Université de Strasbourg

École Doctorale Mathématiques, Sciences de l'Information et de l'Ingénieur

Laboratoire des sciences de l'ingénieur, de l'informatique et de l'imagerie (ICube)

THÈSE présentée par :

Juan Manuel MARTINEZ CAAMAÑO

Soutenue le : **29 Septembre 2016**

pour obtenir le grade de : **Docteur de l'Université de Strasbourg**
discipline/spécialité : **Informatique**

Fast and Flexible Compilation Techniques for Effective Speculative Polyhedral Parallelization

Thèse dirigée par :

M. Philippe CLAUSS	Professeur à l'Université de Strasbourg, Strasbourg, France

Rapporteurs :

M. Henri Pierre CHARLES	Directeur de Recherche informatique au CEA, Grenoble, France
M. Fabrice RASTELLO	Directeur de Recherche à Inria, Grenoble, France

Examinateurs :

M. Philippe HELLUY	Professeur à l'Université de Strasbourg, Strasbourg, France
M. Erven ROHOU	Directeur de Recherche à Inria, Rennes, France

Special thanks to the CTS Tram A, where the Code-Bones idea was born.

Contents

List of Figures

Chapter 0

Résumé en français

0.1 Introduction

Durant les deux dernières décennies, le paysage des architectures des ordinateurs a changé de manière abrupte. Précédemment et afin de satisfaire la demande toujours croissante de performance, les constructeurs ont agi sur l'augmentation des fréquences d'horloge des processeurs. C'est ainsi que la performance a pu être améliorée avec très peu d'intervention du programmeur ou du compilateur. Cependant, cela n'est aujourd'hui plus possible. Une telle augmentation de fréquence nécessite un accroissement du voltage, qui est devenu impossible à cause de contraintes physiques. Regrouper de multiples cœurs dans le même processeur est l'approche désormais adoptée pour répondre à la demande de performance.

Les architectures multi-processeurs possèdent une histoire riche dans le domaine du calcul haute performance. Mais désormais, elles représentent le courant dominant des architectures et touchent un champ plus large d'applications. Cependant, pour exploiter efficacement un tel matériel, le logiciel doit exhiber des régions parallèles explicitement. Ces régions peuvent être spécifiées par le programmeur ou découvertes par un compilateur. En plus de la parallélisation, l'exploitation des mémoires caches, présentes dans tout processeur moderne, est devenue encore plus importante : le logiciel s'exécutant sur plusieurs cœurs a besoin d'encore plus de bande passante mémoire pour effectuer ses calculs, ce qui élargit encore plus l'écart de performance entre la mémoire et le processeur.

Même si de multiples langages de programmation et extensions existent (OpenMP, TBB, OpenCL, CUDA, par exemple), la programmation parallèle reste une tâche ardue. Le programmeur doit aborder des questions difficiles telles que la sélection d'un bon algorithme, la garantie d'une sémantique correcte, et l'adéquation aux caractéris-

```
for( row = 1; row <= left->Size; row++ ) {
    pElement = left->FirstInRow[row];
    while( pElement ) {
        for( col = 1; col <= cols; col++ ) {
            result[row][col] +=
                pElement->Real * right[pElement->Col][col];
        }
        pElement = pElement->NextInRow;
    }
}
```

Listing 1: Noyau de multiplication de matrices creuses

tiques de l'architecture matérielle sous-jacente.

Afin d'aider le programmeur dans cette tâche difficile, des compilateurs spécialisés en parallélisation automatique ont été développés. Avec très peu d'intervention du programmeur, ces compilateurs sont capables d'extraire des régions parallèles à partir d'un programme séquentiel. Des exemples de tels optimiseurs sont Pluto et Polly. Traditionnellement, ils se focalisent sur les boucles de type «for», qui accèdent à des tableaux multi-dimensionnels via des fonctions affines des indices des boucles englobantes. De tels programmes peuvent être abstraits à l'aide du *modèle polyédrique* [13], qui est un cadriciel mathématique dédié au raisonnement sur la parallélisation et l'optimisation de boucles. Les programmes qui adhèrent à ce modèle bénéficient de transformations d'optimisation agressives.

Mais qu'en est-il des programmes qui contiennent des boucles «while», des pointeurs, ou des conditionnelles imprédictibles ? De tels programmes sont courants dans les logiciels généralistes, mais très difficiles à optimiser. Les analyses précises sont impossibles, ce qui empêche toute optimisation évoluée et en particulier la parallélisation automatique, même si le comportement dynamique du programme adhère au modèle polyédrique. Certaines informations cruciales ne peuvent être connues que lorsque le programme ciblé est en cours d'exécution.

Certains programmes, même s'ils ne possèdent pas les propriétés statiques polyédriques nécessaires, peuvent malgré tout exhiber des phases d'exécution où le comportement mémoire est compatible avec le modèle polyédrique. Le noyau de programme du listing 1 implémente un produit de matrices creuses. La boucle «while» du nid de boucles traverse une liste chaînée, où chaque élément représente une ligne d'une matrice. Lors de la compilation, il est impossible de savoir quelle location mémoire sera touchée par cette boucle et combien d'itérations seront exécutées.

Cependant, pour certaines données d'entrée (une matrice diagonale ou une sous-matrice carrée, par exemple), il est possible que le comportement soit compatible avec

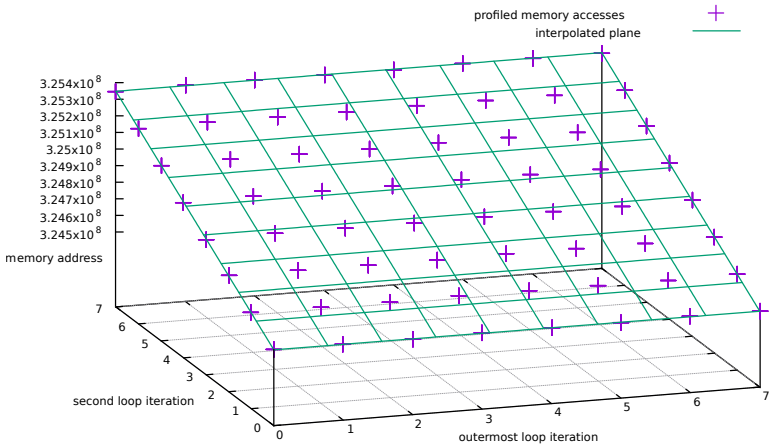

Figure 1: Adresses collectées et plan d'interpolation, pour une instruction mémoire accédant à une liste chaînée, dans l'exemple de la multiplication de matrices creuses.

le modèle polyédrique. Sur la figure 1, on montre les adresses mémoire accédées par une certaine instruction mémoire, et leur plan d'interpolation. On remarque un comportement compatible, au moins pour les accès qui sont représentés. Afin de détecter de telles phases, puis de les optimiser, le modèle polyédrique doit être adapté à son utilisation dynamique.

Une autre approche est employée dans les systèmes de parallélisation spéculative TLS (*Thread Level Speculation*). Ceux-ci exécutent de manière optimiste des régions de programme en parallèle, avant de connaître toutes les dépendances. Des mécanismes matériels et logiciels surveillent les accès mémoire. Si une dépendance est violée, un mécanisme de recouvrement restaure l'exécution jusqu'à un point consistant précédent. Ensuite, une re-exécution séquentielle est lancée.

Le succès de tels systèmes est limité. La plupart des systèmes TLS appliquent un schéma simple de parallélisation, tel que l'exécution en parallèle de tranches de la boucle la plus externe du nid de boucles ciblé. Il leur est impossible de traiter les programmes où une transformation, telle que la torsion de boucle (*skewing*), est nécessaire à l'extraction de parallélisme. Ils sont également incapable d'améliorer la localité des données, ce qui entraîne de faibles performances en pratique. De plus, les mécanismes utilisés pour détecter les violations de dépendances tendent à devenir des goulets d'étranglement de ces systèmes, entraînant une quantité importante de communications entre les threads.

Plusieurs facteurs empêchent les systèmes TLS d'appliquer des optimisations plus agressives. Tout d'abord, peu de choses sont faites pour considérer les dépendances, même si elles interviennent selon des motifs prédictibles. Un modèle de prédiction est nécessaire pour les inclure dans le raisonnement, et pour rendre possible la détermi-

nation d'une transformation optimisante. Pour instancier une telle transformation, un mécanisme de génération de code à la volée est nécessaire. Il doit être capable de générer du code dans un temps raisonnable. Notons que dans un tel système, tout surcoût temporel peut annihiler le bénéfice d'une optimisation. La plupart des systèmes existants ne remplit qu'une de ces exigences.

1. Une approche consiste à privilégier la vitesse au détriment de la flexibilité en générant plusieurs versions binaires optimisées lors de la compilation, au prix d'un fichier binaire de grande taille. Cette approche limite le nombre de transformations possibles à celles qui ont été pré-calculées.

2. Une autre approche possible est de générer entièrement le programme optimisé au cours de l'exécution, ce qui impose un surcoût temporel important.

3. Enfin, il est également possible de générer des squelettes paramétrés de code. En donnant les valeurs appropriées aux paramètres d'un tel squelette, plusieurs transformations peuvent être instanciées.

Cette dernière approche est une balance acceptable entre vitesse et flexibilité, bien qu'elle interdise les transformations non supportées par les squelettes générés à la compilation. Un nombre important de squelettes est nécessaire afin de supporter n'importe quelle combinaison de transformations qui altèrent la structure du nid de boucles, telles que le pavage, la fusion, la fission, ou le ré-ordonnancement des instructions. De plus, certaines transformations, telle que le déroulement de boucle, dépendent de paramètres d'exécution qui ne peuvent pas être prédits à la compilation. Finalement, cette approche impose toujours une limitation à l'exploitation dynamique du modèle polyédrique.

Apollo est un système TLS capable d'appliquer des transformations polyédriques agressives. Il met en oeuvre un modèle de prédiction et un mécanisme rapide de génération de code juste-à-temps, afin d'optimiser avec succès des programmes comprenant des pointeurs ou des boucles «while». Dans la section 0.2, nous présentons la plateforme Apollo. Notre contribution principale est appelée *Code-Bones*: il s'agit d'un mécanisme ambitieux de génération de code qui supporte n'importe quelle transformation polyédrique, tout en restant rapide. Il est présenté en section 0.3. Finalement, dans les sections 0.4 et 0.5, nous présentons nos résultats expérimentaux et nos conclusions finales.

0.2 Apollo: quand les polyèdres rencontrent la spéculation

Traditionnellement, le modèle polyédrique échoue face à des programmes qui comportent des indirections, des accès mémoire par pointeurs, ou des bornes de boucle imprédictible, qui sont courants dans les programmes généralistes. Cependant, de tels programmes peuvent exhiber des phases qui sont compatibles avec le modèle polyédrique : les bornes et les adresses accédées prennent des valeurs qui correspondent à des fonctions affines des indices des boucles englobantes. Les compilateurs effectuant des analyses statiques ne peuvent pas exploiter de tels comportements, alors qu'une plate-forme dynamique de compilation peut détecter et tirer avantage de telles phases. Apollo est une plate-forme hybride de compilation qui procède ainsi avec succès.

Pendant l'exécution du programme cible, Apollo surveille le comportement pour détecter ces phases compatibles. Même si une telle phase a été détectée, on ne peut pas certifier que les itérations futures auront le même comportement que les précédentes. Ceci est la pierre angulaire de la spéculation. Apollo construit un modèle qui prédit comment les itérations futures devraient se comporter, puis détermine une transformation et génère le programme optimisé selon ce modèle. La transformation sélectionnée, et le programme optimisé généré, restent valides aussi longtemps que les itérations exécutées correspondent au modèle de prédiction. En cas de spéculation fausse, un recouvrement (*rollback*) est lancé et le programme original est re-exécuté. Tout ces mécanismes nécessaires à l'optimisation spéculative de programmes imposent un surcoût temporel périlleux pour tout système dynamique. Cependant, les gains de performance obtenus grâce à la parallélisation et à l'amélioration de la localité des données peuvent largement compenser ce surcoût. Apollo, à notre connaissance, est l'unique plate-forme capable de déployer l'entière puissance du modèle polyédrique au cours de l'exécution du programme cible.

0.2.1 Seulement un #pragma

Pour utiliser Apollo, le programmeur doit encadrer les nids de boucles ciblés à l'aide d'une directive #pragma dédiée. A l'intérieur du #pragma, toutes les sortes de boucles sont autorisées, boucles «for», boucles «while», ou boucles «goto». À la figure 2, nous montrons un aperçu de la plate-forme. À la compilation, le programmeur compile son programme source annoté avec Apollo. Ensuite, le système *runtime* d'Apollo orchestre l'exécution et optimise dynamiquement le programme.

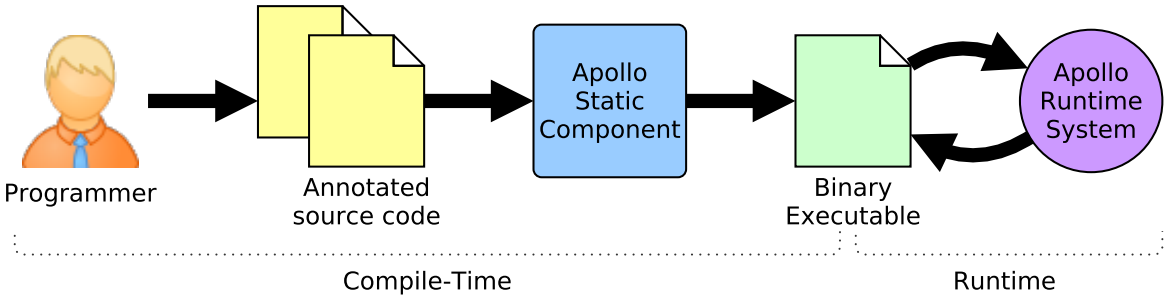

Figure 2: Aperçu de la plate-forme Apollo.

0.2.2 Préparation lors de la compilation

Pour permettre une parallélisation spéculative, le programme original doit tout d'abord subir une préparation. La composante statique d'Apollo est chargée de cette tâche. Son rôle est d'inclure dans le code exécutable final, toutes les structures de données qui sont nécessaires à une parallélisation spéculative efficace. Toutes ces informations seront ensuite accessibles au système *runtime*.

La première étape effectuée par la composante statique est d'optimiser le code à l'aide d'optimisations classiques, fournies par le compilateur LLVM. Ensuite, des informations statiques sont extraites des nids de boucles cibles et inclues dans l'exécutable. Ces informations concernent, par exemple : les nombres et types d'instructions mémoire, la structure des nids de boucles, et des informations d'*aliasing*.

Pour chaque boucle d'un nid ciblé, Apollo insère un *itérateur virtuel*, qui démarre à zéro et qui est incrémenté de un à la fin de chaque itération. Ces itérateurs permettent à Apollo de traiter toutes les sortes de boucles de la même façon. Plus important encore, les itérateurs virtuels vont jouer le rôle de base, au sens mathématique, pour la prédiction des accès mémoire, des bornes de boucles et des valeurs de scalaires.

Une copie du code original est utilisée pour créer une version instrumentée du code. De plus, plusieurs copies du code sont générées pour en dériver des squelettes de code. Les squelettes de code sont des codes incomplets qui supportent une classe fixée limitée de transformations optimisantes. En les complétant lors de l'exécution, Apollo peut instancier une combinaison particulière de transformations. Néanmoins, ces squelettes ne peuvent supporter l'éventail complet des transformations permises par le modèle polyédrique. C'est pour cette raison que ce mécanisme a été remplacé par l'utilisation de *Code-Bones*.

L'approche *Code-Bones* est la contribution principale de ce travail de thèse, qui est détaillée à la section suivante. De nombreux *Code-Bones* sont dérivés du code original, et embarqués dans l'exécutable dans leur représentation *bitcode* de LLVM. Lors de

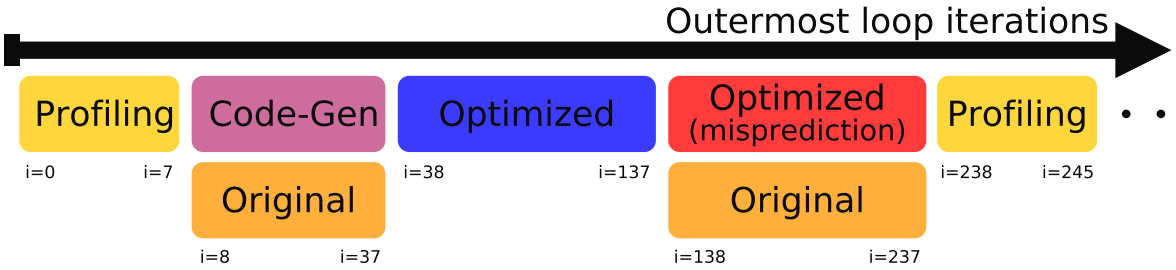

Figure 3: Succession de morceaux alternant entre comportements.

l'exécution, les *Code-Bones* sont chargés en mémoire et assemblés en un nouveau nid de boucles, instanciant ainsi toute combinaison possible de transformations de boucles. En utilisant le compilateur juste-à-temps de LLVM, Apollo génère finalement le code binaire optimisé.

0.2.3 Exécution par 'morceaux' d'itérations

Afin d'être réactif aux changements de comportement, Apollo exécute le nid de boucles cible par *morceaux*, ou *tranches*, d'itérations, alternant ainsi entre versions de code à chaque morceau. Un morceau est un ensemble d'itérations contiguës de la boucle la plus externe. Au début de l'exécution d'un nid, Apollo profile quelques itérations à l'intérieur d'un petit morceau, typiquement de l'itération 0 à l'itération 7, en utilisant la version instrumentée du code. Le morceau suivant peut être exécuté avec une version différente du code, par exemple une version optimisée, de l'itération 8 à 107. De cette manière, le nid est exécuté en une succession de morceaux. Entre ces morceaux, le contrôle est rendu au système *runtime*, où Apollo peut prendre une décision quant à l'exécution du morceau suivant. Dans notre approche, les morceaux sont exécutés séquentiellement, mais les itérations à l'intérieur d'un morceau peuvent être exécutées en parallèle.

Lors de cette exécution par morceaux, Apollo alterne entre des comportements différents afin de détecter et de s'adapter à des phases d'exécution différentes (voir la figure 3). Le comportement du morceau courant et le résultat de son exécution déterminent le morceau suivant.

1. Au début de l'exécution d'un nid de boucles, ou après la terminaison d'un morceau où la version originale a été exécutée, un morceau de profilage est lancé afin de capturer le comportement du code. A la fin de cette exécution profilée, Apollo passe à la phase de génération de code.

2. Lors de la phase de génération de code, un modèle de prédiction est construit et

une transformation est déterminée. A partir de celle-ci, un code binaire optimisé est généré à l'aide du compilateur juste-à-temps de LLVM. En cas de succès, Apollo est prêt à exécuter le code optimisé. Sinon, l'exécution se poursuit à l'aide du code original. En parallèle à tout cela, un thread en tâche de fond exécute la version originale du code, afin de masquer, au moins partiellement, le surcoût temporel correspondant.

3. Avant d'exécuter un code optimisé, Apollo effectue une sauvegarde préventive des zones d'écriture mémoire prédites (grâce au modèle de prédiction). Dès que la sauvegarde a été réalisée, Apollo exécute le code optimisé.

4. Les morceaux optimisés sont exécutés jusqu'à la détection d'une erreur de spéculation. Les itérations à l'intérieur d'un morceau optimisé sont généralement exécutées en parallèle.

5. Lorsqu'une fausse spéculation est détectée, un recouvrement est effectué afin de rétablir l'état mémoire antérieur à l'exécution du morceau fautif.

6. Le même morceau est re-exécuté à l'aide de la version originale du code, qui est, bien évidemment, sémantiquement correct.

0.2.4 La construction d'un modèle de prédiction

Contrairement à la plupart des systèmes TLS, Apollo construit un modèle qui prédit le comportement du nid de boucles. En assumant la validité de cette prédiction, Apollo déduit des dépendances entres les itérations et les instructions, puis applique des optimisations de code agressives, qui impliquent un ré-ordonnancement des itérations et des instructions.

Ce modèle prédit : (1) les accès mémoire, (2) les bornes de boucles, et (3) les scalaires de base.

1. Les accès mémoire Il y a trois modélisations possibles pour les accès mémoire: (i) linéaire, (ii) par tube, et (iii) par intervalle. Dans la figure 4, nous représentons ces 3 modèles.

Lorsqu'il est possible d'interpoler les adresses mémoire collectées à l'aide d'une fonction linéaire, un accès mémoire est prédit comme étant (i) linaire. Les accès futurs sont prédits comme respectant exactement la fonction d'interpolation. Cependant, les accès mémoire peuvent ne pas correspondre exactement à un motif linéaire parfait. Dans ce cas, un hyperplan de régression est calculé en utilisant la méthode des moindres

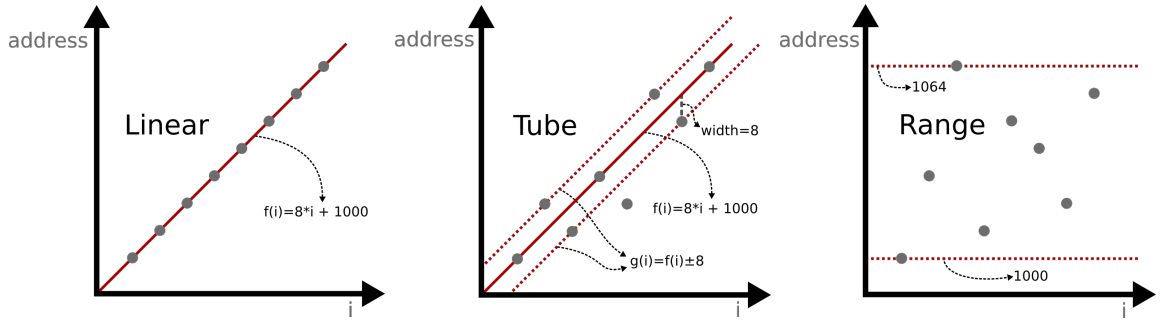

Figure 4: Les trois modélisations des accès mémoire.

carrés. Le coefficient de corrélation de Pearson est ensuite calculé. S'il est supérieur à 0,9, les accès mémoire futurs sont prédits comme étant situés à l'intérieur d'un tube, sinon, ils sont prédits comme étant situés à l'intérieur d'un intervalle. Ce critère a été établi par expérimentations [29]. Un tube consiste en un hyperplan de régression, une largeur de tube et un alignement. La largeur de tube est la déviation maximale observée par rapport à l'hyperplan de régression, arrondie supérieurement au prochain multiple de la taille d'un mot. Un intervalle consiste en une adresse mémoire maximum et une adresse mémoire minimum entre lesquelles tous les accès mémoire sont prédits comme étant situés.

2. Les bornes de boucle Il y a deux modélisations possibles : (i) linéaire ou (ii) par tube. La figure 5 représente visuellement les différents types de prédictions pour les bornes de boucle. Remarquons que la borne inférieure des itérateurs virtuels est toujours zéro, et que seule la borne supérieure est prédite. La prédiction linéaire des bornes est similaire à celle des accès mémoire.

Dans le cas de tubes, un hyperplan de régression est calculé. Puis deux hyperplans en sont dérivés permettant de prédire un nombre maximum et minimum d'itérations exécutées par une boucle. Lors de la recherche d'une transformation, l'espace d'itérations est divisé en deux, et toutes les itérations situées en dessous de la borne minimum prédite peuvent être transformées. Par contre, les itérations situées entre les bornes minimum et maximum prédites doivent être exécutées séquentiellement, jusqu'à atteindre le condition de sortie de boucle.

3. Les scalaires de base Il y a trois modélisations possibles : (i) linéaire, (ii) semi-linéaire et (iii) réduction.

À nouveau, le cas linéaire est similaire au cas linéaire des accès mémoire. Un scalaire semi-linéaire est caractérisé par un incrément constant à chaque itération de sa boucle père, mais une valeur initiale en début de boucle qui ne peut pas être prédite. La

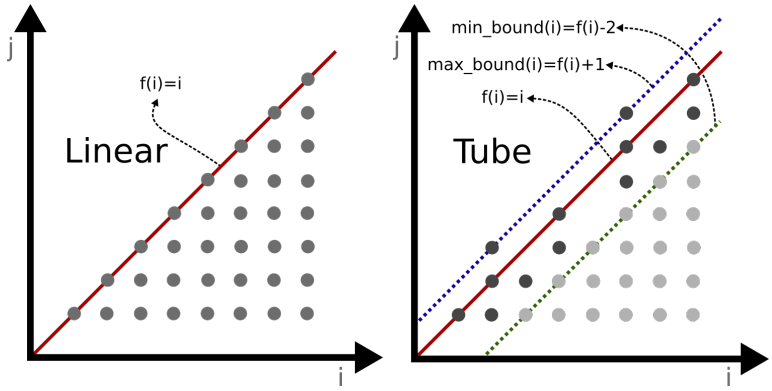

Figure 5: Les deux modélisations des bornes de boucle.

location mémoire nécessaire au calcul de cette valeur initiale doit être prédite comme étant fixée tout au long de l'exécution de la boucle. Tout autre comportement est traité comme une réduction. Malheureusement, les outils polyédriques sous-jacents utilisés par Apollo ne savent pas du tout traiter les réductions, ce qui empêche tout génération de code optimisé lorsqu'elle surviennent.

0.2.5 La sélection d'une transformation

Pour choisir une transformation, Apollo s'en remet à l'optimiseur polyédrique appelé Pluto, qui est capable de déterminer une transformation de boucle qui vise à la fois une bonne localité des données et l'exhibition de parallélisme. Pluto expose son moteur interne à travers un interface de bibliothèque, qui prend en entrée une représentation OpenScop du nid de boucle cible. Ce processus étant lié de près au mécanisme des *Code-Bones*, il est présenté en section 0.3.

0.2.6 La génération de code optimisé

Apollo emploie deux mécanismes de génération de code optimisé : les squelettes de code, développés initialement dans le prototype pré-Apollo VMAD, et les *Code-Bones*, qui relèvent d'une approche plus flexible présentée dans cette thèse. Le deuxième mécanisme a depuis remplacé le premier.

