étude d'un unique compilateur** dans la suite de ces travaux. Nous avons sélectionné pour cela le compilateur LLC (*Low-Level Compiler*) du cadriciel LLVM, permettant la génération de code machine à partir de la représentation intermédiaire LLVA de LLVM. Il est aujourd'hui très largement utilisé dans un nombre important de projets, aussi bien académiques qu'industriels, dont une partie de l'étendue a déjà été présentée dans le premier chapitre. Construit comme un compilateur optimisant, dans l'optique de faire bénéficier aux applications des informations disponibles à l'exécution par l'intermédiaire de phases d'optimisations, ce compilateur est aussi bien utilisé aujourd'hui dans des solutions de virtualisation,

comme VMKIT, que dans des projets ayant une optique de performances brutes, comme le développement de solutions OpenCL pour ARM, l'utilisation d'OpenGL pour MAC OS X, ou encore la parallélisation d'OpenCL. La communauté l'utilisant est une des plus puissantes du domaine et ce compilateur est reconnu comme étant un des plus performants, permettant par exemple de générer un code équivalent aussi rapidement que GCC, avec une portabilité facilitée et une faible empreinte mémoire. LLC sert de brique de base au développement de nombreux compilateurs dynamiques, les développeurs concevant leurs propres solutions de moteur de compilation dynamique autour de celui-ci.

Il convient, dans le cadre de ce manuscrit, de bien faire la distinction entre LLC, et LLI, qui inclut LLC et le moteur de compilation dynamique développé par la communauté LLVM. Ce moteur n'est qu'une version expérimentale d'utilisation de la compilation dynamique, constitué d'une surcouche de routines appelant les différentes fonctions de LLC, et n'est pas complètement fonctionnel sous ARM. Les performances obtenues pour celui-ci sont loin de l'attractivité obtenue avec d'autres solutions basées sur LLC. Il est donc important de garder à l'esprit que nous ne nous focalisons dans cette étude que sur le compilateur LLC, dont la maturité est largement approuvée et éprouvée aujourd'hui par la communauté, et non sur son implémentation existante dans LLI.

Dans le cadre de l'ensemble des études menées durant cette thèse, nous nous sommes concentrés sur la version 2.9 du compilateur LLVM. Les versions de LLVM évoluant très rapidement (une nouvelle version tous les six mois environ), nous n'avons pas cherché à suivre leur évolution et avons décidé de nous fixer sur la dernière version existante au moment du démarrage des travaux.

Le compilateur LLC ne réalise pas la partie raccordement en mémoire (*linking*) de la compilation et s'arrête au niveau de la génération du fichier objet. Aussi, toutes les expérimentations réalisées sur les performances de LLC ne concerneront que la partie strictement dédiée à la génération de code, de la représentation intermédiaire en mémoire au code machine au moment de sa mise en mémoire.

Nous avons mis en place au chapitre précédent une version autonome x86 de LLC, s'affranchissant de tous les appels systèmes et rapatriant l'ensemble des librairies nécessaires à son fonctionnement, dans l'optique d'en générer un binaire ELF autonome. Basés sur ces travaux, nous avons généré une version autonome équivalente pour cible ARM, que nous utilisons dans ce chapitre ainsi que dans l'ensemble des expérimentations qui suivront dans le prochain chapitre. Elle permet la génération de code machine ARM à partir de la représentation intermédiaire LLVA de LLVM.

### 4.2.3 Présentation des résultats de l'étude

Les résultats obtenus dans le cadre de cette étude ont permis de mettre en avant **trois portions de code critiques** au sein du compilateur LLVM, au niveau de leur temps d'exécution. Cette criticité est la conséquence de la forte irrégularité dans la gestion des flots de données et de contrôle mise en avant au chapitre précédent, liée à la forte quantité de données abstraites manipulées et à l'utilisation massive de pointeurs au sein des codes de compilation.

En analysant les instructions les plus récurrentes lors des différentes exécution de LLC, et surtout en analysant les portions de code correspondantes, il apparaît que **la gestion des tableaux associatifs et de l'allocation dynamique de la mémoire constituent les deux premiers points critiques** qui ont pu être mis en avant au cours de cette analyse, au sein des fonctions relatives à la gestion des différents types de données abstraites. Les tableaux associatifs sont des types de données abstraites permettant l'association de paires de (clé, valeur) et la manipulation de structures de données complexes. Chacune des clés peut être associée à une ou plusieurs valeurs.

Ces tableaux permettent ainsi l'association de différents types entre eux. Nous verrons dans la suite qu'il existe deux principaux types d'implémentation pour ces tableaux associatifs : les tables de hachage (association (clé, valeur) par l'intermédiaire d'une fonction de hachage) et les arbres binaires quasi-équilibrés. Les tableaux associatifs sont un problème récurrent dans un large domaine d'applications, qui s'étend bien au delà de la compilation dynamique. Cette représentation est en effet fortement utilisée dans l'ensemble des applications manipulant des données abstraites et faisant appel pour cela massivement à l'utilisation de pointeurs, comme par exemple les nouvelles applications de traitement d'image incluant de l'analyse de contenu.

Ces tableaux associatifs sont notamment manipulés au sein de la librairie standard STL C++ [125] à travers les conteneurs `std::[multi]map` et `std::[multi]set`. `std::map` permet le stockage d'éléments de type (clé, valeur). `std::set` permet le stockage d'éléments de type clé seulement. Un *Set* peut donc être vu comme une *Map* avec une valeur de type `void*`, c'est d'ailleurs souvent cette implémentation qui est choisie. Ces deux conteneurs sont des conteneurs à associativité unique, assurant que deux éléments au sein du tableau associatif ne peuvent être identiques. Pour une utilisation d'éléments identiques, il existe deux conteneurs complétant respectivement `std::map` et `std::set` : `std::multimap` et `std::multiset`.

Nous verrons dans la suite de ce chapitre que LLVM dispose déjà d'un très grand nombre d'optimisations logicielles visant à limiter l'impact de cette gestion des tableaux associatifs et de l'allocation dynamique de la mémoire, en proposant notamment de s'affranchir des librairies standards. Toutefois, il est possible de constater sur la figure 4.1 que ces deux points représentent encore en moyenne 24 % du temps total d'exécution du compilateur LLC, avec respectivement 16 % en moyenne pour l'allocation dynamique de la mémoire et 8 % en moyenne pour la gestion des tableaux associatifs. Nous pouvons constater sur cette figure un comportement légèrement différent pour le test relatif au code susan, avec une prédominance de l'impact de la gestion des tableaux associatifs sur l'allocation dynamique de la mémoire, contrairement aux autres codes. Ce résultat s'explique par le volume de code plus important de cet algorithme qui réalise par conséquent un plus grand nombre de manipulation des données que les autres, d'où l'accroissement du pourcentage de temps relatif à la gestion des tableaux associatifs, alors que celui alloué à l'allocation mémoire reste dans la moyenne des pourcentages obtenus pour les autres codes.

Concernant le troisième point critique, celui-ci est lié à **la gestion des différents graphes d'instructions de l'application à compiler**. Cela se manifeste par la présence systématique d'instructions **liées au chargement, au décodage et à la manipulation d'opérandes** relatives à ces instructions, aussi bien au niveau de la représentation intermédiaire qu'au niveau du code machine, dans notre relevé des opérations les plus exécutées sous LLVM. Ce résultat apparaît comme logique de par la constante manipulation des différentes instructions, aux différents niveaux de représentation, à travers toutes les phases de génération de code et d'optimisation. Lorsque nous dressons la liste des 64 instructions les plus exécutées sous LLVM, les instructions renvoyant à des étapes de chargement et de décodage des instructions de l'application figurent quasi systématiquement dans cette liste pour la série de tests considérée.

Toutefois, il a été possible d'observer lors de cette étude que ces instructions ne sont pas contenues au sein d'une partie bien définie de l'algorithme de LLC. Elles sont en effet réparties dans l'ensemble du compilateur. Les développeurs ont choisi de ne pas centraliser ces manipulations dans un noyau dédié mais de les effectuer de manière distribuée dans les différents noyaux nécessitant ces opérations de gestion des instructions. La détermination du pourcentage du temps d'exécution de LLC que représentent ces manipulations n'est donc pas aussi triviale que la détermination de celui relatif à

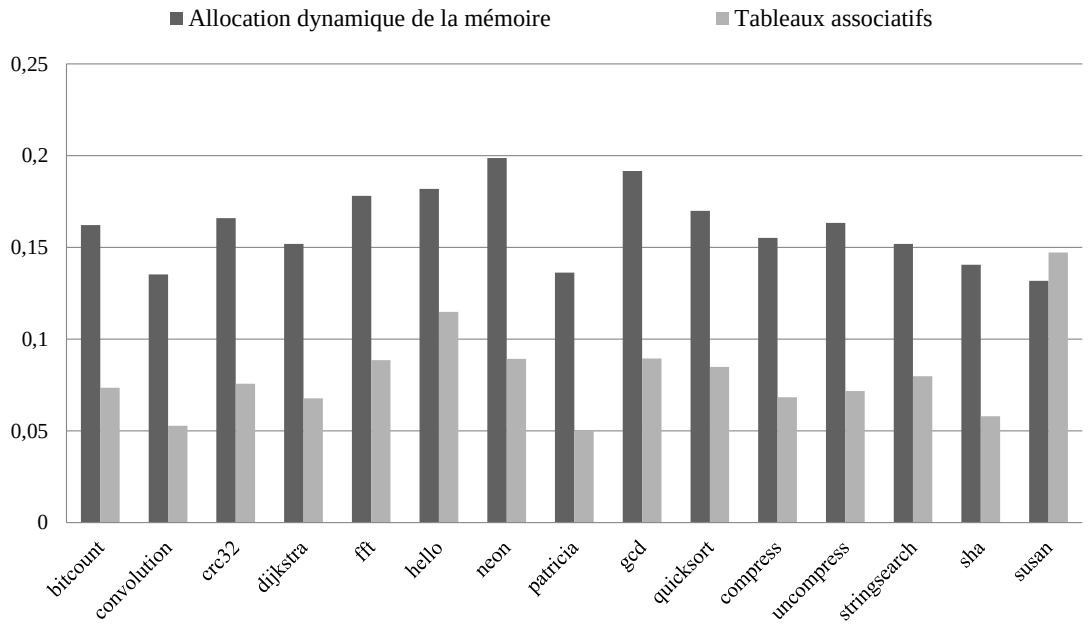



FIGURE 4.1 – Pourcentage du temps d'exécution total relatif à la gestion des tableaux associatifs et à l'allocation dynamique de la mémoire sous LLC pour la série de tests considérées.

la gestion des tableaux associatifs et à l'allocation dynamique de la mémoire, confinées aux fonctions de gestion des différents types de données abstraites. En conséquence, la quantification de l'influence de cet aspect est délicate à réaliser. Nous ne pouvons donc obtenir un pourcentage précis comme celui mis en avant pour les précédents points critiques mais nous estimons, à travers l'analyse des performances de LLC et une analyse à gros grain des différentes portions de code sollicitant ces opérations, que ces étapes de chargement et de décodage des instructions de l'application à compiler peuvent représenter jusqu'à 20 % du temps total d'exécution du compilateur LLC.

### 4.2.4 Conclusion sur cette étude

Cette étude des portions de code critiques au sein de LLVM a permis de mettre en avant trois points principaux quant à leur influence sur les temps d'exécution de LLC. Les deux premiers concernent l'influence de la gestion des tableaux associatifs et de l'allocation dynamique de la mémoire, représentant en moyenne 24 % du temps d'exécution de LLC. Les tableaux associatifs sont aujourd'hui largement répandus pour la manipulation des données abstraites, aussi bien dans le cadre de la compilation que dans la gestion de l'allocation mémoire, et même pour la gestion des langages à haut niveau d'abstraction comme le JavaScript. Leur optimisation a donc été largement explorée mais les problématiques dans les systèmes embarqués demeurent, comme les résultats obtenus dans le chapitre précédent le démontrent. Le troisième point principal concerne la gestion du graphe des instructions de l'application à compiler, représentant jusqu'à 20 % du temps d'exécution de LLC. Des opérations récurrentes de chargement et de décodage sont systématiquement réalisées à travers l'ensemble des différentes phases de génération et d'optimisation, aussi bien au niveau représentation intermédiaire qu'au niveau code machine. Elles présentent une grande régularité dans la façon avec laquelle elles sont réalisées, mais l'importante irrégularité des codes de compilation dynamique rend leur gestion très complexe pour des processeurs embarqués, comme nous l'avons vu au chapitre précédent. Ces trois points constituent donc de forts potentiels d'optimisation des codes de compilation dynamique et sont ceux que nous avons choisi

d'explorer dans le cadre de ces travaux, à travers l'approche présentée et justifiée dans la prochaine section de ce chapitre.

## 4.3 Justification de l'utilisation d'une solution basée sur la mise en place d'accélérations matérielles dans l'embarqué

L'objectif de cette partie est **de mettre en avant et de justifier l'intérêt** de l'approche retenue dans le cadre de cette thèse, consistant à **la mise en place d'accélérations matérielles** pour la compilation dynamique dans les systèmes embarqués. Notre objectif est de proposer la réalisation de ces accélérations en se basant sur l'identification des portions critiques de code de LLC, et plus globalement des codes de compilation dynamique, réalisée dans la section précédente. Nous détaillons cette approche retenue pour nos travaux à la fin de cette section. Mais nous nous attachons tout d'abord à la justification de cette volonté de s'orienter vers la mise en place de solutions matérielles en analysant les avantages, les inconvénients et surtout les limites des solutions existantes. Nous allons voir que les optimisations au niveau de la conception des systèmes, autrement dit les optimisations au niveau matériel, restent un domaine relativement peu exploité pour la compilation dynamique et que notre approche s'inscrit dans une démarche jusqu'à présent non explorée, alliant à la fois flexibilité et performances.

### 4.3.1 Limites des optimisations logicielles

La première approche analysée d'optimisation de la compilation dynamique dans les systèmes embarqués concerne **les optimisations logicielles**. Nous avons pu notamment mettre en avant les optimisations algorithmiques, comme celles réalisées sous LLVM, les optimisations par compilation scindée, l'optimisation de la gestion mémoire ou la spécialisation de code. Toutes ces optimisations permettent d'accroître les performances des codes de compilation dynamique et de réduire l'impact des portions les plus critiques en termes de temps d'exécution. Toutefois, l'ensemble des études mettent en avant les limites des gains obtenus pour la gestion de ces codes. La raison principale réside dans la problématique des efficacités énergétique et surfacique dans leur gestion, conséquence des phénomènes précédemment mis en avant et de l'absence de mécanismes aussi sophistiqués que ceux à disposition sur les architectures généralistes pour les gérer (simplicité des mécanismes de prédiction et limitation des tailles de caches). Elle réside également dans l'optique de performances avec laquelle ces optimisations ont été réalisées, ne prenant pas suffisamment en compte les aspects inhérents à l'embarqué.

Les optimisations logicielles, et notamment **algorithmiques**, ont aussi pour défaut majeur d'accroître le volume de code à gérer au sein des compilateurs et donc à l'exécution dans le cadre de la compilation dynamique. Le code du compilateur LLC représente pas moins d'un million trois cent mille lignes de code. Les développeurs de LLVM ont notamment mis en place une très importante quantité d'optimisations logicielles pour la gestion des tableaux associatifs et l'allocation dynamique de la mémoire. Ils ont remplacé l'utilisation des tableaux associatifs de la librairie standard STL C++ (`std::[multi]map`, `std::[multi]set`) par pas moins de dix-sept conteneurs spécialisés de tableaux associatifs (ADT, *Abstract-Data Types*) en fonction des différents cas d'utilisation (neuf pour `std::[multi]map` et huit pour `std::[multi]set`). L'utilisation de la librairie standard n'est réservée qu'aux cas les moins intenses. Ces différents conteneurs sont listés dans le tableau 4.1.

Les développeurs se basent, entre autres, sur le constat du faible nombre d'éléments contenus dans ces tableaux et de leur faible taille, permettant leur rangement dans de

TABLE 4.1 – Conteurs spécialisés pour la gestion des tableaux associatifs dans LLC, remplaçant respectivement `std::[multi]map` et `std::[multi]set`. Ces deux derniers ne sont utilisés que pour les cas non couverts par ces conteneurs spécialisés

| `std::map` | `std::set` |
|---|---|
| DenseMap | DenseSet |
| StringMap | SmallSet |
| IndexedMap | SmallPtrSet |
| ValueMap | SparseSet |
| IntervalMap | SparseMultiSet |
| MapVector | FoldingSet |
| IntEqClasses | SetVector |
| ImmutableMap | UniqueVector |
| | ImmutableSet |

simples tableaux plutôt que dans des arbres binaires équilibrés comme c'est le cas dans la librairie STL C++. Leurs analyses mettent en avant un temps d'accès réduit dans ce cas de figure en comparaison avec celui nécessaire dans le cas d'un rangement de type arbre binaire trié.

Concernant **l'allocation dynamique de la mémoire**, le code de LLC tente de limiter au maximum les recours à celle-ci à travers l'utilisation d'allocateurs spécialisés. Il est notamment intéressant de citer parmi eux l'allocateur `RecyclingAllocator` permettant de conserver une copie des éléments récemment désalloués, afin d'en éviter une réallocation dans le cadre d'une nouvelle utilisation de ces éléments. Cela permet de limiter au maximum les allocations et désallocations réalisées par LLC. L'ensemble des conteneurs préalablement mentionnés permettent également de limiter au maximum les allocations dynamiques de la mémoire en proposant l'allocation statique des tableaux de rangement. Une analyse du remplissage de ces tableaux permet, lorsque c'est nécessaire, de réaliser une augmentation ou une réduction de la taille de ceux-ci en fonction de l'évolution des besoins. Ces opérations d'allocation ou de désallocation n'interviennent donc pas à chaque manipulation comme c'est le cas pour la librairie standard.

Outre **l'accroissement du volume de code à manipuler**, ces solutions ont aussi pour défaut majeur **d'accroître la complexité de l'algorithme**. Ainsi, pour LLC, les développeurs souhaitant effectuer des manipulations au sein même de la gestion des tableaux associatifs doivent impérativement se familiariser avec l'ensemble des différents conteneurs et adapter leur implémentation à ceux-ci. L'utilisation de la librairie standard n'offrant que des résultats limités en termes de performances, l'utilisation des conteneurs est indispensable à la mise en place d'une solution optimisée.

Malgré toutes ces optimisations, nous avons pu mettre en avant dans la partie précédente que la gestion des tableaux associatifs et l'allocation dynamique de la mémoire représentent encore une part très significative du temps d'exécution de LLC, montrant ainsi leurs limites.

### 4.3.2 Illustration des limites de ces optimisations à travers une étude de spécialisation de code

#### Présentation de l'étude

Nous nous proposons dans cette partie de mettre en avant **les limites d'une opportunité d'optimisation logicielle des codes de compilation dynamique**, à travers une étude publiée au Workshop international ADAPT '13 [126], organisé dans

le cadre de la conférence HIPEAC '13. Les optimisations logicielles bas-niveau, comme la spécialisation de code, sont aujourd'hui largement adoptées pour les applications régulières dans les systèmes embarqués, notamment pour les optimisations d'ordonnancement d'instructions, de gestion de cache, ou d'utilisation d'instructions SIMD (*Single Instruction, Multiple Data*). Mais concernant les applications irrégulières, comme la compilation dynamique, ces optimisations bas-niveau sont beaucoup plus rares et difficiles à mettre en place, de par le manque de mécanismes et de ressources mémoires à disposition sur ces architectures pour gérer des codes utilisant massivement des pointeurs et manipulant une grande quantité de données abstraites. Nous nous intéressons ici à la mise en place **d'une optimisation bas-niveau des tableaux associatifs**, dont nous venons de montrer la prédominance concernant leur temps d'exécution sur le compilateur LLC, en proposant **l'utilisation de code auto-adaptatif**.

#### Introduction à la notion d'arbres rouges et noirs

LLC dispose déjà d'un certain nombre d'optimisations logicielles pour la gestion de ces tableaux associatifs, en multipliant notamment, comme nous venons de le voir, les différentes implémentations de ceux-ci en fonction des différents cas d'utilisation. Nous nous sommes focalisés dans cette étude sur l'implémentation des tableaux associatifs telle que proposée dans la librairie standard STL C++ [127, 128] à travers *Map* et *Set*, ainsi que dans Linux [129], à savoir celle basée sur **les arbres rouges et noirs**. La clé du tableau associatif, à laquelle est associée la valeur, permet d'effectuer le tri au sein de l'arbre. Ces arbres rouges et noirs sont des arbres binaires triés auxquels est rajoutée une notion de coloration des noeuds afin d'assurer l'équilibrage de l'arbre grâce aux propriétés suivantes :

- il existe deux colorations possibles pour chacun des noeuds : rouge ou noir ;
- le noeud d'origine (noeud racine) de l'arbre est forcément noir ;
- les extrémités de chaque branche de l'arbre sont constitués de noeuds *NULL*, colorés en noir ;
- les noeuds enfants (gauche ou droite) d'un noeud rouge sont noirs ;
- chaque chemin de l'arbre, de la racine à chaque feuille de l'arbre, contient le même nombre de noeuds noirs.

Ces propriétés permettent d'assurer un quasi-équilibrage de l'arbre. La quatrième propriété assure en effet que l'arbre ne peut jamais contenir deux noeuds rouges consécutifs, il peut donc au mieux y avoir une succession de noeuds noirs sur l'ensemble d'une branche et au pire une alternance de noeuds noirs et rouges. Avec la cinquième propriété, cela assure le fait qu'il ne peut donc pas y avoir plus d'un rapport deux entre le chemin le plus long et le chemin le plus court, de la racine à une des feuilles de l'arbre.

Cette caractéristique majeure des arbres rouges et noirs assure une complexité inférieure à celle des arbres binaires triés classiques, en $O(logn)$, aussi bien pour les opérations de lecture (*get*) que d'écriture dans l'arbre (*set*). Les résultats obtenus par le choix de cette implémentation pour les tableaux associatifs montrent qu'il s'agit en moyenne de la meilleure implémentation pour ces derniers. Toutefois, un défaut majeur de cette solution est la nécessité de lourdes opérations d'équilibrage de l'arbre pour chaque modification (insertion ou suppression de noeuds) au sein de celui-ci. Un exemple d'arbre rouge et noir est donné sur la figure 4.2.

Sur cette figure, le noeud d'entête *header* n'est pas représenté. Ce noeud est doté d'une structure identique à celle des autres noeuds mais où le champ parent pointe sur le noeud racine de l'arbre, le champ gauche sur le noeud à l'extrême gauche de l'arbre (clé la plus petite de l'arbre), le champ droite sur le noeud à l'extrême droite de l'arbre (clé la plus grande de l'arbre). Les champs clé et couleur sont inutilisés.

L'intérêt de ce noeud d'entête est de servir de point d'entrée de l'arbre rouge et noir pour accéder rapidement à celui-ci lors de sa manipulation. Par ailleurs, ce noeud a la propriété d'être invariant durant toute la durée de vie de l'arbre. Puisqu'il ne contient pas de clé, contrairement au noeud racine, il ne peut être amené à changer au cours des manipulations de l'arbre, et notamment des rééquilibrages.

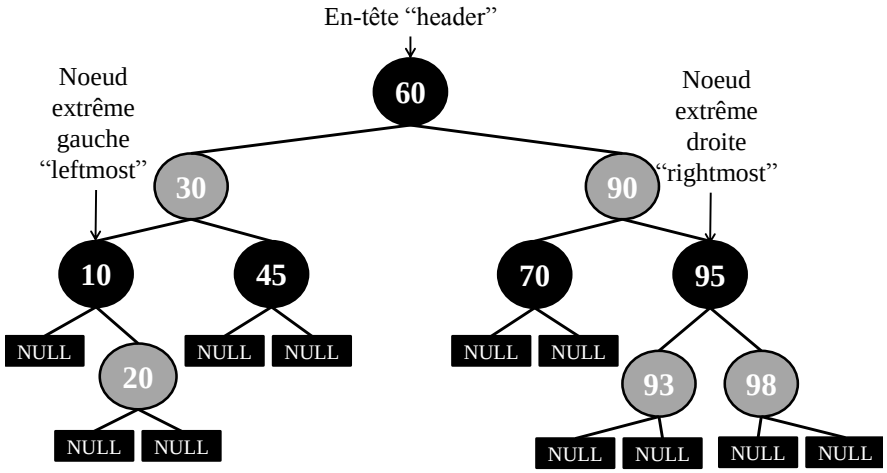



FIGURE 4.2 – Exemple d'arbre rouge et noir où les éléments sont triés en fonction de leur clé. *Header* représente la racine de l'arbre, *Leftmost* et *Rightmost* représentent respectivement les noeuds les plus à gauche et à droite de l'arbre.

### Spécialisation de code proposée

En se focalisant sur les algorithmes de gestion de ces arbres rouges et noirs, nous cherchons à mettre en avant dans cette étude des fonctions communes de gestion afin de proposer une spécialisation des plus fréquentes d'entre elles au sein des opérations de lecture (*get*) et d'écriture (*set*). L'analyse a permis d'identifier la fonction en charge de la recherche du noeud avec une clé immédiatement supérieure ou égale à une clé de référence passée en paramètre, la fonction `lower_bound`.

Cette fonction est systématiquement utilisée dans le cadre d'une recherche de noeud, aussi bien pour les opérations de lecture que d'écriture. Composée d'une boucle lui permettant de parcourir l'ensemble de l'arbre, sa simplicité permet d'envisager une incorporation peu complexe de la totalité des opérations qu'elle réalise directement au sein de la structure des noeuds de l'arbre, permettant d'obtenir un code auto-adaptatif. L'objectif est ainsi de transformer la structure initiale de données, gérée par le code statique de l'application, en **structure de code spécialisé exécutable**, contenant les données directement dans le flot d'instructions machines. Les modifications apportées à la structure de données initiale sont présentées sur la figure 4.3, avec à gauche de la figure la structure de données initiale et à droite notre nouvelle structure, incluant le code de `lower_bound`. Le code assembleur inclus dans la structure correspond au code équivalent de la fonction `lower_bound` dans un jeu d'instructions Thumb2.

Nous présentons ci-dessous l'implémentation originale en C de `lower_bound` :

```
1 static inline
2 rb_tree_node_t* map_lower_bound( map_t* self, void* __k )
3 {
4   return rb_tree_lower_bound( &self->tree, rb_tree_root( &self->tree ), rb_tree_end( &
        self->tree ), __k );
5 }
```

```
6
7
8  rb_tree_node_t* rb_tree_lower_bound ( rb_tree_t* self, rb_tree_node_t* __x,
       rb_tree_node_t* __y, void const* __k )
9  {
10         while(__x!=0){
11                 if(!(self->key_less( &__x->value[0], __k )))
12                         __y = __x, __x = __x->left;
13                 else
14                         __x = __x->right;
15         }
16         return __y;
17 }
```

Voici notre implémentation en pseudo C, `x_rb_tree_node_t` représentant notre pseudo code inclus directement en assembleur dans la structure de noeuds présentées sur la figure 4.3 :

```
1  typedef rb_tree_node_t* (*x_rb_tree_node_t)(rb_tree_node_t*, uint32);
2
3  static inline
4  rb_tree_node_t* map_lower_bound( map_t* self, void* __k )
5  {
6      x_start=rb_tree_root(self->tree);
7      y_start=rb_tree_end(self->tree);
8      if(x_start)
9                          return ((x_rb_tree_node_t)x_start)(y_start, __k);
10     return y_start;
11 }
12
13 rb_tree_node_t*
14 x_rb_tree_node_t(rb_tree_node_t* __y, uint32_t __k )
15 {
16     if (__k <= MYKEY) {
17         __y = __x;
18         return ((x_rb_tree_node_t)MYLEFT)(__y, __k); // or return __y if _x->left == 0
19     }
20     return ((x_rb_tree_node_t)MYRIGHT)(__y, __k); // or return __y if _x->right== 0
21 }
```

La *Map* représente un ensemble de tableaux associatifs dans laquelle ces derniers sont manipulés au sein d'un arbre rouges et noirs.

### Résultats obtenus

Les expérimentations sur notre nouvelle proposition de structure ont été réalisées directement sur une cible réelle équipée d'un processeur ARM Cortex-A9 cadencé à 400 MHz et contenant 32 KO de mémoire cache, respectivement pour les données et pour les instructions. Un système d'exploitation Linux (Debian) est installé sur la carte incluant ce processeur. Afin de réaliser un programme de test réaliste pour mettre en situation notre proposition, nous avons réalisé un certain nombre de mesures sur LLC nous permettant de mettre en avant un ratio d'une écriture pour trois lectures, ces dernières ayant un ratio de succès de deux pour un (deux lectures réussies pour un échec, conséquence de la non présence du noeud dans l'arbre). Le test proposé se charge donc de générer une table de valeurs aléatoires écrites dans l'arbre (*set*) et étant ensuite lues aléatoirement (*get*), obéissant aux ratios mesurés sur LLC.

En raison de **l'absence de cohérence entre le cache de données et le cache d'instructions dans les processeurs ARM**, le transfert de notre nouvelle structure de ce premier vers ce second, engendré par l'inclusion de code exécutable au sein de

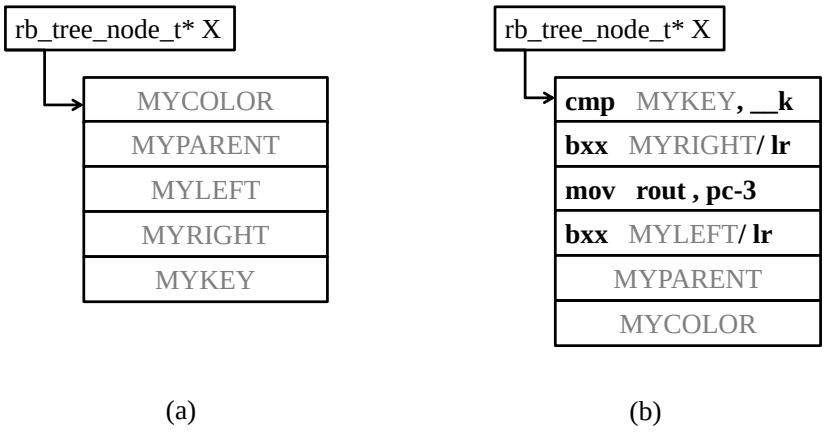



FIGURE 4.3 – (a) Structure originale des noeuds des arbres rouges et noirs, contenant seulement les données ; (b) Structure proposée incluant les données (en noir) et le code exécutable relatif à la fonction `lower_bound` (en gris), ___k représente la clé de référence passée en paramètre.

la structure, ne peut être réalisé automatiquement. Par conséquent, la cohérence doit être gérée manuellement. Aucune instruction accessible au niveau utilisateur dans le jeu d'instructions ARM ne permet de réaliser la purge du cache d'instructions. Le seul moyen d'assurer la cohérence entre les deux caches est donc d'utiliser l'appel système Linux `__clear_cache`. Cette fonction doit être utilisée à chaque modification d'un noeud au sein de l'arbre, autrement dit pour toutes les opérations d'écriture (*set*), conduisant à une forte dégradation des performances pour leur réalisation (liée à l'appel système). La criticité de cette gestion a notamment été abordée pour Android qui fournit sa propre fonction optimisée de purge du cache, plus rapide que l'appel système Linux de base. Les informations relatives à cette problématique de cohérence de cache sous ARM sont disponibles en ligne [130].

Afin de ne pas subir la pénalité induite par la fonction `__clear_cache` sur les opérations d'écriture, notre test s'est focalisé uniquement sur les opérations de lecture. Les opérations d'écriture ne pourraient être valables que si nous pouvions disposer d'un niveau d'accès supérieur au processeur ARM, nous permettant d'utiliser directement l'instruction de purge du cache d'instructions et donc d'assurer la cohérence directement au niveau processeur, sans utiliser d'appel système Linux. Cette impossibilité de l'utiliser illustre d'ailleurs le fait que les processeurs embarqués de type ARM ne sont actuellement pas conçus pour ce genre d'optimisation bas-niveau, permettant l'utilisation à grain fin de code auto-adaptatif.

Deux tests ont été réalisés afin **d'évaluer les gains potentiels obtenus** par l'utilisation de notre nouvelle structure basée sur la spécialisation de code. Le premier test a été réalisé pour une *Map* contenant 512 éléments, dans laquelle nous avons fait augmenter le nombre de lectures, en tenant compte des ratios préalablement mesurés. Les mesures portent sur la version originale, sans spécialisation de code, et sur celle obtenue par nos travaux avec la structure modifiée incluant le code exécutable de `lower_bound`. Les résultats, donnés en nombre de cycles par opérations de lecture, sont reportés sur la figure 4.4. Il est possible de constater, en comparant les deux solutions, que la spécialisation de code permet d'obtenir des gains de l'ordre de 45 % pour un faible nombre d'opérations de lecture et une stabilisation autour de 35 % jusqu'à 64 opérations de lecture. Pour les opérations de lecture suivantes, nous observons une décroissance des

performances et donc du gain obtenu, conséquence d'un début de débordement du cache d'instructions et surtout de la mémoire cache d'adresses cibles pour les branchements (BTAC, *Branch Target Address Cache*).

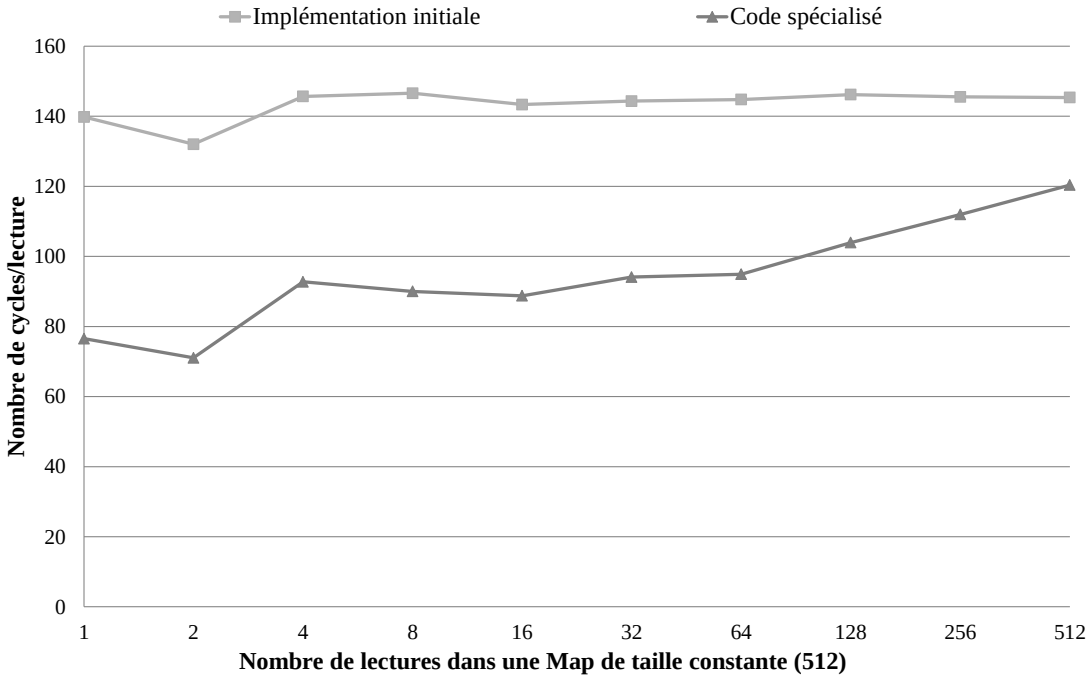



FIGURE 4.4 – Comparaison du nombre de cycles nécessaires par opération de lectures pour la solution initiale et notre code spécialisé pour une *Map* de 512 éléments, pour un nombre croissants d'opérations de lecture au sein de la *Map*. Résultats donnés en cycles/opérations.

Afin d'évaluer **l'impact asymptotique de notre optimisation**, en comparaison avec la version originale, sur les ressources matérielles, nous avons augmenté la taille de la *Map* jusqu'à 16384 éléments. Les résultats, présentés sur la figure 4.5, montrent une dégradation des performances de l'ordre de 25 % par rapport à la version originale pour un grand nombre d'éléments dans la *Map* avec multiplication des opérations de lecture. Alors que l'implémentation d'origine ne sollicite que le cache de données, notre solution s'appuie quant à elle sur le cache d'instructions, sur le prédicteur de branchement et sur le cache d'adresses cibles pour les branchements (BTAC). Bien que les caches d'instructions et de données fassent tous les deux la même taille (32 KO), les débordements de ce dernier étant beaucoup plus fréquents dans les architectures, celui-ci dispose de nombreux mécanismes permettant d'en limiter les effets. En conséquence, la dégradation des performances pour la version d'origine apparaît plus tardivement et se manifeste de manière beaucoup moins importante que dans le cadre de notre solution pour un grand nombre d'éléments, comme il est possible de l'observer sur la courbe 4.5. Ces résultats laissent même à penser que **le prédicteur de branchement et le BTAC deviennent contre-productif** pour des *Map* de grande taille.