Un squelette de code est une copie paramétrée du code original. Il est paramétré par le modèle de prédiction et par une transformation polyédrique. En donnant des valeurs à ces paramètres, on peut instancier différentes transformations. Ces squelettes contiennent également des instructions vérifiant la validité du modèle de prédiction. À la compilation, plusieurs squelettes de code sont générées afin de supporter plusieurs types de transformations. Cependant, chaque squelette ne supporte qu'une certaine

```
for(i = 0; i < n; ++i)
    for(j = 0; j < n; ++j)
        a[i][j] = a[i][j-1] + a[i-1][j]
```

Listing 2: Code original

```
pm = ... // the prediction model
tx[][] = ... // the transformation
ca[] = pm->linear_function_a //predicted linear function
for(x = lower_x(pm, tx); x < upper_x(pm, tx); ++x) {
    for(y = lower_y(pm, tx, x); y < upper_y(pm, tx, x); ++y) {
        //obtain the iterators in the original iteration space
        i = tx[0][0] * x + tx[0][1] * y
        j = tx[1][0] * x + tx[1][1] * y

        //compute the predictions
        a_pred_0 = ca[0] * i + ca[1] * j + ca[2]
        a_pred_1 = ca[0] * i + ca[1] * (j-1) + ca[2]
        a_pred_2 = ca[0] * (i-1) + ca[1] * j + ca[2]

        //verify the predictions
        if(a_pred_0 != a[i][j]) rollback()
        if(a_pred_1 != a[i][j-1]) rollback()
        if(a_pred_2 != a[i-1][j]) rollback()

        //original computation
        *a_pred_0 = *a_pred_1 + *a_pred_2
        }
}
```

Listing 3: Squelette de code dérivé du code du Listing 2

combinaison de transformation de boucles, car la structure de boucle est fixée lors de la compilation. Par exemple, alors qu'un certain squelette supporte la combinaison d'une torsion et d'un échange de boucles, il ne peut pas supporter d'autres transformations telles que le pavage ou la fission de boucle. Bien que ces dernières transformations pourraient être supportées par des squelettes supplémentaires, ceux-ci engendrerait un fichier exécutable de très grande taille. De toute manière, cette stratégie ne peut pas fournir la flexibilité nécessaire pour couvrir toutes les combinaisons possibles de transformations de boucles, telles que celles permises par les compilateurs polyédriques statiques comme Pluto.

Le pseudo-code du Listing 3 montre le squelette de code dérivé du code du Listing 2. On peut remarquer que ce squelette est paramétré par le modèle de prédiction pm et par l'inverse de la transformation tx. A partir des itérateurs x et y qui parcourent l'espace d'itération transformé, et en utilisant l'inverse de la transformation, on retrouve les valeurs des itérateurs originaux i et j dans l'espace d'itération original. Il est alors aisé

de calculer la prédiction pour chaque accès mémoire. Pour vérifier que le modèle de prédiction est valide, les adresses mémoire effective sont comparées à leur prédiction. Si elles sont différentes, un recouvrement est signalé et l'exécution est annulée.

Considérons que nous voulions instancier un échange de boucles sur le code du Listing 4.4. La matrice de transformation correspondante, et son inverse, est $\mathsf{tx} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. A partir de cette matrice, de nouvelles bornes de boucles sont calculées en utilisant l'algorithme d'élimation de variables Fourier-Motzkin. Le résultat de cet algorithme sera interprété par les fonctions lower et upper. Notons qu'une conséquence de l'utilisation de ce squelette de code est que la transformation de pavage n'est pas supportée, car la structure de boucle est fixée. De plus, on ne peut pas instancier une transformation différente pour le code de vérification, car il réside dans la même boucle que les instructions qu'il vérifie.

Une approche différente et plus flexible est réalisée grâce aux *Code-Bones*. Au lieu de fournir une structure de boucle à remplir, ils fournissent de blocs constituants plus petits, utilisés pour l'assemblage d'une boucle entièrement nouvelle au cours de l'exécution.

Afin de comparer cette nouvelle approche aux squelettes de code, nous montrons au Listing 4 les *Code-Bones* dérivés du même code original. Un *code-bone* regroupe des instructions liées afin de reconstruire des instructions similaire au code source en langage C. Ces *code-bones* peuvent être agglomérés afin d'instancier n'importe quelle transformation supportée par le modèle polyédrique. Dans le code du Listing 5, une combinaison de transformations de fission, échange et déroulement de boucles est appliquée. Contrairement au squelette de code précédent, des transformations différentes sont appliquées au code de vérification et au code de calcul. Remarquons les différences de *code-bones* générés pour les calcul originaux et la vérification. Cela nous permet de les ordonnancer différemment, et d'exploiter des propriétés spécifiques du code de vérification pour l'optimiser encore plus.

Dans la section suivante, nous détaillons notre technique de génération de code basée sur les *Code-Bones*.

0.3 Code-Bones: génération de code rapide et flexible pour la parallélisation polyédrique spéculative

La principale raison des faibles performances obtenues par la plupart des systèmes TLS, est qu'ils sont incapables d'appliquer des transformations agressives d'optimisation

```
computation_bone(pm, i, j) {
    ca[] = pm->linear_function_a //coefficients

    a_pred_0 = ca[0] * i + ca[1] * j + ca[2]
    a_pred_1 = ca[0] * i + ca[1] * (j-1) + ca[2]
    a_pred_2 = ca[0] * (i-1) + ca[1] * j + ca[2]

    *a_pred_0 = *a_pred_1 + *a_pred_2
}

verification_0_bone(pm, i, j) {
    ca[] = pm->linear_function_a //coefficients
    a_pred_0 = ca[0] * i + ca[1] * j + ca[2]
    if(a_pred_0 != a[i][j]) rollback()
}

verification_1_bone(pm, i, j) {
    ca[] = pm->linear_function_a //coefficients
    a_pred_1 = ca[0] * i + ca[1] * (j-1) + ca[2]
    if(a_pred_1 != a[i][j-1]) rollback()
}

verification_2_bone(pm, i, j) {
    ca[] = pm->linear_function_a //coefficients
    a_pred_2 = ca[0] * (i-1) + ca[1] * j + ca[2]
    if(a_pred_2 != a[i-1][j]) rollback()
}
```

Listing 4: *Code-Bones* dérivés du code du Listing 2

```
for(i = 0; i < N; ++i)
    for(j = 0; j < N; ++j)
        verification_0_bone(pm, i, j);
        verification_1_bone(pm, i, j);
        verification_2_bone(pm, i, j);

for(j = 0; j < N; ++j)
    for(i = 0; i < N; i+=2)
        computation_bone(pm, i, j)
        computation_bone(pm, i+1, j)
```

Listing 5: Transformation avancée en utilisant les Code-Bones du Listing 4 (fission, échange et déroulement de boucles).

et de parallélisation. C'est pourquoi, uniquement des parallélisations simples sont effectuées et des questions telles que la localité des données ne sont pas abordées, bien qu'il soit notoirement admis que le succès d'une programmation parallèle dépend fortement de telles questions.

Combiner parallélisation spéculative et modèle polyédrique fixe un contexte particulier pour la génération de code. Nous avons rencontré troix challenges majeurs au cours du développement de notre processus de génération de code.

1. Exprimer une représentation intermédiaire dans le modèle polyédrique Apollo est construit au-dessus du compilateur LLVM. Par conséquent, la composante statique d'Apollo manipule des programmes dans la forme intermédiaire de LLVM. Cette forme se veut être de bas niveau, de telle sorte que toute construction de programmation de haut niveau puisse être exprimée. Par exemple, les boucles «for» et «while» sont exprimées uniformément comme des sauts conditionnels entre des blocs de base dans le graphe de flot de contrôle.

Les instructions du code C sont exprimées en plusieurs instructions contiguës plus élémentaires. Après l'application d'optimisations classiques de compilation, ces instructions sont réparties à travers plusieurs blocs de base, et même plusieurs boucles. Les instructions sont ré-organisées et certaines d'entre-elles sont même fusionnées afin d'éliminer des calculs redondants. Ainsi, une forme concise d'instruction en C se retrouve diluée dans la représentation intermédiaire.

Par contre, le modèle polyédrique représente un nid de boucles comme des instances d'itérations de ces instructions de type C. De telles instructions sont vues comme des unités atomiques, et seules les fonctions linéaires des accès mémoire sont exposées. Une conséquence majeure est que nous ne pouvons pas exprimer directement notre forme intermédiaire dans le modèle polyédrique.

Notre approche résout ce problème en extrayant des *code-bones* de la forme intermédiaire. Ces *code-bones* sont vus comme des unités atomiques qui n'interfèrent pas entres-elles, et qui sont équivalentes à des instructions de type C. Cette abstraction permet une expression directe dans le modèle polyédrique.

2. Exploiter des opportunités uniques d'optimisation Le code optimisé généré par Apollo est composé de deux types principaux d'instructions : les instructions qui effectuent les opérations du code original, et qui résultent en des écritures mémoire ; et les instructions dédiées à la vérification de la validité du modèle de prédiction. Ces différents types d'instructions entraînent différentes opportunités d'optimisation, qui sont particulières dans le contexte d'Apollo.

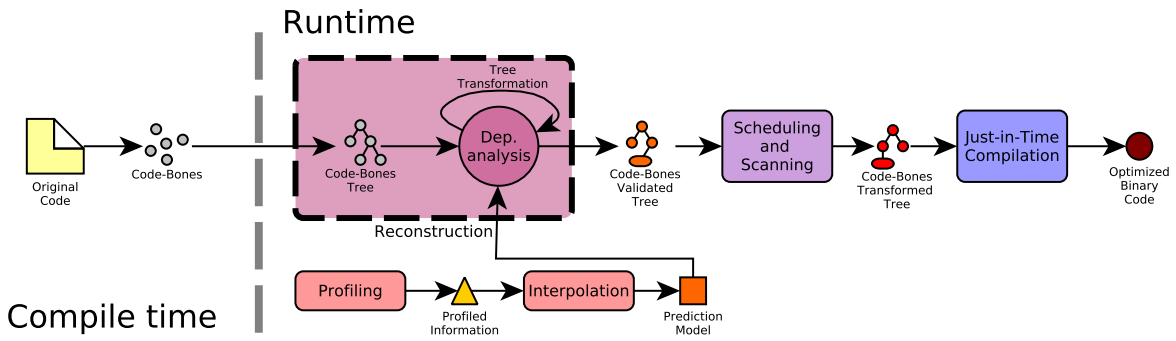

Figure 6: Aperçu du flot de génération de code d'Apollo.

3. Rapide! Dans un contexte d'optimisation dynamique, le surcoût temporel est un souci majeur. Cependant, les outils polyédriques disponibles n'ont pas été développés dans l'optique d'une utilisation *runtime*, et sont souvent trop lents. Plusieurs décisions et optimisations ont du être appliquées pour contourner ce problème. Néanmoins, nous pensons qu'il y a un grand et crucial écart à remplir par de nouveaux outils adaptés à l'analyse et à l'optimisation dynamiques.

0.3.1 Code-Bones: construction de blocs élémentaires pour les transformations polyédriques d'une forme intermédiaire de compilateur

Notre approche pour la génération de code propose d'utiliser des blocs de construction permettant d'instancier toute transformation. Nous appelons ces blocs des *Code-Bones*.

À la compilation, de multiples *code-bones* sont générés. Puis, à l'exécution, une transformation est déterminée et implémentée en instanciant et en assemblant les *codes-bones*. La figure 6 décrit les différentes tâches effectuées pour générer finalement du code exécutable optimisé. Nous expliquons ci-dessous chacune de ces étapes.

1. Reconstruction Une structure de boucle qui mime la boucle originale est créée, complémentée par des informations dynamiques obtenues grâce au modèle de prédiction. Tous les *code-bones* de calcul sont insérés dans ce nid de boucles. Chacun possède une position particulière, donnée par sa boucle parente et l'*id* de son instruction d'écriture associée.

Toutes les instructions prédites des *bones* doivent être vérifiées. Tout d'abord, les prédictions non-linéaires sont traitées en insérant un *bone* de vérification juste avant le *bone* qui contient l'instruction non-linéaire correspondante. Ensuite, les deux *bones* sont fusionnés en un seul. Dans ce nouveau *bone* généré dynamiquement, la vérification

garde la véritable exécution du *bone*. Une fois que toutes les instructions non-linéaires sont vérifiées, les *bones* qui vérifient les prédictions linéaires sont également insérés. Ils n'ont pas besoin d'être fusionnés. L'ajout de *bones* de vérification se répète jusqu'à ce que chaque instruction prédite est traitée.

De plus, les *bones* contiguës peuvent être fusionnés, afin de réduire le nombre final d'instructions polyédriques à ordonnancer. Ainsi, nous pouvons réduire le surcoût temporel nécessaire à la sélection d'une transformation et à la génération du parcours d'itérations associé. Par contre, la fusion de *bones* réduit l'éventail des transformations possibles.

2. Validation L'ordre dans lequel apparaissent les accès mémoire dans ce code peut être différent de l'ordre du code original. De plus, certaines lectures peuvent être répétées si elles prennent part à plusieurs *code-bones*. Cela peut violer des dépendances internes à une itération du code original. Les dépendances entre itérations sont bien sûr satisfaites, car l'ordre des itérations n'a pas encore été modifié. Ainsi, le nid de boucles reconstruit doit être vérifié concernant les dépendances intra-itération du code original. S'il n'y a pas de violation, le nid reconstruit est équivalent au nid original, et nous pouvons passer à sa transformation optimisante.

3. Conversion en une représentation polyédrique Maintenant, nous avons un nid de boucles qui contient des instructions de type C bien définies. Ces instructions s'exécutent de manière atomique et sont caractérisées uniquement par leurs accès mémoire. Il est assez simple d'obtenir le domaine polyédrique et les fonctions d'ordonnancement pour chaque *bone* d'un tel nid. Ils sont ensuite encodés au format OpenScop afin d'être communiqués à Pluto.

Cependant, pour les accès mémoire, Apollo tente de retrouver le plus d'informations possibles concernant la structure de donnée qui est accédée. Cela non seulement accélère la phase de sélection d'une transformation, mais aussi améliore la qualité de la transformation sélectionnée.

Plusieurs étapes sont réalisées :

1. La reconnaissance de tableaux : en utilisant des informations d'analyse statique d'alias et les intervalles d'adresses prédits pour les instructions mémoire, plusieurs ensembles d'instructions ayant des accès qui se recouvrent sont définis. Chaque ensemble est associé à son propre tableau, et ne se superpose pas aux autres ensembles.

2. Le calcul des tailles de mots : pour chaque tableau, le plus grand commun diviseur

(PGCD) des coefficients de la fonction linéaire d'accès et de la taille de mot de chaque accès, est calculé. Le résultat est utilisé pour dériver une nouvelle fonction d'accès, en divisant les coefficients par le PGCD.

3. Le recouvrement des dimensions : pour les tableaux accédés uniquement à travers des fonctions linéaires, il est parfois possible de retrouver les dimensions des tableaux multi-dimensionnels. Notre implémentation est inspirée de [21]. Lors de l'exécution, les valeurs de tous les coefficients des fonctions d'accès linéaires sont connues, ce qui simplifie la tâche.

4. Ordonnancement Une fois que la représentation polyédrique du code a été construite, nous pouvons invoquer l'algorithme du compilateur Pluto pour déterminer une transformation d'optimisation et de parallélisation. Pluto propose plusieurs options qui doivent être combinées pour obtenir la meilleure transformation. Il n'y a pas une combinaison unique qui outrepasse d'autres combinaisons. De plus, la meilleure combinaison peut dépendre du code cible ou du matériel. Cependant, de nombreuses expérimentations nous ont conduits à définir une combinaison d'options qui engendre des codes bien optimisés dans la plupart des cas.

5. Optimisation de la vérification Les *code-bones* qui vérifient la prédiction de la transformation sont quelque chose d'unique à Apollo. Ils exposent de nouvelles opportunités d'optimisation que les *bones* de calcul ne possèdent pas. Ils ont deux propriétés majeures à exploiter :

- les *bones* de vérification n'écrivent pas en mémoire ;
- la plupart du temps, les *bones* de vérification ne participent pas aux dépendances.

Les optimisations expliquées dans la suite concernent les *bones* de vérification qui accèdent à la mémoire à travers des fonctions linéaires, et qui ne participent pas aux dépendances. Ce type de *bone* est le plus fréquent parmi ceux que l'on rencontre en pratique.

1. La première optimisation consiste à déplacer tous les code-bones de ce type dans un nid de boucle à part, afin d'être exécutés avant le reste du code. Ce nid de boucle est optimisé séparément avec Pluto, en demandant une fission maximale, sans pavage. Plusieurs expérimentations nous ont montré qu'il s'agissait de l'approche donnant la meilleure optimisation. Dans le nid résultant, toutes

les boucles sont parallèles. Puisqu'il est exécuté avant le reste du code (qui participe à des dépendances), il permet une détection anticipée de toute erreur de spéculation.

2. Pour la seconde optimisation, nous considérons les même *code-bones* que précédemment, mais dont les coefficients de leurs fonctions linéaires sont à zéro pour un certain niveau de boucle. Dans ce cas, vérifier uniquement une itération pour ce niveau de boucle est suffisant, car pour ce niveau, l'entrée du *code-bone* reste invariante.

3. Le calcul d'une adresse cible peut être composée de sous-parties qui sont clairement linéaires par construction. Ainsi, la vérification de telles sous-parties linéaires peut consommer du temps inutile de calcul. La dernière optimisation réduit la vérification aux sous-parties qui ne sont pas définies directement comme étant linéaires, en supprimant les boucles relatives aux sous-parties linéaires, et en réduisant les expression définissant l'adresse cible aux sous-parties qui nécessitent une vérification.

6. Parcours La représentation OpenScop transformée est transmise à CLooG afin de calculer les boucles de parcours. La sortie de CLooG est une structure de données similaire à un arbre de syntaxe qui décrit comment itérer pour chaque instruction de la représentation polyédrique.

7. Compilation juste-à-temps La dernière étape de notre processus de génération de code est de générer du code exécutable binaire. Pour cette tâche, nous utilisons le compilateur LLVM. Il procure un compilateur juste-à-temps et un cadriciel pour appliquer des optimisations sur notre programme cible, avant la génération finale de code natif. Le processus consiste à générer du code final en représentation intermédiaire LLVM à partir de la sortie de CLooG et des *code-bones*. Cette représentation intermédiaire finale est ensuite transmise au compilateur juste-à-temps pour la génération de code binaire. Une passe dédiée transforme la sortie de CLooG en forme intermédiaire. Elle correspond à certaines constructions du C, comme des boucles «for», des conditions «if» et des instructions (dans notre cas des *code-bones*). La traduction de ces constructions en forme intermédiaire est triviale, exceptée pour les *code-bones*. Nous traduisons une invocation de *code-bone* en un appel au *code-bone* (rappelons qu'un *code-bone* est représenté comme une fonction LLVM). S'il s'agit d'un *bone* de vérification, alors une branche vers l'instruction de recouvrement est insérée. Enfin, une série d'optimisation

est appliquée, incluant la propagation de constantes, la combinaison d'instructions, la réduction de force, le déroulement de boucles, la vectorisation, etc. parmi d'autres.

0.4 Expérimentations

Dans cette section, nous évaluons la plate-forme Apollo, améliorée par notre processus de génération de code utilisant les *code-bones*. Les programmes d'essais ont été exécutés sur deux machines différentes :

- **Lemans**, un serveur multi-cœur généraliste, doté de deux processeurs AMD Opteron 6172 de 12 cœurs chacun (24 cœurs au total).
- **Armonique**, une puce embarquée multi-cœur, dotée d'un processeur ARM Cortex A53 64-bit de 8 cœurs.

L'ensemble des programmes d'essai a été construit à partir de plusieurs suites de bancs d'essai, de telle manière que les programmes sélectionnés incluent un nid de boucle noyau principal et qu'ils mettent en lumière les capacités d'Apollo. Pour le programme SPMatmat, cinq entrées différentes ont été utilisées afin de mettre en évidence différents aspects d'Apollo.

Les figures 7 et 8 montrent les accélérations obtenues par Apollo par rapport aux programmes séquentiels originaux, lorsque les *code-bones* sont utilisés et lorsque les squelettes de code sont utilisés. Nous les comparons au meilleur temps d'exécution obtenu par le code séquentiel original compilé soit avec GCC-5.3 ou Clang 3.8. Nous considérons la moyenne de cinq exécutions, en utilisant le code cible original compilé avec Clang et le niveau d'optimisation 3 (`-O3`) comme notre référence de base (1× sur l'axe y).

En jaune, nous montrons la meilleure moyenne parmi les codes produits par GCC et Clang, à partir de cinq exécutions du code cible original. En rouge, nous montrons l'accélération moyenne, à partir de cinq exécutions, en utilisant Apollo avec le mécanisme des squelettes de code. En bleu, nous montrons l'accélération moyenne, l'accélération maximale, et l'accélération minimale, obtenues à partir de cinq exécutions, en utilisant Apollo avec le mécanisme des *code-bones*.

En général, l'approche par *code-bones* donne de meilleurs résultats que l'approche des squelettes de code, grâce aux transformations d'optimisation qui ne sont pas supportées par les squelettes disponibles. Lorsque la transformation est appliquée via les deux approches, des performances similaires sont obtenues. Les deux approches obtiennent des accélérations par rapport à la version séquentielle.

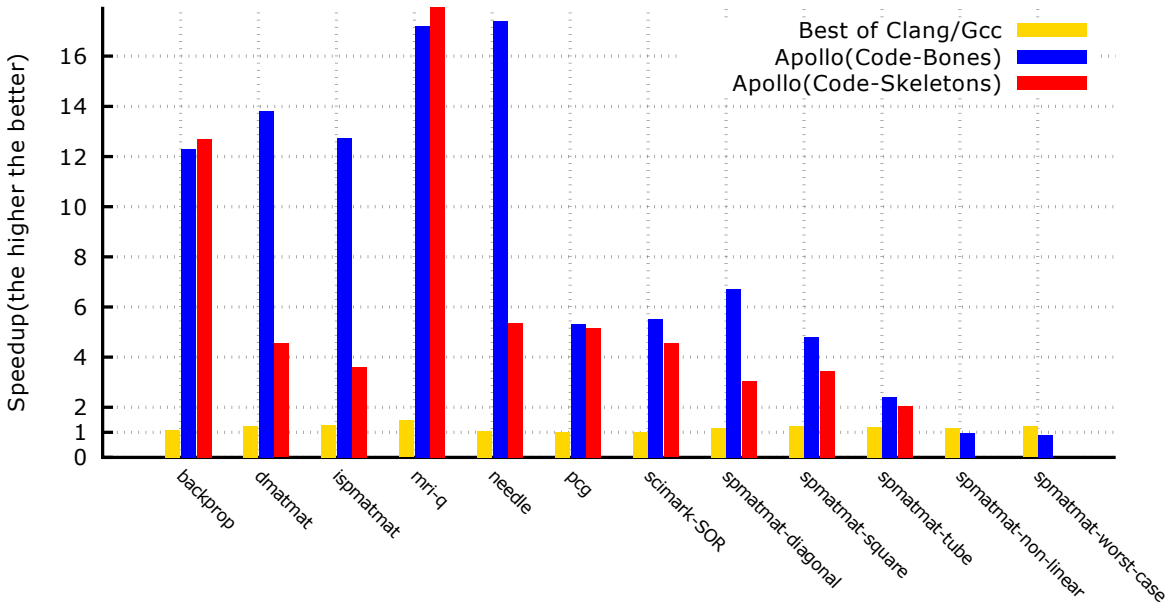

Figure 7: Accélération obtenues sur Lemans avec 24 threads.

La performance obtenue lorsque la matrice d'entrée exhibe des accès mémoire modélisés en tubes est montrée sous le label spmatmat-tube. Avec cette entrée, deux accès mémoire sont modélisés en tubes. Cependant, pendant l'exécution, la plupart des accès se retrouvent en dehors des tubes de prédiction et ainsi reposent sur des vérifications plus complexes. Cela impose un surcoût de vérification important. Une amélioration de performance par rapport à la version séquentielle est obtenue seulement lorsque beaucoup de cœurs sont utilisés. Les deux approches de génération de code obtiennent des performance très similaires, car le goulet d'étranglement est ici le code de vérification, et non pas la bande passante mémoire. Ainsi, l'amélioration procurée par les transformations d'optimisation portant sur la localité des données est insignifiante.

Les deux derniers labels, spmatmat-non-linear et spmatmat-worst-case, exhibent des scénarios où Apollo échoue à optimiser le code. Avec la première entrée, après interpolation et régression, des scalaires de base sont modélisés comme des réductions. Puisque ces scalaires ne sont pas supportés par les outils polyédriques, Apollo recommence l'exécution en utilisant la version originale du code. Pendant toute l'exécution, Apollo alterne entre exécution instrumentée et exécution originale. Avec la deuxième entrée, Apollo réussit à générer du code optimisé, mais dès qu'il est exécuté, une erreur de spéculation est détectée et un recouvrement est signalé.

Sur Lemans, pour les deux entrées, l'exécution finale est plus lente de 20% au pire

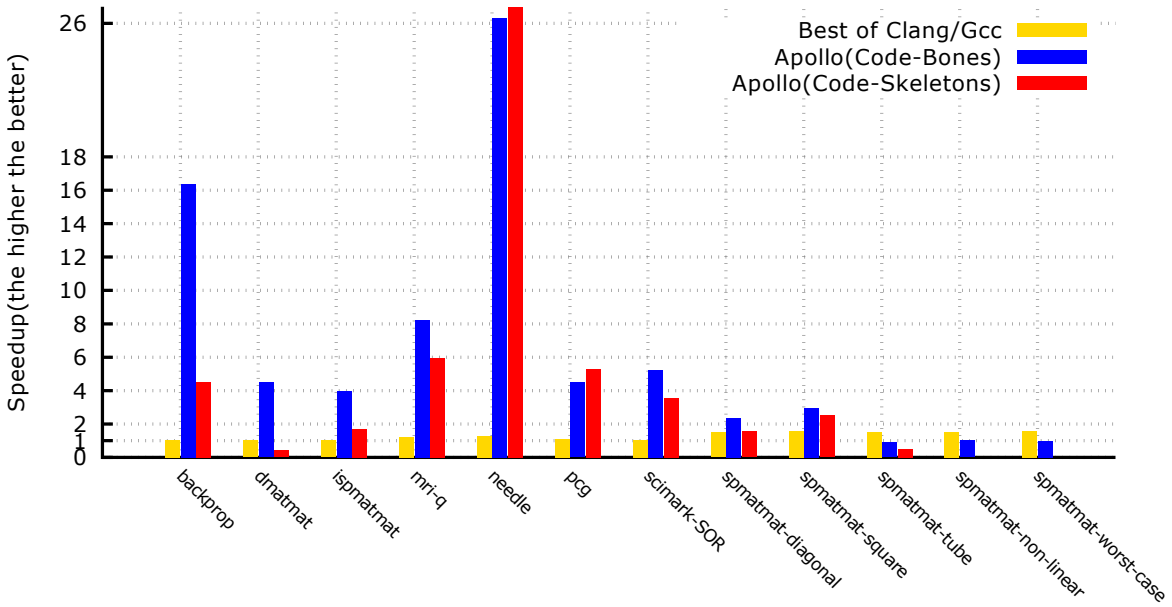

Figure 8: Accélération obtenues sur Armonique avec 8 threads.

cas que le code séquentiel original compilé avec Clang, et plus lent de 10% en moyenne. La version compilée avec GCC outrepasse les autres. Puisque Apollo est construit par dessus Clang, la performance que Apollo peut atteindre reste liée à Clang. Sur Armonique, la performance est très proche au code séquentiel original compilé avec Clang.

Dans les figures 9 et 10, nous montrons les pourcentages du temps d'exécution total passés dans les différentes phases d'Apollo. Ces mesures correspondent à des exécutions en 8 threads. Le temps passé pour la génération de code est montré séparément dans une deuxième colonne, car cette phase s'exécute en parallèle de la version originale du code. C'est pourquoi, pour toutes les expérimentations, le temps passé à exécuter la version originale est bien sûr supérieur au temps utilisé pour la génération de code.