### Conclusion de l'étude

Ces résultats permettent de mettre en avant **les limites des gains potentiels offerts par les optimisations logicielles** sur des codes semblables à ceux de la compilation dynamique. Les architectures embarquées en particulier ne disposent pas des mécanismes permettant à la fois de gérer efficacement ces codes et les optimisations proposées. Dans le cas d'étude qui vient d'être proposé, c'est la sollicitation exceptionnelle des différents mécanismes de chargement du processeur et de son cache d'instructions,

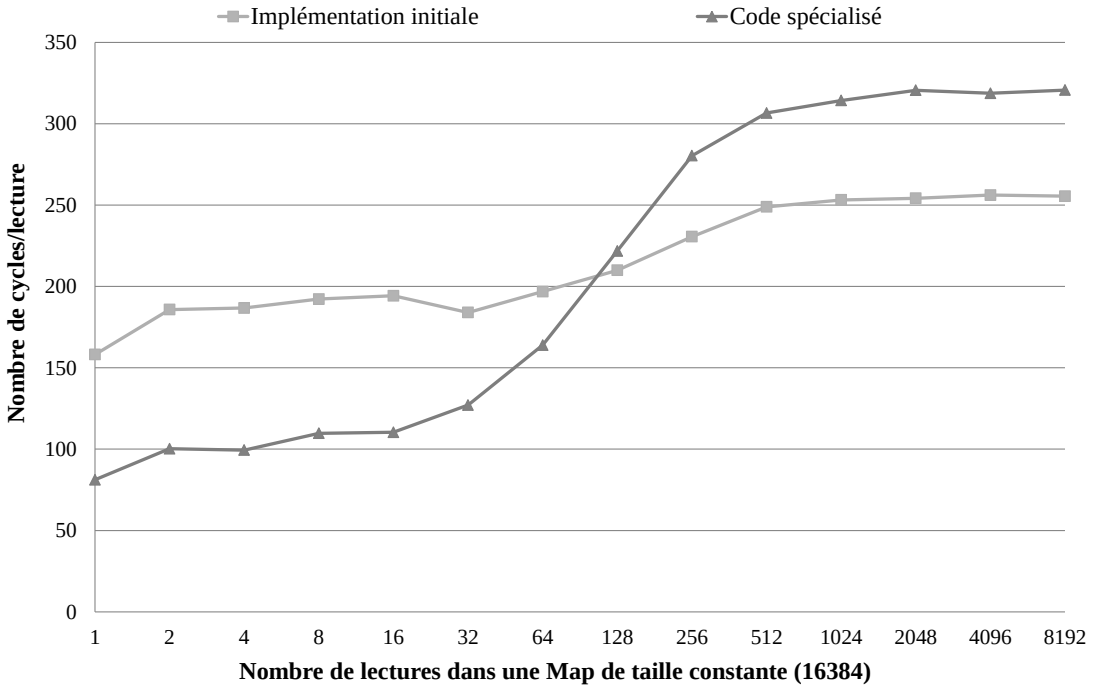



FIGURE 4.5 – Comparaison du nombre de cycles nécessaires par opération de lectures pour la version initiale et notre code spécialisé pour une *Map* de 16384 éléments, pour un nombre croissants d'opérations de lecture au sein de la *Map*. Résultats donnés en cycles/opérations.

engendrée par la spécialisation de code, qui limite les gains obtenus. A partir de ces résultats, il est possible de soulever la question de **l'opportunité de la mise en place de composants matériels spécifiques à la compilation dynamique**. Les effets sont doubles : fournir à la compilation dynamique ses propres ressources, lui permettant de ne pas utiliser celles allouées à l'exécution et donc de limiter son impact à ce niveau, et lui fournir les optimisations matérielles adéquates sur les architectures embarquées pour la rendre au moins aussi attractive que sur les architectures généralistes, de type station de travail ou serveur. Ces optimisations visent notamment à compenser la simplicité des mécanismes de prédiction et la faible taille des caches d'instructions de ces architectures. La prochaine partie de ce chapitre est justement dédiée à une analyse des optimisations existantes au niveau système, présentées dans le chapitre 2, afin de mettre en avant les forces et les faiblesses de celles-ci.

### 4.3.3 Limites des optimisations au niveau système

La seconde approche mise en avant quant aux optimisations de la compilation dynamique dans les systèmes embarqués concerne **les optimisations au niveau système**. L'objectif est de mettre en place des ressources dédiées en charge notamment de la compilation dynamique et plus globalement des services de virtualisation. Cette approche présente l'avantage majeur de limiter l'impact de la compilation dynamique en s'affranchissant du partage des ressources normalement allouées à l'exécution entre phases d'exécution et de compilation. Toutefois, ici encore, la question des efficacités énergétique et surfacique pour les codes de compilation dynamique est critique puisque subsiste le problème de l'invalidation éventuelle de l'exécution de certaines portions de code en attendant leur compilation, si cette dernière n'est pas réalisée suffisamment rapidement.

**Deux types d'optimisations** ont été mises en avant : la mise en place de **ressources dédiées spécialisées** et la mise en place, plus récente, de **ressources dédiées**

**standards**.

Concernant **les ressources dédiées spécialisées**, il est possible de mentionner l'ensemble des processeurs Java, ainsi que Transmetta Crusoe et Daisy que nous avons déjà évoqués précédemment. A l'image des processeurs et extensions Java, ces solutions n'ont connu qu'un succès très limité malgré les atouts offerts quant à leurs efficacités énergétique et surfacique. Très fortement développés au début des années 2000, le seul résultat subsistant de leur développement est ARM Jazelle, extension de processeur ARM permettant l'accélération par un jeu d'instructions étendu de la gestion des JVM ainsi que de l'exécution des programmes Java. Le défaut majeur de ces solutions concerne leur total manque de flexibilité. Toujours à l'image des processeurs Java, la réutilisabilité de ces solutions dans des domaines applicatifs différents est impossible, un processeur Java se cantonnant le plus souvent à la gestion d'une unique JVM et par conséquent à l'exécution de programmes Java seulement.

**L'optimisation plus récente consistant au déploiement de ressources dédiées standardisées**, avec notamment l'apparition de projets comme Ildjit et les travaux de Cao et al., semble être aujourd'hui une alternative prometteuse pour la démocratisation de la compilation dynamique. Associant au découplage, permettant de dissocier ressources de compilation et d'exécution, un critère de flexibilité en utilisant des ressources non spécialisées, cette solution permet une réutilisation facilitée de ces ressources pour différentes technologies. Ting Cao, par exemple, propose le transfert de services récurrents de virtualisation (ramasse-miettes, interprétation et compilation dynamique) sur ces ressources, adaptables à l'ensemble des machines virtuelles. Ildjit se contente quant à lui des ressources non utilisées d'architectures multi-coeurs pour réaliser la compilation dynamique des différentes portions de l'application.

Le défaut majeur de cette optimisation est la limitation des efficacités énergétique et surfacique offertes par celle-ci dans la gestion des codes de compilation dynamique, conséquence d'une utilisation de ressources standards embarquées et des limites précédemment mises en avant dans leur capacité à gérer ces codes. Ce point est notamment soulevé par Ting Cao à la fin de son étude. Son analyse des performances met en avant les gains importants pouvant être obtenus par ses travaux (13 % de réduction de l'impact de la virtualisation et 22 % d'accroissement des efficacités énergétique et surfacique) mais appuie sur le fait que ceux-ci sont majoritairement obtenus par le transfert des services de ramasse-miettes et d'interprétation. La forte complexité inhérente à la compilation dynamique limite les gains obtenus sur des ressources aux performances limitées. Ting Cao estime que les phases de compilation ne peuvent être gérées plus efficacement par les ressources embarquées et que seule une augmentation significative des performances de ces ressources permet de réaliser plus rapidement ces phases. Elle souligne toutefois les fortes opportunités offertes par la compilation dynamique quant aux optimisations et aux gains en performances envisageables sur les applications.

### 4.3.4 Présentation et justification de notre approche

A la lumière des résultats qui ont pu être mis en avant, nous pouvons conclure sur les fortes limitations, en l'état actuel, des gains obtenus par les propositions existantes. Toutefois, nous estimons, contrairement à Ting Cao, qu'il existe une alternative à l'augmentation brute des performances en essayant d'optimiser les efficacités énergétique et surfacique des codes de compilation dynamique sur les architectures embarquées. Les études réalisées mettent en avant le problème d'adéquation entre la complexité de ces codes et l'architecture des processeurs embarqués pour la gérer. C'est pourquoi nous proposons, dans le cadre de cette étude, **de démontrer la faisabilité de la mise en place d'extensions matérielles** à ces processeurs, sous la forme d'unités fonctionnelles ou de co-processeurs, permettant d'accroître ces efficacités et de gérer de manière

optimale la complexité des codes de compilation dynamique. Bien que notre analyse de l'état de l'art nous permette de nous positionner en faveur de la mise en place de ressources dédiées à la compilation et aux services de virtualisation, telles que proposées par Ting Cao ou dans le cadre de Ildjit, et que notre approche se base elle aussi sur cette solution, il est tout à fait concevable d'envisager l'utilisation de nos accélérations matérielles sur les ressources normalement allouées à l'exécution.

L'objectif de notre approche est de combiner à la fois flexibilité et performances. L'association d'extensions matérielles dédiées à la compilation dynamique à une ressource standard permet de combiner ces deux points. Nous nous proposons de baser nos accélérations sur les analyses réalisées dans ce chapitre, ayant mis en avant l'importance de l'impact de la gestion des tableaux associatifs, de l'allocation dynamique de la mémoire et de la gestion du graphe des instructions de l'application à compiler. Notre objectif est de mettre en place des accélérations permettant d'accroître l'efficacité de la gestion des codes de compilation dynamique avec un impact minimal sur la consommation et la surface, et d'augmenter significativement les performances obtenues pour ces codes afin de justifier l'utilisation de matériel dédié face à l'utilisation d'une ressource type processeur embarqué plus puissante.

L'approche retenue dans cette thèse se base sur l'utilisation d'**un processeur dédié à la compilation dynamique**, et potentiellement par extension aux services de virtualisation, similaire aux propositions de Ting Cao, auquel nous souhaitons greffer nos extensions. Le choix de l'architecture mono-coeur est la conséquence directe des résultats mis en avant dans le chapitre 3. La forte irrégularité des codes de compilation dynamique laisse en effet peu de place à une parallélisation de ceux-ci, les dépendances entre les différentes phases de compilation étant extrêmement fortes. La mise en place d'une parallélisation engendrerait une forte complexité de déploiement du compilateur, pour un gain que nous estimons limité sur les performances. D'ailleurs, les seules solutions basées sur la parallélisation dans l'état de l'art concernent une parallélisation à gros grain, au niveau des noyaux de l'application à compiler, pour lesquels il s'agit de dupliquer les tâches de compilation dans leur totalité (cas de Ildjit).

Notre objectif est donc de proposer un système de compilation dynamique, couplant un processeur dédié et nos propositions d'accélération, aussi bien sous la forme d'unités fonctionnelles que sous la forme de co-processeurs. Ce système est en charge par exemple de la réalisation des phases de compilation dynamique et/ou par extension de virtualisation d'une architecture hétérogène complexe embarquée, de type architecture multi ou many-coeurs. Son incorporation peut être réalisée aussi bien dans une optique d'accroissement des performances brutes (compilation multi-étages : dynamisme des applications et sensibilité au jeu de données) que dans une optique de virtualisation (interopérabilité : gestion de l'hétérogénéité, déploiement dynamique d'applications, via OpenCL par exemple).

Le système mis en place vient s'associer à cette architecture par l'intermédiaire du bus de central de cette dernière. Les requêtes au système sont gérées et envoyées par l'architecture multi-coeurs, via son service d'exécution (par exemple le système d'exploitation). Le système, une fois la compilation réalisée, envoie le code machine dans la zone mémoire prévue à cet effet pour la cible. Un schéma de l'ensemble incluant notre système de compilation dynamique est proposé sur la figure 4.6.

## 4.4 Conclusion

En conclusion de ce chapitre, nous avons pu mettre en avant **trois grands points critiques** relatifs aux temps d'exécution au sein du compilateur LLC. La gestion des tableaux associatifs et de l'allocation dynamique de la mémoire représente pratiquement

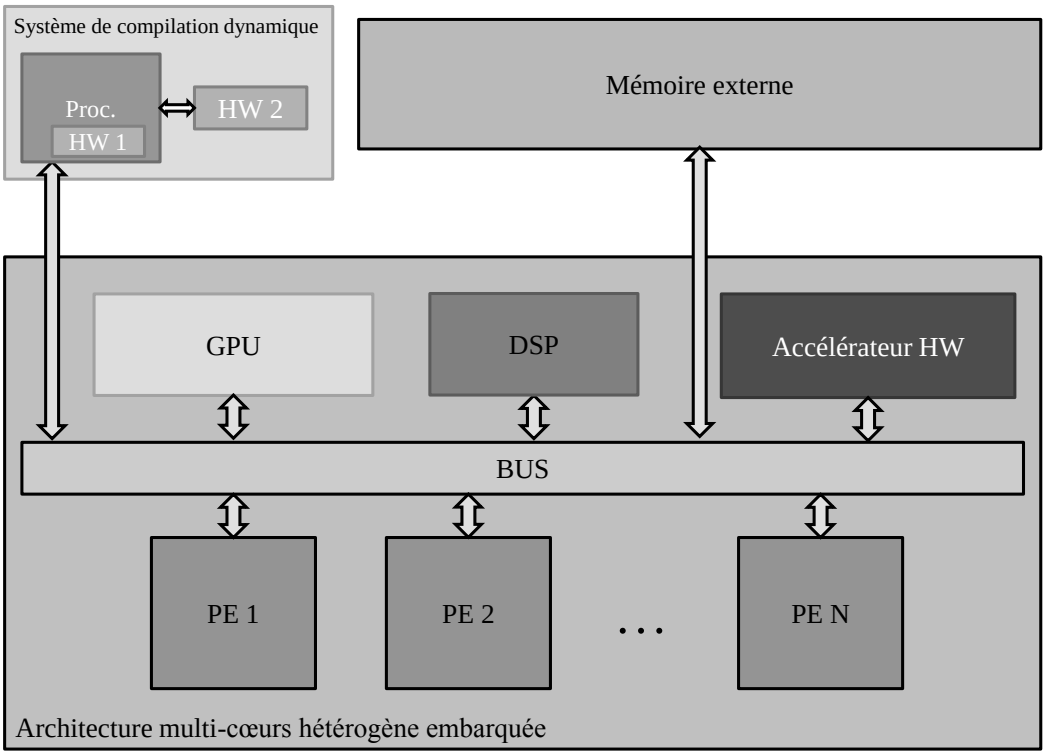



FIGURE 4.6 – Schéma de l'architecture proposée incluant le système de compilation dynamique, composé du processeur dédié à la compilation dynamique et/ou aux services de virtualisation et des accélérateurs proposés, sous la forme d'unités fonctionnelles et/ou de co-processeurs.

un quart du temps d'exécution en moyenne pour ce compilateur. L'autre point concerne la gestion du graphe des instructions de l'application à compiler, au niveau de sa représentation intermédiaire et de son code machine, qui représente, selon nos estimations, environ 20 % du temps total d'exécution. Les nombreuses optimisations existantes, qu'elles soient logicielles ou au niveau système, mettent en avant la problématique des efficacités énergétique et surfacique dans la gestion des codes de compilation dynamique en raison de leur forte complexité, liée aux irrégularités mises précédemment en avant. Les optimisations existantes s'accordent sur la limitation des gains qu'engendre cette complexité. Les processeurs embarqués ne disposent pas des mécanismes de prédiction et des tailles de caches permettant de la gérer contrairement aux architectures généralistes, type station de travail ou serveur. Nous avons pu mettre en avant ce défaut des ressources à travers une étude d'optimisation logicielle basée sur la spécialisation de code.

En se basant sur ces conclusions, nous proposons dans le cadre de cette thèse **la mise en place d'accélérations matérielles couplées au processeur en charge de la compilation dynamique**. Notre objectif est d'offrir à la fois de la flexibilité et un gain en performances significatif sous contraintes embarquées (surface et consommation), justifiant l'utilisation d'extensions matérielles en remplacement d'un processeur plus puissant. L'idée d'accélérer entièrement la compilation dynamique n'est évidemment pas concevable, sa forte irrégularité et sa forte complexité revenant quasiment à concevoir un processeur complet et une solution très peu portable, à l'image des processeurs Java et du manque de flexibilité qui les caractérise.

Notre proposition porte sur la mise en place de deux accélérateurs. Le premier d'entre eux porte sur l'accélération de la gestion des tableaux associatifs et de l'allo-

cation dynamique de la mémoire. Cet accélérateur et la démarche ayant conduit à sa réalisation sont présentés dans le chapitre suivant (Chapitre 5). Nous détaillons notamment la base commune mise en avant entre les deux points afin de proposer un accélérateur unique. Basée sur une implémentation de type unité fonctionnelle du processeur dédié à la compilation dynamique, nous montrons les gains qu'il est possible d'obtenir par son utilisation et nous évaluons le surcoût en consommation et en surface silicium que son implémentation entraîne. Nous proposons enfin une mise en application de notre accélérateur dans un cadre applicatif réel d'utilisation de la compilation dynamique. Le dernier chapitre (Chapitre 6) est quant à lui consacré à la présentation du concept de notre second accélérateur, portant sur la gestion du graphe des instructions de l'application à compiler. Conçu sous la forme d'un co-processeur couplé au processeur dédié à la compilation dynamique, nous présentons les phases de gestion que cet accélérateur permet de gérer au niveau de la représentation intermédiaire et du code machine, son fonctionnement et sa structure interne, avant d'introduire une étude d'implémentation en cours de réalisation sur ce sujet.

# Chapitre 5

# Accélération matérielle de la gestion des tableaux associatifs et de l'allocation dynamique de la mémoire

**Sommaire**

## 5.1 Introduction

Les études réalisées dans le chapitre précédent ont permis de mettre en avant l'impact de : 1) **la gestion des tableaux associatifs** et 2) **l'allocation dynamique de la mémoire** sur les temps d'exécution du compilateur LLC développé au sein du cadriciel LLVM. Ce compilateur sert aujourd'hui de brique de base à un très large panel de compilateurs dynamiques, aussi bien en milieu académique qu'industriel.

En se basant sur ces conclusions, nous proposons à travers ce chapitre la mise en place d'un accélérateur matériel pour la gestion de ces deux points critiques. Pour cela, nous avons adopté l'approche suivante, qui nous a amené à :

- analyser les optimisations logicielles existantes sous LLC qui ont été présentées dans le chapitre précédent ;
- proposer une interface commune en débarrassant ce compilateur de ces optimisations logicielles en introduisant une version normalisée de LLC, ne faisant appel qu'aux librairies standards ;
- instaurer une uniformisation de l'implémentation des tableaux associatifs dans ces librairies, les tableaux associatifs étant aussi employés pour l'allocation dynamique de la mémoire, basée sur l'utilisation des arbres rouges et noirs présentés dans le chapitre précédent (Section 4.3.2) ;
- proposer un accélérateur matériel de ces arbres rouges et noirs ;
- évaluer les gains obtenus.

L'objectif final de ce chapitre est l'obtention d'un accélérateur permettant de remplacer et de dépasser les performances des optimisations logicielles initiales de LLC, tout en fournissant une solution portable et réutilisable puisque basée sur une accélération des librairies standards. Ces travaux ont été publiés dans le cadre de **la conférence ASAP '13** [131] et font l'objet d'un **dépôt de brevet** intitulé *"Accélérateur Matériel pour la Manipulation d'Arbres Rouges et Noirs"*.

## 5.2 Retour sur les optimisations logicielles existantes de LLC

L'objectif de cette partie est de revenir sur les **optimisations logicielles proposées par LLC**, au niveau algorithmique, dans le cadre de la gestion des tableaux associatifs et de l'allocation dynamique de la mémoire. Ces optimisations ont déjà été brièvement présentées dans le chapitre précédent afin de mettre en avant les optimisations algorithmiques réalisées par les développeurs de LLVM. Ces optimisations sont cloisonnées au sein de LLC dans le répertoire ADT (*Abstract Data Type*), contenant

la définition de tous les types de gestion des données abstraites, autrement dit les différents conteneurs remplaçant ceux fournis par la librairie standard STL C++ [125] (`std::[multi]map` et `std::[multi]set`).

### 5.2.1 Retour sur les librairies ADT de LLC

Nous avons listé dans le chapitre précédent l'ensemble des conteneurs mis en place par les développeurs de LLVM afin de remplacer `std::[multi]map` et `std::[multi]set`. Cet ensemble est rappelé à titre informatif dans le tableau 5.1. L'utilisation des conteneurs de la librairie standard STL C++ est réservée aux cas non couverts par ces différents conteneurs, correspondant aux cas les moins critiques quant à leurs performances (notamment pour les éléments de grande taille).

TABLE 5.1 – Conteurs spécialisés pour la gestion des tableaux associatifs dans LLC, remplaçant respectivement `std::[multi]map` et `std::[multi]set`. Ces derniers ne sont utilisés que pour les cas non couverts par ces conteneurs spécialisés.

| `std::map` | `std::set` |
|---|---|
| DenseMap | DenseSet |
| StringMap | SmallSet |
| IndexedMap | SmallPtrSet |
| ValueMap | SparseSet |
| IntervalMap | SparseMultiSet |
| MapVector | FoldingSet |
| IntEqClasses | SetVector |
| ImmutableMap | UniqueVector |
| | ImmutableSet |

Ces conteneurs présentent notamment la particularité de limiter au maximum les implémentations de type arbres triés pour les tableaux associatifs, contrairement à la librairie STL C++. Ils utilisent à la place **une implémentation sous la forme de tables de hachage**, introduisant la notion de tableaux sans ordre pour le rangement des données, avec un accès de la clé à la valeur par utilisation d'une fonction de hachage. La complexité inhérente aux accès à travers cette implémentation est en $O(1)$ et dans le pire cas en $O(n)$, alors que la complexité dans les arbres triés se limite à une complexité en $O(logn)$, $n$ correspondant au nombre total d'éléments. Toutefois, la manipulation sous forme d'arbres triés nécessite l'utilisation de fonctions de rééquilibrage complexifiant l'utilisation de ce type d'implémentation. Aussi, pour les cas d'éléments de taille limitée et en nombre restreint, les temps d'accès et de gestion de ces arbres sont plus importants que les temps d'accès à un élément d'un tableau classique.

La librairie standard se base sur **la manipulation permanente de pointeurs** et sur l'utilisation de l'allocateur dynamique à chaque insertion ou suppression d'un élément dans l'arbre. Pour "rentabiliser" cette manipulation plus lourde, ainsi que les fonctions de tri de l'arbre qui l'accompagnent, il est nécessaire de gérer des éléments d'une taille significative et en quantité suffisante. C'est pour cela que les développeurs de LLVM ont favorisé l'implémentation basée sur les tables de hachage et les tableaux sans ordre pour les cas des conteneurs manipulant un nombre réduit d'éléments de petite taille (cas le plus fréquent dans les parties critiques du compilateur LLC). Ces tableaux permettent de limiter la manipulation de chaînes de pointeurs, contrairement aux implémentations de type arbre. Les développeurs fournissent différents conteneurs s'adaptant aux différents types de données manipulées et aux spécificités des différents cadres d'utilisation.

Les conteneurs ont notamment la propriété d'optimiser la recherche de couples (clé, valeur) pour des clés de petite taille, accélérant leur recherche au sein de tableaux. Ces tableaux sont alloués au préalable par le compilateur avec une taille fixe. Des fonctions d'agrandissement et de diminution de la taille des tableaux ont été mises en place, afin de respecter un certain nombre de conditions bien précises avant de procéder à un appel de l'allocateur mémoire. Cela évite les allocations et les désallocations récurrentes comme effectuées dans le cadre de la librairie standard STL C++ pour chaque ajout ou suppression de noeud au sein de l'arbre trié. Ces opérations sont extrêmement coûteuses en performances, spécifiquement pour l'algorithmie embarquée. Pour comprendre la vision que peuvent avoir les développeurs de LLVM sur l'utilisation de la librairie STL C++, nous nous permettons de citer cette définition, extraite du manuel programmeur de LLVM [132] :

*"*`std::set` *is a reasonable all-around set class,* ***which is decent at many things but great at nothing****.* `std::set` *allocates memory for each element inserted (thus it is very malloc intensive) and typically stores three pointers per element in the set (thus adding a large amount of per-element space overhead). It offers guaranteed log(n) performance, which is not particularly fast from a complexity standpoint (particularly if the elements of the set are expensive to compare, like strings), and has extremely high constant factors for lookup, insertion and removal. The advantages of* `std::set` *are that its iterators are stable (deleting or inserting an element from the set does not affect iterators or pointers to other elements) and that iteration over the set is guaranteed to be in sorted order.* ***If the elements in the set are large, then the relative overhead of the pointers and malloc traffic is not a big deal, but if the elements of the set are small,*** `std::set` ***is almost never a good choice.****"*

Concernant `std::map`, les développeurs de LLVM renvoient à leur définition de `std::set` que nous venons de citer.

Si nous regardons plus en détail les spécificités de l'allocation mémoire sur LLC, nous nous apercevons qu'ici encore les développeurs se détournent de l'utilisation des librairies standards, comme l'allocateur C Doug Lea *dlmalloc* [133], et proposent **la mise en place de leurs propres allocateurs spécialisés**. Parmi ces allocateurs, il est possible de citer le `RecycleAllocator`, dont nous avons déjà mentionné les fonctions au chapitre précédent : la conservation des éléments récemment désalloués afin d'éviter leur ré-allocation en cas de réutilisation immédiate.

Toutes ces optimisations ont pour conséquence d'accroître le volume de code manipulé au sein du compilateur. LLC, dont le fichier C++ de définition ne contient pas plus de quelques centaines de lignes de code, manipule au total pas moins d'un million trois cent mille lignes de code en réalisant toutes les phases de compilation. Cette spécialisation massive de LLC au niveau de la gestion des tableaux associatifs et de l'allocation dynamique de la mémoire entraîne également une complexité accrue de l'algorithme pour les développeurs souhaitant s'appuyer sur ce compilateur pour leurs projets. Elle nécessite en effet une étape de familiarisation avec les différents conteneurs afin de remplacer une éventuelle utilisation des librairies standards au sein de leurs développements dans une optique d'optimisation des performances.

## 5.3 Proposition d'une nouvelle version normalisée de LLC

La première étape abordée dans le cadre de ces travaux est la proposition de la mise en place d'une nouvelle version de LLC, basée uniquement sur l'utilisation de la librairie standard STL C++ pour la gestion des tableaux associatifs et sur l'allocateur C Doug Lea pour l'allocation dynamique de la mémoire. En d'autres termes, nous cherchons à mettre en place une version normalisée de LLC, affranchie de ses optimisations

logicielles. La réalisation de cette version normalisée a été faite à partir d'une copie du code de la version autonome de LLC utilisée dans le chapitre précédent.

### 5.3.1 Pourquoi normaliser LLC

L'objectif à travers ces travaux de **normalisation du compilateur LLC** est double.

Le premier point concerne la volonté de **trouver une base commune d'accélération**. Les optimisations logicielles réalisées par LLC sont extrêmement spécifiques d'un conteneur à l'autre. Dans le cadre de notre étude, nous cherchons à établir la possibilité d'accélérer matériellement certaines des opérations les plus critiques des codes de compilation dynamique, parmi lesquelles la gestion des tableaux associatifs et l'allocation dynamique de la mémoire. Définir une accélération sur les optimisations logicielles existantes reviendrait à trouver une base commune d'accélération potentielle entre toutes les différentes optimisations logicielles, afin de proposer un accélérateur adaptable aux contraintes de l'embarqué en surface et en consommation. Or, l'idée de trouver une accélération commune au sein d'optimisations très spécifiques est une étape au mieux extrêmement complexe, au pire impossible, pour des gains potentiels sans doute marginaux de par les faibles opportunités d'accélération (la base commune étant extrêmement réduite voire inexistante, l'importance des phases accélérées s'en trouvera considérablement limitée). Ces optimisations logicielles sont de plus sujettes à d'éventuelles évolutions d'une version de LLVM à l'autre. Même si leur niveau de maturité rend aujourd'hui ces évolutions marginales, le fait de ne pas utiliser de librairies standards rend possible, bien que peu probable, une éventuelle refonte de ces conteneurs lors de l'apparition d'une nouvelle version de LLVM. Définir un accélérateur matériel sur des portions de code pouvant connaître une telle évolution fait prendre le risque de concevoir un module pouvant devenir obsolète pour les versions futures de LLVM.

L'autre point concerne **l'aspect portabilité et réutilisabilité de la solution** mise en place. Baser notre proposition d'accélération sur les optimisations logicielles spécifiques à LLC reviendrait à en limiter l'utilisation à ce seul compilateur. Toutefois, nous pensons que la criticité de la gestion des tableaux associatifs et de l'allocation dynamique de la mémoire est un problème récurrent pour de nombreux domaines applicatifs (bases de données, interprétation de langages de scripts, etc.). C'est pourquoi nous estimons que notre proposition d'accélération peut bénéficier à un plus large domaine algorithmique que celui initialement considéré de la compilation dynamique.

Bien évidemment, une solution envisageable est de proposer aux développeurs de ce type d'application d'utiliser les conteneurs de LLVM afin d'implémenter leurs tableaux associatifs et leur allocateur. Cela demande toutefois un effort non négligeable de développement pour un gain potentiel que nous estimons limité, réduisant fortement l'attractivité de cette solution. Il est plus intéressant selon nous pour un développeur de pouvoir bénéficier de manière transparente de cette accélération en le laissant se baser sur l'utilisation des librairies standards, aujourd'hui largement répandues. Ces librairies standards offrent de plus chacune une implémentation uniformisée de leur gestion respective des tableaux associatifs et de l'allocateur mémoire, offrant de plus grandes perspectives d'accélérations à fortes efficacités énergétique et surfacique.

En d'autres termes, notre objectif est de nous appuyer sur ces librairies standards pour proposer une accélération matérielle réutilisable plus performante que les optimisations logicielles proposées par les développeurs de LLVM. Notre idée est ainsi de transférer ces optimisations du logiciel à l'accélérateur mis en place dans le cadre de cette étude.

### 5.3.2 Normalisation proposée

Notre normalisation de LLC se base sur l'utilisation de la librairie standard STL C++ pour la gestion des tableaux associatifs et sur la librairie C Doug Lea pour l'allocation dynamique de la mémoire.

Pour **la gestion des tableaux associatifs**, nous proposons ainsi de remplacer l'utilisation des dix-sept conteneurs initialement implémentés dans LLC par l'utilisation des quatre conteneurs de la librairie standard, `std::[multi]map` et `std::[multi]set`. Nous nous sommes pour cela focalisés sur les fichiers sources contenant l'ensemble de ces conteneurs, qui sont cloisonnés dans le répertoire ADT du générateur de code LLC. En procédant par héritage, nous avons imposé un usage systématique des quatre conteneurs de la librairie standard, en tenant compte des spécificités de chacun des conteneurs optimisés pour le remplacer par le conteneur le plus adapté à chaque fois. Le travail a essentiellement été réalisé dans le fichier DenseMap.h, dans lequel la majeure partie des définitions des conteneurs sont réalisées, les autre conteneurs héritant directement de ceux-ci avec des choix d'implémentations spécifiques. A noter notamment que, comme pour la librairie standard, la gestion de DenseSet est fortement couplée à celle de DenseMap, un *Set* pouvant être ici aussi vu comme une *Map* dont la valeur est un `void*`. D'ailleurs, DenseSet hérite directement de DenseMap. L'illustration d'une partie des modifications proposées est présentée dans la portion de code suivante, mettant en avant les modifications effectuées en début de structure DenseMap :

```
1  struct DenseMap
2  {
3    typedef haadt::map<KeyT, ValueT> base_type;
4
5    typedef typename base_type::iterator iterator;
6    typedef typename base_type::const_iterator const_iterator;
7    typedef typename base_type::value_type value_type;
8    typedef typename base_type::size_type size_type;
9    typedef typename base_type::key_type key_type;
10   typedef typename base_type::key_compare key_compare;
11   typedef typename base_type::allocator_type allocator_type;
12
13   typedef typename allocator_type::template rebind<value_type>::other _Pair_alloc_type;
14
15   typedef haadt::_Rb_tree<key_type, value_type, std::_Select1st<value_type>,
16                           key_compare, _Pair_alloc_type> _Rep_type;
17 ...
18 };
```

Nous pouvons y voir la création d'un nouvel espace de noms (*namespace*) HAADT, pour *Hardware Accelerated Abstract Data Types*. Cet espace contient une copie de la librairie standard STL C++ pour les seuls conteneurs nous intéressant : `std::[multi]map` et `std::[multi]set`. Puisque nous travaillons sur une version complètement autonome de LLC, nous avons rapatrié l'ensemble des fichiers nécessaires au fonctionnement de ces conteneurs, avec notamment les fichiers de gestion de l'implémentation des tableaux associatifs, c'est-à-dire ceux relatifs à la gestion des arbres rouges et noirs. Cet espace contient les fichiers suivants : haadt_map, haadt_map.hh, haadt_multimap.hh, haadt_multiset.hh, haadt_set, haadt_set.hh, haadt_tree.hh et haadt_tree.cpp. Nous obtenons ainsi quatre nouveaux conteneurs issus de la librairie standard et qui sont les seuls conteneurs manipulés par LLC. Nos travaux pour les types de données abstraites sont résumés sur la figure 5.1.

Concernant **l'allocation dynamique de la mémoire**, nous proposons l'utilisation systématique de l'allocateur C Doug Lea, *dlmalloc* [133]. Cet allocateur est aujourd'hui utilisé aussi bien dans la librairie GNU C standard que dans la librairie NewLibC [134]

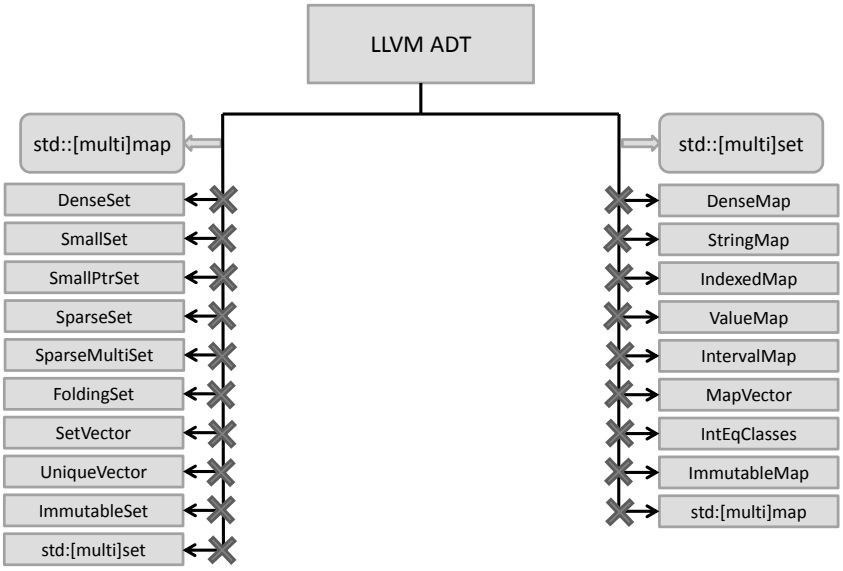



FIGURE 5.1 – Présentation schématique de nos travaux de normalisation de LLC pour les ADT, remplaçant l'ensemble des conteneurs spécifiques à LLC par ceux de la librairie standard STL C++.

ou dans Firefox. Il a été développé en 1987 par Doug Lea, nous reviendrons en détail sur les implémentations réalisées au sein de cet allocateur dans la suite de ce chapitre. Nous opérons donc la ré-implémentation au sein de LLC de l'opérateur `new[]`, afin que celui-ci utilise uniquement `malloc`, ainsi que l'utilisation systématique des fonctions `malloc`, `realloc`, `calloc` et `free`, et cela quelle que soit la taille des éléments manipulés, contrairement à l'implémentation originale de LLC qui utilise des allocateurs spécifiques en fonction de la taille des éléments et notamment des conteneurs associés. Voici les fonctionnalités associées à chacune des fonctions :

- `malloc` : réservation dynamique d'une zone mémoire, sans initialisation automatique ;
- `realloc` : modification de la taille de la zone mémoire précédemment allouée par `malloc` avec création si nécessaire d'une nouvelle zone et libération automatique de l'ancienne ;
- `calloc` : réservation dynamique d'une zone mémoire avec initialisation automatique de l'ensemble de la zone à 0 ;
- `free` : libération dynamique d'une zone mémoire.