Pour la plupart des expériences, la majorité du temps est pris par les régions parallèles et optimisées du code cible, et seule une petite fraction du temps est consommé par la phase d'instrumentation. Le temps nécessaire à la sauvegarde mémoire et au recouvrement est négligeable par rapport aux autres phases.

Pour le programme `pcg`, une partie importante du temps est prise par la phase de génération de code. Ceci est la conséquence du nombre important de *code-bones* qui sont considérés pour cet exemple. Cependant, du code parallèle est finalement généré et l'exécution optimisées procure malgré tout une amélioration de performance.

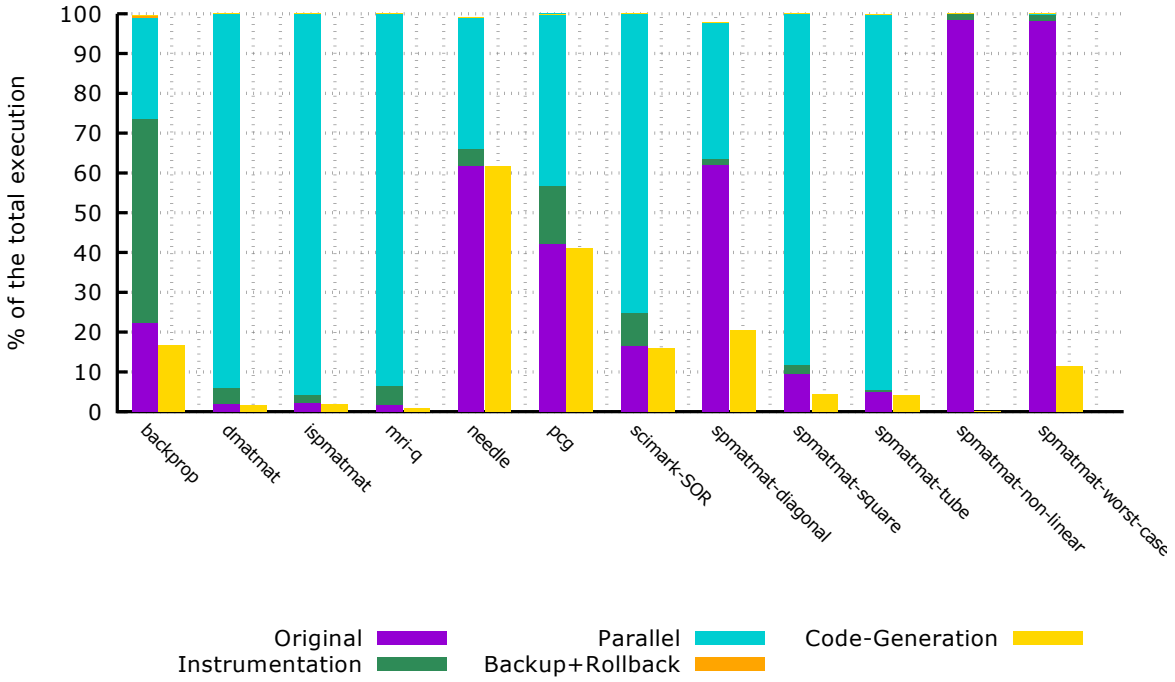

Figure 9: Pourcentages du temps d'exécution total passés dans chaque phase d'Apollo, avec 8 threads, sur Lemans.

Pour spmatmat-diagonal, un temps important est pris pour l'exécution de la version originale du code. Avec cette entrée, le nid de boucle noyau montre deux phases : une première phase de comportement linéaire qui est optimisée avec succès, et une deuxième phase où il est impossible de générer avec succès du code optimisé (il y a un scalaire de base modélisé comme une réduction). Ainsi, la phase finale est accomplie en utilisant la version originale du code.

À la figure 6.25, nous montrons les mesures de temps pour le programme SPMatmat, avec une matrice d'entrée résultant en des accès modélisés en tubes. Dans ce cas, le temps passé dans les régions parallèles est bien plus important que pour les autres phases. Cependant, cela s'explique par la faible performance obtenue avec ce programme, à cause du surcoût temporel important du à la vérification. En fait, le temps pris par la génération de code reste important, ainsi que nous le détaillons dans la section suivante.

Enfin, pour les deux derniers exemples, spmatmat-non-linear et spmatmat-worst-case, la plupart du temps est passé par l'exécution de la version originale du code. Pour le premier, une petite partie du temps sert à la génération de code, qui s'interrompt après l'interpolation/régression. Une exécution optimisée n'est jamais atteinte. Cependant, pour l'autre entrée, qui est une matrice carrée diagonale, la génération de code consomme significativement plus de temps, particulièrement parce que cette phase est

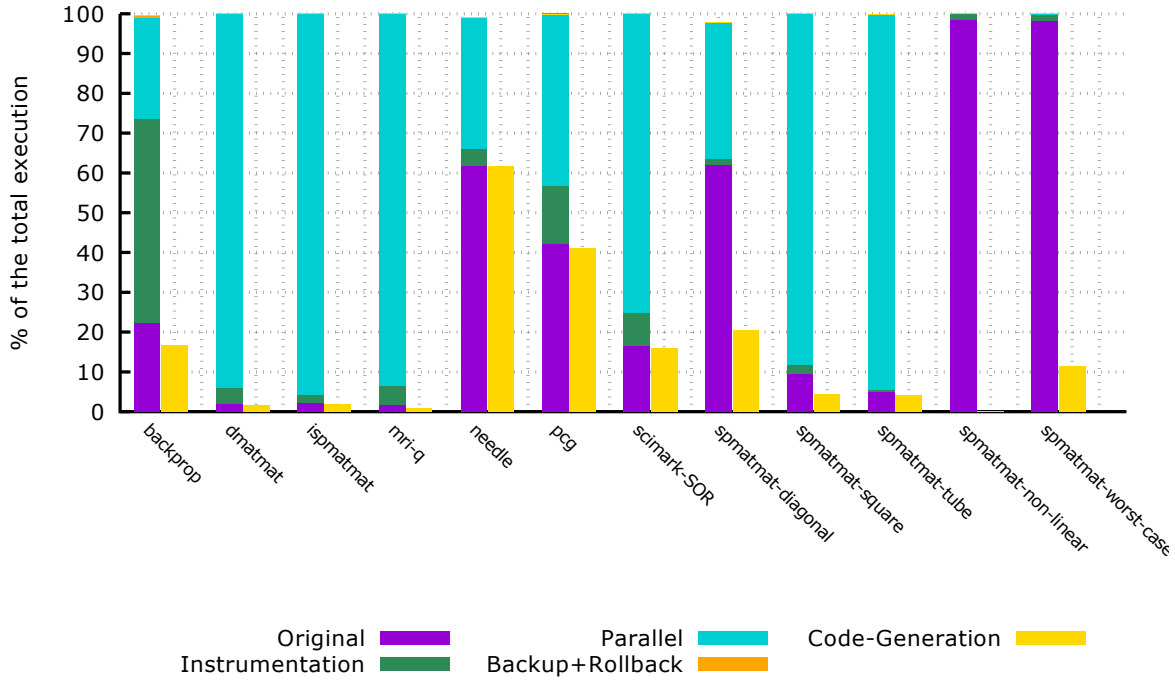

Figure 10: Pourcentages du temps d'exécution total passés dans chaque phase d'Apollo, avec 8 threads, sur Armonique.

exécutée plusieurs fois après chaque exécution du code original. Une très petite partie du temps est passée dans la phase optimisée car, dès le début de l'exécution, une mauvaise spéculation est détectée.

0.5 Conclusions

Nous avons implémenté une plate-forme de parallélisation spéculative appelée Apollo, capable d'optimiser dynamiquement un nid de boucle, dès que celui-ci, au cours de son exécution, exhibe un comportement compatible avec le modèle polyédrique.

La contribution principale de cette thèse est la stratégie de génération de code qui est rapide et flexible. Elle élargit le champ des optimisations possibles dans Apollo. Nous avons appelé notre stratégie *code-bones*.

La précédente stratégie de génération de code d'Apollo, appelée squelettes de code, sacrifiait la flexibilité au profit de la performance. Ainsi, seule un petit sous-ensemble d'optimisations polyédriques était supporté. Notre stratégie étend potentiellement cet ensemble à toute optimisation polyédrique, tout en restant efficace.

Nous avons évalué notre proposition sur plusieurs programmes sur un serveur généraliste x86 et sur une puce embarquée multi-cœur ARM64. Nous avons présenté

des mesures de performance et de surcoût temporel pour chacun des programmes, sur les deux machines. De ces résultats, nous montrons que les *code-bones* surpassent ou égalisent avec leurs compétiteurs principaux, à savoir la stratégie des squelettes de code. En général, les surcoûts restent faibles. Cependant, lorsque beaucoup de *code-bones* participent au nid de boucles cible, notre stratégie fait face à certaines limitations. Mais les résultats sont généralement très prometteurs.

Avec Apollo, les codes dynamiques peuvent désormais bénéficier pleinement du modèle polyédrique.

Chapter 1

Introduction

In the past two decades the landscape of computer hardware changed abruptly. Before that, to satisfy the ever increasing demand for performance, hardware designers relied on increasing the clock frequency of the processor. Thus software performance was boosted with very little programmer or compiler intervention. However, this is no longer possible. Such increase in frequency requires an increase in voltage, which became impossible due to physical constraints. Packing multiple cores into the same processor is the current approach to fulfill the performance demand. Multi-processors already have a rich history in the high performance computing domain; but now they entered into the mainstream and spawn across a wider range of applications. However, to take advantage of such hardware, software must explicitly exhibit parallel regions. These can be marked by the programmer or automatically detected by a compiler. Along with parallelization, exploiting cache memories present in all modern processors became even more important: software executing in multiple cores now requires more memory bandwidth to perform their computations, this enlarging even more the performance gap between memory and processors.

Even if multiple programming languages and extensions exist (OpenMP, TBB, OpenCL, CUDA, for example), parallel programming remains a difficult task. The programmer is required to handle complicated issues such as selecting a proper algorithm, ensuring correct semantics, and being aware of the underlying hardware characteristics. To aid the programmer in this arduous task, compilers dedicated to automatic parallelization were developed. With very little programmer intervention, these compilers are able to extract parallel regions from a serial code. Examples of such optimizers are Pluto or Polly. Traditionally, they focus on for-loops, accessing multidimensional arrays through affine functions of the enclosing loop indices. Such codes can be abstracted using the *Polyhedral Model* [13], which is a mathematical framework used to

reason about loop optimization and parallelization. Codes that adhere to this model benefit from aggressive optimizing transformations.

But what if the code contains while-loops, pointers, or unpredictable conditionals? Such codes are common in general purpose software, but very difficult to optimize. Precise analyses are impossible, thus preventing advanced optimizations and particularly automatic parallelization; even if the dynamic behavior of the code adheres to the Polyhedral Model. Some crucial information can only be known while the target code is executing.

A different approach is taken by *Thread Level Speculation* systems. They optimistically execute regions of the code in parallel, before all the dependencies are known. Hardware and software mechanisms keep track of memory accesses, and if a dependency is violated, a recovery mechanism restores the execution until a previous consistent point. Then, sequential re-execution is initiated. Yet, these approaches had limited success. Most TLS systems use straightforward parallelization schemes, as executing slices of the outermost target loop in parallel. It is impossible to deal with codes where a transformation – such as skewing – is required to exhibit parallelism. They also fail to improve data locality, yielding poor performance in practice. Furthermore, the mechanisms used to detect dependency violations tend to be a bottleneck of such systems, yielding a huge amount of inter-thread communications.

Several factors prevent TLS systems from performing more aggressive optimization. First, very little is done to deal with dependencies, even if they occur in predictable patterns. By being able to predict them, it is then possible to select an optimizing transformation. To instantiate this transformation, a mechanism to generate optimized code on-the-fly, during the program execution, is needed. This mechanism should be able to generate code in a reasonable amount of time; note that, in such a system, any time overhead may harm the benefits from optimization. We distinguish three main approaches:

1. One approach is to trade speed for flexibility by generating multiple optimized binary versions of the code at compile time, at the price of a large fat binary file. This approach limits the number of available transformations to the few precomputed ones.

2. Another approach is to generate the entire optimized code at runtime, which imposes a huge overhead.

3. Finally, another strategy is to generate parametrized *skeletons* of code at compile-time. By setting appropriate values to the parameters of a skeleton, different

transformation can be instantiated.

This last approach, provides an acceptable balance between speed and flexibility, although, it forbids transformations that are not supported by the skeletons that were generated at compile time. A huge number of skeletons is required to support any combination of transformations that alter the structure of loop nests, such as tiling, fusion, fission, or reordering of the statements. Furthermore, some transformations, like unrolling, depend on runtime parameters that cannot be predicted at compile time. In the end, this approach still imposes a limit if one aims to take full advantage of the polyhedral model at runtime.

Apollo is a TLS system able to perform aggressive polyhedral transformations. It features both a prediction model and a fast just-in-time code generation mechanism to successfully optimize codes with pointers, or while-loops. Apollo stands for ***A****utomatic speculative* ***Pol****yhedral* ***L****oop* ***O****ptimizer*. It can optimize codes with very little programmer effort[1]. Apollo is based on a previous prototype framework called VMAD [15].

In this manuscript, we present *Code-Bones*: an ambitious mechanism that supports the runtime generation of code from any polyhedral transformation, while still being fast. Adapting the polyhedral model for speculative parallelization is a tricky task, since all the existing polyhedral model related tools were developed for a compile time usage in mind, and are not well suited for a runtime usage. In this manuscript, we detail the faced related challenges and how we dealt with them.

In parallel with this work, Aravind Sukumaran-Rajam – a former PhD student – worked on extending the mentioned prediction model to support non-linear behaviors [29]. His work was adapted and improved to cooperate with our new code generation mechanism.

The first two Chapters, 2 and 3, introduce briefly the polyhedral model and the thread-level-speculation systems, which are the key concepts for understanding Apollo. Chapter 4 presents how Apollo combines the polyhedral model with thread-level-speculation. *Code-Bones*, the dynamic code generation mechanism of Apollo and main contribution of this thesis, are presented in Chapter 5. Chapters 6 addresses the experimental results of this work. Finally, in Chapter 7, we end this manuscript with conclusions and perspectives about our work.

[1]Fancy to know how Apollo looks from the programmer's point of view? Go to 4.1.

Chapter 2

The Polyhedral Model

Typical compiler's intermediate representations abstract the input code as syntax trees or control-flow-graphs. Such representations have a major drawback: statements appear only once even if they execute several times inside a loop, impeding to reason about individual statement iterations. In contrast, the polyhedral model allows the representation of individual statement iterations as integer points inside a polyhedron, which facilitates the analysis and transformation of loops and loop nests.

Several programming libraries, such as Polylib [24], ISL [34], or CLooG [1], implement functions applying on unions of polyhedra. These functions include:

- set operations (union, difference, intersection...),
- image and preimage with respect to an affine function,
- counting the number of integer points included in a parametric polyhedron (Ehrhart polynomial [11]),
- lexicographic minimum/maximum (parametrically),
- scanning integer points belonging to unions of polyhedra.

In this Chapter, we only present an overview of the polyhedral model. Section 2.1 gives some required basic notions. How a loop nest is represented in this model is addressed in Section 2.2. There are several related tools, among which we briefly introduce those used by Apollo in Section 2.3. Finally in Section 2.4, we conclude by addressing some limitations of static polyhedral compilers that motivate our framework Apollo.

2.1 Definitions

Definition 1 (Affine function)**.** A function $f : \mathbb{K}^m \to \mathbb{K}^n$ is said to be affine iff $\exists$ a matrix $A \in \mathbb{K}^{n \times m}$ and a vector $\vec{b} \in \mathbb{K}^n$ such that:

$$\forall \vec{x} \in \mathbb{K}^m, f(\vec{x}) = A\vec{x} + \vec{b}.$$

Definition 2 (Affine hyperplane)**.** An affine hyperplane of an n-dimensional space $\mathbb{K}^n$ is a subspace of dimension $n-1$, defined by a linear equation $\vec{x} \in \mathbb{K}^n$:

$$\vec{a} \cdot \vec{x} = b,$$

where $\vec{a} \in \mathbb{K}^n$ $(\vec{a} \neq 0)$ and $b \in \mathbb{K}$.

Definition 3 (Affine half-space)**.** An affine hyperplane of equation $\vec{a} \cdot \vec{x} = b$ divides the space into two half-spaces, defined by the inequalities:

$$\vec{a} \cdot \vec{x} \geq b \text{ and } \vec{a} \cdot \vec{x} \leq b,$$

where $\vec{a} \in \mathbb{K}^n$ $(\vec{a} \neq 0)$ and $b \in \mathbb{K}$.

Definition 4 (Convex Polyhedron)**.** The intersection of a finite number of affine half-spaces defines a convex polyhedron, each half-space providing a face of the polyhedron. Formally, the polyhedron $P \subset \mathbb{K}^n$ can be expressed as a set of m affine constraints in a matrix $A \in \mathbb{K}^{m \times n}$ and a vector $\vec{b} \in \mathbb{K}^m$:

$$P = \{\vec{x} \in \mathbb{K}^n | A\vec{x} + \vec{b} \geq 0\}.$$

Definition 5 (Parametric Polyhedron)**.** A parametric polyhedron is denoted by $P(\vec{p})$, and parametrized by the vector $\vec{p} \in \mathbb{K}^p$. It can be defined by a matrix $A \in \mathbb{K}^{m \times n}$, a matrix of symbolic coefficients $B \in \mathbb{K}^{m \times p}$, where p is the dimension of the vector of parameters p and a vector of constants $\vec{b} \in \mathbb{K}^m$ as:

$$P(\vec{p}) = \{\vec{x} \in \mathbb{K}^n | A\vec{x} + B\vec{p} + \vec{b} \geq 0\}.$$

2.2 Polyhedral representation of a loop nest

The code in Listing 2.1 implements the multiplication of two $n \times n$ matrices. This example is used several times in this Chapter for illustration purposes.

```
for(i = 0; i < n; ++i) {
    for(j = 0; j < n; ++j) {
        C[i][j] = 0.0; //S1
        for(k = 0; k < n; ++k)
            C[i][j] += A[i][k] * B[k][j]; //S2
    }
}
```

Listing 2.1: Matrix-Matrix multiplication kernel

As mentioned before, the polyhedral model allows the representation of individual statement iterations. A statement can be a single instruction or multiple consecutive instructions. The polyhedral model does not handle the semantics of the instructions inside each statement, only its memory references. In Listing 2.1, there are two statements – $S1$ and $S2$ –, and three for-loops surrounding these statements. The dynamic instances of a statement correspond to the possible combinations of values for the surrounding iterators (i and j for S1; i, j and k for S2). In this example, for $S1$: $S1_{0,0}$, $S1_{0,1}$, ..., $S1_{0,n-1}$, ..., and $S1_{n-1,n-1}$; similarly, for $S2$: $S2_{0,0,0}$, $S2_{0,0,1}$, ..., $S2_{0,0,n-1}$, ..., and $S2_{n-1,n-1,n-1}$.

Definition 6 (Iteration vector)**.** The iteration vector of a statement S is the n-dimensional vector of values of the iterators of the n loops enclosing S.

Vectors $(0,0)$, $(0,1)$ and $(n-1,n-1)$, for example, are possible iteration vectors for $S1$.

The polyhedral model represents statements in loop nests as a combination of three essential constructs: the iteration domain, the schedule function and the access functions. In the following subsections (2.2.1, 2.2.3 and 2.2.2), we detail these components. In subsection 2.2.4, we also describe the dependency polyhedron, which is crucial for choosing a transformation.

2.2.1 Iteration Domain

The Polyhedral Model allows to represent the dynamic instances of each statement of a loop nest. These instances are captured by the *iteration domain*, the first construct detailed in this section.

Definition 7 (Iteration Domain)**.** The set of all iteration vectors for a statement S is called the iteration domain, and it is denoted by $\mathcal{D}^S$.

$$\begin{cases} i & \geq 0 \\ j & \geq 0 \\ n-1-i & \geq 0 \\ n-1-j & \geq 0 \end{cases}$$

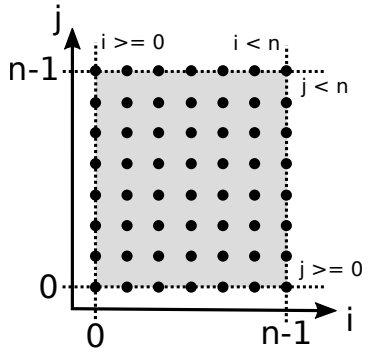

Figure 2.1: Domain constraints for $S1$ (left). Graphical representation of its iteration domain (right).

$$\mathcal{D}^{S1}(N) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2 \middle| \left(\begin{array}{cc|c|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 1 & -1 \\ 0 & -1 & 1 & -1 \end{array} \right) * \left(\begin{array}{c} i \\ j \\ \hline N \\ \hline 1 \end{array} \right) \geq 0 \right\}$$

Figure 2.2: Iteration domain for statement $S1$.

Consider the statement $S1$ in the previous example. The iteration domain is expressed as a system of linear constraints and may be represented graphically, as shown in Figure 2.1. Each integer point in the graphical representation stands for an instance of the statement. The iteration domain is expressed as a parametric polyhedron in Figure 2.2.

The polyhedral representation is possible only if the loop bounds can be expressed as affine functions of the enclosing loop indices, and if the guards of all the conditionals are affine inequalities of the enclosing loop indices. If this is the case, it is possible to express the domain as a set of affine constraints, which have a simple polyhedral representation. For a statement S, the iteration vector $\vec{x}$ is defined in $\mathbb{Z}^d$, where d is the depth of the innermost loop enclosing S; the vector $\vec{p}$ stands for the vector containing the p parameters of the loop nest. Let n be the number of affine constraints; the set of constraints is encoded in the matrices $A \in \mathbb{Z}^{n \times d}$, $B \in \mathbb{Z}^{n \times p}$ and the vector $\vec{b} \in \mathbb{Z}^n$:

$$\mathcal{D}^S(\vec{p}) = \left\{ \vec{x} \in \mathbb{Z}^d \middle| A\vec{x} + B\vec{p} + \vec{b} \geq 0 \right\}$$

2.2.2 Access Function

It is not only important which instances of a statement are executed – encoded in the iteration domain – but also which are the memory locations accessed by each instance. This is the key for performing precise dependency analysis. In the polyhedral model, referenced memory addresses are encoded as accesses to multi-dimensional

arrays through linear functions of the enclosing loop iterators.

Definition 8 (Access Function)**.** For a statement S at depth d accessing a m-dimensional array, its access function is defined as:

$$f(\vec{x}) = M\vec{x} + \vec{m},$$

with $M \in \mathbb{Z}^{d \times m}$.

The access function maps each point of the iteration domain with an array access. Continuing with our example, $S1$ contains one memory write to C[i][j] encoded as follows:

$$f^{S1}_{W,C[i][j]}(i,j,N) = \left(\begin{array}{cc|c|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{array}\right) * \left(\begin{array}{c} i \\ j \\ \hline N \\ \hline 1 \end{array}\right) = \begin{pmatrix} i \\ j \end{pmatrix}$$

Similarly, $S2$ contains four memory accesses, a read and a write to C[i][j], one read from A[i][k] and another from B[k][j]:

$$f^{S2}_{R,C[i][j]}(i,j,k,N) = f^{S2}_{W,C[i][j]}(i,j,k,N) = \left(\begin{array}{ccc|c|c} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{array}\right) * \left(\begin{array}{c} i \\ j \\ k \\ \hline N \\ \hline 1 \end{array}\right) = \begin{pmatrix} i \\ j \end{pmatrix}$$

$$f^{S2}_{R,A[i][k]}(i,j,k,N) = \left(\begin{array}{ccc|c|c} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{array}\right) * \left(\begin{array}{c} i \\ j \\ k \\ \hline N \\ \hline 1 \end{array}\right) = \begin{pmatrix} i \\ k \end{pmatrix}$$

$$f^{S2}_{R,B[k][j]}(i,j,k,N) = \left(\begin{array}{ccc|c|c} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{array}\right) * \left(\begin{array}{c} i \\ j \\ k \\ \hline N \\ \hline 1 \end{array}\right) = \begin{pmatrix} k \\ j \end{pmatrix}$$

2.2.3 Scheduling Function

Loop transformations [35], such as interchange or skewing, consist of modifying the execution order of the statement instances into another one that exhibits parallelism

or improves data locality. The iteration domain and the access functions are not enough to express this change of the execution order; what misses is a way to express the execution order between instances of statements. To determine this order, a *time-stamp* is associated to each statement instance such that if the time-stamp of one statement is greater than the time-stamp of another, the latter executes first. The *scheduling function* assigns a multidimensional time-stamp to each statement instance.

Definition 9 (Scheduling Function)**.** The *scheduling function* of a statement S, known also as the *schedule* of S, is a function that maps each dynamic instance of S to a time-stamp, expressing the execution order between the statements:

$$\forall \vec{x} \in \mathcal{D}^S, \theta^S(\vec{x}) = T\vec{x} + \vec{t}.$$

The associated time stamp allows to order the instances of the statements according to the lexicographical order, denoted as $\ll$:

$$(a_1, \ldots, a_n) \ll (b_1, \ldots, b_m) \iff \exists i, 1 \leq i \leq min(n, m), \forall k, 1 \leq k < i, a_k = b_k \wedge a_i < b_i.$$

In contrast to the iteration domain, the scheduling function can also express the textual ordering of the statements. For our matrix multiplication example, the scheduling functions are:

$$\theta^{S_1}(i, j) = (0, i, 0, j, 0)$$
$$\theta^{S_2}(i, j, k) = (0, i, 0, j, 1, k, 0)$$

For both statements, the iteration domain does not capture which of these statements executes first, regarding their relative positions in the program. The scheduling function captures this by interleaving constants between the iterators, to express their textual order. The two first constants indicate that the statements reside in the same couple of outer loops, while the third indicates that for a given pair (i, j), all instances of S_2 are executed before any instance of S_1.

Cohen *et. al.* [12, 4] have proposed a normalized representation of this function. The scheduling function θ^S of a statement S (at depth d and with p parameters) is represented as a matrix with $d + p + 1$ columns and with one extra level for each loop level (so $2d + 1$ rows). This representation is useful for expressing composition of transformations, as it is decomposable into three sub-matrices:

- The iteration ordering matrix: $A^S \in \mathbb{Z}^{d \times d}$, representing the iteration vectors.
- The matrix of parameters: $\Gamma^S \in \mathbb{Z}^{d \times p+1}$, representing the global variables.