Nous avons rapatrié l'implémentation de l'allocateur Doug Lea au sein de notre espace HAADT, proposant l'inclusion au sein de celui-ci des fichiers suivants : heap.h, allocator.hh, allocator.cpp et contenant l'ensemble des définitions des différentes fonctions manipulées, ainsi que la gestion de l'allocation dynamique de la mémoire.

Nous obtenons ainsi **une nouvelle version de LLC, appelée version normalisée**. Nous proposons la création d'un nouvel espace de noms, appelé HAADT, permettant de fournir aux éventuels utilisateurs l'ensemble des fichiers nécessaires à cette normalisation au sein d'un répertoire et d'un espace de noms uniques. Ce choix permet de cloisonner complètement nos modifications au sein du compilateur afin d'en limiter l'impact sur sa structure.

Les modifications apportées pour les types de données abstraites sont essentiellement contenues au sein du répertoire ADT de définition des conteneurs originaux de LLC. Même si nous avons essayé de limiter au maximum les modifications au sein du compilateur, certains autres fichiers doivent également être modifiés afin que ces der-

niers utilisent nos conteneurs et non les conteneurs originaux. Toutefois, la très grande majorité de ces modifications concernent uniquement la simple modification de l'espace de noms `std` par `haadt` pour les cas d'utilisation des conteneurs de la librairie standard directement dans le code du compilateur (sans passer par les conteneurs de LLC). Cela permet de rendre systématique l'utilisation de nos fichiers de définition et d'assurer l'autonomie de la version normalisée proposée de LLC. La liste des autres fichiers à modifier est extrêmement limitée au regard du nombre total de fichiers contenus au sein du compilateur.

Concernant l'allocation dynamique de la mémoire, les modifications à apporter touchent cette fois-ci un plus grand nombre de fichiers, mais sont extrêmement récurrentes puisqu'il s'agit là encore essentiellement d'ajouter un préfixe `haadt::` qui donne accès à notre propre version de l'allocateur C Doug Lea et dont les fichiers de définition sont directement inclus au sein du compilateur.

Finalement, cette normalisation de LLC, qui nous a demandé plusieurs mois de travail afin d'identifier les différentes modifications à apporter et de rendre l'ensemble fonctionnel, ne représente qu'un temps de développement limité pour les utilisateurs souhaitant l'inclure au sein de leurs propres versions. Ce choix de réduire au maximum l'impact de nos modifications sur l'ensemble du code est également très utile pour passer d'une version à une autre de LLVM, les changements de version étant extrêmement fréquents (une nouvelle version tous les six mois environ). Il est donc largement envisageable selon nous d'imaginer un déploiement à plus grande échelle de cette version normalisée pour les utilisateurs qui le souhaitent et d'en proposer un suivi pour les versions futures de LLVM (nous sommes actuellement basés sur la version 2.9, datant de 2011).

Bien évidemment, cette version débarrassée des optimisations logicielles va automatiquement souffrir d'importantes pertes en performances en comparaison avec la version initiale. Nous évaluerons les pertes dans la partie relative aux évaluations des performances de ce chapitre. L'objectif de l'accélérateur est de dépasser ce surcoût induit et de proposer des gains significatifs par rapport à la version optimisée logiciellement, justifiant ainsi l'utilisation de matériel dédié et spécialisé dans le cadre de la gestion des tableaux associatifs et de l'allocation dynamique de la mémoire.

## 5.4 Identification des accélérations potentielles au sein des librairies standards

L'objectif de cette section est de proposer **une analyse des librairies standards, librairie STL C++ et allocateur mémoire C Doug Lea**, introduites dans la version normalisée de LLC dont nous venons de détailler la mise en place. Nous cherchons à travers cette analyse à identifier des pistes d'accélérations potentielles au sein de ces librairies. Notre objectif étant de proposer une solution à fortes efficacités énergétique et surfacique, puisque se destinant à une utilisation dans un contexte embarqué, nous souhaitons mettre en avant des pistes d'accélérations significatives et communes aux deux librairies, afin de proposer un seul et unique accélérateur pour l'ensemble.

### 5.4.1 Mise en évidence de la gestion de l'allocation dynamique de la mémoire par les tableaux associatifs

L'analyse de l'allocateur C Doug Lea a permis de montrer que **les données étaient manipulées au sein de cet allocateur par l'intermédiaire de tableaux associatifs**. Ces tableaux associatifs permettent entre autres la gestion des zones libres de la mémoire en fonction de leur taille dans le cadre d'une allocation de la mémoire.

L'allocateur Doug Lea ne gère que les portions de mémoire non-allouées et l'allocation de ces dernières. Autrement dit, dès qu'une opération de type `malloc` est effectuée, la zone allouée n'est plus gérée par l'allocateur. Elle sera à nouveau gérée lorsqu'elle sera libérée par la fonction `free`. La robustesse de cet allocateur permet de rendre cette gestion suffisante pour assurer un bon fonctionnement de la gestion de l'allocation mémoire. Les zones allouées de la mémoire ne sont absolument pas gérées par l'allocateur jusqu'à leur libération.

Une zone mémoire allouée est manipulée sous la forme d'une structure de données alignées de 8 octets, contenant une entête et un espace mémoire utilisable. Lorsque cette zone mémoire est désallouée, elle est manipulée par l'allocateur sous la forme d'une structure de données contenant des informations relatives à sa taille et aux liens avec les autres zones libres de la mémoire. Elle est insérée au sein de la *Map* contenant, sous la forme de tableaux associatifs, l'ensemble des zones libres. L'implémentation choisie pour ces tableaux associatifs repose sur l'utilisation d'une table de hachage pointant sur une liste doublement chaînée contenant ces zones libres.

### 5.4.2 Analyse de l'implémentation des tableaux associatifs dans les librairies standards

Concernant **la librairie standard STL C++**, cette analyse a déjà été réalisée et présentée au chapitre précédent (Chapitre 4). Notre tentative d'optimisation logicielle de la gestion des tableaux associatifs pour la compilation dynamique nous a amené à la proposition d'une spécialisation de code pour la gestion des arbres rouges et noirs (Section 4.3.2), utilisés pour l'implémentation des tableaux associatifs dans la librairie STL C++.

Pour rappel, **les arbres rouges et noirs** sont des arbres binaires triés auxquels est rajoutée une propriété de coloration permettant d'assurer l'équilibrage de l'arbre. Ainsi, le chemin le plus long au sein de l'arbre n'est jamais plus du double du chemin le plus court. Cet aspect permet d'assurer une complexité d'accès à l'arbre en $O(logn)$. Durant l'étude réalisée dans le précédent chapitre, nous avons pu constater que les arbres rouges et noirs sont l'implémentation uniforme qui a été choisie par les concepteurs de la librairie STL C++. La tentative proposée d'optimisation logicielle par spécialisation de code a permis de mettre en avant des opérations de gestion simples et régulières pour ces arbres. Les fonctions les plus courantes consistent à parcourir l'arbre, rechercher un noeud via sa clé à partir d'un noeud ou d'une clé de référence, insérer ou supprimer des noeuds, trouver une position au sein de l'arbre et réaliser l'équilibrage de cet arbre.

### 5.4.3 Proposition d'une uniformisation de l'implémentation des tableaux associatifs par les arbres rouges et noirs

Nous venons de mettre en avant **une utilisation systématique des tableaux associatifs** dans les deux librairies pour la manipulation de leurs données. Toutefois, l'implémentation de ces tableaux associatifs est réalisée différemment entre ces deux librairies : soit par table de hachage avec une liste doublement chaînée soit par l'utilisation d'un arbre binaire trié coloré dit arbre rouge et noir.

Afin de proposer une base commune dans l'implémentation de ces tableaux associatifs dans l'optique d'une conception d'un accélérateur commun aux deux librairies, nous avons réalisé une **modification de l'implémentation de ces tableaux au sein de l'allocateur mémoire Doug Lea**, afin que celui-ci repose également sur **les arbres rouges et noirs**.

Des études portant sur la modification de l'implémentation des tableaux associatifs au sein de cet allocateur mémoire ont déjà été réalisées, notamment par Emery D.

Berger, Benjamin G. Zorn et Kathryn S. McKinley dans [135]. Dans cette étude, des mesures de performances sont réalisées pour différentes implémentations, et notamment celle basée sur l'utilisation des arbres rouges et noirs. Les résultats démontrent que l'implémentation sous la forme de tables de hachage et de listes doublement chaînées reste en moyenne la meilleure implémentation possible. Les autres implémentations ne permettent d'obtenir des gains que dans certains cas et sont en moyenne moins performantes.

Toutefois, nous souhaitons, contrairement à cette étude réalisée pour une exécution simple sur processeur, proposer une accélération matérielle de ces arbres rouges et noirs. Aussi, la possibilité de permettre à l'allocateur mémoire de bénéficier de cet accélérateur est un aspect non négligeable justifiant une modification de l'implémentation des tableaux associatifs au sein de l'allocateur, même si celle-ci apparaît comme sous-optimale.

L'objectif est de proposer une modification transparente pour l'utilisateur en ne modifiant pas l'interface d'utilisation de l'allocateur, et notamment des différentes fonctions `malloc`, `calloc`, `realloc` et `free`. Les modifications se focalisent sur l'intérieur même de l'allocateur en changeant l'implémentation proposée. Pour cela, nous modifions la structure de gestion des zones libres de mémoire et nous rajoutons tous les éléments relatifs à la gestion sous la forme d'arbre (gestion parents, fils gauche/droite, etc.), en faisant appel aux fonctions de gestion des arbres contenues dans les fichiers haadt_tree.cpp et haadt_tree.hh (gestion de l'insertion, de la suppression, de la recherche avec parcours d'arbre, de l'équilibrage, etc.). Dans le cadre de l'allocation mémoire, la clé utilisée dans les tableaux associatifs pour la librairie standard correspond à la taille de la zone mémoire libre considérée.

Cette modification est illustrée par la nouvelle structure de gestion des zones libres de la mémoire (*free memory chunks*) afin de l'adapter à une utilisation sous la forme d'un noeud d'arbre rouge et noir :

```
1   typedef struct chunk_s {
2     int32_t color;
3     struct chunk_s* parent; /* location of previous used chunk
4                 (if used), parent node (if free). */
5     struct chunk_s* left; /* left chunk (if free) */
6     struct chunk_s* right; /* right chunk (if free) */
7                 intptr_t   key;
8   } chunk_t;
```

En comparaison avec l'ancienne structure pour une implémentation sous la forme de liste chaînée :

```
1  typedef struct chunk_s {
2    uintptr_t prev_size; /* Size of previous chunk (if free). */
3    uintptr_t size; /* Size in bytes, including overhead. */
4    struct chunk_s* fd; /* double links -- used only if free. */
5    struct chunk_s* bk;
6  } chunk_t;
```

Les raisons nous ayant poussés à préférer l'implémentation sous la forme d'arbres rouges et noirs sont directement tirées des conclusions de l'étude présentée dans le chapitre précédent sur la tentative de spécialisation de code que nous avons réalisé sur une des fonctions de gestion de ces arbres. Nous avons tout d'abord pu mettre en avant les limites des optimisations logicielles possibles sur les codes irréguliers et notamment ceux relatifs à la gestion des tableaux associatifs par les arbres rouges et noirs. Cette limitation illustre le manque de matériel sur les processeurs embarqués pour gérer ce genre de codes et la nécessité de mettre en place des accélérateurs spécifiques à cette gestion.

Nous avons pu également mettre en avant la grande simplicité, la forte régularité et la redondance des opérations réalisées au sein des différentes fonctions de gestion de ces arbres rouges et noirs, offrant de grandes opportunités d'accélération. Nous avons identifié huit fonctions de ce type potentiellement accélérables, réalisant les opérations élémentaires suivantes : comparaison de clés, parcours au sein de l'arbre, modification de la structure de noeud (ajout/suppression de noeud, rééquilibrage de l'arbre).

Il est à noter, concernant l'allocation dynamique de la mémoire, la grande similitude entre le fonctionnement de `malloc` et de `lower_bound`, cette dernière ayant déjà été présentée au chapitre précédent (Chapitre 4). Dans les deux cas, à partir d'un élément de référence, soit une taille, soit une clé de référence, il convient de trouver la zone libre de mémoire ou le noeud avec l'élément immédiatement supérieur ou égal à cet élément de référence. Dans le cadre de l'allocation dynamique de la mémoire, il s'agit de la zone libre de mémoire de taille immédiatement supérieure ou égale à celle de l'élément à insérer. Dans le cadre de la gestion des arbres rouges et noir, il s'agit de la clé du noeud. Les fonctions `malloc`, et par propagation `realloc` et `calloc`, sont donc très facilement modifiables pour être adaptées à une implémentation des tableaux associatifs sous la forme d'arbres rouges et noirs en utilisant directement la fonction `lower_bound`.

Ces éléments mettent en avant la nécessité de l'introduction de matériel dédié pour la gestion des arbres rouges et noirs et permettent d'imaginer la réalisation d'une accélération simple, afin d'obtenir des gains significatifs tout en satisfaisant les contraintes inhérentes à l'embarqué (il ne s'agit pas de concevoir une accélération d'une complexité proche de celle d'un processeur complet, pour des questions d'intérêt, de consommation et de superficie).

### 5.4.4 Conclusion sur ces études

Ces études nous ont permis de proposer une nouvelle version normalisée de LLC, basée uniquement sur l'utilisation des librairies standards STL C++ et Doug Lea, respectivement pour la gestion des tableaux associatifs et l'allocation dynamique de la mémoire. De plus, nos analyses ont permis de mettre en avant l'utilisation systématique des tableaux associatifs au sein de l'allocateur Doug Lea. Nous proposons une implémentation commune de ces tableaux associatifs au sein de ces librairies, basée sur les arbres rouges et noirs, en modifiant celle de l'allocateur Doug Lea initialement basée sur l'utilisation de tables de hachage et de listes chaînées. Cette base commune nous permet d'envisager **la proposition d'une accélération matérielle pour la gestion de ces arbres rouges et noirs**, que nous présentons dans la prochaine section de ce chapitre.

## 5.5 Accélération matérielle en charge de l'accélération de la gestion des arbres rouges et noirs

En nous basant sur les résultats de la section précédente, nous bénéficions maintenant de **librairies standards utilisant la même implémentation des tableaux associatifs**, utilisés pour la gestion des données abstraites qu'elles manipulent. Cette implémentation est basée sur **les arbres rouges et noirs**. La régularité et la simplicité des opérations de gestion relatives à ces arbres rendent possible la mise en place d'une accélération matérielle de ces opérations, justifiée par leur utilisation massive. L'objectif de cette section est de présenter une proposition d'accélération tirée de ces conclusions, ainsi que les possibilités d'implémentation de cet accélérateur. Nous présentons également les adaptations réalisées au sein de la structure de noeud de ces arbres afin de la

rendre adaptée à une utilisation sur du matériel dédié.

### 5.5.1 Nouvelle structure de noeud pour les arbres rouges et noirs

La structure initiale de noeud pour les arbres rouges et noirs, telle que présentée sur la partie gauche de la figure 5.2, et déjà présentée dans le chapitre précédent, est composée d'une partie contrôle de taille fixe, 128 bits, et d'une partie variable dans laquelle est contenue la clé (à laquelle est associée la valeur du tableau associatif). La partie contrôle se compose des champs permettant d'assurer le lien avec le noeud parent, le fils gauche et le fils droite. Elle contient également un champ couleur pour la propriété de coloration relative au noeud (rouge ou noir). Chacun de ces champs est codé sur un entier de 32 bits dans le cadre d'une utilisation de la librairie sur une architecture 32 bits. Le champ clé est quant à lui de taille variable, dépendant de la taille de celle-ci et donc de son type. Cela nous donne finalement une structure de taille variable, d'au minimum 128 bits.

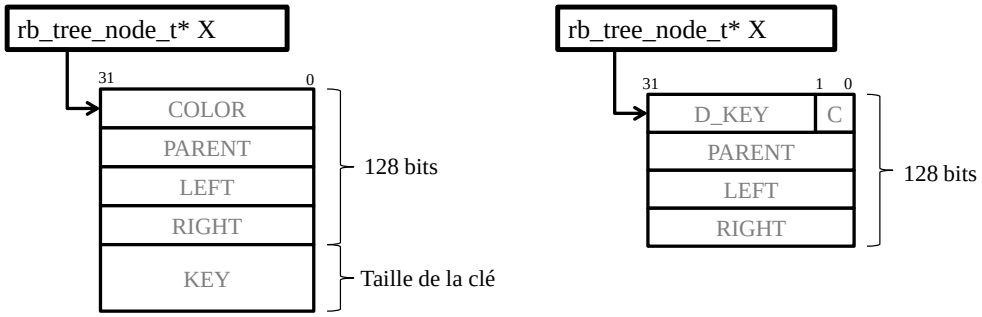
Or, dans l'optique de la mise en place d'une accélération matérielle de la gestion des arbres rouges et noirs, il est préférable que chacun des noeuds présente une taille de structure constante et prédéfinie. Cela permet de faciliter les manipulations de ces structures au sein de l'accélérateur et rend notamment envisageable une manipulation au sein de registres d'une taille optimale, correspondant précisément à la taille des structures manipulées. Cela évite ainsi de rendre impossible l'utilisation de registres (dans le cas d'une structure de taille trop importante) ou d'utiliser des registres plus grands que prévus (pour une structure de taille inférieure), avec les conséquences engendrées quant aux performances et à la surface de la solution.

C'est pour cela que nous proposons **la création d'une nouvelle structure répondant à cette contrainte de taille fixe**. Nous remplaçons pour cela la clé initiale, de taille variable, par une nouvelle clé résumée, de taille fixe et nommée D_KEY pour *Digest Key* de 31 bits. Cette clé résumée représente la clé initiale en préservant l'ordre de tri au sein de l'arbre. Toutefois, la réduction de la taille de la clé transforme les propriétés d'ordre de l'arbre, bien que ce dernier soit conservé. L'ordre initial de l'arbre est en effet total, c'est à dire que tous les éléments de l'arbre peuvent être comparés les uns avec les autres afin d'en déterminer l'ordre. En proposant l'inclusion d'une clé résumée, nous introduisons un ordre partiel sur l'arbre, conséquence de la compression de l'information sur 31 bits. Il est par conséquent impossible de déterminer complètement l'ordre entre deux clés par comparaison de celles-ci. Si les informations de supériorité et d'infériorité sont conservées (une clé de 31 bits inférieure ou supérieure à une autre correspondra bien à une clé de 128 bits inférieure ou supérieure à cette même clé), les informations d'égalité deviennent quant à elle partielles (deux clés égales sur 31 bits ne correspondront pas forcément à deux clés égales sur 128 bits), nous contraignant à repasser sur 128 bits pour vérifier cette égalité. Cela nécessite le recours à un post-traitement logiciel pour résoudre cette ambiguïté d'ordonnancement. Toutefois, nos expérimentations sur le sujet montrent que ces cas de figure sont rares.

Le choix des 31 bits est conditionné par une autre modification au sein de la structure de noeud des arbres rouges et noirs. Le champ de couleur dans la structure initiale est encodé sur 32 bits. Or il n'existe que deux couleurs possibles au sein de l'arbre. Nous nous proposons donc d'encoder cette couleur sur un seul et unique bit, afin notamment d'éviter une augmentation inutile du coût et de la complexité de l'accélérateur. En l'associant à la clé résumée, avec ses 31 bits sur les bits de poids forts et le bit de couleur sur le bit de poids faible, nous obtenons un nouveau champ de 32 bits, remplaçant le champ initial de la couleur. Le champ clé initial, de taille variable, est, quant à lui, supprimé de la structure.

Nous obtenons ainsi une nouvelle structure, présentée à droite de la figure 5.2, **de**

**taille fixe de 128 bits**. Outre le fait que cette structure soit forcément d'une taille inférieure ou égale à la taille de la structure initiale et de taille fixe, elle présente aussi une propriété intéressante au niveau de l'accès mémoire puisqu'elle est d'une taille multiple d'une puissance de 2, permettant un accès direct à la donnée sans avoir à procéder à un quelconque masquage.




(a) Structure initiale

(b) Structure proposée

FIGURE 5.2 – Structure initiale et structure proposée de noeud dans les arbres rouges et noirs dans le cadre d'une utilisation sur processeur 32 bits. La structure initiale, à gauche, est composée de quatre champs de contrôle de taille fixe et une clé de taille variable. La structure proposée, à droite, propose la suppression du champ clé initial et l'introduction d'une clé résumée sur 31 bits associée à la couleur sur 1 bit.

Voici un exemple d'implémentation de la structure dans le cadre de l'allocation dynamique de la mémoire, à mettre en comparaison avec la structure préalablement présentée :

```
1  typedef struct chunk_s {
2    uintptr_t key; /* 31..2: size in words, 1: used, 0: color (if free), n/a (if used).
         */
3    struct chunk_s* parent; /* location of previous used chunk (if used), parent node (
         if free). */
4    struct chunk_s* left; /* left chunk (if free) */
5    struct chunk_s* right; /* right chunk (if free) */
6  } chunk_t;
```

Le défaut majeur introduit par cette nouvelle structure est l'obligation pour le développeur de gérer la particularité du premier champ, en étant notamment capable d'assurer l'extraction des 31 bits de poids forts pour récupérer la clé et la séparer du bit de couleur, correspondant au bit de poids faible. Un pseudo exemple de méthode permettant de réaliser cette opération est donné ci-dessous :

```
1 template <>
2 struct HAMapInfo <mykey_t> {
3        int get31msb( mykey_t const& __k ) {
4        return __k[31:1];
5        }
6 };
```

### 5.5.2 Présentation des fonctions accélérées

L'analyse réalisée sur l'algorithme de gestion des arbres rouges et noirs a permis de mettre en avant **huit fonctions C++** potentiellement accélérables. Voici la liste de ces huit fonctions :

- `_Rb_tree_increment` : chercher, dans un arbre rouge et noir, le noeud "successeur" d'un noeud donné, c'est-à-dire le noeud présentant une clé de valeur immédiatement supérieure. Le paramètre de la fonction est l'adresse du noeud dont le successeur doit être trouvé. La valeur de retour de la fonction est l'adresse de ce noeud successeur.
- `_Rb_tree_decrement` : chercher, dans un arbre rouge et noir, le noeud "prédécesseur" d'un noeud donné, c'est-à-dire le noeud présentant une clé de valeur immédiatement inférieure. Le paramètre de la fonction est l'adresse du noeud dont le prédécesseur doit être trouvé. La valeur de retour de la fonction est l'adresse de ce noeud prédécesseur.
- `_R_tree_rebalance_for_erase` : chercher, dans un arbre rouge et noir, un noeud dont l'adresse est fournie en entrée, le supprimer et assurer le rééquilibrage de l'arbre en tenant compte de cette suppression de noeud. Les paramètres de la fonction sont l'adresse de l'entête de l'arbre et celle du noeud à supprimer. La valeur de retour de la fonction est l'adresse du noeud de l'arbre supprimé, correspondant au deuxième paramètre de la fonction.
- `_Rb_tree_insert_and_rebalance` : insérer, dans un arbre rouge et noir, un noeud dont l'adresse est fournie en paramètre de la fonction et assurer le rééquilibrage de l'arbre en tenant compte de ce nouveau noeud. Les paramètres de la fonction sont l'adresse du point d'accès à l'arbre (noeud d'entête) et celle du noeud à ajouter.
- `_M_lower_bound` : chercher, dans un arbre rouge et noir dont l'adresse d'entête est fournie en premier paramètre de la fonction, le premier noeud dont la clé est supérieure ou égale à une clé de référence fournie en tant que deuxième paramètre de la fonction et retourner l'adresse de ce noeud.
- `_M_upper_bound` : chercher, dans un arbre rouge et noir dont l'adresse d'entête est fournie en premier paramètre de la fonction, le premier noeud dont la clé est strictement supérieure à une clé de référence fournie en tant que deuxième paramètre de la fonction et retourner l'adresse de ce noeud.
- `_S_maximum` : chercher, dans l'arbre rouge et noir manipulé au moment de l'appel de cette fonction, le noeud à l'extrême droite de la branche considérée à partir du noeud fourni en paramètre de la fonction. Si celui-ci correspond au noeud racine de l'arbre, la fonction retourne le noeud à l'extrême droite de l'arbre, avec la clé maximale. La fonction retourne l'adresse de ce noeud.
- `_S_minimum` : chercher, dans l'arbre rouge et noir manipulé au moment de l'appel de cette fonction, le noeud à l'extrême gauche de la branche considérée à partir du noeud fourni en paramètre de la fonction. Si celui-ci correspond au noeud racine de l'arbre, la fonction retourne le noeud à l'extrême gauche de l'arbre, avec la clé minimale. La fonction retourne l'adresse de ce noeud.

Comme nous l'avons mentionné précédemment, les fonctions relatives à l'allocation dynamique de la mémoire, à savoir `malloc`, `calloc`, `realloc` et `free`, réalisent pratiquement les mêmes opérations que celles effectuées par la fonction `_M_lower_bound` de gestion des arbres rouges et noirs. Elles peuvent donc bénéficier directement de son accélération.

### 5.5.3 Implémentations possibles de l'accélérateur

**Deux types d'implémentation** de l'accélérateur ont été identifiées. La première implémentation possible concerne **la mise en place d'un co-processeur**, associé au processeur en charge de la compilation dynamique et plus globalement des services de virtualisation, pour lequel nous souhaitons accélérer la gestion des tableaux associatifs et de l'allocation dynamique de la mémoire. Les communications entre le processeur,

la mémoire et l'accélérateur sont assurées via un bus de communication. L'accélérateur est alors traité par le processeur comme un périphérique et communique avec celui-ci à l'aide de fonctions systèmes et de requêtes. L'avantage majeur de cette implémentation est sa facilité de mise en place en comparaison avec la solution suivante, puisque tous les processeurs permettent aujourd'hui l'inclusion de nouveaux périphériques. Elle offre également une portabilité simplifiée de l'accélération d'un processeur à un autre, avec un niveau d'intrusion limité au sein de ce dernier, limitant les modifications à y apporter. En revanche, son défaut majeur concerne les latences introduites au niveau des communications via le bus.

L'autre implémentation porte sur **la création d'une unité fonctionnelle**, au coeur même du processeur, en incluant notamment l'accélérateur au sein de son pipeline avec les autres unités fonctionnelles. La sollicitation de l'accélérateur se fait par le développement d'une extension du jeu d'instructions initial, en proposant une série d'instructions spécialisées pour cet accélérateur. L'avantage de cette implémentation est l'absence de latences de communication entre le processeur et l'accélérateur, les communications se faisant via les instructions spécialisées et les registres existants du processeur pour la récupération des différents paramètres propres aux fonctions accélérées. Elle permet également de ne pas augmenter le trafic de données avec l'incorporation de requêtes de communication puisqu'il s'agit simplement de détourner les données initiales des unités fonctionnelles de base vers la nouvelle unité. Cependant, son défaut majeur est son absence totale de cloisonnement entre l'accélérateur et le processeur, limitant fortement la portabilité de l'accélération. L'ajout d'une unité fonctionnelle engendre en effet une quantité significative de modifications au coeur même du processeur, allant de modifications micro-architecturales significatives à la refonte partielle de la chaîne de compilation, en passant par la chaîne de débogage du processeur.

Dans les deux cas, un point essentiel est **le partage de la mémoire** entre le processeur en charge de la compilation dynamique et l'accélérateur matériel. En effet, les deux unités vont manipuler les différents noeuds des arbres rouges et noirs. Il est donc indispensable d'assurer la cohérence au niveau des données manipulées, les modifications devant se répercuter sur les deux unités. Dans le cas d'une implémentation type unité fonctionnelle, il est possible d'assurer directement cette cohérence par l'intermédiaire du partage du cache de données entre les deux entités, autrement dit le partage du premier niveau de mémoire (L1). Ainsi, aucun mécanisme de cohérence n'est nécessaire, les modifications de part et d'autre se répercutant aussitôt sur l'autre unité. Dans le cas d'une implémentation de type co-processeur, il est important d'assurer la cohérence des données manipulées entre les deux unités. Par exemple, si elles partagent toutes les deux une mémoire de type L3, chaque modification réalisée par l'accélérateur devra être répercutée sur tous les niveaux de mémoire du processeur, jusqu'à sa mémoire cache L1, pour assurer la cohérence. Cela oblige à mettre en place ces mécanismes de cohérence au sein du processeur. L'implémentation type unité fonctionnelle est donc plus simple à mettre en place à ce niveau.

### 5.5.4 Présentation de l'implémentation choisie

Nous avons choisi, dans le cadre de cette étude, de retenir **l'implémentation sous la forme d'unité fonctionnelle**. Elle constitue, selon nous, l'implémentation la plus intéressante à mettre en place compte-tenu des raisons précédemment évoquées, notamment sur les questions de performances et de cohérence des données en mémoire. Comme déjà mentionné, notre objectif est de mettre en place un système de compilation dynamique à fortes efficacités énergétique et surfacique, en associant nos accélérations à un petit processeur embarqué en charge des services de virtualisation et notamment de la compilation dynamique.

Nous avons donc décidé de nous baser sur le processeur précédemment utilisé pour analyser les portions critiques de code sous LLC : le ARM Cortex-A5 [124], déjà présenté au chapitre précédent. Le choix de ARM est bien évidemment consécutif à la prédominance de l'usage de ces processeurs aujourd'hui dans le domaine embarqué. Le choix de ce processeur en particulier est lié au fait que nous bénéficions, comme déjà mentionné, d'un simulateur de jeu d'instructions fonctionnel pour celui-ci, associé à un simulateur de cache. De plus, le Cortex-A5, plus petit processeur de la famille Cortex-Ax, est un processeur RISC simple à exécution dans l'ordre (*in-order*), correspondant parfaitement au type de processeur que nous visons dans l'optique d'une mise en place d'une ressource dédiée à la compilation dynamique. Pour rappel, la version utilisée dans le cadre de notre étude comprend un cache de données et un cache d'instructions de 32 KO chacun. Le simulateur, après incorporation et instrumentation des instructions spécialisées proposées au sein de celui-ci, est utilisé pour valider le fonctionnement de l'accélérateur et évaluer les gains qu'il permet d'obtenir.

La figure 5.3 présente l'inclusion de notre accélérateur comme nouvelle unité fonctionnelle au sein du pipeline du Cortex-A5. Nous pouvons y voir son intégration au milieu des autres unités fonctionnelles existantes sur le processeur, ainsi que le partage du premier niveau de cache du processeur, avec un accès par mots de 128 bits à celui-ci, correspondant à la taille de la nouvelle structure de noeud mise en place. Ces unités fonctionnelles sont les suivantes :

- unités de chargement et de décodage des instructions *Fetch* et *Decode* ;
- unité *Issue* en charge de la sélection des instructions décodées dans la file *Queue* ;
- unité de gestion des opérations entières *Int* ;
- unité de gestion des opérations de multiplication *Mul* ;
- unité en charge des lecture/écriture depuis/vers la mémoire *Load/store*
- unité d'accélération des arbres rouges et noirs *RBT* ;
- unité en charge de la mise à jour des registres du processeur *Write-Back*.

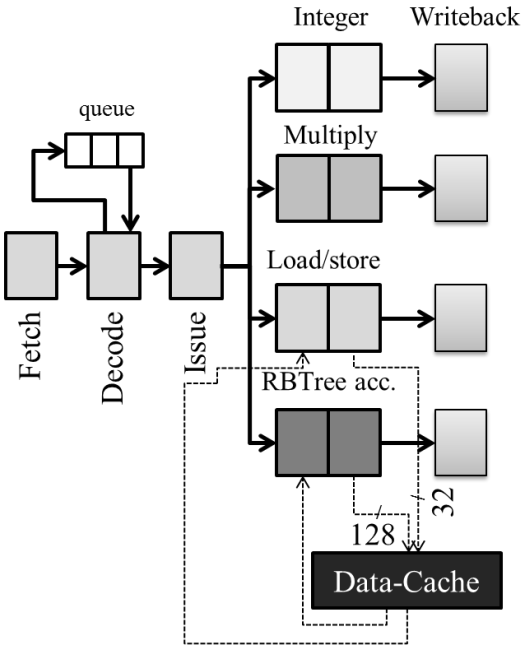



FIGURE 5.3 – Nouveau pipeline du ARM Cortex-A5, incluant notre accélérateur matériel de la gestion des arbres rouges et noirs comme unité fonctionnelle du processeur. Le partage avec le premier niveau de cache du processeur est également explicité sur cette figure.

Chacune des unités *Int*, *Mul*, *Load/store* et *RBT* effectue les opérations qui lui

sont propres en deux cycles. Nous mettrons en avant le découpage réalisé au niveau de l'accélérateur entre les opérations du premier cycle et du second cycle. L'objectif étant de ne pas ralentir le processeur au niveau de sa fréquence maximale de fonctionnement, nous verrons dans la partie suivante que nous proposons la mise en place d'instructions multi-cycles, notamment pour les opérations itératives nécessitant de parcourir l'arbre, afin de garantir une exécution des opérations élémentaires de l'accélérateur dans ces deux cycles (les itérations se faisant au fur et à mesure de l'avancée dans le pipeline et de la succession des opérations élémentaires). Les interactions avec le pipeline sont cloisonnées à la lecture des registres généraux du processeur au début de l'instruction et à l'écriture du résultat dans un registre général à la toute fin de son exécution (*write-back*). Il n'y a donc aucune interaction entre le processeur et l'accélérateur durant la réalisation des différentes opérations élémentaires inhérentes à l'instruction exécutée, si ce n'est le gèle du pipeline (conséquence de l'utilisation d'un processeur *in-order*) et les différents échanges de données avec la mémoire.

## 5.6 Présentation de l'accélérateur

Cette section est dédiée à la présentation de l'accélérateur proposé pour la gestion des arbres rouges et noirs, implémenté sous la forme d'une unité fonctionnelle de processeur. Nous présentons tout d'abord l'extension de jeu d'instructions proposée avant de présenter la structure globale de l'accélérateur puis les différentes unités le composant.

### 5.6.1 Présentation de l'extension du jeu d'instructions

Le choix de l'implémentation sous la forme d'une unité fonctionnelle de l'accélérateur matériel de la gestion des arbres rouges et noir nous amène à proposer une extension du jeu d'instructions initial, afin de fournir les instructions spécialisées relatives aux fonctions accélérées. Nous proposons l'inclusion de **neuf nouvelles instructions spécialisées** au jeu d'instructions ARM pour les huit fonctions identifiées. Ces instructions sont répertoriées dans le tableau 5.2. La première colonne présente la syntaxe de ces instructions, semblable à celle des instructions ARM. La seconde colonne décrit l'opération réalisée par l'instruction, relative à la fonction accélérée par celle-ci. La dernière colonne met en avant quelles parties de la librairie standard STL C++ et de l'allocateur mémoire Doug Lea sont concernées par cette accélération. Les registres Rd, Rn, Rm, Rs sont des registres du processeur utilisés pour la récupération des paramètres propres aux fonctions accélérées par ces instructions spécialisées et pour l'écriture de la valeur de retour de ces fonctions. La communication entre le processeur et l'unité fonctionnelle est assurée par une machine à états finis (FSM, *Finite State Machine*) au sein de l'unité de contrôle, que nous présentons plus en détails dans la suite de cette section.

Concernant les instructions `RBTINS<L|R> Rn, Rm, Rs`, celles-ci ont la particularité de lire trois registres du processeur auquel est associé l'accélérateur. Toutefois, les processeurs ARM supportent cette possibilité sans ajout de matériel spécifique puisque certaines instructions ARM, comme `MLA` (Multiply and Accumulate), nécessitent déjà l'accès à trois registres simultanément en lecture. Les instructions spécialisées introduites lisent au maximum trois registres et écrivent au maximum dans un registre du processeur auquel est couplé l'accélérateur.