- The statement ordering vector: $\beta^S \in \mathbb{Z}^{d+1}$, which represents the textual position of S.

$$\theta^S(\vec{xp}) = \left[\begin{array}{c|c|c} 0\dots0 & 0\dots0 & \beta_1 \\ A^S_{1,1}\dots A^S_{1,d} & \Gamma^S_{1,1}\dots\Gamma^S_{1,p} & \Gamma^S_{1,p+1} \\ 0\dots0 & 0\dots0 & \beta_2 \\ A^S_{2,1}\dots A^S_{2,d} & \Gamma^S_{2,1}\dots\Gamma^S_{2,p} & \Gamma^S_{2,p+1} \\ \vdots\ddots\vdots & \vdots\ddots\vdots & \vdots \\ A^S_{d,1}\dots A^S_{d,d} & \Gamma^S_{d,1}\dots\Gamma^S_{d,p} & \Gamma^S_{d,p+1} \\ 0\dots0 & 0\dots0 & \beta_{d+1} \end{array}\right] * \left(\begin{array}{c} x_1 \\ \vdots \\ x_d \\ \hline p_1 \\ \vdots \\ p_p \\ \hline 1 \end{array}\right)$$

In this way, various transformations can be expressed directly: for example, loop interchange or skewing by modifying A^S; by modifying Γ^S, one generates loop shifting transformations; and by modifying β^S, one can reorder the statements, apply fusion or fission.

To continue with our example, a valid polyhedral transformation is to separate both statements into different nests (loop fission), and to do a loop interchange between iterators j and k for statement $S2$. Both outermost loop nests are marked as parallel. The resulting schedule and associated code are shown in Figure 2.3. Notice how vector B^{S2} has been modified to express the loop fission; and how matrix A^{S2} is used to express the interchange.

2.2.4 Dependency Polyhedron

Loop transformations must be valid regarding dependencies that are defined by the original loop nest, to ensure that the new code preserves the original semantics.

Definition 10 (Dependency between statements)**.** Two statements S and R are said to be dependent, if there exists two instances $S(\vec{x_S})$ and $R(\vec{x_R})$, where $\vec{x_S}$ and $\vec{x_R}$ belong respectively to the iteration domain of S and R, and are such that $S(\vec{x_S})$ and $R(\vec{x_S})$ access the same memory location and at least one access is a write. Notice that S and R may reference the same statement.

If in the original execution order $S(\vec{x_S})$ executes before $R(\vec{x_R})$, R is said to be dependent on S. S is called the *source* and R the *target* of the dependency.

To preserve the original semantics, the execution order of two dependent statements in the transformed parallel execution must be the same as in the original sequential

$$A^{S1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \Gamma^{S1} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \beta^{S1} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$A^{S2} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \Gamma^{S2} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \beta^{S2} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

```
#pragma omp parallel for
for(i = 0; i < n; ++i)
    for(j = 0; j < n; ++j)
        C[i][j] = 0.0; //S1
#pragma omp parallel for
for(i = 0; i < n; ++i)
    for(k = 0; k < n; ++k)
        for(j = 0; j < n; ++j)
            C[i][j] += A[i][k] * B[k][j]; //S2
```

Figure 2.3: Optimized and parallelized Matrix-Matrix multiplication kernel. New scheduling function (Top) and generated code associated to the new scheduling function (Bottom).

execution. On the other hand, two independent statements can be executed in any order, and hence in parallel.

Dependencies can be classified in three categories:

- **Read-after-Write**, abbreviated RAW, or flow-dependency.
- **Write-after-Read**, abbreviated WAR, or anti-dependency.
- **Write-after-Write**, abbreviated WAW, or output-dependency.

Read-after-Read dependencies are not considered regarding program semantics, since the memory is not altered through such a dependency. Thus, any order of execution of the statements is allowed. However, it is still worth considering them to improve data locality. Various algorithms have been developed to remove WAR and WAW dependencies, such as: privatization, renaming or expansion. These strategies try to relax the constraints imposed by these dependencies, thus enabling more optimization opportunities.

The dependency relationships can be expressed as a graph, called the *dependency graph*. Intuitively, there is one vertex for each statement in this graph. An edge exists between two vertices if there is a dependency between them. For each edge e of this graph, the exact dynamic instances that are dependent can be expressed as a polyhedron, called the *dependency polyhedron* $\mathcal{P}^e$. The details regarding the dependency

polyhedron exceed this introduction to the polyhedral model. More information can be found in [13].

2.3 Polyhedral software tools

One benefit of using the polyhedral model as a program representation is to take advantage of several libraries implementing related operations. For our software developments, we used the following libraries.

Polylib [24] is a polyhedral library that implements numerous functions on parametric polyhedra, in particular, unions, intersections, images, pre-images, computing the vertices of a polyhedron, etc. Also, it can be used to compute the Ehrhart polynomials, for counting integer points inside a parametric polyhedron.

OpenScop [2] is a portable format for polyhedral representation of loop nests. It simplifies exchanges among polyhedral tools that use this format.

Candl [3] is a library devoted to dependency analysis. For a loop nest, encoded in the OpenScop representation, it is able to compute the dependency graph and the associated dependency polyhedra.

Pluto [5] is an automatic loop optimizer based on the polyhedral model. It accepts as input an OpenScop representation of a loop nest, and supports a wide range of loop transformations, such as: fusion, fission, interchange, skewing, tiling, unrolling, etc.

CLooG [1] is used to generate the scanning code that iterates over each point of a union of polyhedra. Additionally, it embeds many optimizations that aim to reduce the control overhead of the generated code. It also accepts a polyhedral representation in the OpenScop format. It is embedded in the well-known open-source compiler GCC for its advanced loop optimization features.

2.4 Limitations

Even if powerful, the polyhedral model is restricted to a small set of compute-intensive codes consisting of linear for-loops accessing static multi-dimensional arrays through linear functions. Indirections, pointers, while loops, unknown bounds and multiple

```
for( row = 1; row <= left->Size; row++ ) {
    pElement = left->FirstInRow[row];
    while( pElement ) {
        for( col = 1; col <= cols; col++ ) {
            result[row][col] +=
                pElement->Real * right[pElement->Col][col];
        }
        pElement = pElement->NextInRow;
    }
}
```

Listing 2.2: Sparse Matrix-Matrix multiplication kernel

exits are common constructs occurring into general purpose codes. Unfortunately, they hamper precise static analyses and disqualify the polyhedral model.

However, even if they do not exhibit the convenient polyhedral static properties, some codes may still exhibit phases whose memory behavior is compliant with the polyhedral model during their execution. The code kernel in Listing 2.2 implements a sparse matrix multiplication. The while loop of the nest traverses a linked list, where each element represents a row of a matrix. At compile time, it is impossible to know what memory locations will be touched by that loop or how many iterations will be executed.

However, for some inputs – a diagonal matrix or a square sub-matrix, for example – it is possible that the actual behavior of the loop agrees with the polyhedral model. In Figure 2.4, accessed memory addresses and their interpolating plane, for a given memory instruction, are shown. It highlights a polyhedral compatible behavior, at least for these few iterations.

To detect such phases and then to optimize them, the polyhedral model has to be adapted to a runtime usage. In chapter 4, we explain in detail how this is achieved by our framework Apollo.

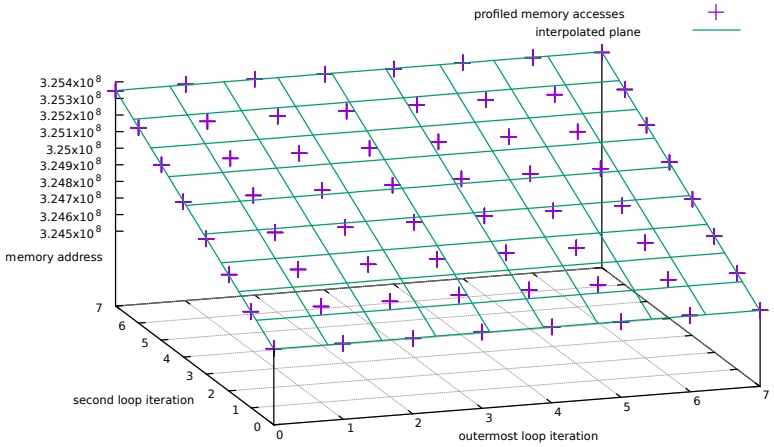

Figure 2.4: Observed addresses and interpolating plane, for one memory operation accessing nodes of the linked list, in the sparse matrix-matrix multiplication example.

Chapter 3

Thread-Level Speculation

Compilers performing automatic parallelization are generally hampered by the limited information available at compile time, while some key properties are only accessible at runtime. This is often the case for codes using pointers, indirections, while-loops, etc... In this case, only sequential code is generated to guarantee a semantically correct execution.

A better solution is to optimistically execute regions of code (or iterations of a loop) in parallel, assuming that no dependencies ever occur between them. While these regions execute, an underlying mechanism monitors every memory access to ensure the same semantics as the sequential execution. If a data race occurs, all the offending threads are halted, and a rollback is initiated. This technique is called Speculative Parallelization or Thread-Level Speculation (TLS).

This chapter describes thread-level speculation. In Section 3.1, we exhibit a motivating example, to highlight the importance of these systems. In Section 3.2, we explain how a traditional TLS system works. We conclude with the limitations of such systems in Section 3.3.

3.1 Motivation

TLS systems may successfully optimize codes when all the dependencies cannot be known at compile time. As an example, consider the code in Listing 3.1. This code contains two memory accesses to array A using two indirections, a store to A[b[i]] and a load from A[c[i]].

A static compiler cannot be able to disambiguate the possible dependencies between the store and the load, thus it conservatively preserves the sequential execution order. In contrast, a TLS system optimistically executes iterations in parallel.

```
for(i = 0; i < N; ++i) {
    A[b[i]] = ...
    ... = A[c[i]]
}
```

Listing 3.1: Code fragment to be parallelized.

Figure 3.1 shows two execution traces for the previous code. The first trace corresponds to the sequential execution, while the second one is the parallel execution launched by a TLS system.

For the speculative execution, a first set of iterations is distributed to 4 available threads. During the execution of this set, all the threads access disjoint regions of memory, hence there is no violation of dependencies. Once completed, the changes performed by all the threads are actually written (*commit*) to array A. Then, a second set of iterations is distributed to the threads.

During the execution of this second set of iterations, threads 3 and 4 access the same element of the array (A[9]). First, thread 4 stores a value at this position, then thread 3 loads a value from it. This violates a Write-After-Read dependency defined by the original iteration order (See the sequential execution in Figure 3.1). Threads 1 and 2 are still allowed to write their modifications to memory; but the other threads are discarded (*rollback*) upon data race detection. To cope with this dependency issue, the execution resumes sequentially from the faulty iteration.

3.2 Overview

In Figure 3.2, we depict the main stages performed by a TLS system. Most systems have two stages: a compile time and a runtime stage. During compilation, relevant properties of the code are extracted, such as data dependencies that can be disambiguated without runtime information. Some systems also include a preliminary offline profiling step in order to provide some dynamic information to the compiler, provided that the same behavior will occur during the actual execution. Most TLS systems do not consider any transformation other than slicing the outermost loop into parallel slices. Therefore, the used code generation mechanism tends to be very simple. They statically generate code snippets dedicated to profiling, re-execution and speculative execution. In contrast, Apollo performs more aggressive loop transformations, such as tiling and fission, that require an advanced and fast mechanism to generate transformed code dynamically.

At runtime, the target code is profiled on some execution samples in order to capture

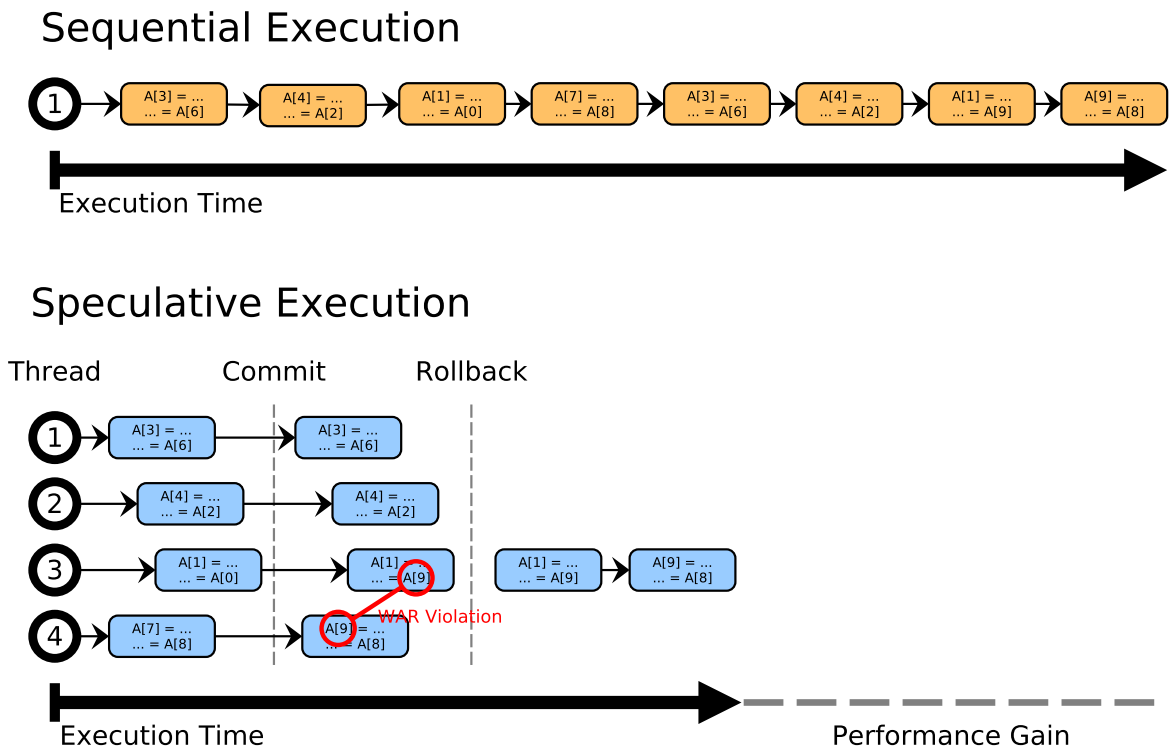

Figure 3.1: Sequential execution (top) and Speculative execution (bottom).

information relevant for parallelization. Classically, dynamic dependency analysis is performed. In our approach, memory accesses and loop trip counts are registered, then linear equalities and inequalities are built from linear interpolation and regression. These (in)equalities predict the values that memory access addresses, loop trip counts and iterators, will take in the rest of the execution. Additionally, our approach uses this information to select an advanced and optimizing code transformation.

Later, during the parallel execution, the speculation is verified in order to ensure correct semantics. Such verification usually consists of detecting conflicting accesses to the referenced memory locations. In our proposal, this is achieved by comparing each actual memory access against the predicted (in)equalities.

In order to recover from a misspeculation, stores are performed in a private buffer (*deferred updates*). Later, if the execution is validated, the data is written to the global memory (*commit*). Another approach is to save a copy of the memory write region before the stores are executed (*backup*), and then store directly in the main memory (*eager updates*).

In case of misspeculation, the incorrect computations have to be canceled and the memory restored to a correct state. In a system implementing deferred updates, one has to simply discard the private write buffers of the faulty threads; if using eager

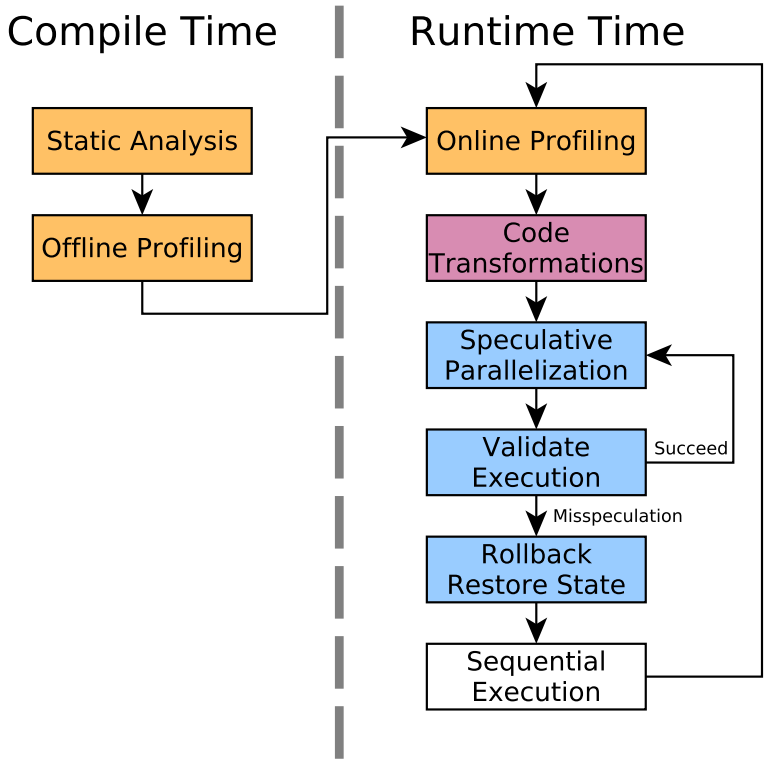

Figure 3.2: Overview of the different stages of a TLS system.

updates, previously backed-up memory has to be copied back to global memory. Then, execution can be resumed. Usually, this is achieved by sequentially re-executing the faulty threads for which a conflicting memory access has been detected. To be efficient, still, the number of misspeculations must remain small.

3.2.1 State of the art

There are two main types of TLS systems: hardware-based and software-based. Typically, hardware-based approaches exhibit lower time overheads and better energy efficiency. However, the available systems have not fully evolved to handle general purpose programs. Apollo is software-based, does not rely on any particular hardware, and can be used in any general purpose CPU, as ARM64 or x86-64. In the following, we only address software based TLS systems, since Apollo is one of them.

POSH [20] is a compilation framework for transforming the target program into a TLS compatible version. The framework also includes a profiler, which tries to identify the tasks where speculative parallelization will not be beneficial due to misspeculation. The LRPD test [26] speculatively parallelizes for-all loops, and performs runtime detection of memory dependencies. However, this technique is limited to array accesses

where array bounds are known at compile time.

Softspec [6] implements some fundamentally similar ideas to Apollo. It is a TLS system that attempts to parallelize for-loops with stride-predictable (incremented by a constant step) memory accesses. This work focuses only on innermost loops. Hence, one-variable interpolating functions are built and used for simple dynamic dependency analysis via the GCD-test. In this work, multiple versions of the original code are generated statically. Among these code versions, there is one for profiling and one for speculative execution.

The VMAD framework [15, 18] is the TLS system that sets the bases for Apollo. Like Apollo, it is also based on the polyhedral model. The system consists of two main modules, (i) a static module and (ii) a runtime module.

The static module consists of a set of LLVM passes and operates on the LLVM IR (Intermediate Representation). The user marks the loop nests of interest using a pragma. For each such marked loop nest, the static module generates a set of code skeletons, each supporting a different set of transformations. An instrumented version of the code is also generated which includes callbacks to the runtime system to communicate the memory addresses accessed and the enclosing iterator values. At runtime, the outermost loop is executed by chunks of iterations, and the instrumented version is selected for profiling the first chunk. Based on the dynamic code behavior, the system selects one polyhedral transformation and instantiates one of the statically generated code skeletons. During the run of the optimized code, the verification mechanism ensures that the speculative model still holds. If the speculation succeeds, the safe state is updated, and the execution continues for the rest of the chunks. If the system detects any violation, a rollback is initiated and thus the safe state is restored. This is followed by the execution of the original sequential code and profiling.

When compared to other TLS systems, VMAD is capable of performing advanced loop transformations by using the polyhedral model. Our framework, Apollo, can be considered as the successor of VMAD. One major difference between Apollo and VMAD, is how the transformation is selected. In VMAD, the compiler proposes a set of code transformations statically. At runtime, when the dependencies are resolved, the first valid transformation from the proposed list is selected. A major drawback is that since the system is targeting dynamic codes, the exact code behavior can only be known at runtime; hence proposing a set of transformations statically is difficult and moreover, the size of this set may itself be high. In addition to this, selecting the first valid transformation, is a sub-optimal solution; there may exist another valid transformation in the proposed list which offers better performance. VMAD was designed as a prototype and thus has a lot of practical issues on both performance and codes

that can be handled. Though the system is designed to support imperfect loop nest(s), in practice, due to the usage of distance vectors and other implementation issues, the number of code kernels that could be handled was limited. Our system overcomes these limitations by using the Pluto compiler dynamically; from a linear prediction model, a polyhedral representation is constructed, and is fed to Pluto to get a valid optimizing transformation. Also, VMAD relied on binary skeletons for optimized code generation. Instead, our system can use bitcode-skeletons, to be instantiated by a Just-In-Time compiler enabling further optimizations, or Code-Bones, the technique presented in this manuscript that greatly increases the number of supported polyhedral transformations. Finally, Apollo is also capable of handling non linear memory accesses and loop bounds using a generic extension of the polyhedral model.

Design details on Apollo are presented in Chapters 4 and 5.

Inspector/Executor

Earlier work on runtime parallelization [25] involved a technique known as *inspector/executor*. As the name implies, this technique involves two processes: (i) inspector and (ii) executor. The inspector can be automatically generated, if the source loop can be distributed into a loop computing the memory addresses that will be accessed and another loop which actually accesses these addresses (i.e., when the address computation and data computation are not contained in the same strongly connected component of the dependency graph). The inspector monitors the execution and collects the memory addresses that will be accessed. From these addresses, data dependencies are computed. The scheduler then finds iteration wise dependency relations and constructs a Directed Acyclic Graph (DAG) in which the vertices denote the statements and the edges denote the dependencies between the statements. A cost model determines whether the loop parallelization is profitable. If possible and profitable, the scheduler then parallelizes the code and the executor executes the code.

A similar strategy, which is one of the contributions of this thesis, was implemented in Apollo to save the overhead of performing a memory backup (discussed in Section 4.8). Two code versions are generated: a verifier and an executor. The verifier contains verification instructions that do not participate in dependencies. These instructions are used to partially validate the prediction model for a given chunk. Normally, this code executes faster than the executor counterpart. Before running each optimized chunk, instead of performing a memory backup, the verifier is first used to validate the prediction model. If it succeeds, then the backup is not performed for the data related to the verified prediction and the executor is launched. In many cases, it is even

```
for(i = 0; i < N; ++i)
    for(j = 0; j < N; ++j)
        A[b[i][j]] = A[b[i][j-1]] + A[b[i-1][j]]
```

Listing 3.2: Small kernel that requires a transformation to become parallel.

possible to completely avoid the backup using such a scheme. If the verifier detects a misspeculation, the chunk is executed using the original version of the code (notice that the rollback phase is not required in this context).

3.3 Limitations

Despite many improvements, little success has been reported from TLS systems. Their performance remains limited for several reasons.

Missed parallelization opportunities Most TLS systems do not perform any transformation, and simply execute a loop in parallel slices. However, some codes *need* a transformation to become parallel. In the code in Listing 3.2, if each position of array 'b' is initialized as b[i][j] = N*i+j, a skewing transformation is required to parallelize the innermost loop of the nest. Apollo is able to overcome this limitation by using a model of the execution to predict the array elements that will be accessed, and by performing powerful polyhedral transformations.

Data locality is not addressed Traditional TLS systems focus on parallelization and ignore data locality optimizations. This becomes a bigger problem as the number of CPU cores increases, leading to poor scalability. In contrast, Apollo considers data-locality and parallelization when choosing a transformation, leading to outstanding performance results. Consider again the example in Listing 2.2; other TLS systems could successfully parallelize the outermost loop of this code, but to achieve better performance, a tiling transformation, which improves data locality, is required.

Misspeculation detection Most TLS systems require their threads to communicate between each other, to detect data races. This communication easily becomes a bottleneck. Even worse, the amount of communicated data can be huge, proportional to the number of iterations and the number of memory accesses. This data exchange, required for verification, competes for the memory bandwidth with the exchanges performed by the actual computation. In contrast, our approach uses a model of the execution. Each instance of every memory access and loop bound must be verified

against this model. Only local data (such as constants, local variables, and iterators) is used to perform the verification of, for example, a memory access. Like this, the communication between threads remains small, only performed to abort execution in case of misspeculation.

In the next Chapter, we introduce Apollo, our framework for speculative and polyhedral parallelization, and detail how our framework solves all these above mentioned issues.

Chapter 4

Apollo: When Polyhedra meet Speculation

Traditionally, the polyhedral model fails to handle codes with indirections, memory accesses through pointers, or unpredictable loop bounds, which are common in general purpose codes. However, such codes may exhibit phases that are compliant with the polyhedral model: bounds and accessed addresses take values matching affine functions of the surrounding loop iterators. Compilers performing static optimization cannot exploit such behaviors, but a runtime compilation framework can detect and take advantage of such phases. Apollo is a hybrid compilation framework that succeeds in doing so.

During the target program's execution, Apollo monitors the behavior to detect these polyhedral compliant phases. Even if such a phase has been detected, one cannot ensure that future iterations will behave the same as the previously executed iterations. This is the cornerstone of the speculation. Apollo builds a model that predicts how future iterations should behave, then it determines a transformation and generates optimized code based on this model. The selected transformation – and the generated code – remains valid as long as the executed iterations agree with the prediction model; if a misspeculation occurs, a rollback is performed and the code is re-executed. All these mechanisms that are required for speculative code optimization impose an overhead, which is hazardous in any dynamic system. However, performance improvements obtained from parallelization and data locality improvement can largely outweigh this overhead. Apollo is, to our knowledge, the unique framework able to unleash the full power of the polyhedral model at runtime.

In this Chapter, we present the Apollo framework. In Sections 4.1 and 4.2, we describe the compile time component of Apollo. Section 4.3 explains the different

```
#pragma apollo dcop
{
    for( row = 1; row <= left->Size; row++ ) {
        pElement = left->FirstInRow[row];
        while( pElement ) {
            ...
        }
    }
}
```

Listing 4.1: Example of the #pragma directive.

```
apollo-static -O3 spmatmat.c -Wl,-E -lruntime -o spmatmat.bin
apollo-static -O3 -S -emit-llvm spmatmat.c -o spmatmat.ll
```

Listing 4.2: Usage of Apollo's static component.

phases punctuating the execution of Apollo. In Section 4.4, we describe the profiling phase. In Sections 4.5 and 4.6, we introduce the *prediction model*, how it is built, and how it is used to select a transformation. The basic ideas about code generation inside Apollo are given in Section 4.7. In this Section, the Code-Bones and Code-Skeletons are presented. The first is the main contribution of this manuscript, while the second is its major competitor. Before any speculative execution of code, several checks and tasks are performed to guarantee that a recovery will be possible in case of misspeculation. These are presented in Section 4.8. Finally, in Section 4.9, some concluding remarks are given.

4.1 Only a #pragma

To use Apollo, the programmer has to enclose the targeted loop nests using a dedicated #pragma directive (shown in Listing 4.1). Inside the #pragma, any kind of loops are allowed, for-loops, while-loops, do-while-loops or goto-loops.

Then, the programmer compiles the code using a specialized compiler. This compiler is an alias over the clang compiler that also loads appropriate passes of Apollo. Two typical commands are shown in Listing 4.2: the first showing how to compile a source code with Apollo and linking it with its related runtime system; the second one is used to obtain the LLVM intermediate representation of the code resulting from Apollo's static transformations.