### 5.6.2 Description globale du système

Suite à la définition des fonction à accélérer préalablement présentées, nous avons identifié une série de **sept fonctionnalités** devant être remplies par l'accélérateur :

TABLE 5.2 – Présentation des instructions spécialisées mises en place pour l'unité fonctionnelle d'accélération des arbres rouges et noirs. `RBTINS` correspond en fait à deux instructions : insertion à gauche ou insertion à droite.

| Instructions | Fonctions | Utilisées par |
|---|---|---|
| `RBTINC Rd, Rm` | *increment*. Ecrire dans le registre `Rd` l'adresse du noeud de l'arbre successeur du noeud de l'arbre à l'adresse contenue dans `Rm`. | map : :iterator, set : :iterator |
| `RBTDEC Rd, Rm` | *decrement*. Ecrire dans le registre `Rd` l'adresse du noeud de l'arbre prédécesseur du noeud de l'arbre à l'adresse contenue dans `Rm`. | map : :iterator, set : :iterator |
| `RBTLOW Rd, Rn, Rm` | *lower bound*. Ecrire dans le registre `Rd` l'adresse du premier noeud dont la clé est supérieure ou égale à la clé de référence contenue dans `Rn`. L'adresse de l'entête de l'arbre considéré est contenue dans `Rm`. | map : :lower_bound, set : :lower_bound, malloc, realloc |
| `RBTUP Rd, Rn, Rm` | *upper bound*. Ecrire dans le registre `Rd` l'adresse du premier noeud dont la clé est strictement supérieure à la clé de référence contenue dans `Rn`. L'adresse de l'entête de l'arbre considéré est contenue dans `Rm`. | map : :upper_bound, set : :upper_bound |
| `RBTDEL Rn, Rm` | *rebalance for erase*. Supprimer le noeud de l'arbre dont l'adresse est contenue dans `Rn` et opérer le rééquilibrage de l'arbre en conséquence. L'adresse de l'entête de l'arbre considéré est contenue dans `Rm`. Le noeud est supprimé de l'arbre rouge et noir, mais la désallocation de ce noeud n'est pas réalisée par cette instruction et reste à la charge de l'appelant. | map : :erase, set : :erase, malloc, realloc, free |
| `RBTINS<L\|R> Rn, Rm, Rs` | *insert (left or right)*. Insérer le noeud dont l'adresse est spécificée dans `Rn`, en partant du noeud d'insertion dont l'adresse est spécifiée dans `Rs` et opérer le rééquilibrage de l'arbre en conséquence. L'adresse de l'entête de l'arbre considéré est contenue dans `Rm`. | map : :insert, set : :insert, malloc, realloc, free |
| `RBTMAX Rd, Rm` | *maximum*. Ecrire dans le registre `Rd` l'adresse du noeud à l'extrême droite de la branche de l'arbre considérée, à partir du noeud dont l'adresse est contenue dans `Rm`. | map : :erase, set : :erase, malloc, realloc, free |
| `RBTMIN Rd, Rm` | *minimum*. Ecrire dans le registre `Rd` l'adresse du noeud à l'extrême gauche de la branche de l'arbre considérée, à partir du noeud dont l'adresse est contenue dans `Rm`. | map : :erase, set : :erase, malloc, realloc, free |

- accéder via la mémoire cache à des champs de données d'une structure de noeud à partir d'un champ d'adresse fourni à la mémoire et les écrire dans des registres prévus à cet effet ;
- envoyer à la mémoire l'ensemble des champs de données de la structure de noeud manipulée ainsi que l'adresse à laquelle aller écrire ces champs ;
- comparer les champs de données d'une structure de noeud à une clé ou adresse de référence contenue dans des registres prévus à cet effet ;
- modifier le champ de couleur d'un noeud au sein du premier champ de données composé de la clé résumée et de la couleur en fonction des règles de coloration des arbres rouges et noir présentées au chapitre précédent (Section 4.3.2) ;
- échanger entre deux structures de noeud des champs de données (parent, fils gauche/droite), dans l'optique d'un rééquilibrage de l'arbre ;
- récupérer dans les registres du processeur les paramètres nécessaires à la réalisa-

tion des fonctions accélérées (clé ou adresse de référence), après récupération de l'instruction spécialisée relative à la fonction à exécuter ;
– fournir au processeur, via ses registres, la valeur de retour correspondant à un champ d'adresse (par exemple l'adresse du noeud recherché).

Afin de réaliser ces fonctionnalités, nous proposons un accélérateur, conçu sous la forme d'une unité fonctionnelle, dont la structure est présentée sur la figure 5.4.

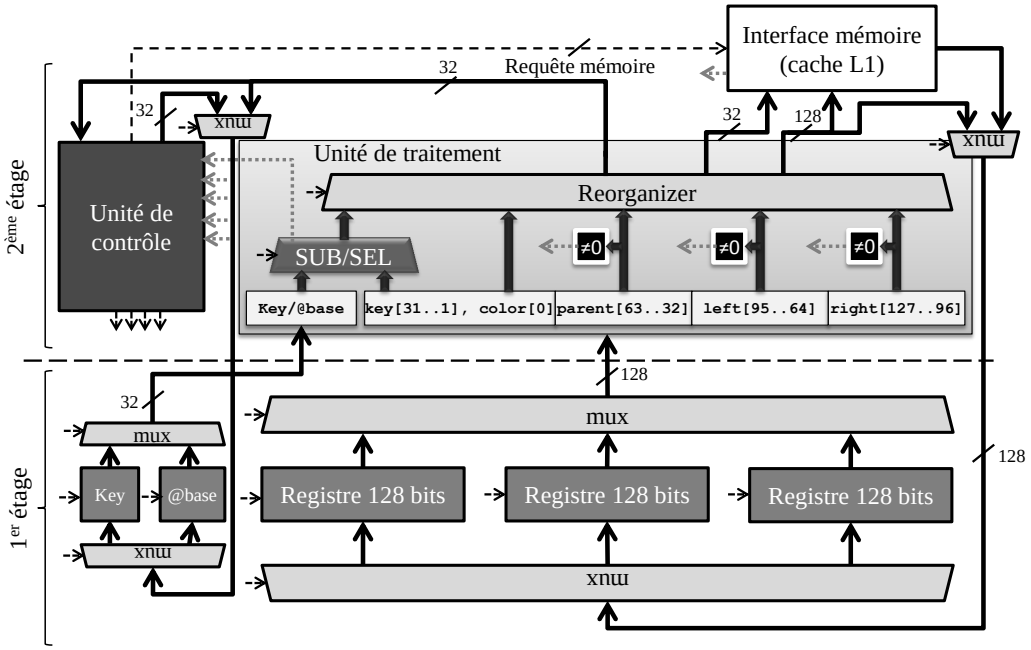



FIGURE 5.4 – Proposition d'accélération matérielle pour la gestion des arbres rouges et noirs. L'accélérateur est implémenté sous la forme d'une unité fonctionnelle du processeur, les communications avec le processeur pour les instructions à exécuter, la récupération des paramètres nécessaires et le retour du résultat sont assurées via la FSM.

Cet accélérateur se compose tout d'abord de **cinq registres**, deux de 32 bits et trois de 128 bits dont les accès en lecture et en écriture se font par l'intermédiaire de multiplexeurs. Ces registres sont visibles dans la partie basse de la figure 5.4.

**Le premier registre de 32 bits** `Key` permet la sauvegarde de la clé de référence, notamment pour les fonctions `lower & upper bound`, ou d'une clé temporaire, avec ou sans le champ couleur associé (31 bits de poids forts + 1 bit de poids faible). Les clés manipulées à ce niveau correspondent aux clés résumées préalablement introduites. Ces clés sont encodées sur 31 bits. Le registre de 32 bits peut contenir soit la clé seule, sur 31 bits, soit le premier champ de données complet de la structure proposée, incluant la couleur, lorsque cela est nécessaire (comparaison de couleur entre noeuds par exemple).

**Le second registre de 32 bits** `@base` a une double fonction : il sert tout d'abord de registre de stockage de l'adresse de référence dans le cadre de l'utilisation de l'allocateur dynamique de la mémoire. Il sert également de registre temporaire de sauvegarde lors de la manipulation des différents noeuds de l'arbre. Cette fonctionnalité est notamment utilisée lors du rééquilibrage de l'arbre, où les champs de données de la structure de noeud vont être modifiés en fonction de ce rééquilibrage (modification du noeud parent ou du fils gauche/droite). L'algorithme procède alors par inversion de champs entre les différents noeuds, d'où la nécessité de sauvegarder ces champs dans un registre temporaire entre les manipulations des noeuds concernés.

Les données chargées dans ces deux registres peuvent provenir soit de l'unité de contrôle, comme opérande de l'instruction ou consécutivement à l'opération précédente,

soit de l'unité de traitement consécutivement à l'opération précédente.

**Les trois registres de 128 bits** permettent le stockage des différentes structures de noeud manipulées lors de l'exécution des instructions précédemment présentées. La structure proposée étant de taille fixe 128 bits, l'ensemble de cette structure peut être contenue au sein de ces différents registres. Le nombre de registres (trois) a été sélectionné après analyse de l'algorithme et la conduite d'expérimentations permettant de déterminer le nombre optimal de registres nécessaires à la bonne exécution des différentes instructions. Il apparaît que trois représente ce nombre optimal. En dessous, des accès mémoires supplémentaires seront nécessaires pour compenser le manque de registres, limitant les gains obtenus via l'accélérateur et pouvant même aller jusqu'à les cacher complètement. Au dessus, le gain apporté est nul, les registres n'étant jamais utilisés, engendrant une utilisation de surface silicium inutile. Les données chargées dans ces trois registres proviennent soit de la mémoire, consécutivement à une requête mémoire en provenance de l'unité de contrôle, soit de l'unité de traitement consécutivement à l'opération précédente. Le choix de la structure de noeud manipulée pour l'opération élémentaire en cours est effectué parmi ces registres via la FSM en contrôlant le multiplexeur en sortie de ceux-ci.

**L'unité de contrôle** et **l'unité de traitement** sont présentées plus en détails dans les parties respectivement consacrées à leur description. Elles se trouvent sur la partie haute de la figure 5.4. L'unité de contrôle pilote l'ensemble de l'accélérateur grâce aux différents signaux d'entrée et de sortie. Elle contrôle notamment l'unité de traitement chargée de réaliser les opérations élémentaires permettant l'exécution des différentes instructions spécialisées. Elle permet également d'assurer la communication avec le processeur auquel l'accélérateur est couplé, notamment pour recevoir les différentes instructions à exécuter et les opérandes qui y sont rattachées, contenues dans les registres de base du processeur, et renvoyer les résultats consécutifs à l'exécution de ces instructions dans ces registres. Cet élément ne figure pas sur la figure 5.4 pour des questions de lisibilité.

**L'interface mémoire** permet la communication avec le cache de données du processeur. En lecture, l'accélérateur, via l'unité de contrôle, vient y récupérer les différentes structures de noeud manipulées. En écriture, l'accélérateur vient soit y écrire la structure de noeud modifiée de 128 bits, soit une adresse provenant d'un champ de données d'une des structures de noeud contenue dans un des trois registres prévus à cet effet.

**Un réseau de distribution** est également présent dans l'accélérateur afin d'assurer l'acheminement des données entre les différents blocs via notamment un réseau de multiplexeurs figurant sur la figure 5.4.

### 5.6.3 Présentation de l'unité de contrôle

Permettant le contrôle de l'ensemble de l'accélérateur, cette **unité de contrôle** se présente sous la forme d'une machine à états finis (FSM, *Finite State Machine*). Elle est composée des différents signaux d'entrée (signaux de condition) et de sortie (signaux de contrôle) et assure la communication avec le processeur auquel est couplé l'accélérateur. La séquence de contrôle exécutée par l'unité de contrôle est déterminée par l'instruction exécutée. La FSM effectue les opérations suivantes :

– Recevoir du processeur (via l'interface non représentée sur la figure présentant l'accélérateur) l'instruction à exécuter parmi celles précédemment présentées, ainsi que ses arguments (opérandes), typiquement une ou deux adresses de noeuds respectifs de l'arbre rouge et noir considéré ainsi qu'éventuellement une clé de référence. Les adresses sont communiquées à la mémoire via l'interface et les données qui y sont rattachées (structure de noeud par exemple) sont écrites dans les

différents registres prévus à cet effet. Une clé de référence peut également être envoyée par la FSM dans le registre *Key*.
- Sélectionner parmi les différents registres ceux qui sont utilisés par l'unité de traitement pour l'opération élémentaire à réaliser, en fonction de la séquence de contrôle sélectionnée (dépendante de l'instruction exécutée).
- Recevoir de l'unité de traitement, et plus précisément des comparateurs à 0 et de l'unité *SUB/SEL*, des signaux de condition résultant des opérations réalisées par ces blocs.
- En fonction de la séquence de contrôle sélectionnée, de l'état interne et des signaux de condition reçus, envoyer aux différents composants de l'unité de traitement les signaux de contrôle.
- Recevoir de l'unité de traitement (ou prélever d'un registre) l'adresse d'un noeud et la transmettre au processeur en tant que résultat de l'instruction.

Afin de donner un exemple du type de séquence de contrôle pouvant être réalisé par la FSM, la figure 5.5 présente la séquence de contrôle relative à l'instruction `RBTLOW`. Cette séquence est constituée de six états et huit transitions.

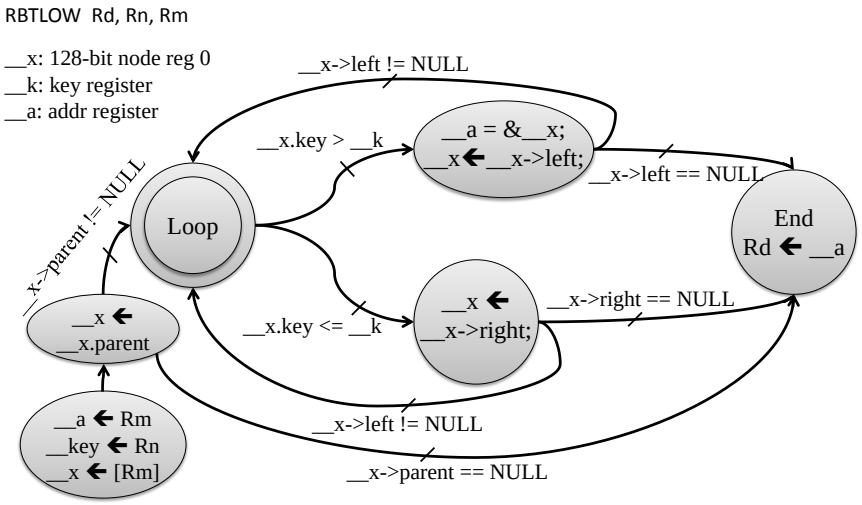



FIGURE 5.5 – Présentation de la séquence de contrôle de l'instruction RBTLOW. Cette séquence est composée de six états et de huit transitions.

L'ensemble de l'analyse réalisée sur l'algorithme des différentes fonctions accélérées a permis de dimensionner une borne maximale pour la taille de la FSM de 161 états et 223 transitions. Toutefois, de nombreuses optimisations ont été pré-identifiées, avec de fortes possibilités de repliement entre les différentes fonctions.

### 5.6.4 Présentation de l'unité de traitement

Cette partie porte sur la présentation de l'unité de traitement, en charge de l'exécution des opérations élémentaires permettant la réalisation des instructions spécialisées multi-cycles.

#### Présentation des éléments composant l'unité de traitement

**L'unité de traitement**, pilotée par la FSM, reçoit ses données en provenance des différents registres de l'accélérateur. La sélection des registres d'origine est effectuée via la FSM en fonction de la séquence de contrôle considérée, de l'état interne et des signaux de condition reçus par la FSM sur l'opération précédente. Cette unité permet la gestion simultanée d'un des deux registres 32 bits (clé de référence/temporaire ou

adresse de référence/temporaire) et d'un des trois registres 128 bits, contenant chacun une structure de noeud.

L'unité de traitement est composée d'**un bloc de sélection et de soustraction *SUB/SEL*** en charge d'opérations de différence et de comparaison à 0 sur le champ clé/couleur en provenance du registre de référence *Key* (clé seule ou clé + couleur) et du premier champ de données de la structure de noeud manipulée, en provenance d'un des registres 128 bits. Cette unité peut également réaliser une comparaison à 0 d'une adresse issue du registre d'adresse *@base* et/ou directement la transmettre à l'unité de réorganisation. Ce bloc reçoit des signaux de contrôle en provenance de la FSM et retourne à cette dernière différents signaux de condition. La sortie de cette unité, outre le signal de condition vers la FSM, contient soit une des entrées mentionnées (*@base*, *Key* ou le champs de données clé/couleur de la structure manipulée), soit la différence ou le résultat de la comparaison à zéro des dites entrées. Cette sortie est ensuite envoyée au bloc de réorganisation *reorganizer*.

**Trois blocs comparateurs** sont également présents au sein de l'unité de traitement sur les trois champs de données correspondant aux adresses vers les autres noeuds de l'arbre contenues dans la structure de noeud manipulée (parent, fils gauche/droite). Ils permettent la comparaison de ces champs à zéro, utilisée notamment lors du parcours d'arbre (pour déterminer par exemple si nous nous trouvons sur des feuilles de l'arbre). La sortie de chacun de ces comparateurs est directement envoyée à l'unité de contrôle comme signal de condition.

Le dernier bloc composant l'unité de traitement est **un bloc de réorganisation appelé *reorganizer***. L'objectif de ce bloc est d'assurer les manipulations et transformations éventuelles au sein de la structure de noeud considérée en fonction de l'opération élémentaire réalisée, et donc de l'instruction exécutée. Il est constitué de cinq entrées : la première est issue de l'unité *SUB/SEL* précédemment présentée. Les quatre autres entrées correspondent aux quatre champs de données issus de la structure de noeud manipulée, en provenance d'un des trois registres de 128 bits, contenant la clé résumée associée à la couleur et les adresses vers les autres noeuds de l'arbre (parent, fils gauche/droite). L'unité reçoit également ses signaux de contrôle en provenance de la FSM. Ce bloc de réorganisation est composé de trois sorties, en plus des signaux de condition envoyés à la FSM :

- une première sortie de 32 bits correspondant à un champ de données de clé, de clé et d'indicateur de couleur ou d'adresse, présent à l'une de ses entrées (première ou seconde entrée), avec modification possible de l'indicateur couleur, à destination de l'unité de contrôle (s'il s'agit de l'adresse de retour pour l'instruction exécutée) et/ou des registres *Key* ou *@base* ;
- une deuxième sortie de 32 bits correspondant à un champ de données d'adresse présent à sa troisième, quatrième ou cinquième entrée (parent, fils gauche ou droite du noeud manipulé), dans l'optique de l'envoie de l'adresse à la mémoire pour le chargement d'une nouvelle structure de noeud en registre ;
- une troisième sortie de 128 bits correspondant à l'ensemble des champs de données composant une structure de noeud (128 bits), obtenus par sélection et permutation éventuelle des champs de données présents à ses entrées (permutation des champs relatifs aux adresses des noeuds parent, fils gauche/droite entre eux ou avec l'adresse du registre *@base* dans le cadre par exemple d'un rééquilibrage de l'arbre), avec modification éventuelle de l'indicateur de couleur (rééquilibrage de l'arbre, modification de la couleur pour respecter les règles relatives aux arbres rouges et noirs présentées dans le chapitre précédent).

**Présentation des opérations élémentaires réalisées par l'unité de traitement**

Afin de réaliser les opérations des fonctions identifiées, accélérées sous la forme d'instructions spécialisées multi-cycles qui ont été présentées précédemment dans cette section, nous avons identifié **huit opérations élémentaires pour l'unité logique de traitement**, correspondant aux opérations réalisées de manière itérative afin d'exécuter les instructions spécialisées :

- comparer une clé de référence, stockée dans le registre de référence *Key*, avec la clé d'un des noeuds contenus dans un des trois registres 128 bits (extraction des 31 bits de poids forts du premier champ), grâce à l'unité *SUB/SEL*, et fournir le résultat de cette comparaison à l'unité de contrôle (FSM), comparaison possible également des champs couleurs ;
- comparer à zéro (ou à une autre valeur prédéterminée) une adresse contenue dans un des champs de données de la structure de noeud, contenue dans un des trois registres 128 bits, et fournir le résultat de cette comparaison à la FSM ;
- comparer à zéro (ou à un) un champ couleur stocké dans un des noeuds contenus dans un des trois registres de 128 bits (extraction du bit de poids faible du premier champ), grâce à l'unité *SUB/SEL*, et fournir le résultat de cette comparaison à la FSM ;
- changer le champ couleur d'un des noeuds stocké dans un des trois registres de 128 bits (accès au bit de poids faible du premier champ de données) ;
- envoyer à la mémoire pour écriture, via l'interface prévue à cet effet, l'ensemble des champs de données, modifiés ou non, d'une structure de noeud issue d'un des trois registres 128 bits ;
- écrire l'adresse temporaire, stockée dans le registre temporaire *@base* dans un des champs de données d'une structure de noeud, issue d'un des trois registres 128 bits (modification du champ parent, ou fils gauche/droite) ;
- écrire une adresse stockée dans un des champs de données d'une structure de noeud, issue d'un des trois registres 128 bits, dans le registre temporaire *@base*, en remplacement de la valeur précédemment contenue dedans.

Toutes les opérations réalisées par l'accélérateur se font en deux étapes, correspondant chacune à un cycle. Le premier cycle concerne le chargement des trois registres de 128 bits et des deux registres de 32 bits, ainsi que la sélection des données manipulées par l'unité logique en provenance de ces différents registres (via le réseau de distribution). Le second cycle correspond à la réalisation d'une des huit opérations élémentaires que nous venons de présenter, effectuées par l'unité de réorganisation *reorganizer* et l'unité de sélection et soustraction *SUB/SEL* qui composent l'unité de traitement. L'ensemble est piloté par l'unité de contrôle (FSM) en fonction des signaux de condition reçus, de son état interne et de l'opération à réaliser.

## 5.7 Evaluation des performances

Nous nous proposons dans cette section d'effectuer une évaluation des performances de notre unité fonctionnelle en charge de l'accélération de la gestion des arbres rouges et noirs que nous venons de présenter. Cette évaluation, pour des raisons de contraintes d'implémentation fortes dans le cadre de la mise en place d'une nouvelle unité fonctionnelle, a été réalisée par instrumentation du simulateur auquel est associé l'accélérateur, le ARM Cortex-A5. Nous disposons comme préalablement mentionné d'un simulateur de jeu d'instructions de précision approximative au cycle de ce processeur, supportant les jeux d'instructions Arm32 et Thumb2 de l'ensemble Armv7, associé à un simulateur de cache (32 KO de cache L1). Pour prendre en compte la simplicité du système que nous désirons mettre en place, il n'y a pas de second niveau de cache (L2) et le cache L1

est directement connecté à la mémoire externe. Dans nos expérimentations, la pénalité d'accès à la mémoire externe en cas d'indisponibilité de la donnée en cache (*cache miss*) est fixée à 64 cycles.

### 5.7.1 Présentation des instrumentations du simulateur

Les instructions spécialisées de notre accélérateur matériel ont été définies sur **la base du jeu d'instruction Thumb2 de ARM**, dans l'espace réservé aux instructions co-processeurs. Ces instructions ont été ajoutées au simulateur. Nos versions de la librairie standard STL C++ et de l'allocateur Doug Lea, inclues dans l'espace de noms HAADT, ont été modifiées afin d'utiliser nos instructions spécialisées accélérées matériellement. Nous avons pour cela remplacé les implémentations logicielles des fonctions accélérées, par l'inclusion d'instructions assembleurs directement au sein du code. Toute la partie logicielle a été compilée en utilisant le compilateur GNU GCC (`arm-none-linux-gnueabi-gcc-4.6.1`), avec le niveau d'optimisation `-O2` (*optimization flag*) et l'option au raccordement (*linker flag*) `-static` afin de produire un binaire ELF complètement autonome, notre simulateur ne fournissant qu'une émulation de système d'exploitation minimaliste. Ces manipulations sont identiques à celles effectuées pour l'évaluation des profondeurs d'indirection et des portions critiques du compilateur LLC, réalisée dans les chapitres précédents.

Les performances de notre accélérateur ont été évaluées via **une instrumentation du simulateur** permettant de mesurer le nombre de cycles passés dans les différentes instructions accélérées matériellement. Le simulateur nous permet également d'évaluer précisément le nombre de cycles passés dans certaines portions précises du code, au niveau du logiciel, suite aux instrumentations réalisées pour les expérimentations du chapitre précédent. Cela nous permet de comparer nos instructions matériellement accélérées avec les portions initiales de code (version logicielle). Comme au chapitre précédent, nos expérimentations sur LLC, au niveau de l'évolution de son temps d'exécution, ne mesurent que les parties strictement concernées par la génération de code : de la représentation intermédiaire LLVA en mémoire au code machine en mémoire (la partie de raccordement n'est pas réalisée par LLC, s'arrêtant à la génération du fichier objet). Nos différentes version de LLC sont toutes basées sur la version autonome mise en place dans les chapitres précédents et permettant la génération de code machine ARM.

### 5.7.2 Méthodologie pour l'évaluation des performances

Nous disposons, avec notre accélérateur, de **trois versions de LLC**. La première est la version originale de LLC, **optimisée logiciellement**. La seconde est la **version normalisée** de LLC préalablement introduite. La troisième est la version **accélérée matériellement** de LLC, qui correspond en fait à la version normalisée mais bénéficiant cette fois-ci de l'accélérateur matériel que nous venons de présenter, alors que les deux premières versions n'en bénéficient pas.

**Trois séries de mesures** ont été réalisées sur LLC, avec en entrée (en tant que code à compiler) la série de tests déjà utilisée au chapitre précédent et dont nous rappelons les éléments : bitcount, convolution (produit de convolution), crc32, dijkstra, fft, hello, neon intrinsincs (tests de vectorisation pour processeur ARM), patricia, gcd (calcul de pgcd), quicksort, adpcm (compression et décompression), stringsearch, sha et susan.

La première série porte sur **l'évolution des temps d'exécution globaux** entre les différentes versions de LLC, permettant ainsi d'évaluer les gains sur le compilateur complet que notre solution permet d'envisager, par rapport à la version initialement optimisée logiciellement et par rapport à la version normalisée.

La seconde série porte sur **l'évolution du temps passé dans la gestion des tableaux associatifs et dans l'allocation dynamique de la mémoire** pour les trois versions de LLC, par rapport au temps total d'exécution de chacune de ces trois versions. Ces mesures permettent d'évaluer l'évolution de la criticité de ces étapes sur le temps total d'exécution.

La dernière série porte sur **l'évaluation du gain brut obtenu à l'aide de notre accélérateur** pour la gestion des tableaux associatifs et l'allocation dynamique de la mémoire, en mesurant l'évolution des temps d'exécution de la librairie standard STL C++ et de l'allocateur C Doug Lea avec et sans l'accélérateur.

Les résultats de ces trois séries de mesures sont présentés dans la partie suivante.

### 5.7.3 Résultats obtenus en performances pour l'accélérateur

Nous présentons ici l'évaluation des performances de notre accélérateur pour les trois séries de mesures présentées précédemment. Tous les résultats chiffrés sont donnés dans les tableaux 5.3, 5.4, 5.5 et 5.6.

#### Première série de mesures

La figure 5.6 présente les résultats obtenus pour la première série de mesures en montrant les gains obtenus sur l'évolution des temps d'exécution des versions logiciellement (version initiale) et matériellement (version normalisée bénéficiant de l'accélérateur) optimisées. Ces gains sont donnés relativement aux temps d'exécution mesurés pour la version normalisée ne bénéficiant pas de l'accélérateur.

Les résultats mettent en avant des gains d'en moyenne 29.1 % pour la version optimisée logiciellement, avec un gain maximal de 34.4 % et un gain minimal de 24 %, contre 49.2 % en moyenne pour la version normalisée accélérée matériellement, avec un gain maximal de 60.4 % et un gain minimal de 41.9 %. Cela représente un gain entre la version accélérée matériellement et la version optimisée logiciellement d'en moyenne 15.5 %, avec un gain maximal de 25.4 % et un gain minimal de 9.4 %.

La version accélérée matériellement, bien que basée sur les librairies standards, offre un gain significatif et surtout stable par rapport à la version logiciellement optimisée, et cela quelque soit le code compilé par LLC. On s'aperçoit également dans le tableau 5.6 que les extremums au niveau des gains obtenus dans la version accélérée matériellement ne correspondent pas forcément à ceux de la version optimisée logiciellement, conséquence de la différence d'approche pour la gestion des tableaux associatifs et de l'allocation mémoire entre les deux versions, et donc de la différence de balance entre les deux.

#### Seconde série de mesures

La figure 5.7 présente les résultats obtenus pour la seconde série de mesures sur l'évolution du temps passé dans la gestion des tableaux associatifs et l'allocation dynamique de la mémoire pour les trois version de LLC. Les résultats sont donnés en pourcentage des temps d'exécution respectifs des trois versions.

Alors que la version initiale de LLC, optimisée logiciellement, passe en moyenne 24 % du temps dans ces deux phases, avec un pourcentage maximal de 29.7 % et un pourcentage minimal de 18.6 %, la version normalisée passe en moyenne 41.2 % du temps dans celles-ci, avec un pourcentage maximal de 45.0 % et un pourcentage minimal de 37.0 %. Ces résultats sont logiques compte tenu de la "désoptimisation" opérée pour passer de la version optimisée logiciellement à la version normalisée, appuyant les propos des développeurs de LLVM sur les performances limitées des librairies standards et sur l'impact de l'allocation dynamique de la mémoire.

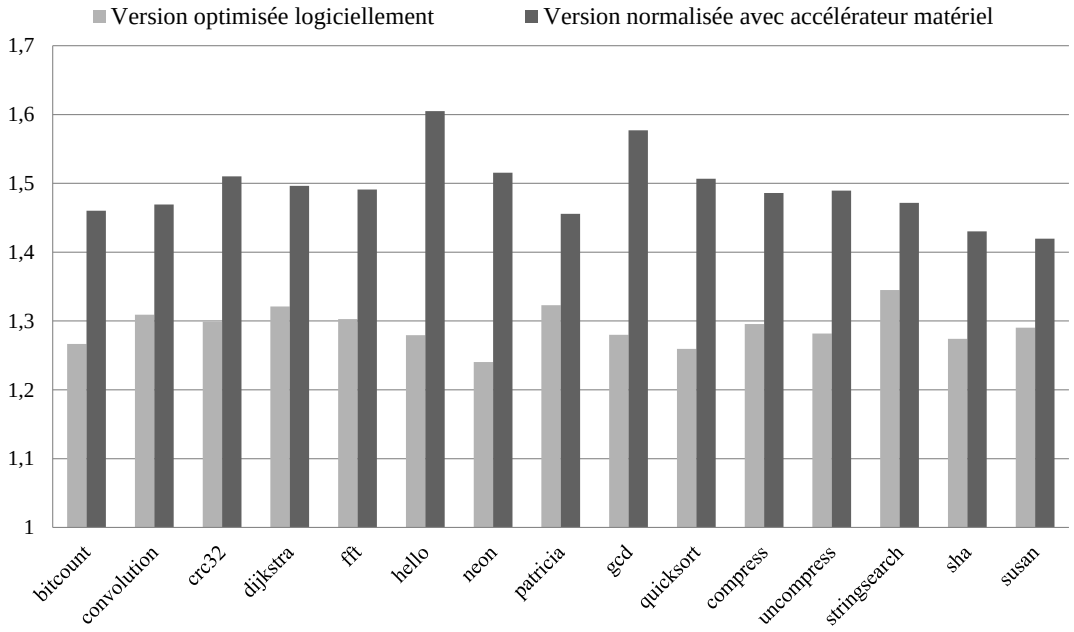



FIGURE 5.6 – Gains obtenus en performances (sur le temps d'exécution) pour la version optimisée logiciellement de LLC et la version normalisée bénéficiant de l'accélérateur matériel. Les gains sont donnés relativement à la version normalisée, s'exécutant sur le processeur seul.

Si l'on considère maintenant notre version normalisée accélérée matériellement, le pourcentage passé dans la gestion des tableaux associatifs et l'allocation mémoire n'est plus en moyenne que de 12.2 % du temps d'exécution total, avec un pourcentage maximal de 20.7 % et un pourcentage minimal de 8.9 %. Ces résultats confirment ceux obtenus pour la première mesure en montrant une baisse significative de l'impact de ces phases critiques sur le temps total d'exécution grâce à l'accélérateur, tout en bénéficiant des avantages de l'utilisation des librairies standards.

### Troisième série de mesures

La figure 5.8 présente les résultats obtenus pour la troisième série de mesures visant à mettre en avant les gains bruts obtenus sur la gestion des tableaux associatifs et l'allocation dynamique de la mémoire par les librairies standards grâce à l'accélérateur proposé. Ces gains sont mesurés par rapport aux temps d'exécution initiaux des librairies standards sans l'accélérateur.

Les résultats mettent en avant un très fort potentiel de gain sur les deux phases grâce à l'accélérateur. Nous obtenons en effet en moyenne un gain de 5.11× pour l'allocation dynamique de la mémoire, avec un gain maximal de 6.41× et un gain minimal de 3.24×. Concernant la gestion des tableaux associatifs, nous obtenons un gain moyen de 5.65×, avec un gain maximal de 6.56× et un gain minimal de 2.87×. Ces gains importants témoignent de la grande attractivité de cette solution pour l'ensemble des codes consommateurs d'allocation dynamique de la mémoire ou de tableaux associatifs.

Les résultats obtenus, combinés à la portabilité de la solution (puisque cachée derrière la librairie standard STC C++ et l'allocateur C Doug Lea), ouvrent un grand champ de possibilités d'utilisation de l'accélérateur. A noter que les gains obtenus pour la compilation du code susan sont plus faibles que les autres. Nous l'expliquons par le volume de code de cette application, bien plus élevé que celui des autres. Il est à noter que la compilation d'un tel volume de code est difficilement envisageable dans le cadre de la compilation dynamique. Les codes compilés, sous la forme de blocs de base ou de

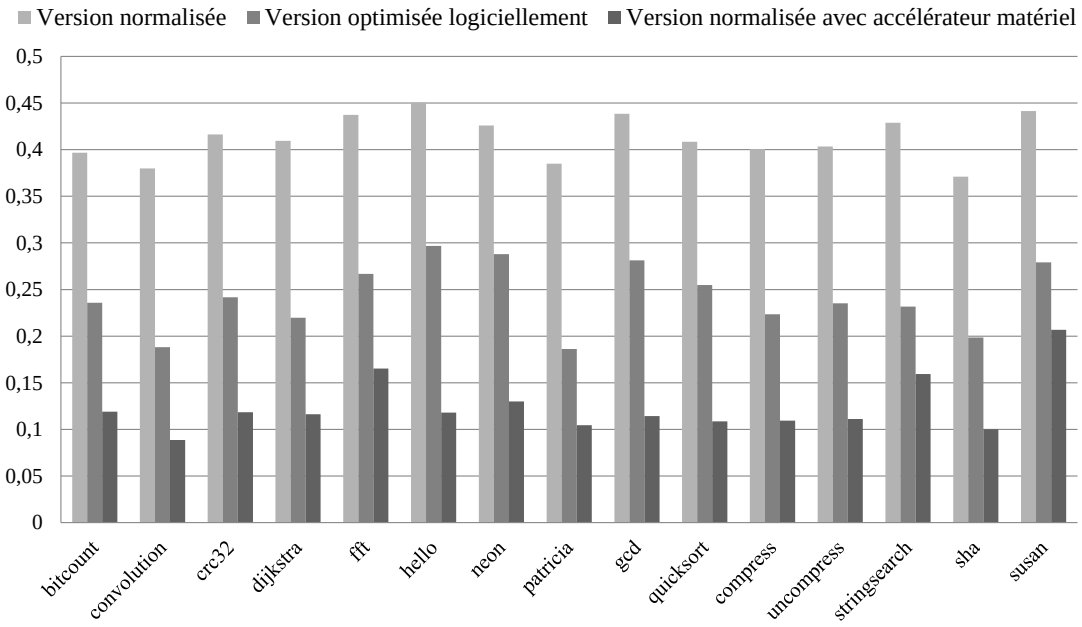



FIGURE 5.7 – Evolution du temps passé dans la gestion des tableaux associatifs et l'allocation dynamique de la mémoire pour les trois versions de LLC. Cette évolution est donnée en pourcentage du temps d'exécution de chacune des trois versions.

méthodes, ont un volume avoisinant bien plus celui des petits noyaux manipulés dans la série de tests, dont les ordres de grandeurs sont donnés dans les tableaux 5.3, 5.4 et 5.5.

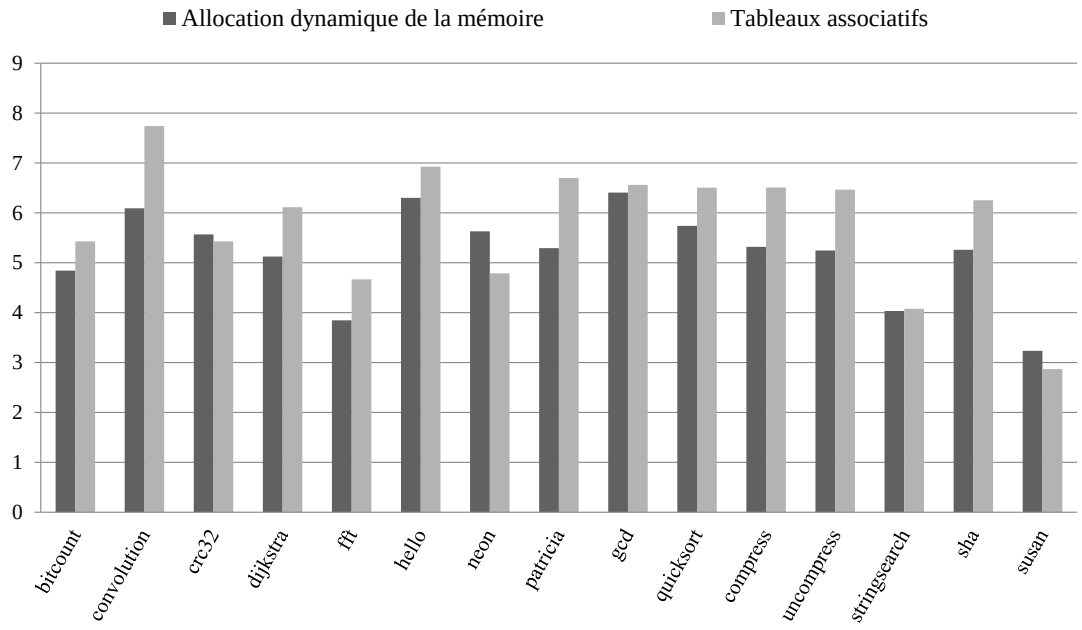



FIGURE 5.8 – Gains obtenus en performances (sur le temps d'exécution) avec l'accélérateur sur la gestion des tableaux associatifs et l'allocation dynamique de la mémoire dans les librairies standards, par rapport à une exécution sans l'accélérateur.