In Figure 4.1, we depict an overview of the framework. At compile time, the programmer compiles his annotated source code with Apollo. Then, the runtime system of Apollo orchestrates the execution and dynamically optimizes the code.

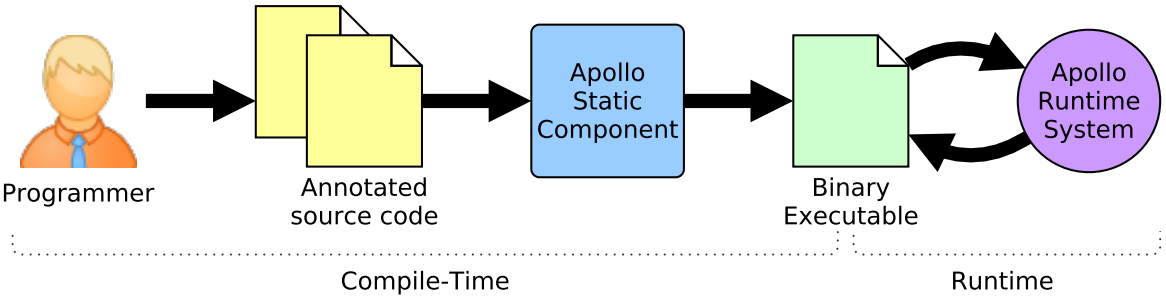

Figure 4.1: Overview of the Apollo framework.

4.2 Compile-time preparation

To enable speculative parallelization, the original code first has to be prepared. The static component of Apollo is in charge of this task. Its role is to embed in the final executable all the data structures and codes that are required to perform speculative parallelization efficiently. All the information embedded in the executable will be later available to the runtime system.

The first step performed by the static component is to optimize the code using classical optimizations, provided by the LLVM compiler. Then, static information is extracted from the target loop nests and embedded in the executable. This information includes for example: number and types of the memory instructions, structure of the loop nest, and aliasing information.

For each loop in a target nest, Apollo inserts a *virtual iterator* starting at zero and incremented by one at the end of each iteration. These iterators allow Apollo to handle any kind of loop in the same way. More importantly, the virtual iterators will play the role of a basis for predicting memory accesses, loop bounds, and scalar values.

A copy of the original code is used to create an instrumented version of the code. Additionally, several copies of the code are generated and Code-Skeletons are derived from them. Code-Skeletons [16, 17] are incomplete codes supporting a fixed and limited class of optimizing transformations. By completing a skeleton at runtime, Apollo is able to instantiate a particular combination of transformations. Nevertheless, these skeletons cannot support the whole range of possible transformations supported by the Polyhedral Model. For this reason, this mechanism has been replaced by the use of Code-Bones.

The Code-Bones [7] approach is the main contribution of this manuscript, which is fully detailed in the next Chapter. Numerous Code-Bones are derived from the original code, and embedded in the executable in their LLVM bitcode representation. At runtime, the Code-Bones are loaded and assembled in a new loop nest, instantiating

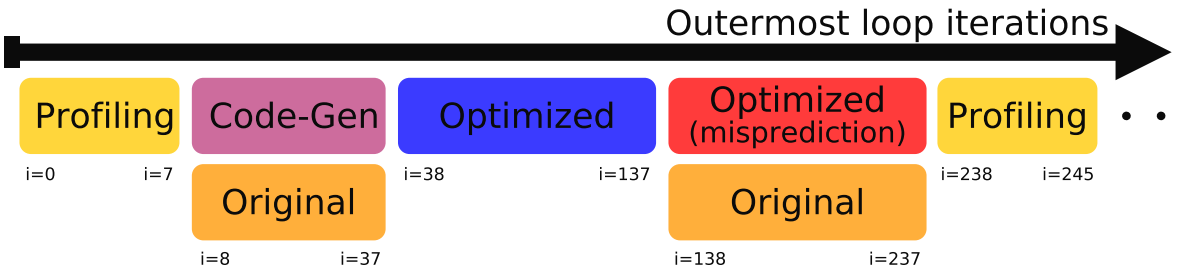

Figure 4.2: Succession of chunks, alternating between behaviors.

any possible combination of loop transformations. By using the LLVM Just-In-Time compiler, Apollo finally generates the optimized binary code.

4.3 Execution by 'chunks' of iterations

To be reactive to changes of its behavior, Apollo executes the target loop nest by *chunks* – or slices – of iterations, switching between code versions at each chunk. A chunk is a set of contiguous iterations of the outermost loop. At the beginning of the execution of a loop, Apollo profiles a few iterations inside a small chunk, typically from iteration 0 to 7, using the instrumented version of the code. The next chunk can be executed using a different code version, as for example an optimized version, from iteration 8 to 107. In this way, the loop is executed as a succession of chunks. Between chunks, the control returns to the runtime system. At this point, Apollo is able to decide about the execution of the next chunk. In our approach, the chunks are executed sequentially, but iterations inside a chunk may be executed in parallel.

During this execution by chunks, Apollo alternates between different behaviors to detect and adapt to different execution phases (see Figure 4.2). The behavior of the current chunk and the result of its execution determine the next chunk. In Figure 4.3, we show the transitions between the different behaviors, that are explained below:

1. At the beginning of the execution of a loop nest, or after completion of a chunk running the original version of the code, a **profiling** chunk is launched to capture the runtime code behavior. Once the profiled execution finishes, Apollo moves to the code generation phase.

2. During the **code generation** phase, a prediction model is built and a transformation is selected. From the selected transformation, optimized binary code is generated using the LLVM Just-In-Time compiler. If successful, Apollo is ready to execute optimized code; if not, the execution continues using the original ver-

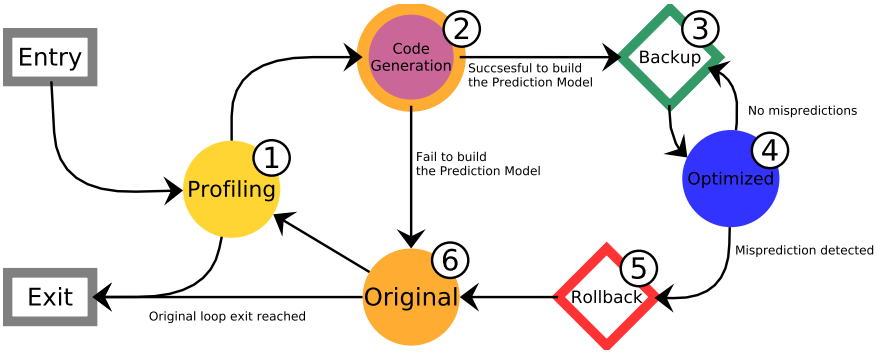

Figure 4.3: Transitions between behaviors of Apollo.

sion of the code. In parallel to this, a background thread executes the original version of the code, in order to mask, at least partially, the related time overhead.

3. Always before executing optimized code, Apollo performs a preventive **backup** of the predicted write memory areas (thanks to the prediction model). Once the backup has been performed, Apollo executes the optimized code.

4. **Optimized** chunks are executed until a misspeculation occurs. Iterations inside an optimized chunk are most often run in parallel.

5. When a misspeculation is detected, a **rollback** is performed to recover the memory state prior to the execution of the faulty chunk.

6. The same chunk is re-executed using the **original** version of the code, which is obviously semantically correct.

In the next Sections, we detail each phase of the execution.

4.4 Profiling

During the first phase of the execution of a loop nest, Apollo collects accessed memory addresses, scalar values and loop trip counts to build a prediction model.

At compile time, memory accesses are classified as static or dynamic: if the target memory address of the access can be described as a linear function of the enclosing iterators, then it is considered as static. Instructions that record these linear functions are inserted in the profiling version of the code. Additionally, for each memory access (even if it is static[1]), an instruction recording the target address is inserted. A similar process is applied for loop bounds and scalar values.

[1]The recorded values also help to determine if a memory access is actually executed or not. It does not incur a significant overhead.

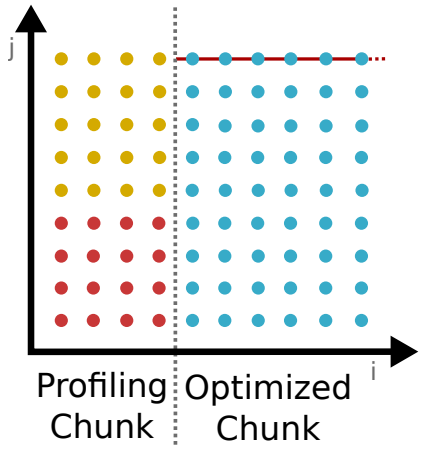

Figure 4.4: Sampled iterations with a 4x4 sample size. In red, iterations for which the values are recorded. In yellow, iterations for which the values are not recorded. In blue are the predicted iterations.

Instrumented code is time consuming. To reduce this overhead, we use a technique called *sampling*. It consists of recording values only for some part of the execution time. In our implementation, the instructions recording the values are guarded with inequalities that compare the virtual iterators with a constant. In consequence, values are recorded only for a subset of all the executed iterations during a profiling chunk execution. Figure 4.4 highlights the iterations for which recording is performed, using a 4×4 sample.

Figure 4.5 shows a simplified intermediate representation of the code that profiles three memory accesses. The accesses are all in the same basic block, and have two parent loops. A unique identifier is assigned to each memory access. This identifier, the target memory address, and the values of the enclosing iterators are registered in the same order as the memory accesses were performed. Notice how the recording of the memory accesses is guarded by a condition depending on the virtual iterators.

If the sample size is set to zero, the recording instructions are never executed. This is equivalent to execute using the original code. By exploiting this, we eliminate the need of a second version of the code for original sequential execution.

4.5 Building a Prediction Model

In contrast to most TLS systems, Apollo builds a model that predicts the behavior of the loop nest. By assuming that this prediction is valid, Apollo deduces dependencies between iterations and instructions, and applies aggressive code optimizations, that involve reordering the execution of iteration and instructions.

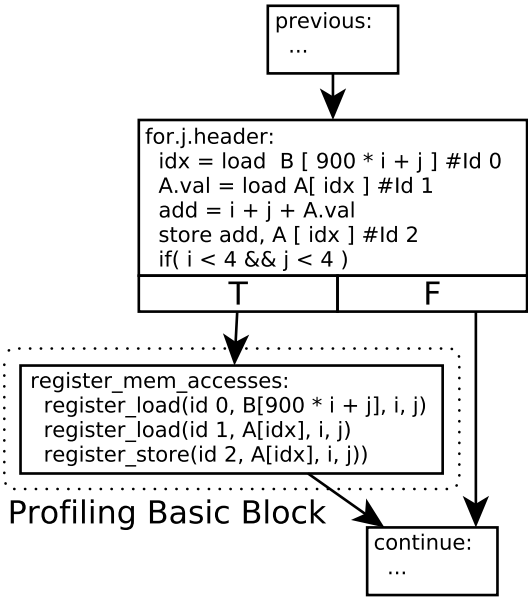

Figure 4.5: Intermediate representation of the profiling code for three memory accesses.

This model predicts: (1) memory accesses, (2) loop bounds, and (3) basic scalars.

Intuitively, memory accesses and loop bounds must be predicted to enable precise dependency analysis for selecting an optimizing transformation. Basic scalars correspond to the ϕ-instructions in the header of each loop. A ϕ is a special instruction in Static-Single-Assignment representations. A ϕ-instruction merges multiple incoming values depending on the last executed basic block. These scalars are updated at each iteration using a value defined in the previous iteration. This imposes a very strong dependency between successive iterations. Predicting the value of these scalars at the beginning of each iteration allows Apollo to remove this dependency, and thus to enable a larger set of possible optimizations.

1. Memory accesses There are three possible modelings for memory accesses: (i) linear, (ii) tube and (iii) range. In Figure 4.6, we depict each model.

When it is possible to interpolate a linear function from the registered memory addresses, a memory access is predicted as (i) linear. Future accesses are predicted to follow exactly the interpolating function.

This linear function is calculated by solving a system of equations. Consider a memory access at depth d of a target loop nest. Every time this access is profiled, the address and the values of its parent virtual iterators are recorded. The result is a list of n profiled tuples, $< mem_i, vi_{1_i}, vi_{2_i}, \ldots, vi_{(d+1)_i} >$, where mem_i corresponds to the profiled address value, and vi_{j_i} to the virtual iterator of depth j. The memory accesses are arranged in a vector $\vec{m} \in \mathbb{Z}^n$ and the virtual iterators in a matrix $\mathcal{V} \in \mathbb{Z}^{n \times d}$. Finally,

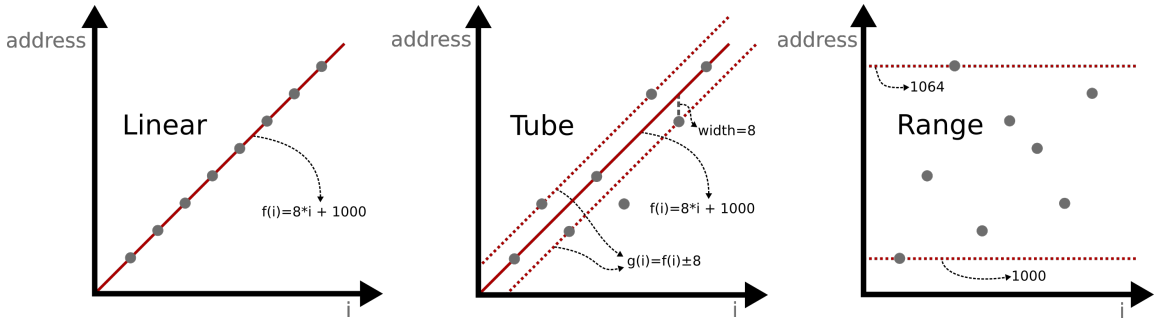

Figure 4.6: Different modelings for memory accesses.

the coefficients of the linear interpolating function are the solution of: $\mathcal{V}\vec{x} = \vec{m}$.

However, memory accesses may not follow a perfect linear pattern. In this case, a regression hyperplane is calculated using the least squares method. The regression hyperplane coefficients (initially of type real) are then rounded to their nearest integer value. Additionally, if a coefficient's modulo is smaller than the word size of the memory instruction, then it is rounded to the nearest value between zero, the word size, or minus the word size of the memory access. The Pearson's correlation coefficient is then computed. If it is greater than 0.9, future memory accesses are predicted to happen inside a tube, otherwise inside a range. This criteria is derived from experimental evaluation [29, 31]. The tube consists of the regression hyperplane, a tube width and a predicted alignment. The tube width is the maximum observed deviation from the regression hyperplane, rounded to the next bigger multiple of the word size.

The range consists of a maximum and a minimum memory address between which all the memory accesses are predicted to occur.

2. Loop bounds There are two possible modelings, (i) linear, or (ii) tube. Figure 4.7 visually depicts the different types of predictions for loop bounds. Notice that the lower bound of the virtual iterators is always 0, and only the upper bound is predicted.

The linear prediction mirrors the memory accesses linear prediction.

For the tube case, a regression hyperplane is computed and its coefficients are rounded to the nearest integer values. Then two hyperplanes are derived predicting a maximum and minimum number of iterations for a loop to execute. These new hyperplanes are parallel to the regression hyperplane, but passing through the maximum positive and negative deviations from the number of executed iterations. When selecting a transformation, the iteration space is divided in two, and all the iterations situated below the minimum predicted number of iterations may be aggressively transformed; on the other hand, the iterations between the predicted minimum and maximum must be executed sequentially, until the loop exit has been reached.

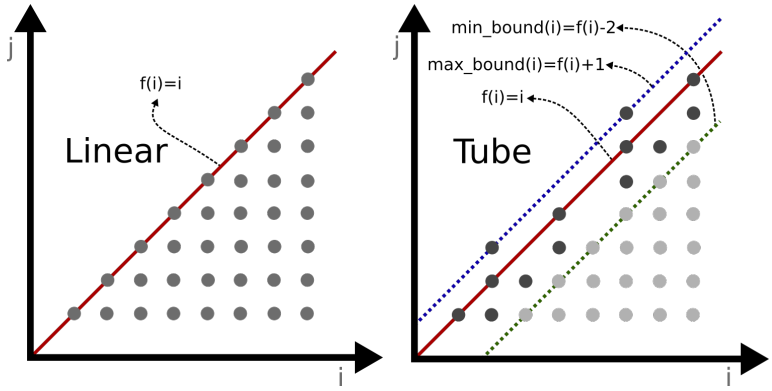

Figure 4.7: Different modelings for the loop bounds.

A previous idea in Apollo [29, 31] is to obtain an hyperplane that predicts the minimum number of iterations to execute. In practice, this hyperplane is just the minimum observed number of iterations. The iterations situated below this minimum can be parallelized, but not transformed, and verified in a similar way to ours, while the iterations above this minimum are executed sequentially, while registering every memory access (even if predicted), in a similar way than a traditional TLS system. In this approach, any dependencies involving an iteration above the hyperplane results in a rollback. This, plus the increased verification overhead, result in an even higher overhead. The approach presented in this manuscript considers also optimizing transformations of such codes, and is not limited to mere parallelization.

3. Basic scalars There are three possible modelings, (i) linear, (ii) semi-linear and (iii) reduction.

Again, the linear case resembles the memory access linear case.

A semi-linear scalar is characterized by a constant increment at each iteration of its parent loop, although the initial value of the scalar at the beginning of the loop cannot be predicted. The memory locations used for computing the initial value of the basic scalar must be predicted to remain unmodified during the execution of the loop. The example in Listing 4.3 shows the behavior that is captured by this prediction; the expression predicting the value of j for each iteration from the virtual iterators is $\mathsf{j} = \mathsf{b[i]} + vi_j$. Note that i will also be predicted. Any other behavior is treated as a reduction. Unfortunately, the underlying polyhedral tools used in Apollo are not able to handle reductions at all, preventing the generation of optimized code when they occur.

```
for(int i = 0; i < N; ++i)
    for(int j = b[i]; j < b[i]+100; ++j)
```

Listing 4.3: Example of a semi-linear scalar "j".

4.6 Transformation selection

To select a transformation, Apollo relies on the state-of-the-art polyhedral optimizer called Pluto, which is able to determine a loop transformation that optimizes for both data locality and parallelism. Pluto exposes its core algorithm through a library interface, taking as input an OpenScop representation of the target loop nest.

The polyhedral representation of a loop nest is based on statements, described by a domain, a schedule (or scattering) function and a set of access functions. However, in the LLVM-IR, there is no clear definition of statements. Statements from the 'C' source code are spread as several instructions across multiple loop levels, and may even share computations between them. Obtaining a polyhedral representation from such codes is not an easy task, which is addressed by the Code-Bones approach. Since building the OpenScop representation is closely related to the Code-Bones mechanism, we will explain it in next Chapter.

Additionally, Pluto is restricted to memory accesses modeled strictly by a single linear function, while memory accesses modeled by a *tube* defined by two linear inequalities, are not directly supported by Pluto. To support them, we modified Pluto to bypass its dependency analysis pass. This modification is also explained in the next Chapter.

4.7 Generating optimized code

To generate optimized code, Apollo owns two mechanisms: Code-Skeletons, developed first in the pre-Apollo prototype VMAD; and Code-Bones, a flexible approach presented in this thesis. The first being replaced by the latter.

A Code-Skeleton [16, 17] is a parametrized copy of the original code. It is parametrized by the prediction model and by a polyhedral transformation; by setting values to the parameters, one can instantiate different transformations. These skeletons also embed instructions verifying the validity of the prediction model (discussed further in Section 4.8). At compile time, several Code-Skeletons are generated to support different transformations. However, each skeleton only supports a certain combination of loop transformations, since the loop structure is fixed at compile time. For example, while a

```
for(i = 0; i < n; ++i)
  for(j = 0; j < n; ++j)
    a[i][j] = a[i][j-1] + a[i-1][j]
```

Listing 4.4: Original code

```
pm = ... // the prediction model
tx[][] = ... // the transformation
ca[] = pm->linear_function_a //predicted linear function
for(x = lower_x(pm, tx); x < upper_x(pm, tx); ++x) {
  for(y = lower_y(pm, tx, x); y < upper_y(pm, tx, x); ++y) {
    //obtain the iterators in the original iteration space
    i = tx[0][0] * x + tx[0][1] * y
    j = tx[1][0] * x + tx[1][1] * y

    //compute the predictions
    a_pred_0 = ca[0] * i + ca[1] * j + ca[2]
    a_pred_1 = ca[0] * i + ca[1] * (j-1) + ca[2]
    a_pred_2 = ca[0] * (i-1) + ca[1] * j + ca[2]

    //verify the predictions
    if(a_pred_0 != a[i][j]) rollback()
    if(a_pred_1 != a[i][j-1]) rollback()
    if(a_pred_2 != a[i-1][j]) rollback()

    //original computation
    *a_pred_0 = *a_pred_1 + *a_pred_2
  }
}
```

Listing 4.5: Derived Code-Skeleton for code in Listing 4.4

given skeleton supports a combination of loop skewing and loop interchange, it cannot support any additional loop transformation as tiling or loop fission. Even if these latter transformations may be supported by an additional skeleton, all such supplementary skeletons would yield an oversized executable file. Anyway, this strategy cannot provide the flexibility required for covering all possible combinations of loop transformations, as those enabled by static polyhedral compilers, as Pluto.

The pseudo-code in Listing 4.5 shows the derived Code-Skeleton for code in Listing 4.4. One can see that this skeleton is parametrized by the prediction model pm and the inverse of the transformation tx. From the iterators – x, y – that scan the transformed iteration space, and using the inverse of the transformation, we recover the values of the original iterators – i, j – in the original iteration space. It is then easy to compute the prediction for each memory access. To verify that the prediction model is valid, the actual memory addresses are compared against their predictions. If they are different, a rollback is signaled and the execution aborts. Finally, the original computation is

performed using the predictions.

Assume we want to instantiate a loop interchange on the code in Listing 4.4. The required transformation matrix, and inverse, is $\mathsf{tx} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. From this matrix, new loop bounds are derived using the Fourier-Motzkin elimination algorithm. The result from this algorithm will be interpreted by functions lower and upper. Notice the consequences of using this Code-Skeleton: tiling transformations are not supported since the loop structure is fixed; and one cannot select a different optimizing transformation for the verification code, since it resides in the same loop as the computation it verifies.

A different and more flexible approach is taken thanks to Code-Bones. Instead of providing a loop structure to fill in, they provide smaller building blocks used for assembling a completely new loop nest at runtime.

To compare against the Code-Skeletons approach, in Listing 4.6 we show the derived Code-Bones for the same code. A Code-Bone packs multiple related instructions to retrieve statements similar to C code. These bones can be arranged to instantiate any transformation supported by the Polyhedral Model. In the code in Listing 4.7, a combination of fission, interchange, and unrolling is applied. In contrast to the skeleton shown before, different transformations are applied to the verification and the computation code. Notice how different bones are generated for the original computation and for the verification. This allows us to schedule them differently, and to exploit some properties of the latter to further optimize them.

4.8 Optimized execution

Once the optimized code has been generated, Apollo is ready to start with the speculative execution. Again, the code is executed by chunks of iterations. However, the number of iterations assigned to optimized chunks is larger than the one used for profiling. Before executing the optimized code, several checks are performed to guarantee that recovery will be possible in case of misspeculation.

The first step is to ensure that the accesses to be performed will always access memory that has been allocated to the process, to avoid an irrecoverable segmentation fault. For this, the ranges predicted to be accessed by each memory instruction are computed, and a load or store is performed on the extremes of each range. If the memory page was not allocated, a segmentation fault is raised and captured by Apollo. If a segmentation fault is detected, the optimized chunk is aborted. Later, Apollo can decide to retry with a smaller optimized chunk or to continue using the original version of the code.

```
computation_bone(pm, i, j) {
  ca[] = pm->linear_function_a //coefficients

  a_pred_0 = ca[0] * i + ca[1] * j + ca[2]
  a_pred_1 = ca[0] * i + ca[1] * (j-1) + ca[2]
  a_pred_2 = ca[0] * (i-1) + ca[1] * j + ca[2]

  *a_pred_0 = *a_pred_1 + *a_pred_2
}

verification_0_bone(pm, i, j) {
  ca[] = pm->linear_function_a //coefficients
  a_pred_0 = ca[0] * i + ca[1] * j + ca[2]
  if(a_pred_0 != a[i][j]) rollback()
}

verification_1_bone(pm, i, j) {
  ca[] = pm->linear_function_a //coefficients
  a_pred_1 = ca[0] * i + ca[1] * (j-1) + ca[2]
  if(a_pred_1 != a[i][j-1]) rollback()
}

verification_2_bone(pm, i, j) {
  ca[] = pm->linear_function_a //coefficients
  a_pred_2 = ca[0] * (i-1) + ca[1] * j + ca[2]
  if(a_pred_2 != a[i-1][j]) rollback()
}
```

Listing 4.6: Derived Code-Bones for Listing 4.4

```
for(i = 0; i < N; ++i)
  for(j = 0; j < N; ++j)
    verification_0_bone(pm, i, j);
    verification_1_bone(pm, i, j);
    verification_2_bone(pm, i, j);

for(j = 0; j < N; ++j)
  for(i = 0; i < N; i+=2)
    computation_bone(pm, i, j)
    computation_bone(pm, i+1, j)
```

Listing 4.7: Advanced transformations using the Code-Bones from Listing 4.6: fission, interchange and unrolling.

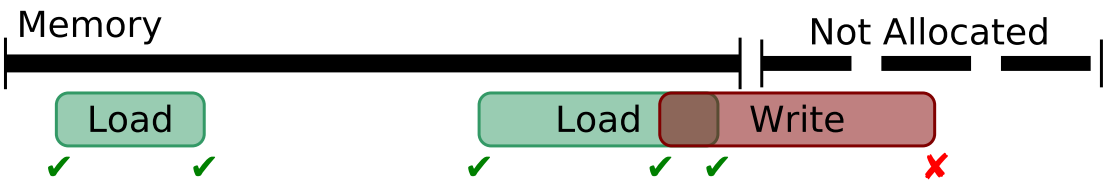

Figure 4.8: Memory instructions must access areas allocated to the process.

Figure 4.8 illustrates this verification. All the memory loads are predicted to happen in a memory area that is allocated to the process, and will not yield a segmentation fault. Then, it is safe to speculatively execute the current chunk.

To compute these memory ranges, for memory accesses modeled as *ranges*, the predicted memory range corresponds directly to the range's maximum and minimum addresses. For memory accesses predicted as *tubes*, the regression line is evaluated at each vertex of the domain, also adding (subtracting) the tube width from the maximum (minimum) values. The number of bytes accessed is finally added to the maximum (a value of type `long`, in 64bits, has a size of 8 bytes, for example). A *linear* access can be considered as a *tube* access with a width of 0.

The next step is to perform a backup of the memory. This allows, in case of misspeculation, to recover the state of memory before the execution of the optimized code. An over-approximation of the memory write areas is used, obtained using the prediction model. Once the write regions are identified, a simple memory copy to a new area of memory is performed. Figure 4.9 depicts this task. Only the write regions are backed up in a fresh memory area.

For memory accesses modeled as *ranges*, the whole memory range is backed up directly. However, for *linear* or *tube* modelings, it is possible to do better. Consider that the interpolated linear function for a memory access is $i + 100 * j$, and the loop upper bounds for i and j are both equal to 10. For this domain and linear function, the predicted extremes are 0 and 1010. Addresses from 11 to 99 are also backed up, although they are never accessed. This excessive amount of useless copies results in very slow backup times, penalizing the performance improvements that can be achieved with Apollo. The solution is to first compute the extreme memory range, but while ignoring the loop whose coefficient is the largest in the linear function, in this case, j. Then, iterate over j and displace the previous range using the coefficient for j. This strategy results in multiple smaller ranges to backup. For the current example, the obtained ranges are:

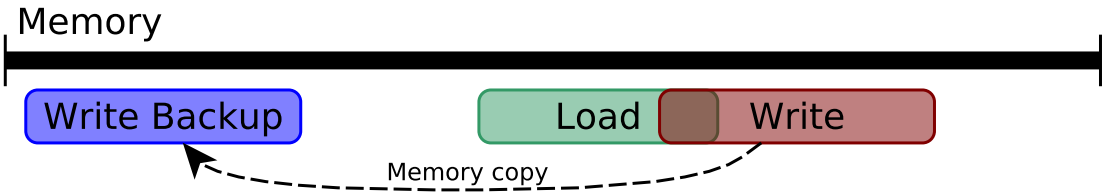

Figure 4.9: Backup of the write regions.

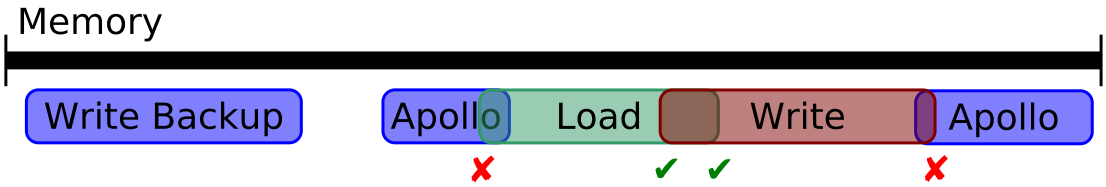

Figure 4.10: The optimized code must not access memory allocated by Apollo for its internal management operations.

$$\{(0 + 100j, 10 + 100j) | \forall j \in [0, 10]\}$$

It is a significant improvement over the backup system originally implemented in VMAD, which not only reduces the backup time, but also the memory consumption of Apollo.

The third step, similar to the first, is to ensure that the memory accesses to be performed will not corrupt the data-structures used by Apollo (as the ones used for misspeculation recovery, for example). Apollo registers *every memory allocation* performed by itself, or by the used libraries, to keep track of the memory ranges in which its data-structures reside. Also, some data structures held in the stack that are required during the entire execution of Apollo are registered. Since keeping track of all the allocations is expensive, dedicated memory pools for common requested sizes are used. Surprisingly, this leads to better performance compared to the situation when no tracking is performed.

Figure 4.10 shows two predicted ranges that overlap with the memory area allocated by Apollo. Thus, it is obviously unsafe to execute the optimized chunk.

Since this check was not implemented in VMAD, it was leading to bugs.

Once this last check is performed, Apollo is ready to execute using the optimized version of the code.

During the execution of the optimized code, all predictions are verified, to ensure that the prediction model remains valid [30]. If the prediction ceases to be valid, the

previously selected transformation may not preserve the semantics of the original sequential execution. When a thread detects a misspeculation, it broadcasts the situation to the other for them to abort. The threads, executing in an optimized chunk, periodically poll a flag that indicates if a misspeculation has been detected. After all the threads have aborted, a rollback is initiated. The rollback consists of copying back all regions from the backup to their original location.

On the other hand, if the execution of the optimized code was successful (no misspeculation), the backup is discarded and a new optimized chunk is initiated.

4.9 Final remarks

Our approach enables more aggressive optimization in codes which do not seemingly agree with the polyhedral model, but that exhibit some phases which turn out to be compatible with the model during their execution. The key relies on the prediction model, built from an on-line profiling phase, used to overcome the limitations of static analysis. Thanks to this prediction model, a polyhedral representation can be derived, and Pluto can be used to choose a transformation. This is the main advantage of Apollo over other TLS systems, since Pluto provides to Apollo the power of selecting an optimizing transformation on-the-fly.

Another main contribution of Apollo is the support of polyhedral optimizations for codes with non-linear behaviors. Sometimes, some behaviors do not perfectly agree with the Polyhedral Model, and Apollo cannot interpolate with linear functions. If this is the case, Apollo computes linear inequalities that bound the possible values. These linear inequalities can be expressed for building the corresponding polyhedral representation.

However, building a polyhedral representation from the prediction model of the loops handled by Apollo is not straightforward. Also, to exploit the optimization opportunities detected by Apollo, a code generation mechanism able to instantiate potentially any polyhedral transformation at runtime is required. This is the subject of discussion in the following Chapter.

Chapter 5

Code-Bones: Fast and Flexible Code Generation for Speculative Polyhedral Parallelization

The main reason why most TLS systems show weak performance is because they do not handle any aggressive optimizing and parallelizing transformation. Hence, mere parallelization is performed and issues like data locality are not addressed, although it is widely agreed that successful parallel programming depends strongly on such issues.

In the previous Chapter, our prediction model is explained. It is a key difference of Apollo regarding most TLS systems. This model allows to speculate about the execution behavior and to reason about data dependencies, as long as the model remains valid. In this Chapter, we explain how this model is used to select an optimizing transformation, and especially how optimized code implementing this transformation is generated.

In Section 5.1, we address some challenges that drove many decisions about our code generation and transformation approach. Section 5.2 gives an overview of our code generator. It works in two phases, one at compile time (explained in Section 5.3) and one at runtime. The runtime phase consists of multiple stages: loop reconstruction (Section 5.4), polyhedral representation (Section 5.5), scheduling (Section 5.6), scanning (Section 5.7), and binary code generation (Section 5.8). In Section 5.9, we discuss some related code generation approaches used in other contexts. Finally, in Section 5.10, we conclude this chapter with some final remarks.

5.1 Challenges

The combination of speculative parallelization and the polyhedral model sets a particular context for code generation. We encountered three major challenges while developing our code generator.

1. Mapping the IR to the Polyhedral Model Apollo is built on top of the LLVM compiler. As a consequence, the static component of Apollo manipulates programs in the LLVM intermediate representation (IR). This IR aims to be low level, such that high level programming constructs can be mapped to it. For example, for-loops and while loops are uniformly expressed as a sub-control-flow graph whose exit node is a conditional jump defining a back-edge to the entry node.

Statements of C codes are mapped as several contiguous elementary instructions. After applying some classical optimizations, these instructions get spread across different basic blocks, and even different loops. Instructions are re-arranged and some of them are even merged to eliminate redundant computations. Thus, the concise form of a statement in C gets diluted in the IR. The value produced by an instruction in the IR is stored in a register and later used by another instruction, without accessing memory. A pair of such dependent instructions must be scheduled together, unless a temporary buffer is introduced, where the result of the first instruction is stored and later read and used by the second instruction. Only then, it would be possible to schedule each instruction independently as a polyhedral statement. This approach would provably results in a better schedule. However, it yields a huge number of polyhedral statements to be scheduled, which is not compatible with the complexity handled by current polyhedral model tools. In [32], it has been shown that the compilation time of Pluto increases in a roughly n^5 complexity in the number of statements in the system. In consequence, Pluto can introduce a high time overhead, which is inadequate for a runtime usage.

An important consequence is that we cannot map directly our IR representation to a polyhedral representation. Our approach handles this issue by extracting *Code-Bones* from the IR (explained in Section 5.3). Code-Bones group several instructions related by definition-use dependencies, that are scheduled together. This provides a good trade-off between flexibility and complexity. Code-Bones are seen as atomic units which do not interfere with each other, resembling the statements in C. This abstraction enables a direct mapping to a polyhedral representation (Section 5.5).

It is unavoidable to relate our work to Polly [14], the LLVM-based polyhedral optimizer. Polly extracts a polyhedral representation from codes in LLVM-IR that

can be accurately analyzed at compile time. But multiple decisions in Polly are not amenable to our approach. First, Polly considers self contained Single-Entry-Single-Exit [19] (SESE) regions as statements. In codes handled by Apollo, instructions required to perform a final store to memory are commonly spread across different loops. It is impossible to extract such ideal Single-Entry-Single-Exit regions in such cases. Our approach handles this issue by copying, inside a single Code-Bone, all the instructions reachable by the definition-use dependencies starting from the final store. For example, a load in an outer loop used by a final store in an inner loop, will be extracted in the Code-Bone associated to the store. Notice that this can violate dependencies in the original code. At runtime, during the *reconstruction* phase of the Code-Bones work-flow, these dependencies are verified to guarantee the original semantics.

Another disadvantage of Polly is pictured in Listing 5.1. In this example, statements S1 and S2 belong to the same SESE region, scheduled together by Polly, while Apollo extracts one Code-Bones for each statement, and schedules them separately. In this case, Polly is less efficient that Pluto, which would schedule both statements separately. Apollo, thanks to Code-Bones, recognizes the same optimization opportunities as Pluto. Moreover, some external tools used by Polly (ISL) are not amenable to a runtime usage. Finally, speculative optimization requires verification code, which is not handled by Polly.

```
for (i=0; i<900; ++i)
    for (j=0; j<900; ++j)
        A[i][j] = ... ; //S1
        B[j][i] = ... ; //S2
```

Listing 5.1: S1 and S2 belong to the same SESE region.

2. Exploiting unique optimization opportunities The optimized code generated by Apollo is composed of two main types of instructions: instruction performing computations of the original code, that result in memory writes; and instructions devoted to verifying the validity of the prediction model. These different types of instructions yield different optimization opportunities (addressed in Section 5.6.1), specific to Apollo.

3. Fast! In a dynamic optimization context, time-overhead is a major concern. However, the available polyhedral tools were not developed for a runtime usage and are often too slow. Many decisions and optimizations had to be applied to mitigate

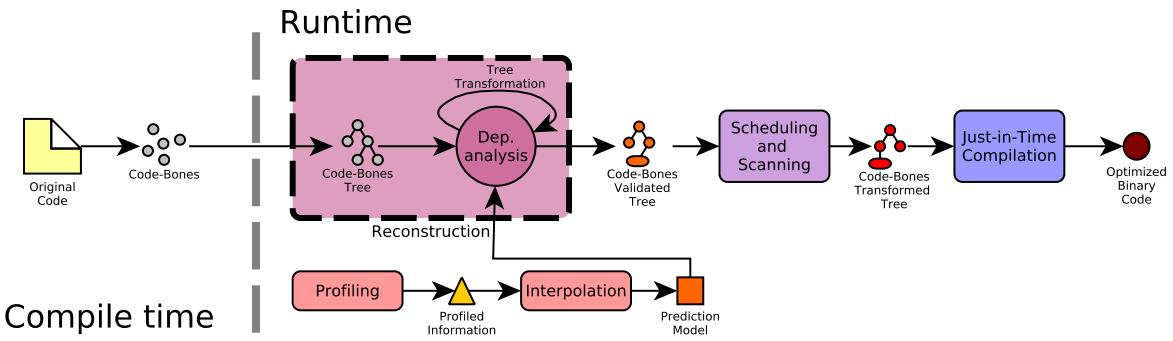

Figure 5.1: Overview of the Code Generation flow of Apollo.

this problem. Nevertheless, we consider that there is a large and crucial gap to fill by new tools which would be adapted to runtime analysis and optimization.

5.2 Code-Bones: Building blocks for polyhedral transformations of a compiler IR

Until now, as a solution for fast runtime code generation, the Apollo framework was using Code-Skeletons [16, 17]. This mechanism was inherited from VMAD and lacks the flexibility required to fully take advantage of the polyhedral model. A Code-Skeleton is tied to a fixed loop nest structure. To support transformations that alter differently the loop nest structure, such as unrolling, tiling, fusion or fission, a different Code-Skeleton is required. If combinations of these transformations are also supported, the number of required Code-Skeletons grows exponentially. The practical solution is to limit the number of supported transformations to a limited number of available skeletons, leading to narrow optimization opportunities.

To support a richer set of transformations, we propose to use building blocks allowing to instantiate any polyhedral transformation. We call these building blocks *Code-Bones* [7].

At compile time, multiple Code-Bones are generated. Then, at runtime, a transformation is determined and implemented by instantiating and assembling them. Figure 5.1 depicts the different tasks that are performed to finally generate optimized executable code. In the following Sections, we explain each of these steps.

5.3 Extraction

The first step is to extract multiple Code-Bones from the original source code.

Any speculatively optimized code is generally composed of two types of computations: (1) computations of the original target code, whose schedule and parameters have been modified for optimization purposes; and (2) computations related to the verification of the speculation, whose role is to ensure semantic correctness and to launch a recovery process in case of wrong speculation. From the control-flow graph (CFG) of the target loop nest, we extract the different Code-Bones.

(1) Each memory write instruction in the original code yields an associated code bone, that includes all instructions belonging to the *backward static slice* of the memory write instruction. In other words, these are all the instructions required to execute an instance of the memory write. Notice that memory read instructions are also included in Code-Bones, since the role of any read instruction is related to the accomplishment of at least one write instruction. This first set of Code-Bones is called *computation bones*. (2) For each memory instruction of the computation bones, that may be a write or a read, an associated *verification bone* is created. Additionally, verification bones for the scalars (one for the initial value and one for the increment) and for the loop bounds are also created. These bones contain instructions devoted to verifying the validity of the prediction model.

As an example, consider the code in Listing 5.2. From this code, the CFG shown in Figure 5.2 (Left) is generated. After having broken the loop back-edges[1], we obtain the tree in Figure 5.2 (Right). Three memory instructions are present in the innermost loop (for.j.header): two loads and one store. In red, we highlight the backward static slice of the store instruction (store add, A[idx]). This store uses three references: add, idx and A. Since they are required for executing the store, instructions that set these references are added to the backward slice. Similarly and recursively, add = i + j + A.val uses three other values for its computation: i, j and A.val. Again, instructions setting them are added to the backward slice. Not only direct definition-use relations must be considered, but also control-flow relations. To reach the basic block for.j.header, the branch instruction in for.i.header has to be executed. Since the execution of the store depends on the execution of this branch, it has also to be added to the backward slice.

```
for (i=0; i<900; ++i)
    for (j=0; j<900; ++j)
        A[B[j]] += i + j;
```

Listing 5.2: Small code example.

[1]Branches from inside the loop that jump to the loop header

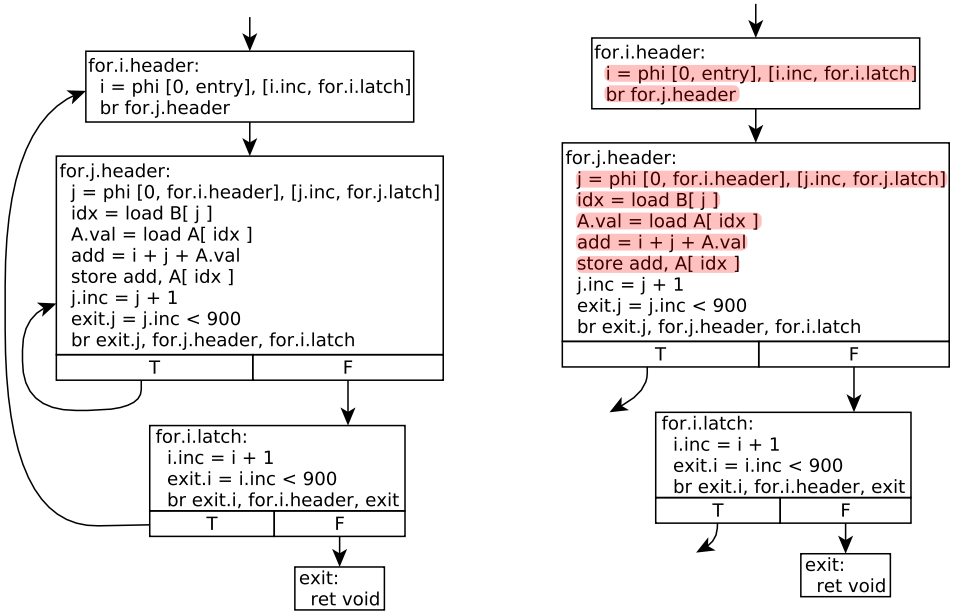

Figure 5.2: Original Control-Flow-Graph (Left). Highlighted backward static slice of the store (Right).

Once the backward slice of an instruction has been identified, it is time to extract it into a code bone. A bone is implemented as a LLVM function, taking as arguments the parameters of the loop nest and the parent virtual iterators. All the instructions from the backward slice are cloned inside the code bone. Predicted instructions – memory address and basic scalars computations, for example – are replaced by their prediction. This results in more efficient code since the computation of the prediction tends to be simpler than the original computations.

In Figure 5.3, we show the generated code bone for the store of our previous example. Notice that the value of i and j are now replaced by their prediction. As presented in the previous Chapter, scalars support two types of predictions: linear and semi-linear (the 'reduction' type is not handled). The code that computes both the linear prediction and the semi-linear prediction is generated for each scalar. The right prediction is selected based on a flag set by the runtime system. This may seem inefficient, since the linear prediction is computed even if the prediction is semi-linear, and vice-versa. Nevertheless, the LLVM Just-In-Time compiler will replace all the calls to get_coefficient by a constant value, and all these useless computations are then eliminated by further optimizations. Every target memory address is replaced by its prediction when predicted exactly using a linear function. In any other case, the memory address computation of the original code is maintained.

In Figure 5.4, we show the verification bone in charge of verifying the store in-

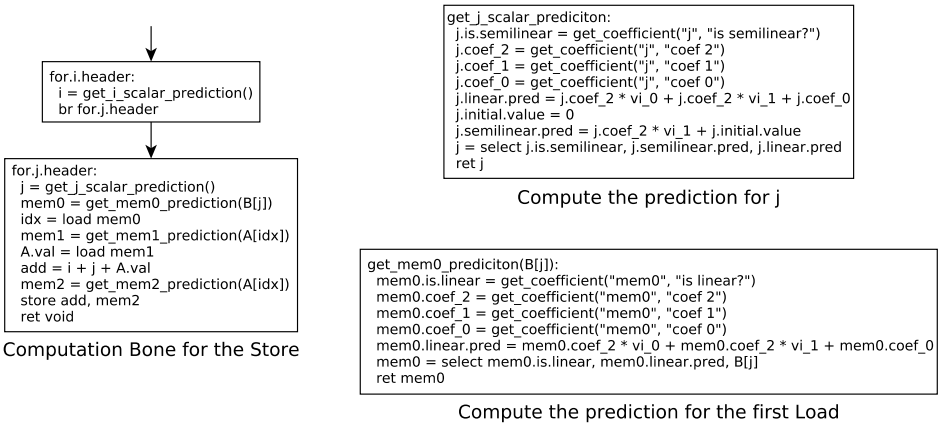

Figure 5.3: Computation bone.

struction. The original access is then verified according to three possible modelings: linear, tube and range. The linear verification consists of comparing the original address against the evaluated predicting linear function. The tube verification consists of verifying that the deviation of the original address from the regression hyperplane is below the expected tube width. The range verification consists of comparing the original address against a lower and an upper bound. If the tube or the range verification fails, there is still hope. In some cases, even if the behavior does not agree with the prediction model, the optimized execution may still remain valid. A load can be safely performed in a memory area predicted to be read-only, since it will not introduce a new dependency. A read-only area is a set of ranges of memory addresses that will not be touched by any memory write instruction, according to their respective predictions. The runtime is invoked to perform these more advanced checks. Finally, this bone returns "false" to signal a misspeculation and trigger a rollback and "true" if the execution can proceed.

All the generated bones are embedded in the executable in their IR representation form. At runtime, the Code-Bones are loaded and used to assemble a new loop nest. In the next Sections, we explain all the stages performed at runtime, until finally obtaining the optimized binary code.

5.4 Reconstruction

At runtime, code generation starts just after the prediction model has been derived from the profiling information. The Code-Bones in their IR representation form are loaded, and each of them is parsed to identify memory access instructions, scalars and verification instructions, which characterize them. Other internal computations can for

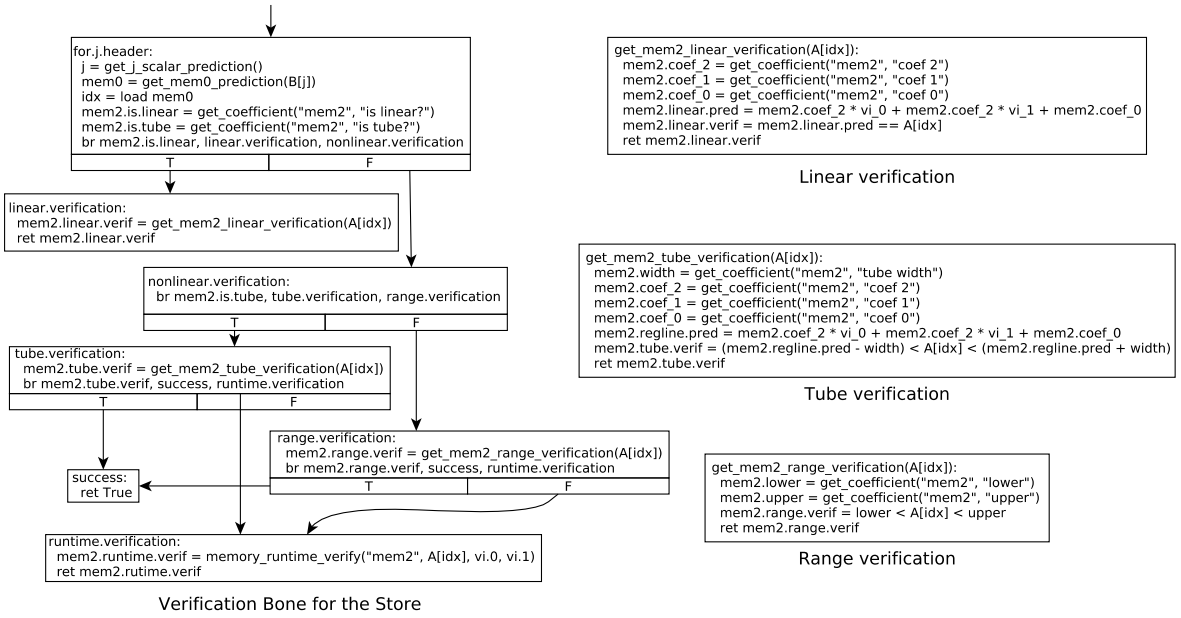

Figure 5.4: Verification bone.

now be completely ignored, since they do not affect the polyhedral representation.

First, the original loop nest is reconstructed as an assembly of Code-Bones. A loop structure that mimics the original loop is created, complemented with dynamic information obtained thanks to the prediction model. Continuing with the Code-Bones extracted for our example in Listing 5.2, a 2-depth loop nest is constructed. Then, all the computation bones are inserted in the loop nest. Each computation bone has a particular position, given by the parent loop and the *id* of its associated store instruction. In case two bones reside at the same loop level, the one whose associated store id is smaller executes first. In our example, there is only one computation bone, inserted in the innermost loop.

All the predicted instructions in the bones must be verified. First, non-linear predictions are handled by inserting a verification bone just before the computation bone containing the related non-linear instruction. Then, the bone that verifies the non-linear prediction and the bone that uses the prediction are fused into a single bone. In this new dynamically generated bone, the verification guards the actual execution of the bone. Remember that a memory access, predicted as non-linear, uses directly the original address computation, while an access predicted as linear uses the predicted memory address. If not guarded by a preventive verification, a faulty write access, predicted as non-linear may occur, without any possible recovery, while all linear predictions take advantage of a previous backup or a previous verification phase (see Section 5.6.1). Once all the non-linear instructions are verified, bones verifying linear

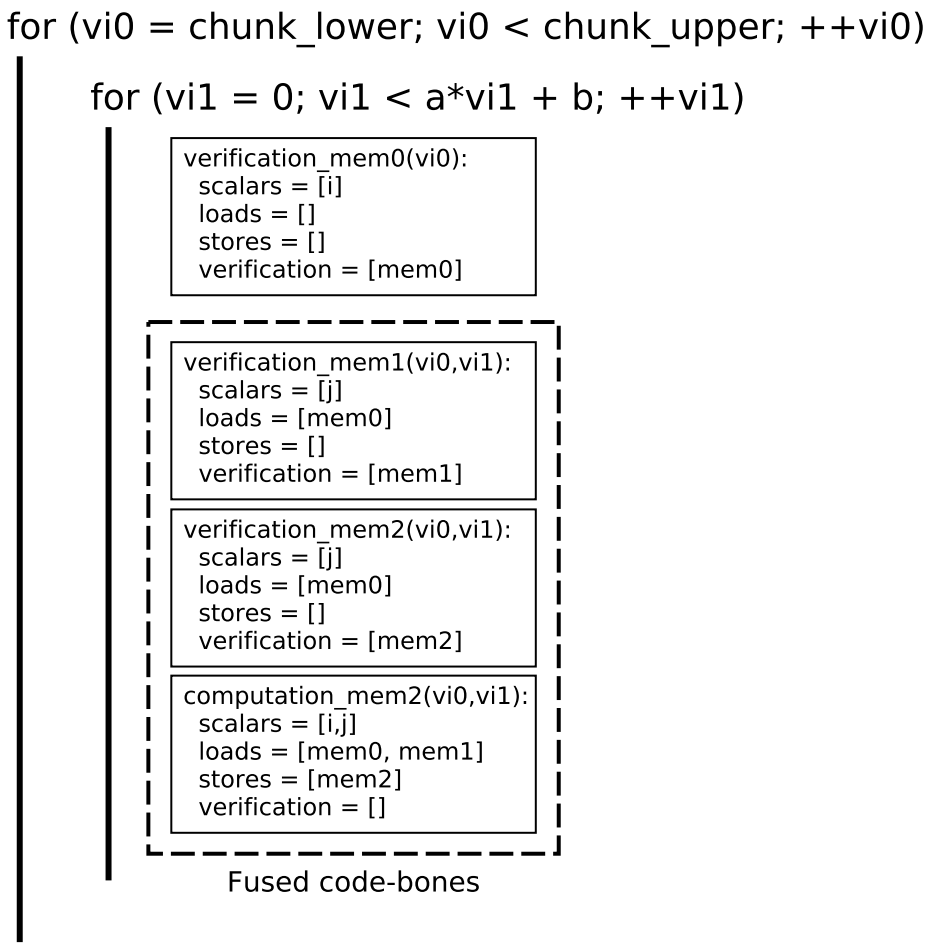

Figure 5.5: Reconstructed loop nest.

predictions are also inserted. They do not require to be fused with the computation bone. The addition of a verification bone is repeated until every predicted instruction has been handled.