TABLE 5.3 – Nombre de cycles mesurés pour la version optimisée logiciellement de LLC et pourcentages relatifs de temps passé dans l'allocation dynamique de la mémoire et la gestion des tableaux associatifs.

| | Nb total cycles | Nb cycles mem. alloc. | Nb cycles tab. asso. | % mem. alloc. | % tab. asso. |
|---|---|---|---|---|---|
| bitcount | 60282780 | 9779945 | 4432223 | 16.2 | 7.3 |
| convolution | 21122644 | 2858724 | 1115197 | 13.5 | 5.3 |
| crc32 | 17383024 | 2884691 | 1316940 | 16.6 | 7.6 |
| dijkstra | 44582243 | 6774867 | 3020881 | 15.2 | 6.8 |
| fft | 81919080 | 14594965 | 7259771 | 17.8 | 8.9 |
| hello | 1494662 | 271918 | 171680 | 18.2 | 11.5 |
| neon | 49787298 | 9893453 | 4445807 | 19.9 | 8.9 |
| patricia | 70663355 | 9632462 | 3526175 | 13.6 | 5.0 |
| gcd | 5313737 | 1018560 | 475753 | 19.2 | 9.0 |
| quicksort | 17703243 | 3008490 | 1502652 | 17.0 | 8.5 |
| compress | 26220848 | 4068982 | 1791510 | 15.5 | 6.8 |
| uncompress | 26037873 | 4254713 | 1867870 | 16.3 | 7.1 |
| stringsearch | 96625702 | 14681804 | 7714900 | 15.2 | 8.0 |
| sha | 52461631 | 7374945 | 3044353 | 14.1 | 5.8 |
| susan | 2875509069 | 378908703 | 423450822 | 13.2 | 14.7 |

TABLE 5.4 – Nombre de cycles mesurés pour la version normalisée de LLC et pourcentages relatifs de temps passé dans l'allocation dynamique de la mémoire et la gestion des tableaux associatifs.

| | Nb total cycles | Nb cycles mem. alloc. | Nb cycles tab. asso. | % mem. alloc. | % tab. asso. |
|---|---|---|---|---|---|
| bitcount | 76358393 | 21033269 | 9532175 | 27.5 | 12.5 |
| convolution | 27650522 | 7576149 | 2955478 | 27.4 | 10.7 |
| crc32 | 22581586 | 6497824 | 2966434 | 28.8 | 13,1 |
| dijkstra | 58902641 | 16743555 | 7465871 | 28,4 | 12.8 |
| fft | 106717634 | 31944524 | 15889722 | 29.9 | 14.9 |
| hello | 1912446 | 526593 | 332474 | 27.5 | 17.4 |
| neon | 61749015 | 18094589 | 8131140 | 29.3 | 13.2 |
| patricia | 93489201 | 26129602 | 9565316 | 27.9 | 10.2 |
| gcd | 6801382 | 2031246 | 948764 | 29.9 | 13.9 |
| quicksort | 22299208 | 6105132 | 3049335 | 27.4 | 13.7 |
| compress | 33971780 | 9571178 | 4214042 | 28.2 | 12.4 |
| uncompress | 33373741 | 9473448 | 4158959 | 28.4 | 12.5 |
| stringsearch | 129956864 | 36971017 | 19427294 | 28.4 | 14.9 |
| sha | 66834318 | 17493226 | 7221146 | 26.2 | 10.8 |
| susan | 3710588193 | 785841598 | 878220184 | 21.2 | 23.7 |

### 5.7.4 Estimation de la taille de l'accélérateur et discussion sur la consommation

Cette partie est dédiée à **l'estimation de la surface silicium de l'accélérateur** mis en place. Afin de réaliser cette estimation, une implémentation de l'accélérateur a été réalisée, selon le mode de réalisation préalablement présenté et pour les fonctions précédemment identifiées.

L'implémentation de la FSM, représentant la majeure partie du travail, a été réalisée à partir de l'algorithme des différentes fonctions visées. La version réalisée offre bien

TABLE 5.5 – Nombre de cycles mesurés pour la version normalisée de LLC avec accélérateur matériel et pourcentages relatifs de temps passé dans l'allocation dynamique de la mémoire et la gestion des tableaux associatifs.

| | Nb total cycles | Nb cycles mem. alloc. | Nb cycles tab. asso. | % mem. alloc. | % tab. asso. |
|---|---|---|---|---|---|
| bitcount | 52289728 | 4341821 | 1756178 | 8.3 | 3.4 |
| convolution | 18818500 | 1243674 | 381823 | 6.6 | 2.0 |
| crc32 | 14952648 | 1166848 | 546286 | 7.8 | 3.7 |
| dijkstra | 39361624 | 3266920 | 1221351 | 8.3 | 3.1 |
| fft | 71566934 | 8304750 | 3403819 | 11.6 | 4.8 |
| hello | 1191638 | 83565 | 48000 | 7.0 | 4.0 |
| neon | 40743548 | 3213632 | 1698464 | 7.9 | 4.2 |
| patricia | 64218458 | 4936808 | 1427257 | 7.7 | 2.2 |
| gcd | 4312483 | 317014 | 144593 | 7.4 | 3.4 |
| quicksort | 14799455 | 1063354 | 468610 | 7.2 | 3.2 |
| compress | 22862949 | 1799029 | 647499 | 7.9 | 2.8 |
| uncompress | 22406025 | 1805716 | 643327 | 8.1 | 2.9 |
| stringsearch | 88302091 | 9168077 | 4763188 | 10.4 | 5.4 |
| sha | 46728245 | 3325482 | 1154835 | 7.1 | 2.5 |
| susan | 2613735219 | 242813845 | 306007678 | 9.3 | 11.7 |

TABLE 5.6 – Accélérations obtenues en performances (sur le temps d'exécution) par les versions optimisées logiciellement et accélérées matériellement et accélérations obtenues par l'accélérateur sur l'allocation mémoire et la gestion des tableaux associatifs par rapport à l'implémentation logicielle initiale. SW : version optimisée logiciellement, HW : version normalisée accélérée matériellement, Normal : version normalisée.

| | Acc. SW/-Normal | Acc. HW/-Normal | Acc. HW/SW | Acc. HW mem. alloc | Acc. HW tab. asso. |
|---|---|---|---|---|---|
| bitcount | 1.46 | 1.27 | 1.15 | 4.84 | 5.43 |
| convolution | 1.47 | 1.31 | 1.12 | 6.09 | 7.74 |
| crc32 | 1.51 | 1.30 | 1.16 | 5.57 | 5.43 |
| dijkstra | 1.50 | 1.32 | 1.13 | 5.13 | 6.11 |
| fft | 1.49 | 1.30 | 1.14 | 3.85 | 4.67 |
| hello | 1.60 | 1.28 | 1.25 | 6.30 | 6.93 |
| neon | 1.52 | 1.24 | 1.22 | 5.63 | 4.79 |
| patricia | 1.46 | 1.32 | 1.10 | 5.29 | 6.70 |
| gcd | 1.58 | 1.28 | 1.23 | 6.41 | 6.56 |
| quicksort | 1.51 | 1.26 | 1.20 | 5.74 | 6.51 |
| compress | 1.49 | 1.30 | 1.15 | 5.32 | 6.51 |
| uncompress | 1.49 | 1.28 | 1.16 | 5.25 | 6.46 |
| stringsearch | 1.47 | 1.34 | 1.09 | 4.03 | 4.08 |
| sha | 1.43 | 1.27 | 1.12 | 5.26 | 6.25 |
| susan | 1.42 | 1.29 | 1.10 | 3.24 | 2.87 |

évidemment de grandes possibilités d'optimisations, avec de nombreuses opportunités de repliement possibles au niveau de cette FSM. Des modifications algorithmiques sont également envisageables au niveau des fonctions existantes afin d'en accroître l'efficacité, conduisant à des évolutions de la FSM. Tous ces points n'ont malheureusement pas pu être abordés suffisamment en détails pour proposer ces modifications au moment de

la rédaction de ce manuscrit.

Nous avons synthétisé notre implémentation de l'accélérateur dans une technologie TSMC 40 nm *Low-Power* (préconisée pour le ARM Cortex-A5) pour une fréquence de fonctionnement à 600 MHz, tout à fait adaptée à un contexte embarqué. Nous disposons, par l'intermédiaire du site de ARM, des chiffres relatifs à la surface du processeur auquel est associé l'accélérateur, le Cortex-A5, dans la même technologie [124].

Les résultats obtenus par synthèse mettent en avant une surface silicium de 7290 $\mu m^2$ pour notre accélérateur. Les chiffres avancés pour le processeur ARM sont de 0.27 $mm^2$ pour le coeur seul (sans cache), de 0.53 $mm^2$ pour le coeur avec 16 KO de cache de données et 16 KO de cache d'instructions et 0.68 $mm^2$ pour le coeur avec les mêmes caches et l'unité vectorielle NEON en plus. Si l'on considère le coeur avec ses caches (tout en gardant à l'esprit que nous avons choisi une taille de cache de 32 KO pour nos expérimentations) et sans l'extension NEON, soit 0.53 $mm^2$, l'ajout de l'accélérateur ne représente qu'un surcoût de 1.38 % en surface silicium.

Concernant la fréquence de fonctionnement maximale supportée, nous avons réussi à synthétiser l'accélérateur jusqu'à 1 GHz. Or la plage de fonctionnement du processeur, annoncée par ARM, est de 530 à 600 MHz en TSCM 40 LP (*Low-Power*). L'accélérateur, sous la forme d'une unité fonctionnelle, peut donc fonctionner aux fréquences de fonctionnement du processeur. Cet accélérateur ne se retrouve pas sur son chemin critique.

Du point de vue **de la consommation**, nous n'avons pu aller jusqu'à l'implémentation fonctionnelle de notre solution au niveau RTL. Nous n'avons donc pas pu réaliser une sollicitation de notre accélérateur à ce niveau afin d'en estimer la consommation. Toutefois, nos études à ce sujet nous permettent d'estimer que notre accélérateur, tel que proposé sous la forme d'une unité fonctionnelle, n'engendre qu'un surcoût marginal en consommation. Nous estimons en effet que le trafic de données représente le pôle principal de consommation. Or, l'implémentation choisie n'engendre pas de trafic supplémentaire. Celui-ci, avec ou sans l'accélérateur, existe dans tous les cas (pour l'exécution des différentes fonctions de gestion des arbres rouges et noirs). Seule la modification de structure proposée engendre une légère variation de ce trafic (nous ne manipulerons plus que des mots de 128 bits au lieu de manipuler des structures de tailles variables). La consommation de la FSM est quant à elle négligeable, ne représentant que des opérations simples de contrôle.

Pour information, les consommations annoncées sur le site de ARM pour le Cortex-A5 en TSMC 40 LP sont de 0.12 mW/MHz pour la consommation dynamique, avec une efficacité énergétique de 13 DMIPS/mW.

## 5.8 Conclusions et perspectives sur cette proposition

Cette partie est dédiée à la conclusion de l'étude qui vient d'être présentée et des perspectives ouvertes par celle-ci.

### 5.8.1 Conclusions

Nous avons proposé à travers ce chapitre une accélération matérielle de la gestion des tableaux associatifs et de l'allocation dynamique de la mémoire grâce à **un accélérateur de la gestion des arbres rouges et noirs**. Nous avons fait le choix de baser cet accélérateur sur les librairies standards (STL C++ et allocateur C Doug Lea `dlmalloc`) et non sur les optimisations logicielles existantes de LLC, afin de mettre en place une approche réutilisable et bénéficiant à l'ensemble des codes utilisant massivement ces tableaux associatifs. Ces librairies ne constituent pas une solution optimale

mais offrent des performances moyennes satisfaisantes dans de nombreux domaines applicatifs. Ces performances moyennes s'en trouvent accrues en bénéficiant directement de l'accélérateur introduit. Une normalisation de LLC a été réalisée en proposant l'utilisation systématique des librairies standards, nous permettant d'obtenir une version normalisée de référence de ce compilateur.

Le choix a été fait pour l'accélérateur de se concentrer sur l'implémentation des tableaux associatifs sous la forme d'arbres rouges et noirs. Leurs fonctions de gestion présentant une grande régularité, elles ouvrent d'importantes perspectives de mise en place de matériel dédié pour les réaliser. Notre étude a permis de mettre en avant l'utilisation des tableaux associatifs dans la gestion des zones mémoire libres au sein de l'allocateur C Doug Lea. Toutefois, ces dernières utilisent une implémentation sous la forme de tables de hachage et de listes doublement chaînées. Nous avons donc modifié la structure interne de l'allocateur afin que celui-ci se base également sur une implémentation de type arbres rouges et noirs, pour bénéficier d'une base commune potentiellement accélérable.

Une modification de la structure initiale des noeuds des arbres rouges et noirs, pour la rendre manipulable directement par l'accélérateur, a été présentée . Nous introduisons pour cela une clé résumée de taille fixe 31 bits, conservant les propriétés d'ordre de l'arbre et permettant d'obtenir une structure de noeud de taille fixe 128 bits, en modifiant également le champ couleur en l'encodant sur un bit unique. Bien évidemment, la taille de cette structure sera doublée lors d'un éventuel passage sur un processeur 64 bits, l'ensemble des champs étant encodés sur 64 bits, la clé résumée sur 63 bits et la couleur toujours sur un seul bit. Le défaut majeur de cette solution est la nécessité pour les utilisateurs de gérer cette clé sur 31 (ou 63) bits, et notamment l'extraction de celle-ci du champ de 32 (ou 64) bits sur lequel elle est associée à la couleur. Toutefois, cette manipulation représente un effort limité pour l'utilisateur. L'autre défaut réside dans la transformation de l'ordre de l'arbre, passant d'un ordre total à un ordre partiel, conséquence de l'introduction de la clé résumée. Nous avons pu voir cependant que les cas concernés par cette problématique sont peu nombreux, et que la nécessité de repasser par le logiciel intervient donc peu fréquemment pour résoudre ces questions d'ordre.

L'accélérateur en lui-même a été implémenté sous la forme d'**une unité fonctionnelle du processeur auquel il est associé, un ARM Cortex-A5**. Cette approche offre l'avantage de ne pas introduire de latences supplémentaires en communication contrairement à une approche type co-processeur, offrant de meilleures perspectives au niveau de ses performances. Cet aspect ne nécessite également pas de gestion de la cohérence des données manipulées entre le cache et la mémoire accédée par l'accélérateur puisque ce dernier partage directement le premier niveau de cache du processeur. Enfin, d'un point de vue consommation, l'intérêt de ce couplage fort est de ne pas introduire de trafic supplémentaire au niveau des données, ces dernières transitant de la même manière avec et sans accélérateur, alors qu'une approche co-processeur aurait nécessité l'inclusion de requêtes.

Cet accélérateur se compose de cinq registres, d'une unité de traitement, d'une interface mémoire, d'une interface processeur et d'une unité de contrôle pour le piloter. Cette simplicité permet d'assurer un encombrement silicium très faible, équivalente à 1.4 % de la superficie du processeur Cortex-A5 auquel il est couplé. Elle suffit toutefois à assurer l'accélération des principales fonctions de gestion des arbres rouges et noirs et permet l'obtention de gains significatifs, notamment sur le compilateur LLC.

Ainsi, les résultats obtenus mettent en avant un gain maximal de 25 % et un gain moyen de 15 % en performances par rapport à la version optimisée logiciellement. Un autre résultat significatif est l'accélération brute obtenue sur la gestion des tableaux

associatifs et de l'allocation dynamique de la mémoire, pour lesquelles nous réalisons respectivement des gains de 5.65× et 5.11× sur leur exécution en comparaison avec l'implémentation logicielle initiale dans les librairies standards.

Côté consommation, nous estimons que notre système est quasiment neutre puisqu'il n'a qu'une très faible influence sur le trafic des données, qui reste pratiquement identique avec ou sans l'accélérateur. Ce trafic représente selon nous le pôle principal de consommation. La seule interrogation se situe au niveau des cinq registres ajoutés via notre solution, dont la consommation n'a pu être estimée au moment de la rédaction de ce manuscrit. Nous estimons toutefois que cette consommation est fortement corrélée au trafic des données que nous venons de mentionner. La consommation de l'unité de contrôle est négligeable , tout comme celle de l'unité de traitement de par la simplicité des opérations réalisées à son niveau.

Bien évidemment, les gains obtenus, notamment pour LLC, sont loin des ordres de grandeur normalement obtenus pour l'accélération d'une application classique par du matériel dédié, surtout dans le cadre de l'embarqué. Mais il convient de garder à l'esprit que ces accélérations concernent, la grande majorité du temps, des applications extrêmement régulières avec un très fort potentiel de gain offert. Dans le cadre de cette étude, nous cherchons à accélérer des applications très irrégulières et dont la complexité limite la réalisation d'opérations génériques facilement accélérables. Les 15 % de gain en moyenne sont à mettre en regard de l'évolution des performances d'un processeur à un autre, notamment vis-à-vis de l'évolution de sa surface. Ainsi, le passage d'un ARM Cortex-A5 à un ARM Cortex-A8 entraînera un accroissement des performances brutes de 27 % pour une multiplication de la surface par 4 du processeur. Nous avons montré que l'approche largement évoquée, notamment par Cao et al., pour maximiser les performances de la compilation dynamique, consiste à augmenter la puissance du processeur. Or, via notre proposition, nous nous proposons d'accroître **les performances des compilateurs dynamique de 15 %** en augmentant **la surface silicium de seulement 1.4%**, maximisant ainsi les efficacités énergétique et surfacique de notre système.

### 5.8.2 Perspectives et opportunités d'aller vers une implémentation réelle de l'accélérateur

Les opportunités offertes par cet accélérateur sont fortes de par les choix technologiques réalisés pour faciliter sa réutilisation. Tout développeur faisant appel à la librairie standard STL C++ et/ou à l'allocateur C Doug Lea bénéficiera automatiquement de l'accélération. Outre LLC et plus globalement l'ensemble des codes de compilation dynamique, cet accélérateur peut bénéficier à un très large panel de codes utilisant les tableaux associatifs, comme par exemple des codes de traitement d'image utilisant des structures de données complexes, avec manipulations de pointeurs, et réalisant beaucoup d'allocations dynamiques. Cet aspect allocation dynamique de la mémoire demeure une problématique récurrente dans le domaine des systèmes embarqués de par les surcoût en performances engendrés par son utilisation. Cet accélérateur peut être une réponse à ce problème avec les performances offertes par son utilisation et le fort potentiel obtenu au niveau des efficacités énergétique et surfacique.

Une piste de réflexion envisagée pour limiter encore le surcoût silicium de notre solution et une éventuelle augmentation de la consommation serait la réutilisation de registres déjà existants sur le processeur. Il s'avère que l'unité fonctionnelle NEON dispose d'un banc de 32 registres de 128 bits chacun. Il est donc envisageable d'imaginer une modification de cette unité pour pouvoir utiliser ces registres avec notre accélérateur. Toutefois, cela engendre une modification encore plus profonde du processeur, puisqu'elle impacte un banc de registre existant de celui-ci. La question se pose égale-

ment de l'intérêt d'activer l'unité NEON sur ce processeur. Si celui-ci est entièrement dévolu à la compilation dynamique et à certains services de virtualisation, a-t-il besoin d'utiliser cette unité ? Si ce n'est pas le cas, son activation (nécessaire pour notre accélérateur en cas de partage du banc de registre), engendrerait un surcoût en consommation non négligeable (activation de trente deux registres pour trois seulement utilisés). A contrario, si l'unité NEON est fortement utilisée sur ce processeur, monopoliser jusqu'à trois de ses registres ne risque-t-il pas d'impacter les performances de la vectorisation ? Concernant, les deux registres de 32 bits, l'utilisation de registres du banc de registres du processeur ARM ne nous semble pas envisageable, de par l'impact sur les performances que cette monopolisation des ressources engendrerait.

Une implémentation sous la forme d'unité fonctionnelle de processeur constitue un choix beaucoup plus intrusif que celui basé sur la création d'une unité type co-processeur. Les modifications apportées à la structure même du coeur concerné sont significatives puisqu'il convient de revoir la micro-architecture de ce coeur, au niveau du contrôle, au niveau de l'accès aux autres unités, au niveau du pipeline du processeur, etc. Ces modifications nécessitent un accès complet aux sources du processeur, chose difficile pour une cible ARM. Outre ces modifications architecturales, il faut également revoir en partie les chaînes de compilation et de débogage du processeur afin de permettre le ciblage de cet accélérateur et notamment de l'extension du jeu d'instructions proposée.

Nous estimons qu'une implémentation fonctionnelle complètement opérationnelle de notre accélérateur est l'affaire de plusieurs mois de travail pour un groupe d'ingénieurs ayant déjà une bonne connaissance du processeur cible, en l'occurrence le ARM Cortex-A5. Pour des raisons de ressources évidentes, nous n'avons donc pas cherché dans le cadre de cette étude à tenter d'atteindre cette étape d'implémentation fonctionnelle si ce n'est à minima pour les estimations de surface de notre proposition. Toutefois, nous estimons que les perspectives ouvertes par notre accélérateur justifient d'envisager d'aller jusqu'au bout de sa réalisation, à travers, par exemple, une potentielle future collaboration dans le cadre d'un projet industriel.

## 5.9 Mise en cadre applicatif réel de l'accélérateur

Cette dernière section est dédiée à la présentation d'expérimentations effectuées dans l'optique **d'une mise en application réelle de la compilation dynamique et de notre accélérateur**. L'objectif est d'introduire la compilation dynamique dans un contexte d'optimisation des performances d'une application en recompilant des petites portions de code tout en bénéficiant d'informations disponibles à l'exécution. Ces expérimentations ne sont que les prémices de travaux d'évaluation à plus grande échelle de l'accélération mise en place. Les applications visées sont fortement dynamiques et très sensibles à leur jeu de données, comme par exemple les applications de traitement d'image et d'analyse de contenu.

### 5.9.1 Présentation de l'application visée

Nous avons basé nos expérimentations sur l'**application SURF** disponible au sein de notre laboratoire. Cette application permet **la détection et le suivi de points d'intérêt sur une série d'images** en provenance par exemple d'un flux vidéo. Ce principe est notamment utilisé pour la classification, la reconnaissance et l'indexation d'images, ainsi que pour de la reconnaissance d'objets au sein de ces images ou de la calibration de caméras, par exemple dans le cas de la stéréo-vision. Cette application se base sur l'algorithme de Bay et al. [136] permettant de réaliser cette détection et ce suivi par l'intermédiaire de détecteurs basés sur des matrices de Hessian et de

descripteurs obtenus à partir d'une distribution de gradients. Cet algorithme, nommé SURF pour *Speeded-Up Robust Features*, est très dynamique et dépend fortement de son jeu de données, puisque ce dernier va conditionner la taille des blocs de traitement, le nombre de points d'intérêt gérés et la complexité de calcul du descripteur de ces points (orientation, mise à l'échelle, etc.).

Cet algorithme s'articule autour de quatre grandes parties de traitement :
- calcul des images intégrales ;
- détection des points d'intérêt avec prise en compte de l'échelle ;
- calcul de l'orientation du point d'intérêt (fenêtre de gradients d'orientation, calcul des gradients voisins et pondération de ceux-ci) ;
- calcul du descripteur du point d'intérêt.

A ces étapes s'ajoute également l'étape de mise en correspondance entre les points d'intérêt des différentes images.

Pour le traitement, l'image de départ est découpée en blocs de cinquante par cinquante pixels afin de réaliser le calcul des images intégrales et de déterminer les points d'intérêt. Ces points sont ensuite récupérés pour effectuer le calcul de l'orientation et du descripteur qui leur est associé.

Il existe à l'heure actuelle deux implémentations de l'algorithme : une version **mono-tâche** et une version **mutli-tâches**. Nous ne considérons, dans le cadre de cette étude, que **la version mono-tâche** afin d'évaluer dans un premier temps les potentielles optimisations algorithmiques qu'il est possible d'envisager.

### 5.9.2 Identification des optimisations dynamiques potentielles

A l'aide de l'outil Kcachegrind de Valgrind, nous avons réalisé **le profilage de l'application** afin d'en déterminer les phases les plus critiques en temps d'exécution, sur une plateforme x86. L'arbre d'appel obtenu est présenté sur la figure 5.9. Sur cet arbre d'appel figure chacune des fonctions principales de l'application et le nombre total de cycles passés dans dans ces fonctions (pour toutes les occurrences de celles-ci). Sur les branches figurent justement ces nombres d'occurrences d'appel pour chacune des fonctions. Une simple division permet d'obtenir le nombre de cycles pour chaque exécution des différentes fonctions.

Sur une plateforme x86, l'application s'exécute en 261 millions de cycles pour réaliser le traitement de deux images. Sur ces 261 millions de cycles, 105 sont consacrés aux opérations de calcul des descripteurs des différents points d'intérêt. Cette étape représente 40 % du temps total d'exécution de l'application.

Parmis ces 105 millions d'instructions, 36 millions sont dévolues à l'appel et à l'**exécution de la fonction** `exp()` de la librairie math.h pour y réaliser les calculs d'exponentielles nécessaires lors du calcul des Gaussiennes. Ces gaussiennes sont employées pour le calcul des différents descripteurs dans le cadre de la pondération du résultat des filtres de Haar utilisés dans le voisinage du point d'intérêt considéré pour la prise en compte de l'effet d'échelle et de la rotation. Ces calculs d'exponentielles représentent 14 % du temps total d'exécution de l'application pour 249200 occurrences.

Cette fonction, coûteuse en nombre de cycles (147 cycles par occurrence), offre un fort potentiel d'optimisation à la compilation en procédant par évaluation directe de celle-ci, évitant l'appel de la fonction. Cette évaluation n'est possible que si ses paramètres sont connus lors de la compilation, ce qui n'est pas le cas lorsque celle-ci est réalisée statiquement.

Or, dans le cadre d'une compilation dynamique, ces paramètres sont connus et peuvent être pris en compte pour effectuer l'étape de calcul directement lors de sa compilation. Cette optimisation ne présente un intérêt que si elle peut être rentabilisée sur

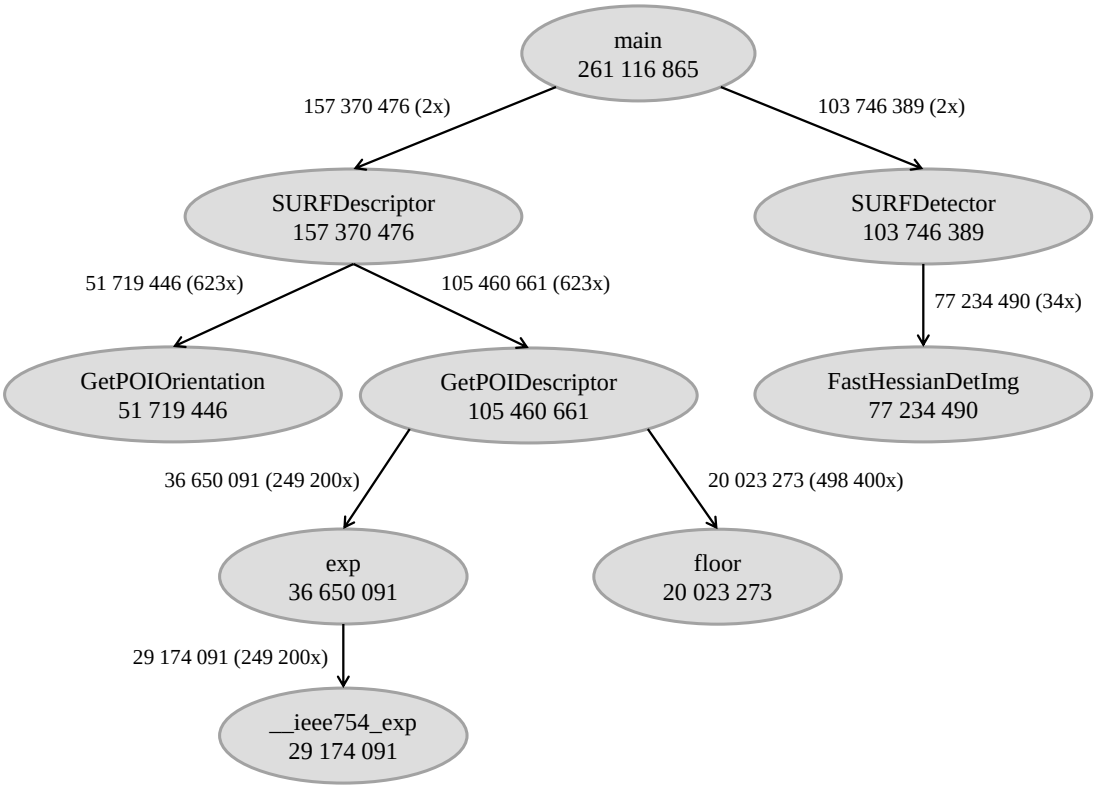



FIGURE 5.9 – Graphe d'appel des fonctions de l'application SURF, faisant figurer le nombre de cycles total passé dans chacune des fonctions et le nombre d'occurrences pour chacune d'entre elles.

une séquence de plusieurs images (permettant d'amortir le surcoût induit par la compilation dynamique en évitant de nouveaux calculs d'exponentielles). Les paramètres de la fonction `exp()` sont très semblables d'une image à une autre dans le cadre d'un flux vidéo, puisque les images se succédant au sein de celui-ci présentent peu de variations au niveau des paramètres de leurs points d'intérêt, notamment au niveau de l'échelle et de l'orientation.

Il est donc fortement envisageable d'effectuer une **recompilation dynamique de la fonction `GetPOIDescriptor`** afin d'y réaliser directement la mise en ligne des résultats des calculs des exponentielles au sein des Gaussiennes pour une série d'images, évitant son appel systématique.

### 5.9.3 Estimation des gains potentiels obtenus par compilation dynamique et par notre accélérateur

Nous avons évalué **les gains potentiels** que permet d'obtenir cette évaluation à la compilation de l'exponentielle sur les temps d'exécution de l'application. Afin de rester dans le cadre du domaine des architectures embarquées, bien que les profilage aient été effectués sur x86 pour des questions d'outils à disposition, nous avons réalisé l'évaluation de ces gains sur une cible ARM Quad-Core Cortex-A9, la même que celle ayant servi dans les troisième et quatrième chapitres de ce manuscrit.

Ne disposant pas d'une chaîne de compilation dynamique complète et fonctionnelle, permettant de réaliser la compilation dynamique de portions de code au cours de l'exécution, il n'a pas été possible de tester directement la mise en application de notre système sur SURF. Ce type de mise en place nécessiterait soit de développer un

moteur de compilation dynamique complet, engendrant des temps de développement conséquents, soit de réutiliser le moteur expérimental fournit par LLC, appelé LLI. Le problème de ce dernier est qu'il réalise de nombreux appels systèmes dont il est difficile de s'affranchir. Or, notre simulateur ne fournissant qu'une émulation de système d'exploitation minimaliste, ce cas de figure est inenvisageable. Seul notre version accélérée matériellement de LLC, et bien évidemment la version initiale, sont opérationnelles. Nous ne pouvons donc mesurer que les temps de compilation seuls, hors exécution. Il est par conséquent nécessaire de réaliser la chaîne de compilation dynamique en quelque sorte manuellement, pour une évaluation à gros grains des performances envisageables.

Les temps de recompilation ont été évalués sous LLC, avec et sans notre accélérateur. Nous avons pour cela généré statiquement la représentation intermédiaire LLVM de la fonction `GetPOIDescriptor`, comme cela aurait été réalisé dans n'importe quel cas de figure dans un cadre réel. Nous avons notamment constaté, en décompilant le code à octet généré via la commande `llvm-objdump`, que la fonction `exp()` figure bien au sein de ce code à octet, dans un bloc de base, et qu'aucune mise en ligne de cette fonction n'a été effectuée. Comme déjà mentionné LLC dispose d'un banc de registre infini pour sa représentation intermédiaire, de type machine à registres. Les paramètres de cette fonction sont donc contenus dans ces registres.

Nous avons généré une série de différentes versions de cette représentation intermédiaire en imposant les paramètres de la fonction `exp()` directement dans la fonction appelante en charge du calcul de la Gaussienne. Ces résultats ont été choisis après profilage de l'évolution de ces valeurs au sein d'une exécution classique de l'application, afin d'obtenir une série de valeurs réalistes. Bien évidemment, ces valeurs imposées engendreront des erreurs sur les calculs de Gaussienne de par leur inexactitude potentielle à un instant donné. Mais nous ne cherchons pas spécifiquement à obtenir un résultat exact au final, simplement à mesurer l'évolution des temps d'exécution. Nous nous sommes toutefois assuré que la réalisation de calculs sur des paramètres inexacts de ce type n'avait aucune influence sur le comportement de l'exécution de l'application.

**Nos expérimentations** ont permis d'évaluer les optimisations réalisées par LLC à partir des différentes versions de représentation intermédiaire de `GetPOIDescriptor` dans lesquelles figurent les différents paramètres de `exp()`, sans que cette dernière n'ait été optimisée. La décompilation du code exécutable fait apparaître que le compilateur réalise comme convenu systématiquement le calcul de l'exponentielle et la mise en ligne du résultat en fonction des paramètres. Mais plus globalement, cette décompilation met en avant le fait que l'ensemble du calcul des Gaussiennes bénéficient de cette optimisation grâce à la connaissance des paramètres de l'exponentielle en recompilant dynamiquement `GetPOIDescriptor`. Les gains attendus sont donc supérieurs à la proportion du temps d'exécution passé dans la fonction exponentielle (14 %), puisque la totalité de la fonction de calcul des Gaussiennes est calculée à la compilation.

Les temps d'exécution de l'application sans l'optimisation apportée sont de 0.515 seconde contre 0.410 seconde avec. Ces temps sont donnés dans les deux cas pour le traitement d'une seule image (détection des points d'intérêt et calcul des descripteurs associés). Cette optimisation permet donc l'obtention d'une accélération de 1.25× grâce à la suppression des appels à la fonction de calcul des exponentielles et à celle de calcul des Gaussiennes. Notons que ce résultat ne tient pas compte du surcoût induit par la compilation dynamique.

Les temps de compilation pour la génération du code exécutable relatif à la fonction de `GetPOIDescriptor` sont de 0.31 seconde dans la version initiale optimisée logiciellement du compilateur, et de 0.25 seconde dans notre version accélérée matériellement, respectant les ordres de grandeurs des gains obtenus dans nos évaluations.

Nous avons, par conséquent, effectué la comparaison entre le temps d'exécution

initial, sans cette optimisation, et le nouveau temps d'exécution du code optimisé auquel est ajouté le surcoût en temps de compilation de LLC. Les résultats obtenus mettent en avant une perte de performances de l'ordre de 28.4 % pour la version initiale de LLC et de 21.7 % pour notre version normalisée bénéficiant de l'accélérateur, pour le traitement d'une image. Cela représente une baisse de 9.4 % des pertes de performances induite par l'introduction de la compilation dynamique.

Afin d'évaluer **le nombre d'images nécessaires pour rentabiliser l'utilisation de la compilation dynamique**, une extrapolation des résultats obtenus a été réalisée. La compilation dynamique n'est réalisée qu'une fois, sur la première image. Les résultats obtenus sont présentés sur la figure 5.10. Sur cette figure il est possible de voir l'évolution des gains obtenus sur l'application grâce à l'utilisation de la compilation dynamique. Ces gains sont donnés par rapport au temps d'exécution initial de l'application, en fonction du nombre d'images traitées. Les courbes évoluent selon les fonctions suivantes :

$$gain_sw(nbimages) = \frac{1}{\frac{tsurfopt}{tsurforigin} + \frac{tcomp_sw}{nbimages*tsurforigin}} \tag{5.1}$$

$$gain_hw(nbimages) = \frac{1}{\frac{tsurfopt}{tsurforigin} + \frac{tcomp_hw}{nbimages*tsurforigin}} \tag{5.2}$$

*gain_sw(nbimages)* correspond à la fonction d'évolution des gains obtenus par la compilation dynamique en utilisant la version optimisée logiciellement de LLC (temps de compilation *tcomp_sw*), en fonction du nombre d'images. *gain_hw(nbimages)* correspond à la fonction d'évolution des gains obtenus par la compilation dynamique en utilisant la version normalisée accélérée matériellement de LLC (temps de compilation *tcomp_hw*), en fonction du nombre d'images. *tsurfopt* et *tsurorigin* correspondent aux temps d'exécution de l'application SURF, respectivement avec et sans l'optimisation réalisée au niveau de la fonction exponentielle.