Additionally, contiguous bones can be fused, in order to reduce the final number of polyhedral statements to schedule. Thus, we can reduce the time overhead required to select a transformation and to generate the associated scan of the iterations. When the number of computation bones participating in the target kernel is bigger than 5, Code-Bones are fused until this limit is reached or until there are no more Code-Bones to fuse. On the other hand, fusing bones also reduces the set of possible optimizing transformation. One alternative for lowering the complexity of polyhedral compilation was proposed in [32]. It consists of sub-polyhedral scheduling using (unit-)two-variable-per-inequality polyhedra which leads to asymptotic gains in complexity. This technique, if implemented, may provide a polyhedral scheduler better suited to our needs.

In Figure 5.5, we show the final result of this process on the example. The order in

which the memory accesses appear in this code can be different from the original code. Furthermore, some load accesses may now be repeated if they take part of multiple Code-Bones (access to mem0 in the example), and may even be moved from an outer to an inner loop. Thus, the reconstructed nest has to be verified against dependencies in the original code. Inter-iteration dependencies can only be violated if the source or the target of a dependency is moved from an outer to an inner loop. Intra-iteration dependencies must also be validated. If no violation occurs, the reconstructed nest is equivalent to the original nest and we can proceed to transform it.

If a violation occurs it is sometimes possible to perform a local transformation to repair the broken dependency. For example, if a dependency is violated between two successive bones, swapping the bones may repair the dependency. If the bones depend on each other, it is necessary to fuse the bones in a particular way: the first bone is inserted just before the store instruction of the second, such that all the load instructions are executed before the stores. If no repairing transformation can be applied, the code generation phase aborts, and a chunk executing the original version of the code is launched.

5.5 Mapping to a Polyhedral representation

Now, we have a loop nest containing well defined statements. It is fairly easy to obtain the polyhedral domain and the scattering function for each bone in such nest. Then, they are encoded in the OpenScop format to be communicated to Pluto.

For all the functions defining memory accesses, the obvious approach is to relate them with accesses to a single common array. However, this simplistic approach leads to a very slow transformation selection times by Pluto. Furthermore, memory instructions often access multiple contiguous memory locations at once. Each access function should then be replicated for each accessed byte; loading a double would then yield 8 accesses!

The solution is to recover as much information about the accessed data structure as possible. This not only speeds up the transformation selection, but also improves the quality of the selected transformation. In this purpose, several steps are performed:

1. Recognize arrays By using static alias analysis information and the ranges predicted to be accessed by memory instructions, different groups of aliasing instructions are built. Each group is associated to its own array, and does not overlap with other groups.

2. Compute word sizes For each array, the greatest common divisor (GCD), of the coefficients of the linear access functions and of the word size of each access, is computed. This value is used to derive new access functions, by dividing the coefficients by the GCD.

3. Recover dimensions For arrays that are only accessed using linear functions, it is sometimes possible to recover the dimensions of multi-dimensional arrays. If successful, the arithmetic complexity of the computations related to dependence analysis and transformation selection is significantly lowered. Our implementation is derived from [21]. At runtime, all the values of all the coefficients of the linear access functions are known, simplifying our task. Our implementation does not need to handle symbolic parameters. For each array, our algorithm considers the non-zero coefficients of the linear access functions as candidate array dimensions sizes. Then, it tries to rewrite the linear access functions as multi-dimensional accesses, where each iterator participates at most in one dimension. To be valid, the range of array elements touched in each dimension must be greater or equal to 0 and strictly smaller than the candidate dimension size.

Figure 5.6 depicts our steps by using an example. (0.) The prediction model is composed exclusively by linear functions for all memory accesses. Notice that the address range accessed by mem0 (1000 to 1083) does not overlap with the range accessed by mem1 (4000 to 8087) and mem2 (4008 to 8095), nor by mem3 (9000 to 13097) and mem4 (9000 to 13087). (1.) Hence, three different arrays are derived. After this step, access mem0 spawns 4 different linear functions of one-byte accesses. (2.) To avoid this, we compute the word size for each access. (3.) Finally, for the second array, associated to mem1 and mem2, it is possible to map the accesses as accesses to a 2-dimensional array. For the third array, it is impossible to rewrite the accesses for the two candidate dimensions (50 and 51). In this case, a virtual iterator participates in more than one dimension. It would be possible to extend our algorithm to handle such cases (up to a certain extent); anyway, these cases are uncommon in practice. Notice that the recovered dimensions may not correspond to the dimensions defined in the original source code and may only be valid in the range defined by the chunk bounds. However, they do not alter the original semantics and improve significantly the transformation selection process.

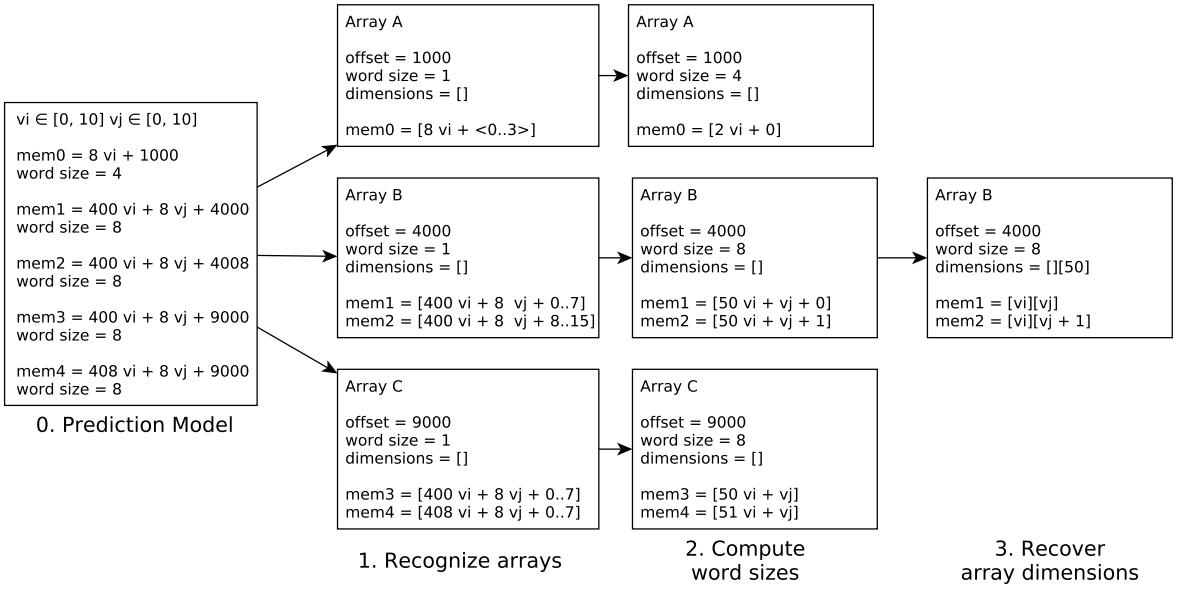

Figure 5.6: Recovering high level access information.

5.6 Scheduling

Once the polyhedral representation of the code has been built, we can invoke the Pluto compiler's algorithm to determine an optimizing and parallelizing transformation.

Pluto exposes multiple options that must be tweaked to result in the best optimizing transformation. There is no unique setup of options that always outperforms other options. Furthermore, the best set may depend on the target code or the hardware.

However, numerous experiments lead us to define a set of options yielding well performing optimized code in most cases. Intra-tile-optimization (–intratileopt) is always activated since it enables Pluto to do loop interchanges to improve data locality. We always enable parallelization (–parallel), unless there is a single core. Loop unrolling (–unroll) is enabled with a factor of 2; bigger factors did not yield any significant performance improvement, while also greatly increasing the code size, harming the LLVM JIT performance. Maximum fission is always set (–nofuse); this configuration provides the best performance results and keeps CLooG's and the LLVM JIT compilation times low. For tiling (–tile), we implemented an heuristic to automatically detect if it is profitable (explained in Section 5.6.2). However, we never found level 2 tiling (–l2tile) to be profitable, and it greatly increases CLooG's execution time. For the rest of the options, we kept the default behavior. Notice that, in a dynamic context, it is not mandatory to obtain the best performing optimized code. One must consider a trade-off between the time taken to obtain optimized code and its execution performance.

We have also faced several limitations when using Pluto.

Some parameters cannot be set through the library interface: the tile sizes cannot be set to a size different from the default, and it is impossible to add arbitrary constraints to the transformation. Even worse, it is impossible to describe a *tube* or *range* of memory accesses with Pluto, although it is possible with tools like *Candl*[2].

To overcome this latter issue, we did the following: first, the OpenScop representation is passed to Candl, to perform the dependence analysis. Then, in the OpenScop representation, all tube and range accesses are replaced by accesses using their regression lines, and the computed dependencies are attached to the OpenScop representation. We modified Pluto to use the attached dependencies and to ignore the access functions encoded in the OpenScop representation. In this way, the transformation is finally selected by Pluto based on the dependencies computed by Candl, and using the equations describing non-linear memory accesses.

5.6.1 Optimization of the Verification

The Code-Bones that verify the prediction of the transformation are something unique to Apollo. These bones expose new optimization opportunities that computation bones do not. They have two major properties to exploit:

- Verification bones do not write into memory.
- Most of the time, verification bones do not participate in dependencies.

The following explained optimizations focus on verification bones whose memory accesses are modeled as linear, and that do not participate in dependencies. This type of bone is the most recurrent verification bone that we handle in practice.

1. The first optimization consists of moving all such kind of Code-Bones into a separate loop nest, to be executed before the rest of the code. This loop nest is optimized separately with Pluto, with maximal fission, and never tiled. From multiple experiments, we found this way to be the best performing optimization approach. In the resulting nest, every loop is parallel. Since it is executed before the rest of the code – that participates in dependencies – it enables an early detection of any misspeculation. We depict this optimization with an example in Figure 5.7, where it is assumed that Verification_2(vi,vj,vk) participates in a dependency, while the rest of the verification bones do not.

2. In the second optimization, we address Code-Bones having the same properties as before, but where all the coefficients of the predicting linear functions at a given

[2]Briefly described in Section 2.3.

```
for(vi=0; vi<N; ++vi) {
  Verification_0(vi);
  for(vj=0; vj<N; ++vj)
    for(vk=0; vk<N; ++vk) {
      Verification_1(vi,vj,vk);
      Verification_2(vi,vj,vk);
      Computation(vi,vj,vk);
    }
}
```

```
for(vi=0; vi<N; ++vi)
  Verification_0(vi);

for(vi=0; vi<N; ++vi)
  for(vj=0; vj<N; ++vj)
    for(vk=0; vk<N; ++vk)
      Verification_1(vi,vj,vk);

for(vi=0; vi<N; ++vi)
  for(vj=0; vj<N; ++vj)
    for(vk=0; vk<N; ++vk) {
      Verification_2(vi,vj,vk);
      Computation(vi,vj,vk);
    }
```

Figure 5.7: Before the optimization 1 (Left). After optimization (Right).

```
for(vi=0; vi<N; ++vi)
 for(vj=0; vj<N; ++vj)
  for(vk=0; vk<N; ++vk)
   if(&(A[vi] + vk) != 400*vi+0*vj+8*vk)
    rollback()
```

```
for(vi=0; vi<N; ++vi)
  for(vk=0; vk<N; ++vk)
    if(&(A[vi] + vk) != 400*vi+8*vk)
      rollback()
```

Figure 5.8: Before optimization 2 (Left). After optimization (Right).

loop level are 0. For that loop level, the input of the code bone remains invariant, since all the predictions and original address computations are not affected by changes in the parent virtual iterator. Hence, verifying only one iteration for this loop level is enough. We depict this optimization with the example in Figure 5.8. From the previous optimized code, we focus on the nest containing Verification_1(vi,vj,vk). This bone verifies an access to an array using an indirection. Virtual iterator vj does not participate in any computation of this bone, since the coefficients multiplying it are equal to 0. We can safely remove this loop.

3. The computation of a target address may be composed of sub-parts that are obviously linear by construction. Hence, the verification of such linear sub-parts may consume useless computation time. The last optimization reduces the verification to sub-parts that are not directly defined as linear, by removing loops related to linear sub-parts, and by reducing the expression defining the target address to the sub-parts that requires verification. The result of this last optimization is shown in Figure 5.9 for the example. Focus on the computation of the original pointer value, A[vi] + vk. We can perform the verification at one iteration for a particular value of vk, and then assume that the verification will be obviously successful for vk+1, vk+2, ... since the pointer is incremented with a constant step. However, for vi it is not possible, since

```
for(vi=0; vi<N; ++vi)
  if(&(A[vi]) != 400*vi)
    rollback()
```

Figure 5.9: Continue from Figure 5.8, after optimization 3.

A[vi] may spawn any value. We can finally remove the vk loop. Thus, we reduce the complexity of the verification from $\mathcal{O}(N^3)$ to $\mathcal{O}(N)$.

5.6.2 Automatic Tiling

To automatically detect if tiling is profitable, Apollo reuses the raw profiling information (instead of the modeling linear functions). A very simple and fast analysis is performed. Apollo computes distance vectors for every memory address collected in the innermost loops. Every dependency type – even Read-After-Read – is taken into account. Finally, if reuses in multiple directions are detected for these accesses, tiling is enabled; if not disabled. This remains as a coarse approach, but practical and successful in most cases.

An alternative to our approach would have been to directly use the prediction model to compute information about reuses to decide if tiling is profitable. However, these computations would rely on complex integer linear programming which are too time-consuming to be performed at runtime.

The tile sizes are always kept to the default value (32) since, as it was already mentioned, it is not possible to change them through Pluto's library interface.

5.7 Scanning

The transformed OpenScop representation is passed to CLooG to compute scanning loops. The output of CLooG is a syntax-tree-like data structure that defines how to iterate over each statement of the polyhedral representation.

CLooG can also optimize the control of the generated code, at the price of increasing the code size. We are compelled to deactivate this option since it greatly increases CLooG's total execution time. Additionally, an increased code size also slows down the LLVM Just-In-Time compiler.

5.8 Just-In-Time compilation

The last step in our code generation process is to finally generate binary executable code. For this task, we use the LLVM compiler. It provides a Just-In-Time compiler and a framework for applying optimizations to our target program before final native code generation.

The process consists of generating final LLVM-IR from CLooG's output and from the Code-Bones in their IR representations. This final LLVM-IR is then passed to the JIT compiler for generating binary code.

The first step is to replace all the coefficients in the Code-Bones with constant values. As mentioned before, the function get_coefficient is used to obtain these values inside the Code-Bones. All the invocations of this function are replaced by their result. The new constant values expose new optimization opportunities, since many of them are equal to zero or one, or are powers of two, participating in multiplications, additions or branch conditions.

A dedicated compiler pass transforms CLooG's output to LLVM-IR. CLooG's output corresponds to some constructs in C code, like for-loops, if-conditions and statements (in our case, Code-Bones). Translating these constructs to IR is obvious, except for Code-Bones. We translate a Code-Bone invocation as a call to the Code-Bone (remember that they are represented as LLVM functions). If the Code-Bone is a verification bone, then a branch to the rollback instruction is inserted. All Code-Bones are marked with the *always inline* property, to force inlining of their bodies. Additionally, metadata is attached to loops marked as *vectorizable* by Pluto, to guide the forthcoming LLVM JIT optimizations. These latter optimizations include constant propagation, instruction combining, strength reduction, loop unrolling, vectorization, loop unswitching, etc... among others. In Figure 5.10, we show a final version of the previous Code-Bones examples. Remember that our initial Code-Bones were merged into a single one, that verifies mem1, then mem2, and finally executes the store that accesses mem2.

Finally, the LLVM Just-In-Time compiler generates binary code which is then launched in a next chunk. The generated binary code will be reused by the successive optimized chunks, until a misspeculation occurs, which invalidates the previous prediction model.

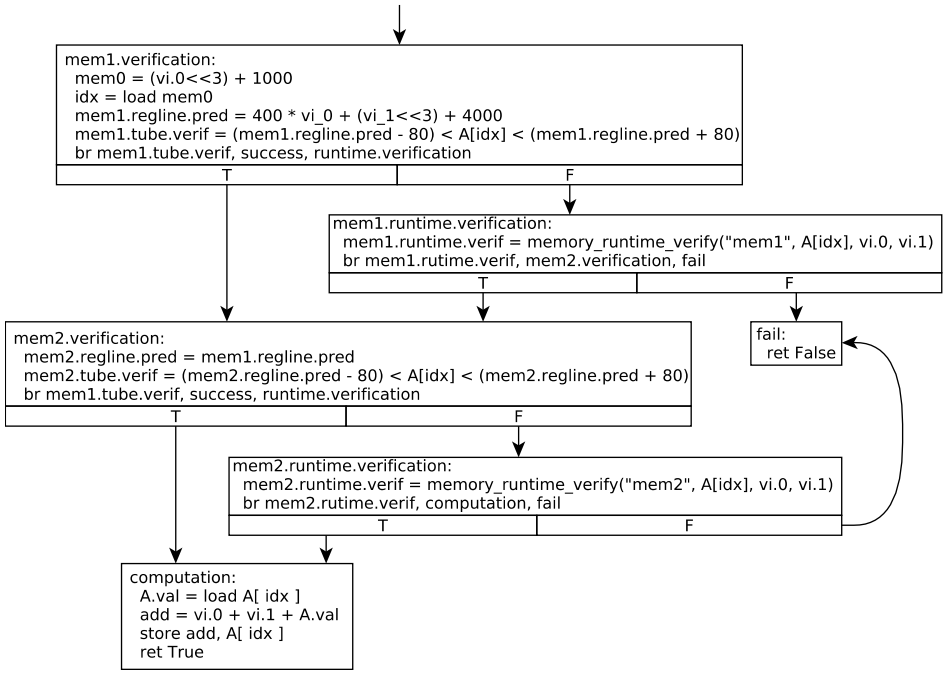

Figure 5.10: Optimized Code-Bone.

5.9 Related work

The work of Nuzman *et al.* [23] shows the feasibility of using fat binaries, which combine the IR together with the statically compiled executable. Hardware-specific optimizations or feedback-directed optimizations are applied to the target code by generating instrumented and optimized versions at runtime. A JIT compiler runs as a background thread, while the main thread keeps performing the main target code computations. However, their implementation does not yet exploit the full potential of the approach. We use a related approach, but dedicated to speculative and advanced loop optimization.

Another work proposes VaporSIMD [22] which is an hybrid mechanism devoted to the adaption of loop vectorization to the current underlying architecture. At compile-time, a bitcode vectorized version of the code is generated. This version contains special annotations to facilitate code generation by the JIT compiler. Then, at runtime, efficient target machine code is generated. Notice that the target code can be fully analyzed at compile-time to be identified as a candidate for vectorization. VaporSIMD does not involve any runtime dependence analysis or speculation. However, neither approaches considers dynamic speculative transformations decided at runtime.

In deGoal [8, 9], lightweight Just-In-Time code generation is achieved by the use of compilettes. A compilette is a specialized code generator that contains only the

strictly necessary processing capabilities to perform the required optimization. In this approach, the programmer writes the compilette in a mix of C with a specialized language. Once the runtime data is available, the application invokes the compilette to generate all the required code. Then, register allocation is performed and machine instructions are selected and scheduled.

5.10 Final remarks

The Code-Bones approach is the last piece of the puzzle to enable polyhedral and speculative parallelization. Before this approach, TLS systems had to conform with mere parallelization or very limited polyhedral transformations through the use of Code-Skeletons. Quickly, we realized that this was insufficient.

Our technique provides the flexibility and the performance required to handle any polyhedral transformation. This new flexibility can be used to cover new behaviors, such as non linear loop bounds, and enables advanced transformations of codes for which it was impossible before.

Furthermore, we show three optimization strategies dedicated to verification code. These optimizations vastly improve performance for the codes handled by Apollo.

All these code generation mechanisms are put at work in the next Chapter, on a set of representative benchmark programs.

Chapter 6

Experiments

In this Chapter, we evaluate the Apollo framework enhanced thanks to our code generation process using Code-Bones. The reported results are obtained from five executions of each benchmark program on two different machines:

- **Lemans**, a general-purpose multi-core server, with two AMD Opteron 6172 processors of 12 cores each (24 in total).
- **Armonique**, an embedded system multi-core chip , with one ARM Cortex A53 64-bit processor of 8 cores.

The set of benchmarks has been built from a collection of benchmark suites, such that the selected codes include a main kernel loop nest and highlight Apollo's capabilities: `SOR` from the Scimark suite [27]; `Backprop` and `Needle` from the Rodinia suite [10]; `DMatmat`, `ISPMatmat`, `Spmatmat` and `Pcg` from the SPARK00 suite of irregular codes [33]; and `Mri-q` from the Parboil suite [28]. In Table 6.1, we identify the characteristics for each program that make it impossible to parallelize at compile-time. Where:

- *Has indirections* means that the kernel loop accesses memory through array indirections.
- *Has pointers* means that the kernel loop accesses memory through pointers.
- *Unpredictable bounds* means that some loop bounds cannot be known at compile-time.
- *Unpredictable scalars* means that the values taken by some scalars cannot be known at compile-time.

Benchmark	Has indirections	Has pointers	Unpredictable bounds	Unpredictable scalars
Mri-q		✓		
Needle		✓		
SOR	✓	✓		
Backprop	✓	✓		
PCG	✓	✓	✓	✓
DMatmat	✓	✓		
ISPMatmat	✓	✓	✓	✓
SPMatmat	✓	✓	✓	✓

Table 6.1: Characteristics of each benchmark program.

For the SPMatmat kernel, five inputs width different data layouts were used to highlight some key features of Apollo.

In Table 6.2, we show the number of computation and verification bones for each program and the transformations that were selected by Pluto at runtime.

Benchmark	Number of Computation Bones	Number of Verification Bones	Selected Optimization
Mri-q	2	1	Interchange
Needle	1	1	Skewing + Interchange + Tiling
SOR	1	6	Skewing + Tiling
Backprop	2	4	Interchange
PCG	21	33	Identity
DMatmat	1	5	Tiling
ISPMatmat	1	8	Tiling
SPMatmat	1	10	Tiling

Table 6.2: Transformations selected by Pluto at runtime and number of Code-Bones of each type for each benchmark.

In Section 6.1, we study the execution-time performance from using Apollo, and in particular, from using the Code-Bones for dynamic code generation. In Subsection 6.1.1, we show how performance is affected by the larger amount of verification required for memory accesses modeled as tubes. In Subsection 6.1.2, we show how the optimization specific to the verification code improves the performance of two selected benchmarks. In Section 6.2, we show the most time consuming phases of Apollo, and the percentage of time spent in each of them. The time overheads for our code generation approach are studied in Section 6.3. Finally, in Section 6.4, we give some conclusions derived from these experiments.

6.1 Performance

The next Figures show the speedups provided by Apollo over the original sequential program either when using Code-Bones or when using Code-Skeletons. Since Apollo is built on top of the LLVM, the original sequential program has been compiled with Clang-3.8. However, we also compare Apollo's execution-time against the best execution time obtained for the original sequential code compiled either with Gcc-5.3 or with Clang-3.8.

We consider the average of five runs, using the original target code compiled using the Clang compiler with optimization level 3 (`-O3`) as our baseline (1× in the y-axis). In yellow, we show the best average among the codes produced by Gcc and Clang, from five executions of the original target code. Notice that this best average may be the baseline when Clang generates the best performing code. In red, we show the average speedup, from five executions, using Apollo with the Code-Skeletons mechanism. In blue, we show the average speedup, the maximum speedup, and the minimum speedup, obtained from five executions, using Apollo with the Code-Bones mechanism.

For Mri-q (see Figure 6.1), both code generation mechanisms reach similar performance on Lemans. This was expected, since both approaches actually perform the same code optimization. However, on Armonique, Code-Bones outperforms the Code-Skeletons approach, even if the same transformation was applied. This may happen because code generated using the Code-Bones tends to be simpler, with less branch instructions, than the counterpart version generated using Code-Skeletons which are too generic. Both approaches greatly obtain a linear or quasi-linear speedup over the sequential version, as the number of cores assigned to the computation increases.

For Needle (see Figure 6.2), the Code-Bones outperforms the Code-Skeletons on both machines, thanks to an optimizing tiling transformation that is not supported by the available skeletons. On Lemans, as the number of available cores increases – specially after 8 cores – the difference between the obtained maximum and minimum speedups grows up to 20×. This is caused by some load balancing issues among the threads. Indeed, some threads remain idle, waiting for the rest of the threads to complete, to be allowed to continue with the following chunk of iterations. The parallel loop of the optimized code corresponds to the second triangular outer loop, which iterates over the tiles. This has three consequences: there is a small number of iterations that is assigned to the parallel loop, since it scans the tiles; the parallel loop resides inside a sequential loop. Thus, at each iteration of the sequential loop, there is a synchronization to be performed among the threads; since the parallel loop is triangular, at some point, the number of iterations to be distributed to the different threads is less

than the number of available threads. Both approaches achieve better performance than the sequential versions compiled with Clang and Gcc, even when a single core is used, thanks to better data locality.

For SOR (see Figure 6.3), Code-Bones again achieve slightly better performance than the Code-Skeletons, thanks to a tiling transformation. Here also, both approaches outperform the sequential versions compiled with Clang and Gcc.

For Backprop (see Figure 6.4) on Lemans, both code generation approaches achieve similar performance, with a small advantage towards the Code-Bones. This was expected since both approaches perform the same optimizing transformation (a loop interchange). However, on Armonique, there is a much bigger difference, even if both approaches apply the same transformation. This might be explained, as with Mri-q, by a less complex optimized code that was generated using Code-Bones. On both machines, for 1 thread, the Code-Bones approach already exhibits some speedups over the serial versions compiled with Clang and Gcc, thanks to the optimizing transformation.

PCG (see Figure 6.5) is a particular kernel yielding a large number of bones participating in the target code. As a consequence, it is impossible for Pluto to provide an optimizing transformation in a reasonable amount of time. To mitigate this problem, when the target code posses more than 5 computation bones, Pluto is only used for dependency analysis to determine which loops in the original code can be run in parallel (Pluto is invoked with flag –identity). Additionally, the Code-Bones where fused at maximum to 21 bones. This code posses several parallel innermost loops. The frequent synchronization at the end of the execution of each parallel loop imposes a time-overhead that harms performance as the number of threads increases. On Armonique, the Code-Skeletons achieve slightly better performance than the Code-Bones approach. However, on Lemans, the Code-Bones achieve significantly better performance, until the synchronization overhead has a very penalizing impact.

For DMatmat and ISPMatmat, in Figures 6.6 and 6.7, the Code-Bones result in much better performance compared to the Code-Skeletons. This happens thanks to tiling. As the number of threads increases, the code generated using Code-Bones continues to scale, while the code generated with Code-Skeletons hits a performance wall due to data locality issues.

The SPMatmat kernel was evaluated with different kinds of inputs: a square matrix, a diagonal matrix, a matrix that yields memory accesses modeled as tubes, and two matrices yielding cases where Apollo fails to optimize the target code. For all these inputs, a tiling transformation was performed.

The square matrix exhibits a single linear phase covering the entire execution, where Apollo obviously succeeds in optimizing. The diagonal matrix exhibits two different

phases along the execution time: an initial linear phase which is successfully optimized, and a final phase with memory accesses modeled as tubes, and a basic scalar modeled as a reduction, which is unfortunately not supported by Apollo. Hence, the original version of the code is run to complete this last phase. Figures 6.8 and 6.9 show the performance results for these two matrices. The Code-Bones reach better performance compared to the Code-Skeletons. Recall that Lemans has 2 processors with 12 cores each. After the 12 threads limit, the available memory bandwidth is not sufficient for the optimized code to continue scaling. Therefore, very weak additional performance is gained beyond this limit.