Il est possible de voir, grâce à ces deux courbes, que le niveau de rentabilité pour l'utilisation de la compilation dynamique est très vite atteint, dès la troisième image dans les deux cas. Ce niveau de rentabilité correspond au passage de la droite $y = 1$, signifiant une accélération par rapport à la version initiale. Avec les deux courbes, et grâce également à la troisième courbe présentant l'évolution du gain de la version utilisant notre accélérateur par rapport à la version logicielle de base, il est possible de voir que ce gain justement a tendance à disparaître dans le temps en tendant vers 1. Avec et sans accélérateur, nous tendons vers les gains initialement mesurés en accélération grâce à l'optimisation introduite (1.25×).

Ce résultat illustre l'influence décroissante de la compilation dynamique (n'ayant été réalisée qu'une seule fois sur la première image) sur le temps total d'exécution au fur et à mesure que celui-ci s'accroît avec l'augmentation du nombre d'images traitées. Toutefois, dans un cadre réel de mise en application, il demeure difficilement envisageable d'attendre vingt-cinq images avant de procéder à une recompilation. Les paramètres de l'exponentielle varient en effet peu d'une image à une autre, mais sur vingt-cinq images cette variation est suffisamment significative pour engendrer d'importantes erreurs sans recompilation. Compte-tenu des résultats obtenus sur la figure 5.10, nous estimons que celle-ci, en intervenant toutes les dix images, permet de rentabiliser sa réalisation en obtenant un gain sur l'application de plus de 90 % du gain optimal (mesuré sans prise en compte du surcoût induit).

### 5.9.4 Conclusion et perspectives sur l'étude

Nous avons réalisé **une mise en application de notre accélérateur dans un cadre applicatif réel d'utilisation de la compilation dynamique**. Les résultats

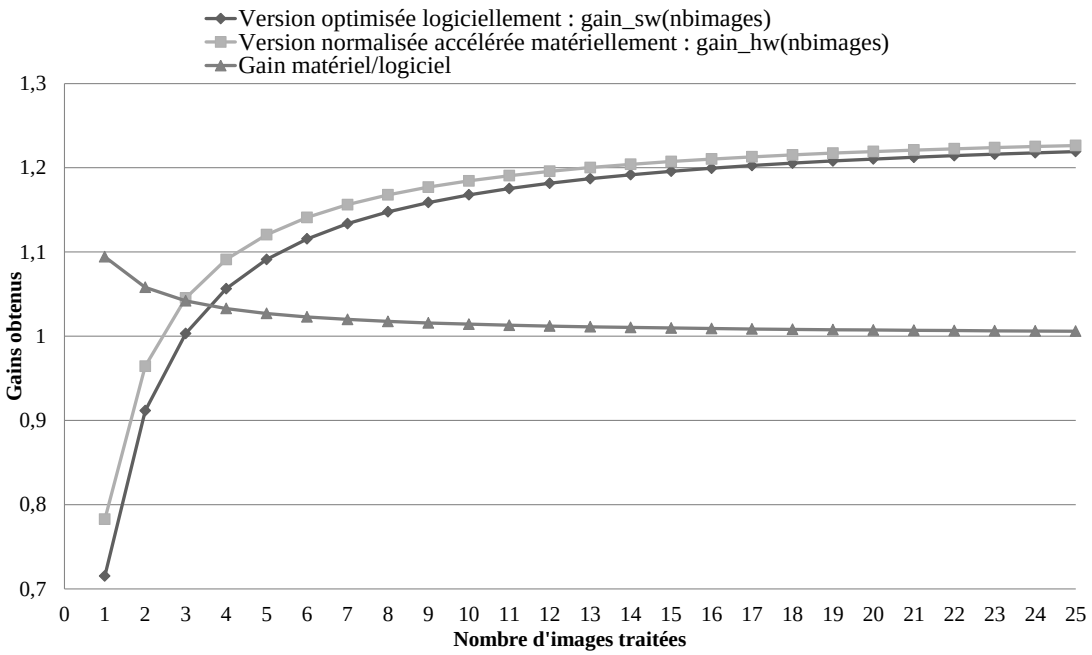



FIGURE 5.10 – Gains obtenus sur l'application SURF par l'utilisation de la compilation dynamique avec LLC. Deux version de LLC sont comparées : la version initiale optimisée logiciellement et notre version normalisée bénéficiant de l'accélérateur matériel proposé dans ce chapitre.

obtenus ont permis de mettre en avant l'intérêt de notre solution dans l'optique de limiter le surcoût induit par la compilation dynamique, avec une diminution de 9.4 % de celui-ci pour une compilation avec LLC, avec et sans l'accélérateur de gestion des tableaux associatifs proposé. La rentabilisation de l'introduction des phases de compilation s'effectue toutefois tout aussi rapidement dans les deux cas, dès la troisième image, comme il est possible de le voir sur la figure 5.10.

Ce résultat, forcément différent de celui espéré, s'explique par la nature de l'application accélérée et surtout par l'optimisation mise en oeuvre. L'implémentation existante, faisant appel à ces calculs d'exponentielles, est clairement sous-optimale. Les gains obtenus par la compilation dynamique sont donc très importants dès la première exécution, alors que celle-ci devrait être amortie sur de plus longs cycles d'exécution (sur un certain nombre d'images et non dès les premières). Ces gains sont trop importants pour mettre en avant un quelconque gain de notre accélérateur quant à la rentabilisation de la compilation dynamique.

Une optimisation envisageable, au niveau de cette application et de son algorithme, avant de faire appel à la compilation dynamique, est de mettre en place des séries de Gaussiennes pré-calculées. Cela permet de s'affranchir du calcul de celles-ci (et des exponentielles associées) en allant directement chercher la valeur dans des tables en fonction des paramètres. Une autre solution consiste à mettre en place des approximations de calculs de Gaussiennes ou d'exponentielles, en procédant par exemple par développement en séries limitées pour ces dernières.

Bien évidemment, une autre piste d'exploration possible concerne les optimisations au niveau de la parallélisation des tâches de l'application. La taille de ces tâches et leur nombre sont très dépendants du jeu de données puisque la parallélisation, dans l'implémentation actuelle multi-tâches est effectuée de la façon suivante :

- Détection des points d'intérêt : assignation d'une tâche par bloc de cinquante par cinquante pixels ;
- Calcul de l'orientation et du descripteur du point d'intérêt : assignation d'une

tâche par sous-ensemble de points d'intérêt (dont la taille est déterminée par un calcul du nombre maximum de points d'intérêt admissible par tâche) ;

Pour la mise en correspondance entre les images, une tâche est également assignée par sous-ensemble de points d'intérêt, obtenus de la même façon que pour le calcul de l'orientation et du descripteur. L'implémentation multi-tâches réalisée se présente sous la forme d'une mise en place de la gestion d'un groupement de tâches, réparties sur les différentes ressources de l'architecture considérée en fonction de la disponibilité de celles-ci. La dépendance des données entre les différentes étapes est gérée par l'intermédiaire de barrières qui sont positionnées entre chacune des grandes étapes du traitement, faisant avancer l'ensemble de celui-ci entre les blocs de manière synchrone. Ainsi, deux barrières sont positionnées. La première après le calcul des images intégrales, pour assurer la distribution des blocs de pixels dans les différentes tâches, dans l'optique de réaliser l'étape de détection (les tâches sont ensuite réparties selon les ressources disponibles). La seconde barrière est positionnée après l'étape de détection et avant l'étape de calcul de l'orientation et des descripteurs, au moment de l'assignation des sous-ensemble de points d'intérêt aux différentes tâches.

Une piste envisageable par la suite concerne donc la possibilité de jouer sur le nombre de tâches à travers le nombre de blocs et de sous-ensembles manipulés et sur la taille de ces blocs et de ces sous-ensembles. Cette approche est notamment valable dans le cadre d'une évolution des ressources disponibles à l'exécution et/ou de la qualité de service désirée : réduction de la résolution de l'image de base (passage de la norme HD à la norme QHD, réduction des filtrages réalisés en interne pour le traitement en supprimant certaines gaussiennes, etc.).

# Chapitre 6

# Concept d'extension matérielle pour l'accélération du graphe des instructions de l'application à compiler

**Sommaire**



## 6.1 Introduction

Les études réalisées dans le chapitre 4 ont permis de mettre en avant l'impact, au sein des codes de compilation dynamique, de la gestion des graphes des instructions de représentation intermédiaire et de code machine de l'application à compiler. Cet impact se manifeste en particulier au niveau des étapes de chargement et de décodage de ces instructions. En ce basant sur ces conclusions, nous nous proposons d'accélérer ces deux étapes. Pour cela, nous avons tout d'abord identifié les différentes étapes relatives à la gestion du graphe de ces instructions (phases de chargement, de décodage et de traduction). Nous nous sommes basés sur le cas de la machine virtuelle Strata [14] qui propose un traducteur dynamique générique avec une gestion unifiée du chargement et

décodage. En s'appuyant sur l'analyse déjà réalisée des efforts existants sur cette question (optimisations logicielles, gestion mémoire, matériel dédié, etc.) nous proposons **un accélérateur matériel pour les phases de chargement et de décodage**. Nous justifions cette proposition selon les deux points suivants, basés sur les études réalisées dans les chapitres 3 et 4 :

- l'impossibilité d'accélérer la totalité des phases de gestion à cause de la complexité et de l'irrégularité inhérentes aux codes de compilation dynamique ;
- la non-accélération de phases non-génériques afin de ne pas impacter la flexibilité et la portabilité de la solution (pas d'accélération de phases dépendantes de la cible notamment) ;

Ce chapitre présente tout d'abord le **concept de cet accélérateur** ayant fait l'objet **d'un dépôt de brevet** intitulé *"Système de compilation dynamique d'au moins un flot d'instructions"*. Dans un deuxième temps, nous introduisons **une étude d'implémentation** en cours de cet accélérateur, réalisée par Florent Berthier dans le cadre d'un stage de fin d'études. Nous présentons également les premières évaluations en performances et en surface silicium de la solution, obtenues durant ce stage.

## 6.2 Identification des différentes étapes de gestion du flot d'instructions à compiler

### 6.2.1 Présentation générale

**La prédominance des manipulations de structures chaînées** au sein des codes de compilation dynamique a permis de mettre en avant l'importance des étapes relatives à ces manipulations en termes de temps d'exécution, notamment concernant les étapes de chargement et de décodage des instructions de l'application à compiler. Ces étapes sont récurrentes dans l'ensemble des systèmes de compilation dynamique. Pour l'illustrer, nous nous intéressons au cas du traducteur dynamique de la machine virtuelle Strata [14], dont la structure est donnée sur la figure 6.1.

Il est possible de voir, sur cette figure de description fonctionnelle du traducteur, que celui-ci possède une sorte **de machine dans la machine**. Nous y retrouvons, en effet, une structure de gestion des instructions à compiler très proche de la structure de pipeline d'une machine classique de type processeur, avec des étapes de chargement, de décodage et de traduction (et non d'exécution) des instructions à compiler. Dans le cas de Strata, ces instructions sont de type instructions de code à octet Java. C'est sur cette machine que nos études se sont focalisées afin d'aboutir à la proposition de cet accélérateur.

Détaillons le fonctionnement du traducteur dynamique de la machine virtuelle Strata, tel que présenté sur la figure 6.1. Le flot d'instructions à compiler est contenu dans la mémoire située à gauche de cette figure, correspondant à la mémoire de programme du système dans laquelle est contenu l'ensemble du code de l'application. Ce flot d'instructions correspond à une séquence de code à octet Java à compiler dynamiquement en code natif. Ce flot est chargé par une unité de capture du contexte en charge de la localisation de l'ensemble de la séquence de code à traduire.

L'adresse de début de cette séquence est ensuite chargée dans le pointeur d'exécution "Nouveau PC". L'étape suivante consiste à vérifier si le code considéré n'a pas déjà été traduit. Dans le cas de la machine virtuelle Strata, les portions de code qui viennent d'être compilées, ainsi que les plus utilisées d'entre elles, sont contenues dans une mémoire cache associée au système (à droite sur la figure). Si le code a déjà été traduit dans le même contexte d'exécution, alors la portion traduite conservée en mémoire cache est directement récupérée et chargée pour exécution.

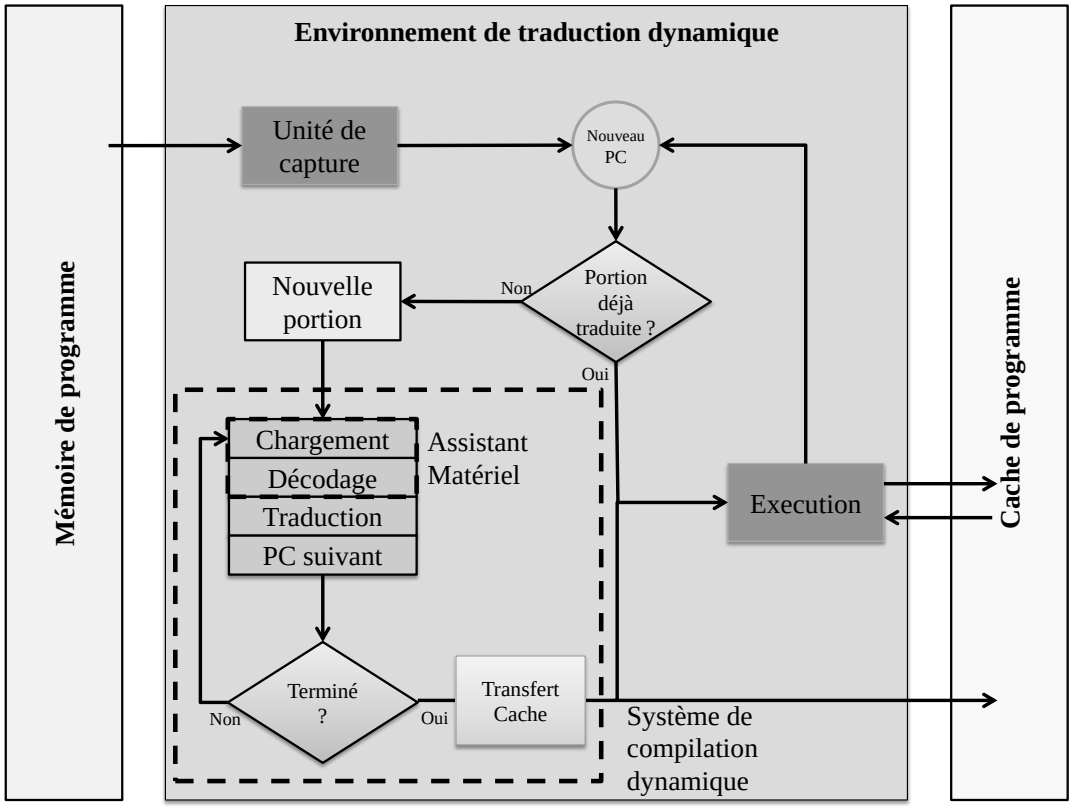



FIGURE 6.1 – Structure d'un traducteur dynamique (extrait de la machine virtuelle Strata).

Si cette séquence n'a pas déjà été traduite, alors elle est considérée comme une nouvelle portion de code à traduire et est récupérée par l'unité en charge de la création de ces portions (qui peuvent être par exemple des blocs de base). La portion va alors être traitée selon une succession d'opérations de chargement, de décodage et de traduction des différentes instructions la composant. Ce traitement peut être réalisé de manière itérative en fonction des optimisations qu'il a été décidé de réaliser par la partie contrôle du traducteur (non représentée ici).

Une fois l'ensemble du traitement réalisé, la portion de code compilée est envoyée en mémoire cache via le contrôleur associé. Puis elle est récupérée au sein de ce cache pour exécution.

### 6.2.2 Justification de notre approche

On peut voir que cette approche fait en quelque sorte la part belle **au stockage des portions de code générées en mémoire**. C'est d'ailleurs sur la base de cette machine virtuelle Strata que sont réalisés les travaux de Baioicchi et al. [13] sur la mise en place de caches sophistiqués pour accentuer les capacités de stockage de ces portions de code (via des systèmes de compression de code et d'analyse des portions les plus fréquemment exécutées). Cette solution vise à limiter l'impact de la compilation dynamique sur les temps globaux d'exécution en limitant ses occurrences. Elle n'est ainsi réservée qu'aux portions non compilées ou aux portions pouvant bénéficier d'importantes optimisations et dont les gains permettront de compenser le surcoût induit.

Contrairement à ces études, qui se focalisent essentiellement sur la partie gestion de la mémoire, nous proposons de nous focaliser sur **l'optimisation de la machine** mise en avant. Cette approche est assez similaire à celle adoptée dans le cadre des études sur les processeurs Java et notamment sur ARM Jazelle [23]. Mais, contrairement à

celle-ci, nous cherchons à mettre en place une solution offrant à la fois performances et flexibilité. Nos analyses sur cette machine au sein des traducteurs dynamiques binaires permettent de mettre en avant une grande généricité dans les étapes de chargement et de décodage.

Quelque soit les systèmes, les opérations réalisées à ce niveau restent les mêmes : chargement des séquences d'instructions, et cela peu importe leur niveau de représentativité (code source, représentation intermédiaire, code machine, etc.), et décodage de ces instructions (extraction de l'opérande ou de l'opération réalisée, récupération des paramètres, etc.). L'étape de traduction est quant à elle beaucoup plus variable puisqu'elle va évoluer en fonction du code manipulé, des optimisations à réaliser et de l'architecture cible.

Notre approche consiste donc à **déporter ces étapes de chargement et de décodage sur l'accélérateur**, dans une optique de performances, et de laisser les étapes de traduction au processeur programmable en charge de la compilation dynamique, pour des questions de flexibilité. Nous illustrons ces choix sur la figure 6.1, l'association proposée des deux ressources formant le système de compilation dynamique introduit au chapitre 4.

## 6.3 Présentation du concept développé pour l'accélération de la gestion du flot d'instructions

L'objectif de cette section est de présenter **le concept d'accélération** proposé dans le cadre de ce chapitre et d'en expliquer l'intégration avec le processeur en charge de la compilation dynamique.

### 6.3.1 Présentation du couplage entre le processeur et l'accélérateur

La décision a été prise de réaliser **un couplage sous la forme d'un co-processeur** entre l'accélérateur des phases de chargement et de décodage et le processeur programmable en charge de la traduction. L'objectif est de mettre en place un ensemble permettant de paralléliser l'exécution de ces phases de compilation entre les deux unités. Les phases de chargement et de décodage sont réalisées en avance de phase par rapport aux portions sur lesquelles travail le processeur. Ainsi, celui-ci dispose directement du résultat de ces opérations lorsqu'il s'attaque à leur traitement, n'ayant plus qu'à réaliser le phase de traduction.

Pendant que le processeur réalise la traduction de la section de code n, l'accélérateur réalise le chargement et le décodage de l'unité n+1. L'accélération proposée est donc double : en plus de réduire les temps d'exécution de ces deux premières étapes de chargement et de décodage, grâce à **la mise en place de matériel dédié** à ces opérations, cette **parallélisation** permet de réduire encore le temps global d'exécution. L'illustration de cette proposition est présentée sur la figure 6.2.

**Ce choix** de la mise en place sous la forme d'un **co-processeur** est lié à trois raisons principales : la portabilité de la solution, la complexité des opérations accélérées et leur spécificité.

Du point de vue **portabilité**, nous cherchons, contrairement à l'accélérateur précédent, à mettre en place une solution beaucoup plus autonome de type périphérique pouvant être facilement transférée d'un processeur à un autre. Cette implémentation permet d'envisager aussi bien le type de solution que celle envisagée dans le cadre de cette thèse et présentée au chapitre 4, consistant en l'association d'un processeur dévoué à la compilation dynamique, qu'une inclusion directe de la solution sur des ressources de la cible (et par exemple dévouée à l'exécution de l'application).

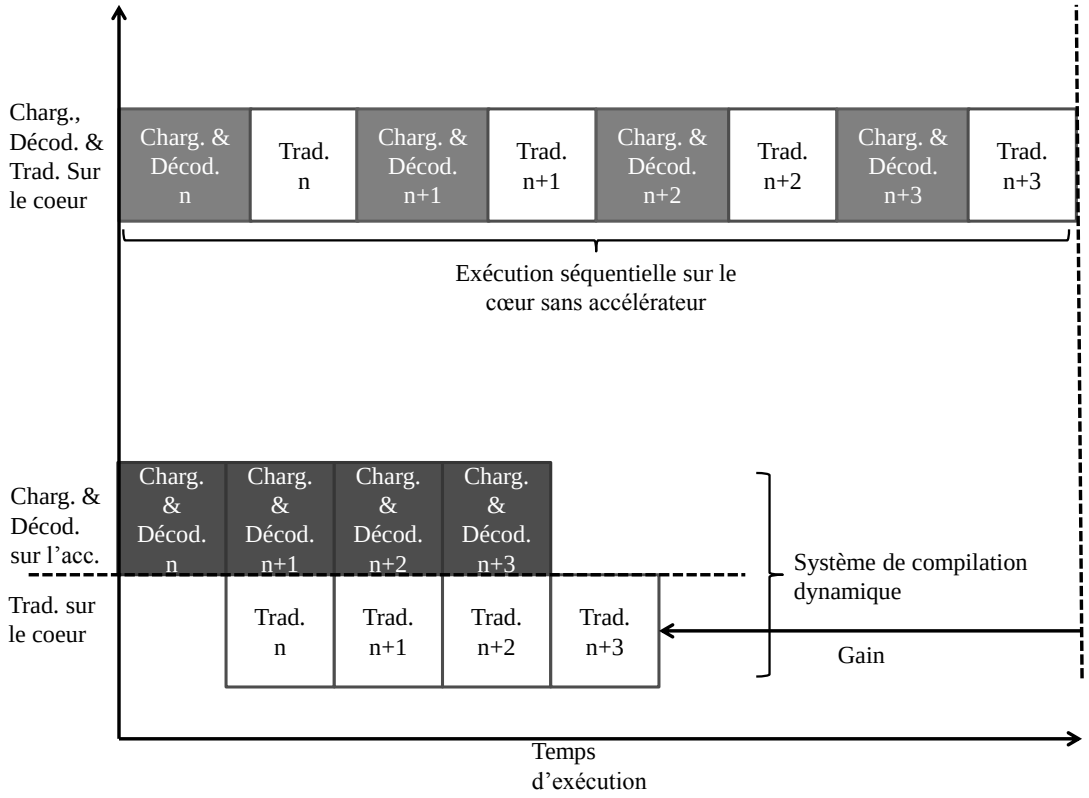



FIGURE 6.2 – Mise en avant de l'intérêt de la déportation et de l'optimisation des phases de chargement et de décodage sur un accélérateur matériel fonctionnant en parallèle avec le processeur associé. Charg. : chargement, Décod. : décodage, Trad. : traduction.

Concernant **la complexité des opérations accélérées**, ce choix est motivé par la difficulté, selon nous, d'implémenter ce type d'accélération sous la forme d'une unité fonctionnelle. Les opérations accélérées sont en effet très complexes (contrairement aux opérations de gestion des arbres rouges et noirs) et il paraît difficile de les réaliser directement dans le pipeline du processeur sous la forme d'instructions spécialisées, sans considérablement ralentir celui-ci. De plus, la spécificité des opérations réalisées (chargement et décodage), laissent à penser à une solution matériellement non triviale, constituée d'opérateurs assez sophistiqués et différents de ce qui est géré en temps normal par des processeurs de type embarqué. Il apparaît en effet difficilement envisageable de mettre en place une solution du type de celle présentée dans le chapitre précédent, constituée uniquement d'opérateurs simples.

Ce choix est également lié au fait que ces opérations accélérées sont beaucoup plus **spécifiques à la compilation dynamique** et ne peuvent visiblement bénéficier qu'à ce domaine algorithmique, contrairement à la solution précédente. En effet, cette dernière peut concerner un panel beaucoup plus large de domaines applicatifs et algorithmiques puisqu'accélérant la gestion des tableaux associatifs, largement utilisés aujourd'hui pour la manipulation des structures de données complexes. Cet accélérateur présente donc, selon nous, un intérêt plus limité que la solution précédente, justifiant plus difficilement la modification en profondeur de la structure d'un processeur qu'engendre la mise en place d'une nouvelle unité fonctionnelle.

### 6.3.2 Présentation de l'accélérateur

L'association du processeur et de l'accélérateur (co-processeur) permet l'obtention d'un système de compilation dynamique. Cet ensemble est intégrable sur une architecture multi-coeurs (hétérogène ou non), comme il est possible de le voir sur la figure 6.3 qui a déjà été introduite au chapitre 4. L'accélérateur proposé dans le cadre de ce chapitre y est mentionné sous le nom *HW 2*.

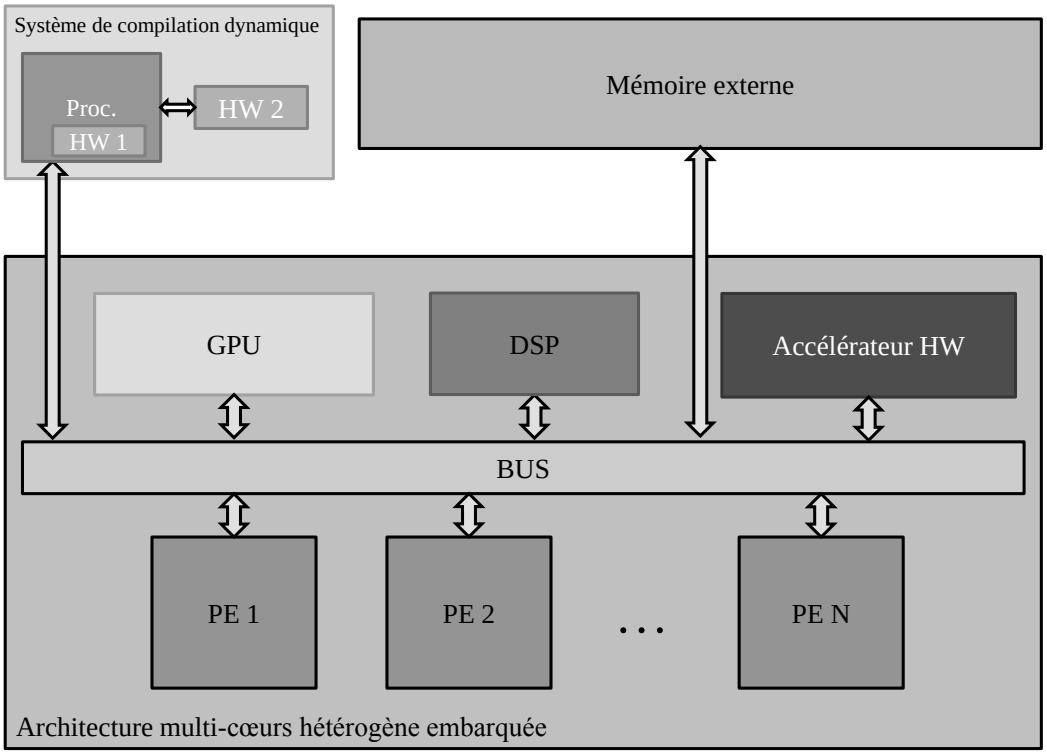



FIGURE 6.3 – Schéma de l'architecture proposée incluant le système de compilation dynamique, composé du processeur dédié à la compilation dynamique et/ou aux services de virtualisation et des accélérateurs proposés, sous la forme d'unités fonctionnelles ou de co-processeurs.

Voici une description **du principe de fonctionnement** du système proposé à gros grain, basée sur la figure 6.4 : la portion de code à traduire, qu'elle soit sous la forme de code source, de représentation intermédiaire ou de code machine est chargée en provenance de la mémoire dans l'assistant matériel. Ce chargement fait suite à une requête en provenance du processeur auquel est associé l'accélérateur, requête elle-même en provenance de l'architecture multi-coeurs associée. Cette requête est générée via le système d'exécution de cette architecture, décidant par exemple de la recompilation d'une portion de code pour cause de fréquence d'exécution et/ou d'optimisation potentielle, grâce à son système de profilage de l'exécution de l'application. Cette requête, envoyée par le processeur à l'accélérateur peut être réalisée pendant que celui-ci est déjà en cours de traduction d'une autre portion de code.

Le code chargé est ensuite analysé par l'unité de parcours de graphe (dont le fonctionnement interne est présenté juste après). Une fois les étapes de chargement et de décodage réalisées par cette unité pour l'ensemble de la portion de code, les résultats sont envoyés dans une mémoire interne de type mémoire cache et une requête d'acquittement est envoyée au processeur, avec l'adresse en mémoire cache du résultat. Celui-ci récupère ensuite le résultat contenant l'ensemble des instructions chargées et décodées

de la portion considérée. Il peut également envoyer simultanément une nouvelle requête à l'accélérateur pour la prochaine portion à traiter.

Une fois le résultat récupéré, il réalise la dernière étape de traduction du code. Cette étape réalisée, il envoie le code généré dans la mémoire de l'architecture cible, et plus spécifiquement dans la zone de mémoire de programme réservée à l'unité sur laquelle doit être exécutée la portion de code. Pour bien comprendre le fonctionnement en avance de phase mis en place, il faut imaginer que pendant que le coeur effectue la traduction d'une portion de code n à générer, l'assistant matériel effectue l'analyse de la portion n+1. Ces étapes terminées, le processeur récupère le résultat et traduit cette portion n+1, pendant que l'accélérateur se charge de la portion n+2. En d'autres termes, cela revient à mettre en place un nouveau pipeline des étapes de chargement/décodage/traduction, en offrant une architecture optimale pour effectuer les deux premières étapes.

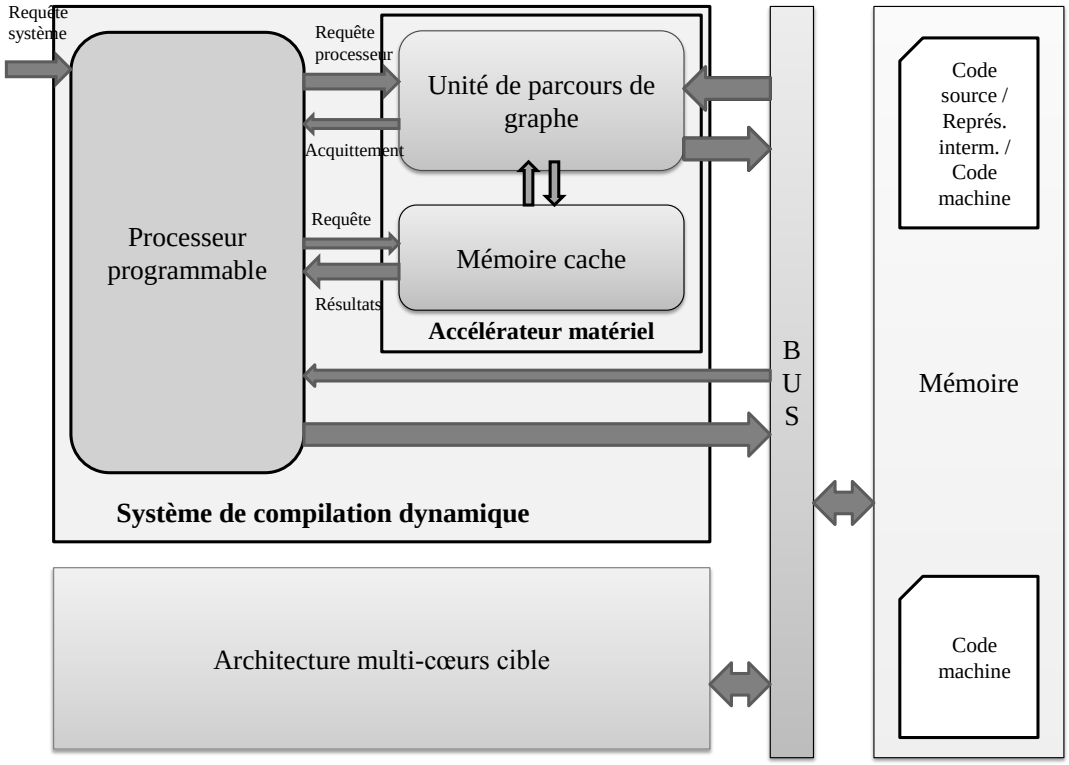



FIGURE 6.4 – Schéma de présentation du système de compilation dynamique proposé, incluant l'unité de parcours de graphe en charge de la gestion des étapes de chargement et de décodage des instructions de l'application à compiler.

Descendons maintenant en granularité pour **la présentation du fonctionnement même de l'unité de parcours du graphe**, illustré par la figure 6.5. Cette unité est composée de trois blocs principaux : **un contrôleur**, **une unité de chargement** et **un décodeur**.

**Le contrôleur** est en charge de la gestion des requêtes en provenance du processeur et de leur émission vers les deux autres unités. Il est également en charge des signaux d'acquittement pour le processeur auquel est couplé l'accélérateur. Le contrôleur reçoit des signaux d'acquittement en provenance des deux autres unités de l'accélérateur. C'est également le contrôleur qui envoie au processeur la requête de fin de traitement des différentes portions de code, avec leur adresse de stockage en mémoire cache pour qu'il puisse en effectuer la récupération avant traduction.

**L'unité de chargement** est en charge de la récupération de la portion de code à analyser en mémoire via une requête à celle-ci. Elle parcourt ensuite la portion afin de réaliser le chargement de l'ensemble des instructions la composant. Elle vérifie en mémoire cache si cette portion n'a pas déjà été chargée et décodée, auquel cas elle envoie directement une requête au contrôleur avec l'adresse correspondant à cette portion qui la renvoie au processeur pour l'étape de traduction, évitant ainsi un décodage inutile de la portion. Il convient de garder à l'esprit que cette mémoire ne contient que les portions chargées et décodées et en aucun cas les portions déjà traduites. La gestion de ces dernières est à la charge du système d'exécution de l'architecture cible. Si une requête est arrivée jusqu'au système de compilation dynamique, c'est qu'une traduction doit être dans tous les cas réalisée (pour la première fois ou pour une optique de ré-optimisation).

**Le décodeur** est en charge quant à lui des étapes de décodage. Cette unité reçoit au fur et à mesure les instructions chargées par l'unité de chargement. Lorsque le décodage d'une instruction est réalisé, le résultat est envoyé dans la mémoire cache associée à l'accélérateur. Ce résultat est également envoyé à l'unité de chargement afin que celle-ci puisse en bénéficier pour le chargement des prochaines instructions (adresse suivante, dans le cas de branchements par exemple). C'est également par ce biais que l'acquittement du décodage de l'ensemble de la portion de code est réalisé, l'unité de chargement envoyant un signal d'acquittement au contrôleur une fois les différentes opérations de chargement et de décodage réalisées. Une autre possibilité, que nous détaillerons dans le paragraphe suivant, concerne l'envoie du résultat directement à l'architecture multi-coeurs cible.

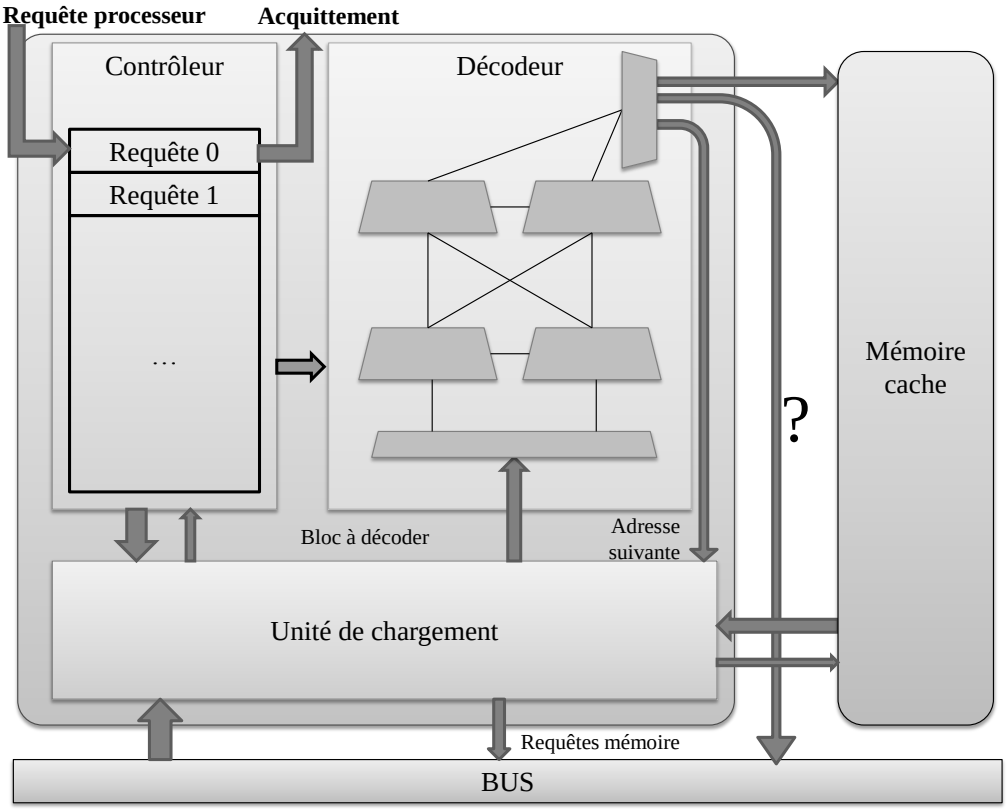



FIGURE 6.5 – Schéma de l'unité de parcours de graphe proposée au sein du système de compilation dynamique, mettant en avant les trois blocs la composant : un contrôleur, une unité de chargement et un décodeur.

Concernant **la granularité de traitement du système mis en place**, trois ni-

veaux ont été retenus. Le premier, au niveau instruction, est assimilable à une simple interprétation. Chacune des instructions de l'application sera chargée et décodée par l'unité, avant d'être traduite par le processeur et envoyée pour une exécution à la volée sur l'architecture multi-coeurs. Dans le cadre du développement d'un décodeur plus sophistiqué, il est possible d'envisager que ce dernier, au moins pour des instructions simples ayant une transcription directe dans le code machine de la cible, réalise directement la partie traduction au moment du décodage. Le résultat peut ainsi être directement envoyé à l'architecture multi-coeurs, sans passer par le processeur. Cette possibilité est mise en avant sur la figure 6.5 via la flèche en sortie du décodeur marquée d'un point d'interrogation. Les deux autres niveaux de granularité sont le niveau bloc de base et le niveau méthode. Le premier permet d'éviter la gestion des branchements au sein d'une portion de code à traiter. Ces branchements ne concerneront en effet que le point de sortie du bloc de base et détermineront le prochain bloc à traiter. Le second permet d'éviter ce découpage préalable sous la forme bloc de base mais oblige à gérer les éventuels branchement au sein du graphe d'instructions de la méthode considérée, obligeant à réaliser la mise en place d'un prédicteur de branchement (c'est ce type d'implémentation que nous présentons dans la prochaine section).

L'accélérateur, tel qu'imaginé dans un premier temps, ne se focalise que sur une prédiction purement déterministe de l'évolution du flot d'instructions, sans aucune spéculation sur les branchements. Une extension possible est de mettre en place des techniques de spéculation pour l'évolution du flux, afin d'augmenter encore la rapidité de la transformation du code et du code traduit en lui-même, en ne traduisant que les chemins effectivement pris lors de l'exécution.

## 6.4 Etude d'implémentation de l'accélérateur dans le cadre de la traduction dynamique binaire

Cette section est dédiée à **la présentation d'une étude d'implémentation du concept d'accélérateur** proposé dans ce chapitre. Ces travaux ont été réalisés par Florent Berthier dans le cadre d'un stage de fin d'études en troisième année d'école d'ingénieur.

### 6.4.1 Présentation du cahier des charges de l'étude réalisée et de son contexte

L'objectif de ce stage est de réaliser l'implémentation fonctionnelle de l'accélérateur des phases de chargement et de décodage des instructions de l'application à compiler. Comme présenté précédemment, cet accélérateur est implémenté sous la forme d'un co-processeur, ajouté comme périphérique au processeur en charge de la compilation dynamique. La description matérielle de l'accélérateur, réalisée en VHDL, a été effectuée dans une optique d'implémentation sur FPGA de la solution. Afin d'évaluer ses performances, il a été décidé de réaliser l'implémentation FPGA de l'ensemble du système de compilation dynamique, incluant le processeur et l'accélérateur (une solution de co-simulation aurait introduit des latences de communication supplémentaires à gérer). Ne disposant pas de modèle de processeur ARM en VHDL, le couplage a été réalisé avec un processeur SPARC LEON 3, à partir de la librairie GRLib [137] développée par Gaisler Research.

### 6.4.2 Présentation du cadre applicatif

Comme mentionné dans le chapitre 4, les phases de chargement et de décodage des instructions de l'application à compiler sont réparties dans l'ensemble des noyaux du

compilateur LLC. Il est apparu comme difficilement concevable, pour un stage d'une durée de six mois, de réaliser la transformation de l'ensemble des noyaux concernés, afin que ceux-ci puissent bénéficier de l'accélérateur développé. Le temps de familiarisation avec le compilateur LLC a été considéré comme trop important vis-à-vis de la durée des travaux.

Ces travaux se sont donc appuyés sur un programme réalisé au sein du laboratoire et basé sur de **la traduction dynamique binaire** de code machine SPARC (et plus spécifiquement de l'optimisation dynamique binaire de ce code machine). L'optimisation réalisée consiste au déroulage des boucles du programme en fonction de paramètres connus au moment de son exécution. Ce programme, comme tous ceux de compilation, réalise des étapes de chargement et de décodage des instructions à recompiler. Dans ce cas d'étude de l'optimisation dynamique binaire, les instructions considérées sont des instructions machines issues du jeu d'instructions Sparcv8.

Le cahier des charges du stage prévoit aussi la modification du programme et l'implémentation de l'accélérateur afin que ceux-ci réalisent la même optimisation dans le cadre de la traduction dynamique binaire, en considérant en entrée du code machine ARM Thumb2 et en générant en sortie du code machine pour processeur SPARC. Toutefois, cette modification n'a pas encore été réalisée au moment de la rédaction de ce manuscrit.

### 6.4.3 Implémentation réalisée de l'accélérateur

Les travaux réalisés par Florent Berthier ont conduit à l'implémentation de l'accélérateur sous la forme d'un périphérique du processeur LEON 3, connecté via l'ensemble de bus de communication utilisé par la GRLib, AMBA 2.0 (ARM). Le périphérique est connecté sur le bus le plus rapide disponible, appelé bus AHB (*Advanced High-performance Bus*).

Le programme gérant la traduction de méthodes entières, une attention toute particulière a été portée à **la gestion des branchements** au sein de celles-ci. Afin de ne pas bloquer l'accélérateur en attendant l'exécution de l'évaluation d'un branchement pour savoir si celui-ci est pris ou non, tous les branchements directs (conditionnels ou non) ont été arbitrairement considérés comme étant pris. Ainsi, l'accélérateur, lors du chargement et du décodage du flot d'instructions, peut continuer son travail lorsqu'il rencontre ce type de branchement. Si jamais cette prédiction s'avère être fausse lors du traitement, alors le processeur ordonne à l'accélérateur de recommencer le chargement et le décodage de l'autre branche du graphe d'instructions, en lui communiquant l'adresse de départ de cette séquence d'instructions. Ce cas entraîne une mise en attente de la portion à traduire pour le processeur, le temps que celle-ci soit chargée et décodée (étapes bénéficiant quoiqu'il arrive de l'accélération matérielle). Cependant, cette mise en attente aurait de toute façon eu lieu en cas de non spéculation.

Concernant les **branchements indirects**, aucune spéculation ne peut-être faite sans connaissance de l'adresse de destination ou de l'adresse contenant la valeur du saut à réaliser. Ces branchements sont donc quoiqu'il arrive bloquants et nécessitent d'attendre qu'ils soient exécutés afin de déterminer l'évolution du flot d'instructions. L'adresse de la prochaine séquence d'instructions à charger et à décoder est alors transmise à l'accélérateur par le processeur. Encore une fois, l'impact se limitera à l'impossibilité de travailler en avance de phase pour l'accélérateur, mais celui-ci permettra tout de même l'accélération des étapes de chargement et de décodage une fois la destination du branchement connue.

Afin de réaliser cette étape de gestion des branchements, l'unité de chargement a été découpée en deux sous-unités. La première, l'unité de pré-chargement, gère le pré-décodage de la prochaine instruction à traiter et notamment la détection des différents

branchements. La seconde, l'unité de chargement, réalise le chargement de l'instruction en fonction du résultat de l'unité de pré-chargement. Cette unité de chargement est également capable de gérer l'alignement des instructions sur 32 bits dans le cadre de la gestion d'instructions Thumb2, sur 16 et 32 bits.

La mémoire cache de stockage des résultats a été implémentée sous la forme d'une mémoire de type FIFO (*First-in, First-out*). Les expérimentations ont montré que le dimensionnement de cette FIFO pouvait se limiter à quatre mots de 128 bits (taille des mots contenant le résultat de chacun instruction chargée et décodée, dont le détail est donné dans la suite de cette section). Contrairement au concept initial présenté en début de ce chapitre, les résultats des instructions chargées et décodées sont envoyés au fur et à mesure au processeur en charge de la traduction (la FIFO permettant de couvrir les éventuels retards et/ou latences).

Pour des questions d'implémentation, et afin de s'affranchir des latences d'accès aux instructions de l'application à charger et à décoder, il a été mis en place une mémoire de programme, fixée à 100 KO, dans laquelle sont contenues ces instructions. Dans une implémentation réelle, l'accélérateur accède directement à la mémoire de programme de l'architecture cible. Une autre raison justifiant ce choix est la volonté de considérer l'accélérateur comme un périphérique de type esclave sur le bus de communication AHB du processeur, facilitant ainsi la gestion des requêtes à son niveau et la communication avec le LEON 3, seul maître sur le bus. Ce mode de réalisation est impossible en cas d'accès par l'accélérateur à une mémoire externe via le bus (celui-ci ne pouvant émettre de requêtes), sauf en multipliant les requêtes par l'intermédiaire du processeur, alourdissant les communications entre blocs.

La structure de l'implémentation proposée pour l'accélérateur est présentée sur la figure 6.6. On y retrouve les différentes unités présentées dans la description du concept d'accélérateur dans la première partie de ce chapitre : l'unité de chargement, le décodeur et le contrôleur. L'ensemble des signaux de communication entre les différentes unités, ainsi que l'interfaçage avec le bus AMBA AHB y sont présentés. Toutefois, nous n'en détaillons pas le fonctionnement dans le cadre de ce manuscrit, l'ensemble de ces travaux ayant été réalisés par Florent Berthier et figurant dans son rapport de stage.

La gestion du chargement de l'ensemble des instructions ARM et SPARC a été implémentée, avec notamment la reconnaissance des branchements (unité de pré-chargement) pour les deux jeux d'instructions et la gestion de l'alignement pour Thumb2 (unité de chargement). Concernant le décodeur, celui du jeu d'instructions Thumb2 n'a pas encore été implémenté au moment de la rédaction de ce manuscrit. En revanche, celui du jeu d'instructions Sparcv8 est complètement fonctionnel. Il a été décidé dans le cadre de ce stage que le résultat de chacune des instructions chargée et décodée serait envoyé en mémoire sous la forme d'un mot de 128 bits contenant :

- l'instruction sur 32 bits (ou sur les 16 bits de poids faibles pour les instructions Thumb2 16 bits) ;
- l'adresse de l'instruction, également sur 32 bits ;
- un bit de stop dans le cas d'une détection de branchement indirect (non géré) ;
- un bit de branchement pour signifier la présence d'un branchement direct (géré) ;
- un bit sur la taille de l'instruction (16 ou 32 bits) ;
- l'encodage de l'instruction sur 6 bits (position de l'instruction concernée dans la table des instructions du programme de traduction dynamique binaire) ;
- un champs vide de 55 bits pouvant contenir des informations complémentaires (par exemple les éventuelles conditions de branchement).

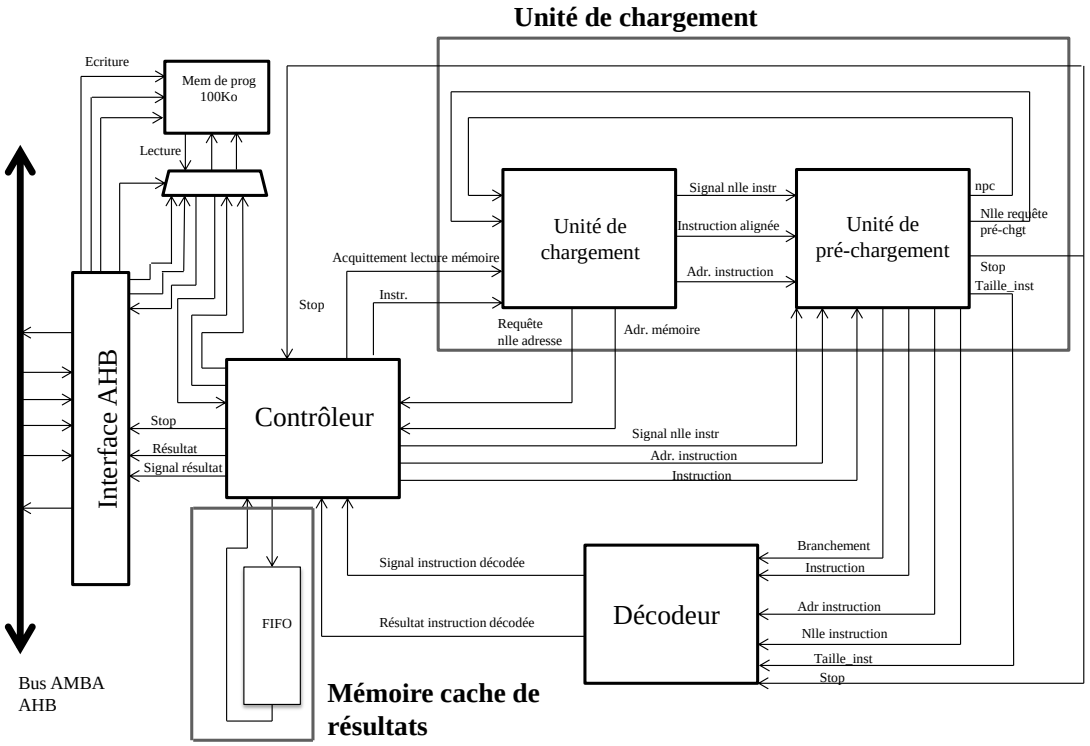



FIGURE 6.6 – Structure de l'implémentation proposée par Florent Berthier. L'ensemble des blocs préalablement présentés y apparaissent. La séparation en deux sous-unités de l'unité de chargement y est également présentée.

### 6.4.4 Premiers résultats expérimentaux

L'implémentation proposée, dont le fonctionnement a été validé par simulation, a ensuite été associée au LEON 3, selon les choix technologiques préalablement présentés. L'ensemble a été simulé avant d'être testé sur FPGA. Le programme de traduction dynamique binaire a été modifié afin de remplacer les implémentations logicielles des fonctions de chargement et de décodage par l'accélérateur. Pour cela, une API (*Application Programming Interface*) a été mise en place pour faciliter son utilisation grâce à l'introduction de fonctions dédiées à la réalisation des différents appels nécessaires à son fonctionnement. Les temps d'accès à l'accélérateur, ainsi que les temps de réalisation des différentes étapes au sein de celui-ci, sont donnés dans le tableau 6.1, en nombre de cycles (résultats obtenus par simulation de l'ensemble).

TABLE 6.1 – Temps d'accès à l'accélérateur et temps de réalisation des différentes étapes au sein de celui-ci. Les résultats sont donnés en nombre de cycles.

| | Opération réalisée | Nombre de cycles |
|---|---|---|
| Accélérateur | Traitement unité de chargement | 8 |
| | Traitement unité de pré-chargement | 2 |
| | Traitement décodeur | 2 |
| | Ecriture résultat FIFO | 3 |
| Processeur | Récupération résultat dans FIFO | 11 |
| | Envoie nouvelle requête de traitement | 2 |

Les évaluations en performances ont été mesurées grâce à l'utilisation des compteurs matériels à disposition sur le processeur LEON 3. Ces évaluations ont été effectuées en réalisant l'optimisation dynamique binaire, via notre programme modifié pour bénéficier de l'accélérateur, d'un petit programme compilé en code machine SPARC et réalisant au sein de boucles des calculs de types multiplications et additions de paramètres. Les résultats obtenus sont donnés dans le tableau 6.2.

TABLE 6.2 – Résultats en performances obtenus pour l'exécution du programme d'optimisation dynamique binaire, avec et sans l'accélérateur. Ces résultats sont donnés en nombre de cycles et en pourcentages relatifs (entre version accélérée et version non accélérée).

| | Nombre cycles total | Nombre cycles chargement et décodage |
|---|---|---|
| Sans accélération | 487089 | 12745 |
| Avec accélération | 471387 | 11084 |
| % réduction temps exécution | 3,2 % | 13 % |

L'observation des résultats permet de mettre en évidence une réduction de 13 % du temps d'exécution des phases de chargement et de décodage par l'utilisation de notre accélérateur. L'impact sur le temps total d'exécution engendre une baise de 3.2 % de celui-ci.

Ces résultats, **bien en deçà des prévisions**, ont conduit à la mesure du pourcentage des temps d'exécution de l'application relatifs à la gestion des étapes de chargement et de décodage. Alors que celles-ci représentent environ 20 % du temps d'exécution pour le compilateur LLC, elles ne représentent que 2.6 % de celui du programme de traduction binaire testé, sans accélération. Cette faible proportion s'explique par la relative simplicité du traducteur, qui ne réalise qu'une seule phase d'optimisation et de traduction, et la faiblesse du nombre d'instructions du programme compilé par le traducteur. Avec accélération, ce chiffre tombe à 2.3 %, ce qui représente un gain théorique de 0.3 % sur l'application totale.

On constate donc **une incohérence** entre l'accélération obtenue sur les étapes de chargement et de décodage, celle obtenue sur l'ensemble de l'application, et le pourcentage relatif de temps passé dans cette gestion par rapport au temps total d'exécution de l'application.

Afin d'évaluer les raisons de cette incohérence, il a été décidé **de mesurer l'impact de l'accélérateur sur l'évolution des erreurs d'accès aux caches du processeur**. Pour cela, nous avons mesuré l'évolution des temps d'exécution de l'application de traduction dynamique binaire en doublant la taille des caches de données et d'instructions.

**Les résultats obtenus** pour un doublement du cache de données (de 8 KO à 16 KO), montrent un gain quasiment nul sur le temps total d'exécution de l'application (0.17 %). Concernant le doublement du cache d'instructions (de 16 KO à 32 KO), il est possible de mesurer un gain significatif sur l'application, de 6.1 %. Ces résultats s'expliquent par l'impact de l'application sur le cache d'instructions du processeur : **le décodage logiciel initialement réalisé par le programme est très lourd et engendre une pollution du cache d'instructions**. Ceci est à mettre en relation avec les phénomènes mis en avant au chapitre 3 au niveau de l'augmentation des erreurs de lecture sur le cache d'instructions pour les codes de compilation dynamique. Nous supposons que l'accélérateur, grâce à la réalisation matérielle du chargement et du décodage en avance de phase des instructions à compiler, **limite ces erreurs** et par conséquent leur impact sur les performances. Ce phénomène constitue une piste

d'optimisation supplémentaire offerte par l'accélérateur, et une explication potentielle aux incohérences obtenues au niveau des gains.

Outre cet effet de bord intéressant mais conduisant toutefois à un gain limité sur l'ensemble de l'application (2.9 % sur les 3.2 % de gain obtenu), c'est surtout la faible perspective d'accélération brute des étapes de chargement et de décodage (13 %) qui limite l'attractivité de la solution telle qu'implémentée. La partie dédiée à la conclusion et aux perspectives de ce chapitre évoque les pistes envisagées afin d'accroître ce gain. Toutefois, nous pensons qu'il est difficile d'envisager l'obtention de gains suffisamment notables sur ces étapes pour justifier une implémentation réelle de l'accélérateur.

Les chiffres relatifs à la surface silicium de la solution ont été mesurés par synthèse en technologie TSMC 40 nm faible consommation (*Low-Power*), identique à celle utilisée pour le premier accélérateur. Ces chiffres portent sur une implémentation de l'accélérateur pouvant réaliser le chargement et le décodage des instructions Sparv8 (le Thumb2 n'est pas considéré car non pleinement fonctionnel au moment de la réalisation de ces mesures). L'interfaçage avec le bus AMBA n'est pas considéré dans ces chiffres pour des questions de commodité, tout comme la mémoire de programme de 100 KO, qui ne sera pas présente dans une implémentation réelle.

La surface totale de l'accélérateur mesurée est de 9470 $\mu m^2$, soit 14765 portes (densité en portes NAND de 1.41 porte/$\mu m^2$ et de 1.67 porte/$\mu m^2$ en bascule flip-flop). Cette surface représente 37 % de la surface du processeur LEON 3 dans sa configuration minimale (environ 40000 portes). Si on met en regard ces chiffres de surface avec ceux du premier accélérateur présenté (7290 $\mu m^2$), il est possible de voir que ceux-ci restent tout à fait raisonnables au niveau du surcoût silicium. Toutefois, afin de pouvoir évaluer la surface réelle de la solution mise en place, il est nécessaire d'évaluer le surcoût silicium induit par l'ajout de la gestion du Thumb2 au sein de l'accélérateur, ce qui devrait être réalisé d'ici la fin du stage. Une importante limitation de cette solution se trouve au niveau de sa fréquence maximale : 244.5 MHz. Face aux fréquences de fonctionnement standards des processeurs embarqués (de l'ordre de 600 MHz), elle constitue un facteur limitant (chemin critique de l'ensemble) et illustre la nécessité d'envisager d'importantes optimisations, que nous détaillons en conclusion de ce chapitre.

Concernant la consommation de l'accélérateur, les résultats ne sont pas encore connus au moment de la rédaction de ce manuscrit.

## 6.5 Conclusion et perspectives

Nous avons présenté, dans ce chapitre, une proposition de mise en place **d'un accélérateur de la gestion du graphe des instructions de l'application à compiler**. Cette étude ayant fait l'objet d'un dépôt de brevet, nous avons tout d'abord présenté le concept tel qu'imaginé lors de ce dépôt. Nous avons pour cela analysé la structure classique d'un traducteur dynamique binaire. Cette étude a permis de mettre en avant la présence d'une machine au sein du traducteur, en charge des étapes de chargement, de décodage et de traduction des instructions de l'application à traiter.

L'accélérateur proposé se présente sous la forme d'un co-processeur réalisant les étapes de chargement et de décodage grâce à un circuit dédié. Le processeur n'est plus alors en charge que des étapes de traduction. Ce choix de découpage est lié à la généricité de ces deux premières étapes, alors que la partie traduction nécessite une importante flexibilité (adaptation à la cible notamment). Nous mettons ainsi en place un système de compilation dynamique, tel que présenté au chapitre 4, associant performances (via l'accélérateur) et flexibilité (via le coeur programmable).

Cette proposition d'accélération est constituée d'une unité de parcours de graphe du flot d'instructions à compiler et d'une mémoire cache de sauvegarde des résultats.

L'unité de parcours se compose de trois blocs distincts : un contrôleur, une unité de chargement et une unité de décodage. L'accélération réalisée est double : elle provient non seulement du gain établi par la mise en place d'un circuit dédié à la réalisation de ces étapes, mais aussi de la mise en place d'un fonctionnement en parallèle de l'accélérateur avec le processeur. Ainsi, pendant que l'accélérateur charge et décode la portion n+1, le processeur traduit la portion n.

En deuxième partie de ce chapitre, nous avons présenté une étude d'implémentation de cette solution, conduite par Florent Berthier au cours d'un stage de fin d'étude. Cette implémentation a été réalisée dans le cadre de la traduction dynamique binaire. Pour des soucis d'accessibilité aux ressources liées à l'implémentation, l'accélérateur a été couplé à un processeur SPARC LEON 3.

Les résultats obtenus pour l'implémentation, à partir d'un programme de traduction dynamique binaire réalisant des optimisations de déroulage de boucle sur les applications compilées, montrent un gain brut de 13 % sur les étapes de chargement et de décodage. Ce gain est bien en deçà des valeurs attendues et permettant de justifier la mise en place de matériel dédié pour ce type de traitement. Un effet de bord de cette accélération a également été mis en avant, avec **une influence notable de l'accélérateur au niveau de la réduction de la pollution du cache d'instructions**. Cet effet de bord remarquable s'explique par le déport matériel de phases de chargement et de décodage logicielles complexes, permettant d'en réduire l'impact sur ce cache. Il constitue une piste d'optimisation supplémentaire offerte par l'accélérateur, à mettre en relation avec les résultats précédemment mis en avant dans le manuscrit (Chapitre 3). Toutefois, les gains obtenus au niveau de l'application sont limités, ceux-ci ne représentant que 2.9 % de réduction du temps total d'exécution de l'application. Les étapes de chargement et de décodage ne représentant, quant à elles, que 2.6 % du temps d'exécution, le gain obtenu en ce qui les concerne sur le temps total d'exécution de l'application n'est donc que de 0.3 %.

Concernant les perspectives ouvertes par cette proposition, un certain nombre de pistes sont envisagées. Elles concernent tout d'abord la réduction du nombre de cycles de chargement d'une instruction, se faisant à l'heure actuelle sur huit cycles, en optimisant les opérations réalisées par l'unité de chargement. Une autre optimisation envisagée est de réaliser l'écriture en mémoire FIFO directement à partir du décodeur, comme proposée initialement, et non en repassant par le contrôleur comme dans l'implémentation actuelle. Enfin, la piste la plus sérieusement envisagée, mais également la plus complexe à mettre en place, concerne la parallélisation de l'ensemble des opérations réalisées à l'intérieure de l'accélérateur, par la mise en place d'un **pipeline d'opérations entre les différentes unités** le composant.

# Conclusion et perspectives

La compilation dynamique est une solution de plus en plus adoptée pour faire face au dynamisme croissant des applications et à l'évolution de la complexité des architectures - et par conséquent aux besoins croissants en termes de déploiement d'application et de virtualisation. Initialement focalisées sur les architectures de types station de travail ou serveur, ces problématiques sont depuis une dizaine d'années apparues sur les architectures embarquées, avec une complexification significative des applications qui y sont portées et de l'hétérogénéité de ces architectures. La compilation dynamique s'est donc répandue sur ce type d'architectures depuis le début des années 2000. Toutefois, la complexité des codes de compilation dynamique a engendré d'importants problèmes de passage à l'échelle sur ces architectures qui ne disposent pas de mécanismes aussi sophistiqués et de tailles de caches aussi importantes que ceux des architectures généralistes pour les gérer.

De nombreuses études ont déjà été réalisées sur la question de l'optimisation de ces codes de compilation sur les architectures embarquées. Deux catégories principales d'optimisation ont été mises en avant : **les optimisations logicielles au niveau des cadriciels et les optimisations au niveau système**. Ces dernières, qu'elles soient basées sur l'utilisation de ressources spécialisées ou standards, offrent de grandes perspectives en termes de gains pour la gestion de ces codes de compilation dynamique. Toutefois, elles souffrent chacune d'un défaut majeur.

Les ressources spécialisées maximisent les performances au détriment de la flexibilité. Nous avons notamment donné l'exemple des processeurs et extensions Java, comme Jazelle, dont le manque de flexibilité a fortement limité l'essor. La mise en place de ressources standardisées fait, quant à elle, apparaître les limites en capacité de gestion par ces ressources de la complexité inhérente aux codes de compilation dynamique, limitant les gains obtenus.

L'objectif de ces trois années de thèse a été de proposer une solution intermédiaire à ces deux approches d'optimisation des ressources dédiées en proposant **la mise en place d'accélérations matérielles couplées à un processeur programmable** en charge de la compilation dynamique, dans l'optique d'offrir à la fois performances et flexibilité au système.

## Synthèse des travaux

Les travaux réalisés dans le cadre de cette thèse ont permis d'aboutir aux **cinq résultats notables** suivants :

- une caractérisation des technologies de compilation dynamique, aux niveaux algorithmique et architectural, avec, entre autres, une étude comportementale au niveau instruction des codes associés ;
- la mise en évidence d'une irrégularité commune au niveau des flots de données et de contrôle entre ces technologies, en particulier dans un contexte embarqué ;

- la démonstration de la possibilité d'optimiser ces codes à l'aide d'opérateurs simples ;
- l'identification de portions critiques de code à l'origine de cette irrégularité, sur lesquelles il est possible de proposer des accélérations matérielles simples ;
- la proposition de deux accélérateurs matériels pour ces portions critiques.

**La caractérisation des technologies de compilation dynamique** a été réalisée sur la base d'un état de l'art et d'un état de la technique de l'ensemble des technologies de compilation dynamique. Quatre technologies ont pu être ainsi mises en avant : les machines virtuelles, la traduction et l'optimisation dynamique binaire, la compilation multi-étages et la compilation dynamique pour les langages typés dynamiquement. L'analyse des spécificités de ces différentes technologies a permis de mettre en avant des caractéristiques communes entre elles, notamment au niveau de la représentation intermédiaire et de sa gestion, amenant à la possibilité de considérer la compilation dynamique comme un domaine algorithmique à part entière. Une synthèse des gains potentiellement atteignables par l'utilisation de la compilation dynamique a été réalisée afin de mettre en avant l'intérêt de son utilisation. Une ouverture vers le contexte embarqué, sur lequel se focalise plus spécifiquement cette étude, a également été présentée.

En se basant sur ces conclusions, une analyse à grain fin (au niveau instruction) des codes de compilation dynamique a été réalisée dans le chapitre 3. Cette analyse a permis de mettre en évidence une **irrégularité commune des codes de compilation dynamique**, en comparaison avec des codes dits conventionnels, au niveau de la gestion des flots de contrôle et de données. Elle a également permis de mettre en avant la possibilité **d'optimiser ces codes de compilation à l'aide d'opérateurs simples**, focalisés sur la gestion des structures récursives et chaînées, afin de compenser la simplicité des mécanismes de prédiction et les faibles capacités des caches des processeurs embarqués. Ces travaux ont fait l'objet d'une publication au Workshop DCE '13 [116].

Suite aux conclusions de l'analyse, une identification des portions critiques des codes de compilation dynamique à l'origine de cette irrégularité a été réalisée. Elle a permis de mettre en avant trois points critiques : **la gestion des tableaux associatifs, l'allocation dynamique de la mémoire, et la gestion du graphe des instructions de l'application à compiler** (au niveau représentation intermédiaire et code machine). Une analyse de l'état de l'art a été réalisée dans l'optique d'analyser les accélérations déjà envisagées au niveau des systèmes embarqués pour les codes de compilation dynamique, aussi bien d'un point de vue logiciel que matériel. Il en est ressorti que la solution visant à mettre en place des ressources dédiées pour les codes de compilation dynamique (et plus globalement pour les services de virtualisation), sur la base de l'utilisation de processeurs embarqués aux capacités de calcul contraintes, apparaît comme offrant le meilleur compromis entre performances et flexibilité.

C'est en se basant sur cette approche d'utilisation de ressources aux capacités limitées et dédiées à la compilation dynamique, que nous avons proposé l'ajout d'optimisations matérielles en vue de compenser les pertes en performances engendrées par l'irrégularité des codes de compilation dynamique. **Deux accélérateurs** ont ainsi été proposés dans le cadre de ces travaux.

Le premier porte sur **la gestion des tableaux associatifs et de l'allocation dynamique de la mémoire**. Cet accélérateur est implémenté sous la forme d'une unité fonctionnelle de processeur. Nos études ont permis de mettre en avant que LLC dispose déjà d'un très grand nombre d'optimisations logicielles en ce qui concerne ces deux points critiques. Pour des soucis de portabilité de la solution mise en place, il a été décidé de se baser sur la seule utilisation de la librairie standard STL C++ et de l'allocateur C Doug Lea. Nous avons donc opéré une normalisation du compilateur

LLC de LLVM afin que celui-ci ne fasse appel qu'à ces librairies, et non à ses différentes optimisations logicielles. Cette approche engendre irrémédiablement une perte en performances, largement compensée par l'accélération mise en place. La gestion de l'allocation dynamique de la mémoire se faisant à l'aide de tableaux associatifs, nous avons proposé une uniformisation de leur implémentation au sein des deux librairies considérées, se basant sur l'utilisation des arbres rouges et noirs.

**Une accélération matérielle de la gestion de ces arbres rouges et noirs** a ainsi été mise en place, grâce à l'incorporation d'une unité fonctionnelle permettant de réaliser sur un circuit dédié les différentes fonctions relatives à cette gestion, et couplée à un processeur ARM Cortex-A5. L'accélérateur, dont la surface totale ne représente que 1.4 % de celle du Cortex-A5, permet l'obtention de gains moyens de 15 % sur l'exécution de LLC. Il permet plus spécifiquement une multiplication par 5 des performances relatives à la gestion des tableaux associatifs par la librairie STL C++ et à l'allocation dynamique de la mémoire par l'allocateur C Doug Lea. Ces résultats, obtenus sur une version se basant uniquement sur les librairies standards, permettent d'envisager son utilisation dans tous les domaines applicatifs utilisant massivement les tableaux associatifs et l'allocation dynamique de la mémoire. Ces travaux ont fait l'objet d'un dépôt de brevet et d'une publication à la conférence ASAP '13 [131].

**Le second accélérateur porte sur la gestion du graphe des instructions de l'application à compiler**, au niveau de sa représentation intermédiaire et de son code machine. Il permet le déport sur un circuit dédié des phases génériques de chargement et de décodage de ces instructions. L'accélérateur, conçu sous la forme d'une unité co-processeur, est couplé au processeur en charge de la compilation dynamique sur lequel sont exécutées les étapes de traduction. Cet ensemble a fait l'objet d'un dépôt de brevet. Une implémentation de cet accélérateur sur une application de traduction dynamique binaire a été réalisée dans le cadre d'un stage de fin d'études. Les gains bruts obtenus sur les étapes de chargement et de décodage restent limités (de l'ordre de 13 %). Toutefois, nous avons pu constater un effet positif du déport de ces étapes sur un circuit dédié au niveau du cache d'instructions, limitant la pollution de celui-ci et impactant positivement le gain global obtenu sur l'application (passant de 0.3 %, grâce à l'accélération brute des deux étapes concernées, à 3.2 %).

Voici une estimation du volume de travail engendré par la réalisation de chacune des étapes marquantes de ces trois années de thèse :

- réalisation de l'état de l'art : 10 mois (7 premiers mois de thèse et 3 mois additionnels répartis durant la réalisation des travaux) ;
- caractérisation des codes de compilation dynamique : 5 mois ;
- normalisation du compilateur LLC du cadriciel LLVM : 5 mois
- conception, développement et évaluation du premier accélérateur : 5 mois
- conception du second accélérateur : 2 mois
- valorisation des travaux réalisés : 6 mois (conférences, séminaires, workshop, école d'été, dépôts de brevet, rédaction d'articles, évaluation à mi-thèse).

A cela s'ajoute bien évidemment les développements annexes ayant permis d'aboutir à ces grandes étapes, les périodes de réflexion et d'analyse des différents travaux réalisés et, bien entendu, la période de rédaction du manuscrit de thèse.

## Discussion

Les résultats mis en évidence dans le cadre de ces travaux démontrent tout d'abord que les opportunités d'optimisation de la compilation dynamique, par le biais de circuits dédiés, existent. Alors que la forte complexité des codes associés rend difficilement concevable leur transfert complet sur un circuit dédié, nous avons démontré

**la possibilité d'envisager le couplage d'un processeur programmable à de multiples accélérations matérielles**, sous la forme d'unités fonctionnelles et/ou de co-processeurs.

L'état de l'art et la caractérisation réalisés au sein de cette étude ont permis d'aboutir à la conclusion que la compilation dynamique peut être considérée comme un domaine algorithmique à part entière. Ils ont aussi permis de mettre en avant la présence de caractéristiques comportementales communes au niveau des codes de compilation dynamique, engendrant notamment une irrégularité au niveau des flots de données et de contrôle. Aussi, nous avons démontré **l'existence de problématiques communes** entre des technologies destinées à des domaines applicatifs très différents (machines virtuelles, traduction binaire, compilation multi-étages, moteurs d'exécution). Les propositions d'optimisation ainsi mises en place dans le cadre de ces travaux peuvent bénéficier à l'ensemble des technologies identifiées, puisque se focalisant sur ces problématiques communes.

### Couplage des accélérateurs

Les deux accélérateurs proposés, le premier sous la forme d'une unité fonctionnelle et le second sous la forme d'une unité co-processeur, couplés à un processeur programmable, visent à accroître les efficacités énergétique et surfacique de l'ensemble formant un système de compilation dynamique. Ce système est ensuite couplé à la cible pour laquelle la compilation dynamique est réalisée, par exemple une architecture embarquée multi-coeurs hétérogène.

Une autre implémentation envisageable est **le couplage direct** de nos deux propositions d'accélération aux ressources de l'architecture cible. Cette approche présente l'avantage de limiter le surcoût silicium induit, puisqu'il n'est pas nécessaire d'implémenter l'ensemble du système. Toutefois, elle présente le défaut majeur de monopoliser une ressource dédiée normalement à l'exécution de l'application, réduisant sa disponibilité pour cette dernière. Elle engendre également un arbitrage à réaliser sur le choix de la ressource à laquelle seront couplées les accélérations proposées.

### Accélérateur des arbres rouges et noir

Pour ce premier accélérateur mis en place, se focalisant sur la gestion des arbres rouges et noirs, nous motivons notre choix de le baser sur les librairies standards afin d'en **faciliter la réutilisation**. Nous pensons que le panel applicatif pouvant en bénéficier est bien plus large que celui de la compilation dynamique. Les tableaux associatifs et l'allocation dynamique de la mémoire sont aujourd'hui très utilisés pour la gestion des structures de données complexes et pour faire face au dynamisme croissant des applications.

Le gain total obtenu sur le compilateur LLC de LLVM ne représente qu'une accélération moyenne de 15 % de ce compilateur. **Ce chiffre peut paraître faible** en regard des ordres de grandeurs constatés pour les applications traditionnellement accélérées (comme dans le domaine du traitement d'image). Même si ces applications connaissent, ces dernières années, un accroissement de leur complexité, et notamment des structures de données qu'elles manipulent, la régularité de leur flot de données et le faible volume d'instructions nécessaire à la manipulation de ces données permettent une optimisation significative de leurs performances (de un à plusieurs ordres de grandeurs). Leur régularité facilite de plus leur mise en circuit (utilisation de ressources SIMD par exemple).

La forte irrégularité des algorithmes de compilation dynamique mise en avant dans le cadre de nos travaux rend peu triviale cette mise en circuit et limite l'ordre de grandeur

des gains envisageables. L'obtention de 15 % de gain avec une augmentation de 1.4 % de l'encombrement silicium apparaît déjà, selon nous, comme un résultat important.

### Accélérateur du graphe des instructions

En ce qui concerne ce second accélérateur proposé, nous avons pu mettre en avant la limite des gains obtenus sur les étapes accélérées de chargement et de décodage. Outre d'importantes opportunités d'optimisation au niveau de l'accélérateur, nous estimons que **le manque de généricité** dans son implémentation limite les bénéfices qu'il est possible d'en tirer. Nous pensons notamment à l'impact positif constaté sur le cache d'instructions, avec une diminution significative de sa pollution, grâce au transfert de la complexité d'une partie des codes de compilation dynamique sur un circuit dédié.

Un accroissement de la généricité de cet accélérateur permettrait de bénéficier, selon nous, de gains plus importants, grâce à un déport accru des phases complexes des codes de compilation dynamique sur un circuit dédié. L'idée n'est pas d'accélérer drastiquement ces phases grâce au circuit dédié, mais plutôt de diminuer la complexité de la partie logicielle laissée au processeur associé. Un autre élément, à l'impact significatif selon nous, et qui pourrait être envisagé avec cet accroissement de généricité, est la mise en place d'une intégration plus forte entre le coeur et l'accélérateur. Une approche type unité fonctionnelle, avec la mise en place d'instructions spécialisées pour les opérations déportées sur le circuit dédié, est possible.

## Perspectives

Les travaux réalisés durant ces trois années de thèse ont permis l'apport de contributions significatives sur la mise en place **d'accélérations matérielles pour la compilation dynamique** dans le but d'en accroître l'attractivité sur **les systèmes embarqués**. Nous pensons que les concepts développés durant ces travaux ouvrent de larges possibilités quant à l'utilisation des différentes technologies identifiées sur ces systèmes. De nombreuses perspectives ont été identifiées afin d'assurer la continuité de ces travaux.

### Poursuite de nos travaux

La première étape concerne tout d'abord la finalisation de l'implémentation fonctionnelle de l'accélérateur des arbres rouges et noirs. Conçue sous la forme d'une unité fonctionnelle, cette solution nécessite des modifications architecturales significatives du processeur auquel elle est couplée, ainsi qu'une légère évolution des chaînes de compilation et de débogage. Toutefois, les perspectives offertes par cette solution nous encourage fortement à poursuivre dans cette voie. Une estimation de consommation, permettant de valider nos déductions sur le sujet, doit également être réalisée à partir de cette implémentation.

Concernant l'accélérateur des étapes de chargement et de décodage, l'essentiel des perspectives concerne l'optimisation de l'implémentation réalisée, afin d'en accroître les performances. Une étape envisagée est la parallélisation des différentes opérations réalisées au sein de l'accélération, en mettant en place une structure de type pipeline. Des recherches sont également à planifier quant à l'accroissement de sa généricité, afin de bénéficier plus largement du déport sur un circuit dédié des phases complexes des codes de compilation dynamique. Enfin, il sera nécessaire de réaliser le portage du compilateur LLC de LLVM sur le système mis en place, afin que ce dernier puisse en bénéficier.

Basé sur ces réalisations, l'objectif sera **de mettre en place un système complet, fonctionnel et optimisé pour la compilation dynamique** en cadre applicatif réel. En se focalisant sur le compilateur LLC, nous souhaitons réaliser une chaîne de compilation dynamique complètement fonctionnelle et procéder à l'inclusion de la gestion des phases de compilation au niveau du système d'exécution de la cible. Nous envisageons d'intégrer cet ensemble dans le cadre d'une couche de virtualisation pour le déploiement d'applications sur architecture multi-coeurs hétérogène embarquée.

### Futur de la compilation dynamique

Notre retour d'expérience sur le domaine de la compilation dynamique nous laisse à penser que son emploi dans cette optique de virtualisation constitue **sa principale perspective d'évolution** dans les années à venir au niveau des systèmes embarqués. Il est d'ailleurs possible de voir une forte activité à ce niveau depuis le démarrage de ces travaux en 2010, avec le développement, par exemple, de nouvelles solutions de portage d'OpenCL (avec entre autres les travaux de ARM sur le sujet) et d'accroissement du support de ce dernier (NVidia, AMD, etc.). La contrainte actuelle de réaliser ce déploiement par l'utilisation d'un processeur hôte généraliste sur lequel s'exécute l'outil de déploiement rend cette opération très lourde et ne permet pas de bénéficier pleinement des avantages d'un déploiement dynamique et automatique.

L'introduction d'une couche de virtualisation permettant d'optimiser ce déploiement directement sur les architectures concernées est, selon nous, un élément clé de la résolution des problématiques qui y sont liées. **LLVM s'est imposé comme une référence** dans ce domaine de par ses performances, sa faible empreinte mémoire et sa portabilité. Le nombre de projets se basant sur l'utilisation de son générateur de code est en constante augmentation, aussi bien dans la recherche industrielle que dans le monde académique. Les performances de ce compilateur sont aujourd'hui équivalentes, quant au niveau d'optimisation et de la vitesse de compilation, à celles obtenus avec GCC. Son support a également été étendu à de nombreuses cibles, allant des processeurs programmables (ARM) aux processeurs graphiques (NVidia). Cela rend possible la gestion par LLVM d'architectures aux jeux d'instructions multiples. Notre étude s'intègre pleinement dans le cadre de ces travaux, et nous pensons qu'une continuité de nos recherches sur le sujet pourrait contribuer fortement à l'essor de ces couches de virtualisation, en accroissant significativement leurs performances.

Concernant l'utilisation de la compilation dynamique dans une optique d'obtention de performances brutes, nous pensons que les approches offrant une forte portabilité et un haut niveau d'abstraction aux utilisateurs, comme LLVM, ne constituent pas une solution d'avenir. Son utilisation dans de nombreux projets visant à accroître l'attractivité des solutions de virtualisation, tend à complexifier significativement sa structure. Nous pensons qu'une approche comme celle avancée par Henri-Pierre Charles avec **l'introduction des "compilettes"** [62], permet l'obtention de gains bien plus significatifs grâce au panel d'optimisations très spécifiques qu'elle offre à l'exécution. Ainsi, ces optimisations à grain fin, très proches et par conséquent très dépendantes de la cible, constituent un champ d'étude prometteur à ce niveau.

L'optimisation des codes de compilation dynamique à l'aide de propositions matérielles constitue selon nous une piste d'exploration fortement prometteuse si elle est suivie dans une optique similaire à celle présentée dans ces travaux et par Ting Cao. Nous avons pu constater le faible succès de la mise en place de solutions complètement dédiées comme les processeur Java, dont seul Jazelle subsiste aujourd'hui. La généricité et la réutilisabilité des solutions, afin d'en assurer **la rentabilité**, est la clé de leur succès, tant les architectures évoluent rapidement aujourd'hui et tant les processus de conception de circuits dédiés sont coûteux. Les possibilités d'optimisations logicielles

sont encore nombreuses, mais les gains qu'elles permettront d'enregistrer seront de moins en moins probants au fur et à mesure de la complexification des codes qu'elles engendreront. De plus, elles ne permettront pas de résoudre les problèmes relatifs à la simplicité des mécanismes de prédiction et à la limitation des tailles de caches des processeurs embarqués dans l'optique de la gestion de ces codes.

### Elargissement des applications de nos travaux

A plus long terme, nous estimons que les solutions matérielles proposées dans le cadre de ces travaux pourraient bénéficier à un plus large panel de domaines applicatifs que ceux relatifs à la compilation dynamique. L'adressage des problématiques liées aux codes présentant des irrégularités au niveau de leurs flots de données et de contrôle est aujourd'hui un enjeu majeur pour beaucoup de développeurs. Nous avons déjà notamment cité le cas des applications de traitement d'image, fortement orientées vers de l'analyse de contenu, avec une forte complexification des opérations réalisées et des données manipulées au sein de celles-ci. La réalisation d'une telle ouverture permettrait de justifier encore davantage la réalisation en circuit de nos propositions, dans l'optique, par exemple, d'une implémentation similaire à celle de Jazelle.

# Bibliographie


[1] ARM. ARM big.LITTLE processing. http://www.arm.com/products/processors/technologies/biglittleprocessing.php. [Online, July 2013].

[2] Texas Instruments. Omap5 Platform. http://www.ti.com/general/docs/wtbu/wtbuproductcontent.tsp?templateId=6123&navigationId=12862&contentId=101230. [Online, July 2013].

[3] Qualcomm. Snapdragon S4. http://www.qualcomm.com.au/products/snapdragon. [Online, July 2013].

[4] Tom Van Vleck. The IBM 360/67 and CP/CMS. http://www.multicians.org/thvv/360-67.html. [Online, July 2013].

[5] SUN Hotspot VM website. http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.html. [Online, July 2013].

[6] Ben Cheng and Bill Buzbee. A JIT Compiler for Android's Dalvik VM. http://www.google.com/events/io/2010/sessions/jit-compiler-androids-dalvik-vm.html, May 2010. [Online, July 2013].

[7] International Organization for Standardization and International Electrotechnical Commission. International Standard ISO/IEC 23271:2006 - Common Language Infrastructure (CLI), Partitions I to VI, 2006. 2nd edition.

[8] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[9] Google V8 JavaScript Engine. http://code.google.com/p/v8/. [Online, July 2013].

[10] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. SPUR: a trace-based jit compiler for CIL. *SIGPLAN Not.*, 45:708–725, October 2010.

[11] Armin Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '04, pages 15–26, New York, NY, USA, 2004. ACM.

[12] The Mathworks. Technology backgrounder: Accelerating Matlab. http://www.mathworks.com/company/newsletters/digest/sept02/accel_matlab.pdf, September 2002. [Online, July 2013].

[13] Jose Baiocchi, Bruce R. Childers, Jack W. Davidson, Jason D. Hiser, and Jonathan Misurda. Fragment cache management for dynamic binary translators in embedded systems with scratchpad. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '07, pages 75–84, New York, NY, USA, 2007. ACM.

[14] Kevin Scott and Jack Davidson. STRATA: a software dynamic translation infrastructure. Technical Report CS-2001-17, University of Virginia Charlottesville, VA, USA, 2001.

[15] Llvm/clang 3.2 compiler competing with gcc. http://www.phoronix.com/scan.php?page=article&item=llvm_clang32_final. [Online, July 2013].

[16] Jae-Jin Kim, Seok-Young Lee, Soo-Mook Moon, and Suhyun Kim. Comparison of LLVM and GCC on the ARM platform. In *2010 5th International Conference on Embedded and Multimedia Computing (EMC)*, pages 1–6, 2010.

[17] Nicolas Geoffray, Gaël Thomas, Julia Lawall, Gilles Muller, and Bertil Folliot. VMKit: a substrate for managed runtime environments. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '10, pages 51–62, New York, NY, USA, 2010. ACM.

[18] Pekka Jääskeläinen, Carlos S. de La Lama, Pablo Huerta, and Jarmo Takala. OpenCL-based design methodology for application-specific processors. In Fadi J. Kurdahi and Jarmo Takala, editors, *ICSAMOS*, pages 223–230. IEEE, 2010.

[19] The Portland Group. PGI OpenCL Compiler for ARM. http://www.pgroup.com/products/pgcl.htm. [Online, July 2013].

[20] Chris Lattner. The LLVM compiler cystem. Bossa Conference on Open Source, Mobile Internet and Multimedia, Recife, Brazil, Mar. 2007.

[21] NVidia. NVIDIA's CUDA/OpenCL PTX Back-End In LLVM 3.2. http://www.phoronix.com/scan.php?page=news_item&px=MTI1NDU. [Online, July 2013].

[22] Albert Cohen and Erven Rohou. Processor virtualization and split compilation for heterogeneous multicore embedded systems. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 102–107, New York, NY, USA, 2010. ACM.

[23] ARM Limited Steve Steele, Java Program Manager. White paper: Accelerating to meet the challenge of embedded Java. http://www.arm.com/products/processors/technologies/jazelle.php, november 2001. [Online, July 2013].

[24] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *J. Syst. Archit.*, 54(1-2):265–286, January 2008.

[25] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[26] Richard L. Wexelblat, editor. *History of programming languages I*. ACM, New York, NY, USA, 1981.

[27] GCC. GNU Compiler Collection. http://gcc.gnu.org/. [Online, July 2013].

[28] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3:184–195, April 1960.

[29] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM.

[30] Joe Armstrong. The development of Erlang. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ICFP '97, pages 196–203, New York, NY, USA, 1997. ACM.

[31] Erik Johansson, Mikael Pettersson, and Konstantinos Sagonas. A high performance Erlang system. In *Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '00, pages 32–43, New York, NY, USA, 2000. ACM.

[32] Markus Mock, Craig Chambers, and Susan J. Eggers. Calpa: a tool for automating selective dynamic compilation. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pages 291–302, New York, NY, USA, 2000. ACM.

[33] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 291–300, New York, NY, USA, 1998. ACM.

[34] Karine Brifault and Henri-Pierre Charles. Efficient data driven run-time code generation. In *proceedings of the Seventh Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, LCR '04, pages 1–7, New York, NY, USA, 2004. ACM.

[35] K. Sajjad, S.-M. Tran, D. Barthou, H.-P. Charles, and M. Preda. A global approach for MPEG-4 AVC encoder optimization. In *14th Workshop on Compilers for Parallel Computing*, Zurich, Switzerland, 2009.

[36] Yves Lhuillier and Damien Courousse. Embedded system memory allocator optimization using dynamic code generation. In Henri-Pierre Charles, Philippe Clauss, and Frédéric Pétrot, editors, *Workshop "Dynamic Compilation Everywhere", in conjunction with the 7th HiPEAC conference*, Paris, France, january 2012.

[37] Apple Inc. (Original authors); Khronos Group (Developpers). OpenCL (Open Computing Language). `http://www.khronos.org/opencl/`. [Online, July 2013].

[38] Kazuhiko Komatsu, Katsuto Sato, Yusuke Arai, Kentaro Koyama, Hiroyuki Takizawa, and Hiroaki Kobayashi. Evaluating performance and portability of OpenCL programs. In *The Fifth International Workshop on Automatic Performance Tuning*, June 2010.

[39] J. Aycock. A brief history of Just-In-Time. *ACM Computing Surveys*, 35:97–113, June 2003.

[40] T.Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition edition, 1999.

[41] Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 144–153, New York, NY, USA, 2006. ACM.

[42] Jikes RVM Team. Jikes RVM web site. `http://jikesrvm.org/`. [Online, July 2013].

[43] M. Arnold, S.J. Fink, D. Grove, M. Hind, and P.F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, 2005.

[44] MichałCierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing judo: Java under dynamic optimizations. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 13–26, New York, NY, USA, 2000. ACM.

[45] Nik Shaylor. A just-in-time compiler for memory-constrained low-power devices. In *Proceedings of the 2nd Java™ Virtual Machine Research and Technology Symposium*, pages 119–126, Berkeley, CA, USA, 2002. USENIX Association.

[46] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioglu, and Erik Altman.

LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, PACT '99, pages 128–, Washington, DC, USA, 1999. IEEE Computer Society.

[47] Carmen Badea, Alexandru Nicolau, and Alexander V. Veidenbaum. A simplified Java bytecode compilation system for resource-constrained embedded processors. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '07, pages 218–228, New York, NY, USA, 2007. ACM.

[48] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[49] G. Agosta, S.C. Reghizzi, G. Falauto, and M. Sykora. JIST: just-in-time scheduling translation for parallel processors. In *Parallel and Distributed Computing, 2004. Third International Symposium on/Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, 2004. Third International Workshop on*, pages 122 – 132, 2004.

[50] Simone Campanoni, Giovanni Agosta, and Stefano Crespi Reghizzi. A parallel dynamic compiler for CIL bytecode. *SIGPLAN Not.*, 43:11–20, April 2008. Article en relation avec ILDJIT.

[51] Michal Cierniak, Brian T. Lewis, and James M. Stichnoth. Open runtime platform: flexibility with performance using interfaces. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, JGI '02, pages 156–164, New York, NY, USA, 2002. ACM.

[52] Raymond J. Hookway and Mark A. Herdeg. Digital FX!32: combining emulation and binary translation. *Digital Tech. J.*, 9(1):3–12, January 1997.

[53] Kemal Ebcioğlu and Erik R. Altman. Daisy: dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, pages 26–37, New York, NY, USA, 1997. ACM.

[54] David R. Ditzel. Transmeta's Crusoe: Cool chips for mobile computing. 2000.

[55] Brian Remick. Dynamic compilation: A comparison of the Transmeta Crusoe processor & the daisy architecture. 2001.

[56] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[57] Apple Rosetta web site. `http://www.apple.com/asia/rosetta/`. [Online, July 2013].

[58] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM.

[59] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, VEE '12, pages 133–144, New York, NY, USA, 2012. ACM.

[60] Derek Bruening, Qin Zhao, and Reid Kleckner. Tutorial: Building dynamic instrumentation tools with DynamoRIO. In *Proceedings of the 9th Annual*

*IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages xxi–, Washington, DC, USA, 2011. IEEE Computer Society.

[61] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[62] Henri-Pierre Charles. Basic infrastructure for dynamic code generation. In Henri-Pierre Charles, Philippe Clauss, and Frédéric Pétrot, editors, *Workshop "Dynamic Compilation Everywhere", in conjunction with the 7th HiPEAC conference*, Paris, France, january 2012.

[63] Oscar Almer, Igor Böhm, TobiasEdler Koch, Björn Franke, Stephen Kyle, Volker Seeker, Christopher Thompson, and Nigel Topham. A parallel dynamic binary translator for efficient multi-core simulation. *International Journal of Parallel Programming*, 41(2):212–235, 2013.

[64] Igor Böhm, Björn Franke, and Nigel P. Topham. Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator. In *ICSAMOS'10*, pages 1–10, 2010.

[65] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM.

[66] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. LLVA: A low-level virtual instruction set architecture. In *In MICRO-36*, pages 205–216, 2003.

[67] Libjit web site. `http://www.gnu.org/s/dotgnu/libjit-doc/libjit_1.html`. [Online, July 2013].

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

[69] Massimiliano Poletto, C. Hsieh, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21:21–2, 1999.

[70] Alexandra Jimborean, Philippe Clauss, JuanManuel Martinez, and Aravind Sukumaran-Rajam. Online dynamic dependence analysis for speculative polyhedral parallelization. In Felix Wolf, Bernd Mohr, and Dieter Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 191–202. Springer Berlin Heidelberg, 2013.

[71] Alexandra Jimborean, Matthieu Herrmann, Vincent Loechner, and Philippe Clauss. VMAD: A virtual machine for advanced dynamic analysis of programs. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 125–126, Washington, DC, USA, 2011. IEEE Computer Society.

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

[73] Luciano Ost, Sameer Varyani, Leandro Soares Indrusiak, Marcelo Mandelli, Gabriel Marchesan Almeida, Eduardo Wächter, Fernando Moraes, and Gilles Sassatelli. Enabling adaptive techniques in heterogeneous MPSoCs based on virtualization. *TRETS*, 5(3):17, 2012.

[74] JavaScript: All about Mozilla's JavaScript engine. `https://blog.mozilla.org/javascript/`. [Online, July 2013].

[75] Mason Chang, Edwin Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Tracing for web 3.0: trace compilation for the next generation web applications. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 71–80, New York, NY, USA, 2009. ACM.

[76] Matsano Security. Attacking clientside jit compilers. `http://www.matasano.com/research/jit/`, 2011. [Online, July 2013].

[77] IonMonkey in Firefox 18. `https://blog.mozilla.org/javascript/2012/09/12/ionmonkey-in-firefox-18/`. [Online, July 2013].

[78] Mozilla Developper Network. Nanojit project. `https://developer.mozilla.org/en-US/docs/Nanojit`. [Online, July 2013].

[79] Ana Azevedo, Alex Nicolau, and Joe Hummel. Java annotation-aware just-in-time (AJIT) complilation system. In *Proceedings of the ACM 1999 conference on Java Grande*, JAVA '99, pages 142–151, New York, NY, USA, 1999. ACM.

[80] G. Agosta, S. Crespi Reghizzi, P. Palumbo, and M. Sykora. Selective compilation via fast code analysis and bytecode tracing. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 906–911, New York, NY, USA, 2006. ACM.

[81] Bruce Childers, Jack W. Davidson, and Mary Lou Soffa. Continuous compilation: A new approach to aggressive and adaptive code transformation. In *in Proceedings of the International Parallel and Distributed Processing Symposium*, pages 205–214, 2003.

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

[83] Naveen Kumar, Bruce Childers, and Mary Lou Soffa. Transparent debugging of dynamically optimized code. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 275–286, Washington, DC, USA, 2009. IEEE Computer Society.

[84] Kim Hazelwood and Artur Klauser. A dynamic binary instrumentation engine for the ARM architecture. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 261–270, New York, NY, USA, 2006. ACM.

[85] Mourad Debbabi, Abdelouahed Gherbi, Lamia Ketari, Chamseddine Talhi, Nadia Tawbi, Hamdi Yahyaoui, and Sami Zhioua. A dynamic compiler for embedded Java virtual machines. In *Proceedings of the 3rd international symposium on Principles and practice of programming in Java*, PPPJ '04, pages 100–106. Trinity College Dublin, 2004.

[86] Oracle. The CLDC HotSpot implementation virtual machine. `http://www.oracle.com/technetwork/java/cldc-hi-whitepaper-150012.pdf`. [Online, July 2013].

[87] The brains behind Apple's Rosetta: Transitive. http://news.cnet.com/The-brains-behind-Apples-Rosetta-Transitive/2100-1016_3-5736190.html. [Online, July 2013].

[88] Qiang Wu, V. J. Reddi, Youfeng Wu, Jin Lee, Dan Connors, David Brooks, Margaret Martonosi, and Douglasw. Clark. A dynamic compilation framework for controlling microprocessor energy and performance. In *INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE*, pages 271–282. IEEE Computer Society, 2005.

[89] Prasad Kulkarni, Matthew Arnold, and Michael Hind. Dynamic compilation: the benefits of early investing. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pages 94–104, New York, NY, USA, 2007. ACM.

[90] Kanemitsu Ootsu, Takashi Yokota, Takafumi Ono, and Takanobu Baba. Preliminary evaluation of a binary translation system for multithreaded processors. In *Proceedings of the International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'02)*, IWIA '02, pages 77–, Washington, DC, USA, 2002. IEEE Computer Society.

[91] Simone Campanoni, Martino Sykora, Giovanni Agosta, and Stefano Crespi Reghizzi. Dynamic look ahead compilation: A technique to hide jit compilation latencies in multicore environment. In *Proceedings of the 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, CC '09, pages 220–235, Berlin, Heidelberg, 2009. Springer-Verlag.

[92] Prasad A. Kulkarni and Jay Fuller. Jit compilation policy on single-core and multi-core machines. In *Proceedings of the 2011 15th Workshop on Interaction between Compilers and Computer Architectures*, INTERACT '11, pages 54–62, Washington, DC, USA, 2011. IEEE Computer Society.

[93] Sathyanarayanan Thammanur and Santosh Pande. A fast, memory-efficient register allocation framework for embedded systems. *ACM Trans. Program. Lang. Syst.*, 26(6):938–974, November 2004.

[94] Artun Pietrek, Florent Bouchez, and Benoît Dinechin. Tirex: A textual target-level intermediate representation for compiler exchange. In *WIR '11: International Workshop on Intermediate Representations*, apr 2011.

[95] Julien Le Guen, Christophe Guillon, and Fabrice Rastello. MinIR, a minimalistic intermediate representation. In Florent Bouchez, Sebastian Hack, and Eelco Visser, editors, *Proceedings of the Workshop on Intermediate Representations (WIR)*, pages 5–12, April 2011.

[96] Xamarin. The Mono Project. http://www.mono-project.com. [Online, July 2013].

[97] Pty Ltd Southern Storm Software. Dotgnu project. http://www.gnu.org/software/dotgnu/. [Online, July 2013].

[98] Marco Cornero, Roberto Costa, Ricardo Fernández Pascual, Andrea C. Ornstein, and Erven Rohou. An experimental environment validating the suitability of CLI as an effective deployment format for embedded systems. In *Proceedings of the 3rd international conference on High performance embedded architectures and compilers*, HiPEAC'08, pages 130–144, Berlin, Heidelberg, 2008. Springer-Verlag.

[99] Roberto Costa and Erven Rohou. Comparing the size of .NET applications with native code. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference*

*on Hardware/software codesign and system synthesis*, CODES+ISSS '05, pages 99–104, New York, NY, USA, 2005. ACM.

[100] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of spill everywhere under ssa form. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '07, pages 103–112, New York, NY, USA, 2007. ACM.

[101] Benoit Boissinot, Sebastian Hack, Daniel Grund, Benoît Dupont de Dine hin, and Fabri e Rastello. Fast liveness checking for ssa-form programs. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 35–44, New York, NY, USA, 2008. ACM.

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

[103] Standard Performance Evaluation Corporation (SPEC). SPEC CPU2000. `http://www.spec.org/cpu2000/`. [Online, July 2013].

[104] Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual machine showdown: Stack versus registers. *ACM Trans. Archit. Code Optim.*, 4(4):2:1–2:36, January 2008.

[105] Nicos Christofides. *Graph theory : an algorithmic approach*. Computer science and applied mathematics. Academic press, London, 1975.

[106] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

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

[108] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. *SIGPLAN Not.*, 44(3):253–264, March 2009.

[109] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 81–, Washington, DC, USA, 2003. IEEE Computer Society.

[110] Gernot Heiser. The role of virtualization in embedded systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, IIES '08, pages 11–16, New York, NY, USA, 2008. ACM.

[111] H.P. Charles and K. Sajjad. HPBCG High Performance Binary Code Generator. `http://code.google.com/p/hpbcg/`, 2009. [Online, July 2013].

[112] Yu Sun and Wei Zhang. Efficient code caching to improve performance and energy consumption for Java applications. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '08, pages 119–126, New York, NY, USA, 2008. ACM.

[113] G. Chen, M. Kandemir, N. Vijaykrishnan, and M.J. Irwin. Energy-aware code cache management for memory-constrained Java devices. In *SOC Conference, 2003. Proceedings. IEEE International [Systems-on-Chip]*, pages 179 – 182, sept. 2003.

[114] Minhaj Khan, H. Charles, and D. Barthou. An effective automated approach to specialization of code. In Vikram Adve, María Garzarán, and Paul Petersen, editors, *Languages and Compilers for Parallel Computing*, volume 5234 of *Lecture Notes in Computer Science*, pages 308–322. Springer Berlin / Heidelberg, 2008.

[115] Ting Cao, Stephen M Blackburn, Tiejun Gao, and Kathryn S McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 225–236, Washington, DC, USA, 2012. IEEE Computer Society.

[116] Alexandre Carbon, Yves Lhuillier, and Henri-Pierre Charles. Scaling-down to embedded systems for dynamic compilation. In Henri-Pierre Charles and Bruce Childers, editors, *2nd International Workshop "Dynamic Compilation Everywhere", in conjunction with the 8th HiPEAC conference*, Berlin, Germany, january 2013.

[117] Donald E. Knuth and Michael F. Plass. Breaking paragraphs into lines. *Software: Practice and Experience*, 11(11):1119–1184, 1981.

[118] Ramesh Radhakrishnan, R. Radhakrishnany, Lizy K. John, Juan Rubio, L. K. Johny, and N. Vijaykrishnan. Execution characteristics of just-in-time compilers, 1999.

[119] CEA LIST. Unisim virtual platforms. `http://unisim-vp.org/site/index.html`. [Online, July 2013].

[120] Jason D. Hiser, Daniel Williams, Wei Hu, Jack W. Davidson, Jason Mars, and Bruce R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 61–73, Washington, DC, USA, 2007. IEEE Computer Society.

[121] Kevin Casey, M. Anton Ertl, and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. *ACM Trans. Program. Lang. Syst.*, 29, October 2007.

[122] Tao Li, Ravi Bhargava, and Lizy Kurian John. Adapting branch-target buffer to improve the target predictability of Java code. *ACM Trans. Archit. Code Optim.*, 2(2):109–130, June 2005.

[123] A. Ketterlin and P. Clauss. Efficient memory tracing by program skeletonization. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 97–106, 2011.

[124] ARM. Cortex-A5 Processor. `http://www.arm.com/products/processors/cortex-a/cortex-a5.php`. [Online, July 2013].

[125] GNU. The GNU C++ Library. `http://gcc.gnu.org/onlinedocs/libstdc++`. [Online, July 2013].

[126] Alexandre Carbon, Yves Lhuillier, and Henri-Pierre Charles. Code specialization for red-black tree management algorithms. In *Proceedings of the 3rd International Workshop on Adaptive Self-Tuning Computing Systems*, ADAPT '13, pages 6:1–6:3, New York, NY, USA, 2013. ACM.

[127] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[128] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. *Foundations of Computer Science, IEEE Annual Symposium on*, 0:8–21, 1978.

[129] Daniel Santos. Generic red-black trees. https://lwn.net/Articles/517979/. [Online, July 2013].

[130] Jacob Bramley. Caches and self-modifying code. http://blogs.arm.com/software-enablement/141-caches-and-self-modifying-code/. [Online, July 2013].

[131] Alexandre Carbon, Yves Lhuillier, and Henri-Pierre Charles. Code specialization for red-black tree management algorithms. In *Proceedings of the IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, ASAP '13, pages 203–210, Washington D.C., USA, 2013. IEEE.

[132] LLVM. LLVM programmer's manual. http://llvm.org/docs/ProgrammersManual.html. [Online, July 2013].

[133] D. Lea. A memory allocator. http://g.oswego.edu/dl/html/malloc.html, 2000. [Online, July 2013].

[134] Newlib community. A memory allocator. http://sourceware.org/newlib/, 2010. [Online, July 2013].

[135] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 1–12, New York, NY, USA, 2002. ACM.

[136] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (SURF). *Comput. Vis. Image Underst.*, 110(3):346–359, June 2008.

[137] Jacob Bramley. GRLib IP library. http://www.gaisler.com/index.php/products/ipcores/soclibrary. [Online, July 2013].

# Publications personnelles

## Conférences avec actes

1. A. Carbon, Y. Lhuillier, and H.-P. Charles, *"Hardware Acceleration for Just-In-Time Compilation on Heterogeneous Embedded systems"*. 24th International Conference on Application-specific Systems, Architectures and Processors (ASAP '13), June 5-7, 2013, Washington D.C. (USA).
2. A. Carbon, Y. Lhuillier, and H.-P. Charles, *"Code specialization for red-black tree management algorithms"*. 3rd International Workshop on Adaptive Self-Tuning Computing Systems (ADAPT '13), January 22, 2013, Berlin (Germany).
3. A. Carbon, O. Héron, K. Ben Chehida and R. David, *"Impact of power management on temperature and reliability evolution for an embedded many-core architecture"*, Workshop proceedings of the 24th International Conference on Architecture of Computing Systems (ARCS '11), February 22-25, 2011, Como (Italy).

## Brevets

1. A. Carbon, Y. Lhuillier, et H.-P. Charles, *"Accélérateur matériel pour la manipulation d'arbres rouges et noirs"*, brevet déposé le 5 juin 2013.
2. A. Carbon et Y. Lhuillier, *"Système de compilation dynamique d'au moins un flot d'instructions"*, brevet déposé le 19 février 2013.

## Conférences sans actes

1. A. Carbon, Y. Lhuillier, and H.-P. Charles, *"Scaling-down to embedded systems for dynamic compilation"*, 2nd International Workshop "Dynamic Compilation Everywhere" (DCE '13), January 22, 2013, Berlin (Germany).
2. A. Carbon, Y. Lhuillier, et H.-P. Charles, *"Accélération de la compilation dynamique pour les cibles embarquées"*, Sixième colloque du GDR SoC-SIP du CNRS, 13-15 juin 2012, Paris (France).

## Posters

1. A. Carbon, Y. Lhuillier, and H.-P. Charles, *"Adapting Dynamic Compilation to Embedded Systems"*, 8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES '12), July 8-14, 2012, Fiuggi (Italy).
2. A. Carbon, Y. Lhuillier, and H.-P. Charles, *"Just-in-time compilation characterization"*, 7th International Conference on High-Performance and Embedded Architectures and Compilers (HIPEAC '12), January 23-25, 2012, Paris (France).

**Alexandre CARBON**

# Accélération matérielle de la compilation à la volée pour les systèmes embarqués


## Résumé

Développée depuis le début des années 60, la compilation dynamique connaît un essor considérable depuis une quinzaine d'année. Cet essor est essentiellement lié à deux aspects : le dynamisme croissant des applications et l'explosion de la demande en solutions de virtualisation. Le transfert de ces problématiques dans le domaine de l'embarqué a conduit à éprouver les technologies de compilation dynamique sur des ressources de calculs spartiates. Toutefois, la gestion de ces algorithmes complexes et irréguliers par des architectures simples (exécution dans l'ordre, peu ou pas de spéculation, hiérarchies mémoire limitées), pose un important problème de passage à l'échelle en termes de performances. En conséquence, les solutions de compilation dynamique sont moins attractives dans ce domaine. Alors que de nombreuses optimisations logicielles ont déjà été proposées dans l'état de l'art, nous proposons, dans le cadre de cette thèse, de mettre en place des accélérations matérielles couplées au processeur en charge de la compilation dynamique afin d'en accroître les performances. Basées sur le compilateur du cadriciel LLVM (LLC), nos analyses ont permis d'identifier deux points critiques en performances : la gestion des tableaux associatifs et de l'allocation dynamique de la mémoire, et la gestion du graphe des instructions à compiler. Deux accélérations ont ainsi été proposées. Concernant la gestion des tableaux associatifs, nous obtenons des gains atteignant 25 % sur LLC pour un surcoût silicium représentant moins de 1.4 % de la surface du processeur associé.


Mots-clés : *compilation dynamique, accélération matérielle, systèmes embarqués, tableaux associatifs, arbres rouges et noirs, parcours de graphe*


## Abstract

Developed since the 60s, JIT compilation is widely used since 15 years. This is the consequence of two main phenomena: the increasing dynamism of applications and the increasing demand concerning virtualization. The transfer of these issues to the embedded domain leads to experience JIT compilation on small and sparse resources. However, the management of JIT compilation algorithms' complexity and irregularity on small resources (in-order processors, limited speculation, limited memory hierarchies) leads to important scaling-down problems in terms of performance. As a consequence, JIT compilation solutions are less attractive in this domain. While several software optimizations have been already proposed in the literature, we propose in this thesis the development of hardware accelerations coupled to the processor in charge of the JIT compilation. The final aim is to propose a more efficient solution in terms of performance with respect to embedded constraints. Based on the LLVM framework compiler (LLC), our experiments highlight two critical points in terms of performance: the associative array and dynamic memory allocation management and the instruction graph handling for instructions to compile and optimize. Two accelerators have been proposed in this way. Concerning the management of associative arrays, we obtain gains up to 25 % on LLC with an area overhead under 1.4 % of the associated processor.


Keywords: *JIT (Just-In-Time) compilation, hardware acceleration, embedded systems, associative arrays, red-black trees, graph crawling*