The performance obtained when the input matrix exhibits memory accesses modeled as tubes is shown in Figure 6.10. With this input, two memory accesses are modeled as tubes. However, during the execution, most accesses fall outside the predicted tubes and thus rely on the more complex runtime verification, where addresses are checked if they are located in a read-only memory area. This imposes a large verification overhead. A performance improvement over the sequential version of the code is reached only when many cores are assigned to the computation. Both code generation approaches achieve very similar performance since the bottleneck is now the verification code, and not the memory bandwidth. Hence, the improvement provided by optimizing transformations that address data locality is insignificant.

The last two inputs exhibit scenarios where Apollo fails to optimize the code, shown in Figures 6.11 and 6.12. With the first input, after interpolation and regression, some basic scalars are modeled as reductions. Since these scalars are not supported by the underlying polyhedral tools, Apollo resumes the execution using the original version of the code. During the whole execution, Apollo alternates between instrumented and original executions. With the second input, Apollo succeeds in generating optimized code, but as soon as it is executed, a misspeculation is detected and rollback is signaled. On Lemans, for both inputs, the final execution time is 20% slower in the worst case than the original sequential code compiled with Clang, and 10% slower in the average case. The version compiled with Gcc outperforms the rest. Since Apollo is built on top of Clang, the performance that Apollo can achieve remains bounded by Clang's. A worst case scenario occurs with a single thread. Since only one thread is available, execution is interrupted to perform code generation, but this overhead is not compensated by the optimized execution of the code, resulting in a high time overhead. On Armonique, the performance stays very close to the original sequential code compiled with Clang.

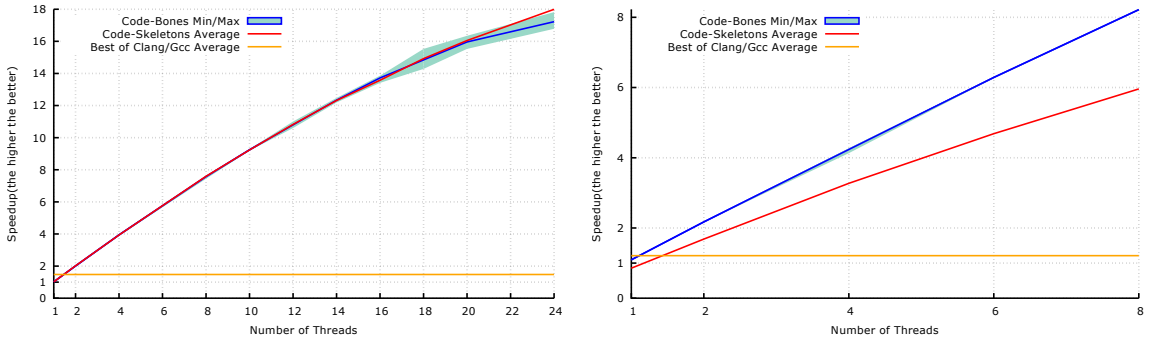

Figure 6.1: Mri-q: Speedup for different thread configurations, for Lemans (left) and Armonique (right).

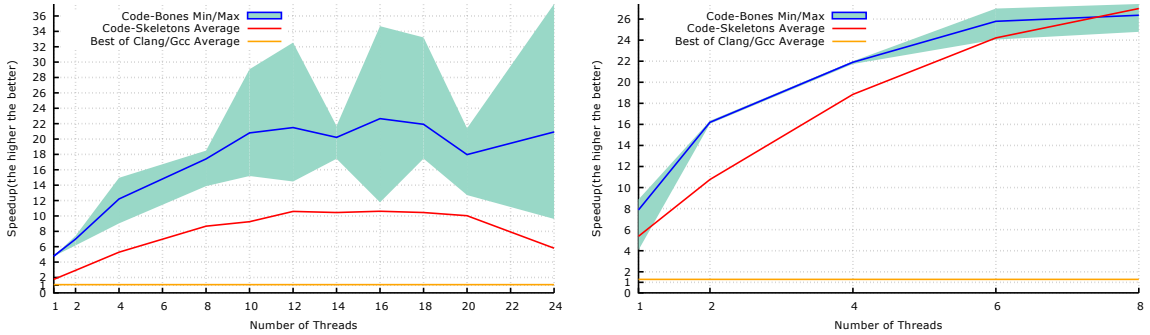

Figure 6.2: Needle: Speedup for different thread configurations, for Lemans (left) and Armonique (right).

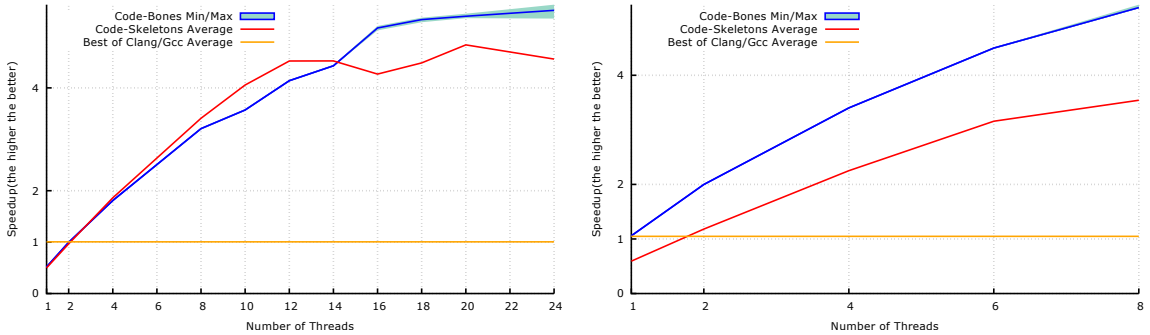

Figure 6.3: SOR: Speedup for different thread configurations, for Lemans (left) and Armonique (right).

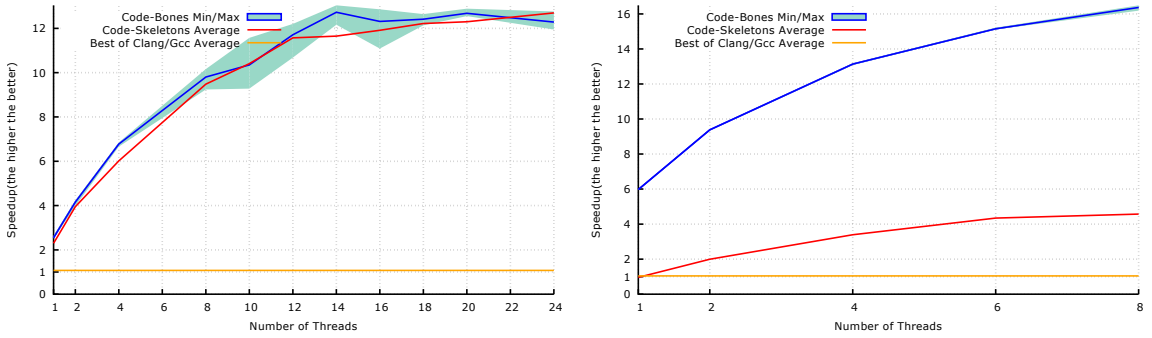

Figure 6.4: Backprop: Speedup for different thread configurations, for Lemans (left) and Armonique (right).

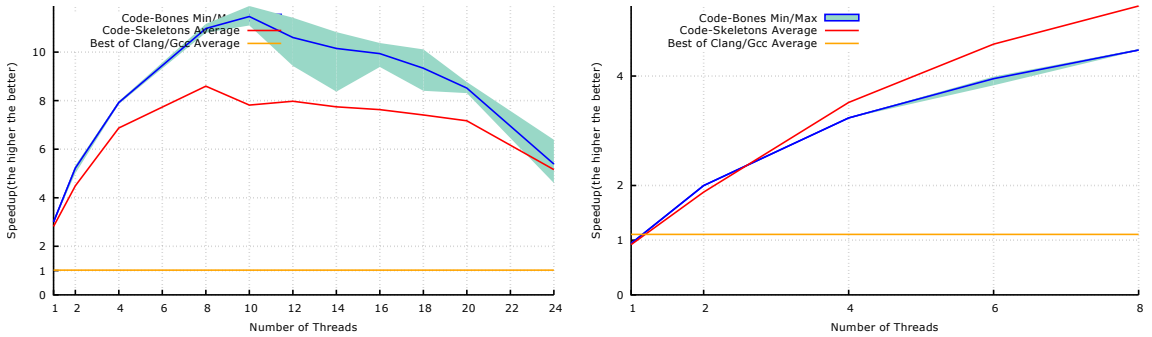

Figure 6.5: PCG: Speedup for different thread configurations, for Lemans (left) and Armonique (right).

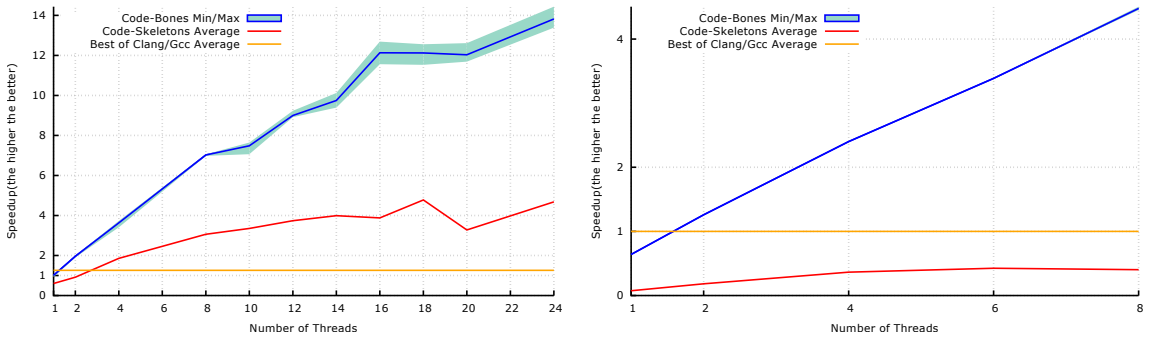

Figure 6.6: DMatmat: Speedup for different thread configurations, for Lemans (left) and Armonique (right).

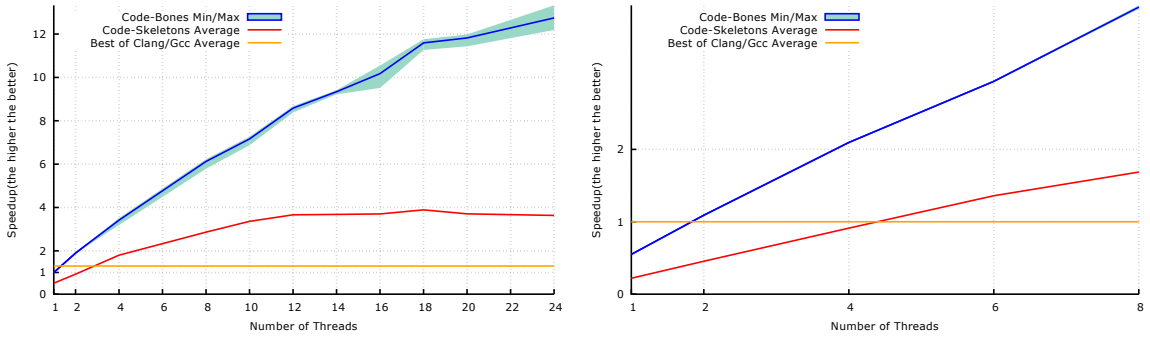

Figure 6.7: ISPMatmat: Speedup for different thread configurations, for Lemans (left) and Armonique (right).

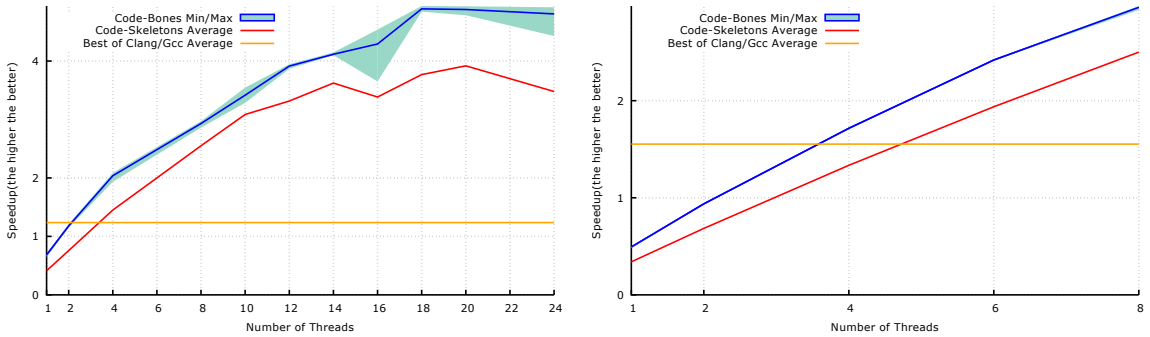

Figure 6.8: SPMatmat Square: Speedup for different thread configurations, for Lemans (left) and Armonique (right).

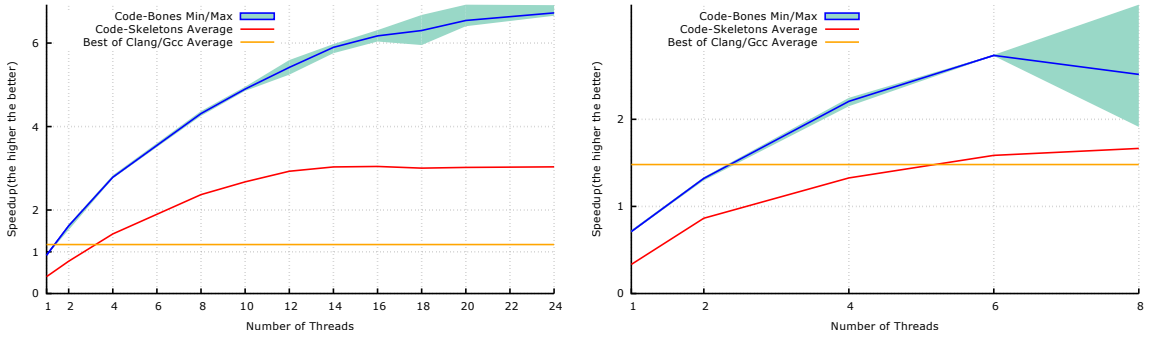

Figure 6.9: SPMatmat Diagonal: Speedup for different thread configurations, for Lemans (left) and Armonique (right).

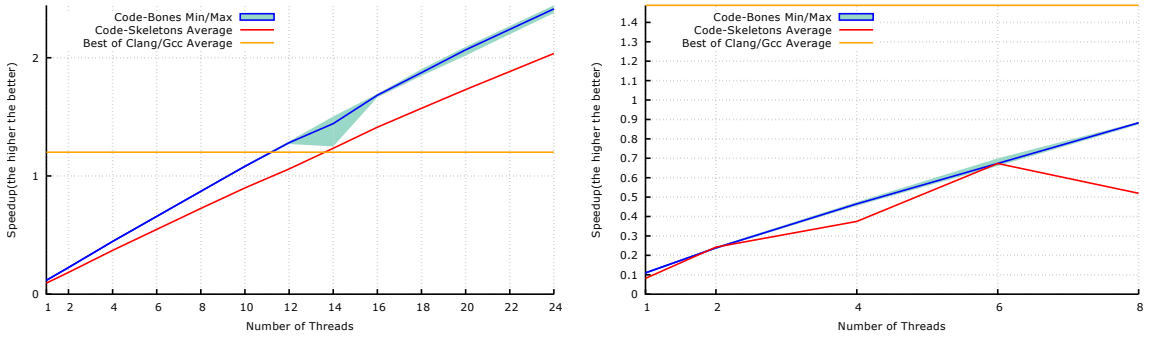

Figure 6.10: SPMatmat Tube: Speedup for different thread configurations, for Lemans (left) and Armonique (right).

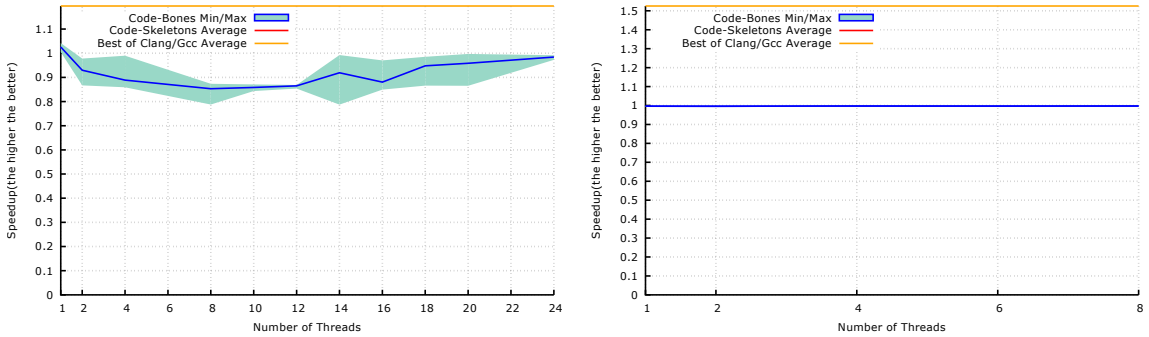

Figure 6.11: SPMatmat Random: Speedup for different thread configurations, for Lemans (left) and Armonique (right).

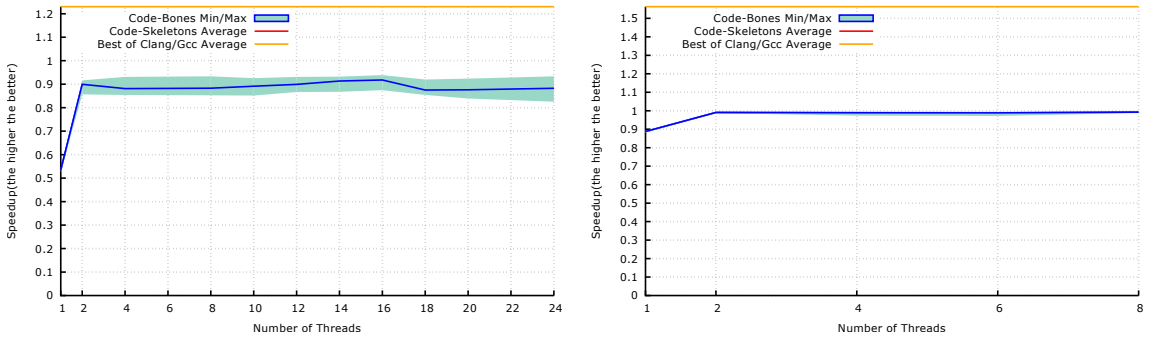

Figure 6.12: SPMatmat Worst Case: Speedup for different thread configurations, for Lemans (left) and Armonique (right).

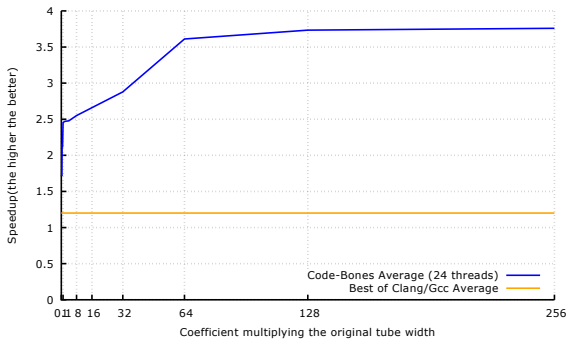

Figure 6.13: SPMatmat Tube: Speedup for 24 threads, varying the tube width, for Lemans.

6.1.1 Performance of the tube verification

In Figure 6.13, for SPMatmat, we show how the performance varies while varying the tube width. For this, we multiply the original tube width by a constant coefficient. The amount of memory accesses predicted to be inside the tube, and requiring a more complex verification, change as we modify this coefficient. Varying the tube width does not affect the applied transformation, in this case, since the memory accesses predicted as tube do not participate in any dependency. As expected, taking values below the purposed tube width results in poor performance. In this particular example, as the execution progresses, the amount of memory accesses that are outside the tube increases. These accesses rely on a more complex runtime check to ensure that no dependency was violated. Thus, considering a tube width larger than the default can reduce the verification overhead and improve performance, until all memory accesses fall inside the tube. After this point, no performance improvement can be expected by augmenting the tube width.

6.1.2 Optimization of the Verification

In Figures 6.15 and 6.14, we show the different speedups for Apollo, using the Code-Bones, with and without the verification dedicated optimization presented in this thesis. The evaluated kernels are SOR and SPMatmat with a square matrix as input.

As expected, we can see that the version where the verification code was optimized outperforms the other version.

For SOR, five out of six verification bones are optimized. These correspond to bones verifying memory accesses, with three parent loops, which do not participate in any dependency. For the five accesses, Apollo is able to remove two out of the three loops: the outermost loop, whose iterator does not contribute to any computation inside

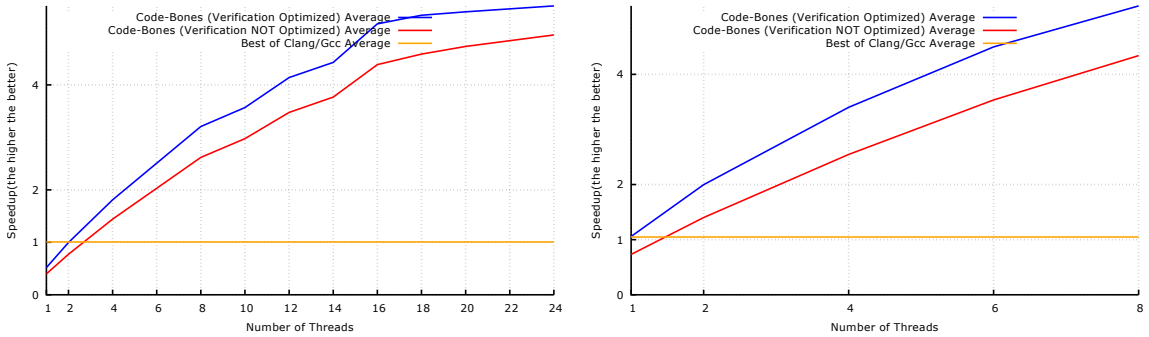

Figure 6.14: SOR: Speedup for different thread configurations, with and without optimizing the verification code, for Lemans (left) and Armonique (right).

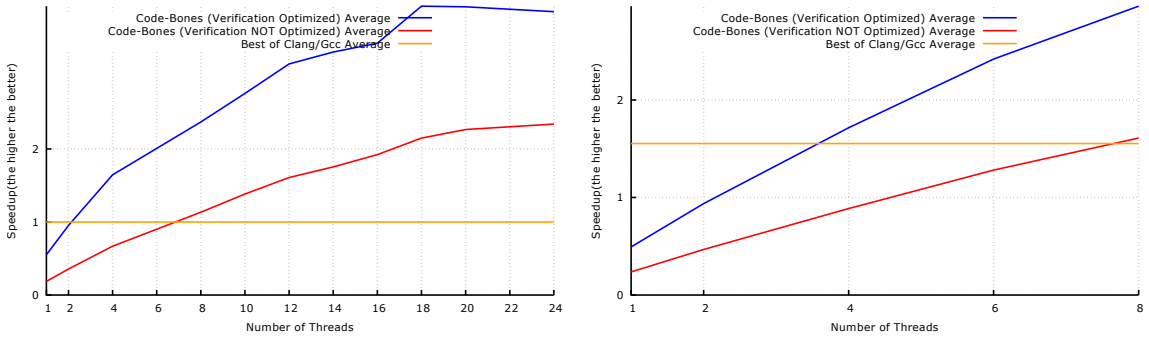

Figure 6.15: SPMatmat Square: Speedup for different thread configurations, with and without optimizing the verification code, for Lemans (left) and Armonique (right).

these bones; and the innermost loop whose computations are an affine combination of computations residing in the second loop.

For SPMatmat, two out of ten verification bones are successfully optimized. Both of them reside in the innermost loop of the 3-depth loop nest. For these bones, Apollo is able to remove a single loop, whose iterator does not contribute to any computation inside the bone.

6.2 Time Overhead

In this Section, we show the percentage of the whole execution time that is spent in the different phases of Apollo. These results correspond to executions with 8 threads. The time spent for code generation is shown separately in a second column, since it runs in parallel with the original version of the code. Therefore, for all the experiments, the time spent executing the original version of the code is obviously larger than the time spent in performing code generation.

For most of the experiments, most of the time is spent in the parallel and opti-

mized region of the target code, and just a small fraction of the time is spent in the instrumentation phase. The time required to perform backup and rollback is negligible compared to the time spent in other phases.

For Mri-q, in Figure 6.16, more than 80% of the time is spent running optimized code. This explains why this benchmark is successfully scaling.

In Figure 6.17, we show the time percentages for Needle. On Lemans, more than 90% of the execution time is spent running optimized code. In contrast, on Armonique, a significant part of the execution time is spent for code generation. This bounds the performance gain that may be reached with Apollo, which is reflected with the flattening of the speedup curve as the number of threads increases. The difference between the times on each machine is caused by inputs with different sizes, since Armonique could not fit the input used with Lemans into memory.

For SOR, in Figure 6.18, a behavior similar to Mri-q can be observed. On Armonique, a significant amount of time is spent in the code generation phase, and in particular while invoking the Just-In-Time compiler. This later issue is addressed in the next Section.

For PCG, in Figure 6.20, an important amount of time is spent in the code generation phase. This is a consequence of the high number of Code-Bones that are handled with this benchmark. In the next Section, we will see that on Armonique, the time spent in this code generation phase is prohibitively expensive. However, parallel code is finally generated and the optimized execution still provides some performance improvement.

The time measurements for DMatmat, ISPMatmat, and SPMatmat, with a square matrix as input, are shown in Figures 6.21, 6.22 and 6.23. The percentages of time spent in each phase are very similar among these kernels. Most of the time is spent in an optimized phase, following the initial instrumentation and code generation phases.

For SPMatmat with a diagonal matrix as input (see Figure 6.24), a significant amount of time is taken by executing the original code. Remember that, with this input, this kernel exhibits two phases: a first phase, exhibiting a linear behavior which is successfully optimized; and a second phase where it is impossible to successfully generate optimized code (there is one basic scalar modeled as reduction). Hence, this final phase is executed using the original version of the code.

In Figure 6.25, we show the time measurements for the SPMatmat kernel, with the input matrix yielding accesses modeled as tubes. In this case, the time spent in the parallel region is much greater than for the rest of the phases. However, this is explained by the weak performance obtained for this benchmark, due to the increased verification overhead. In fact, the amount of time spent in the code generation phase

remains big, as shown in the next Section.

Finally, for the last two examples, in Figures 6.26 and 6.27, most of the time is spent executing the original version of the code. For the first input, which is a random matrix, a short amount of time is spent in the code generation phase, which aborts after interpolation/regression. This input never reaches any optimized code execution. However, for the other input, which is a square diagonal matrix, code generation takes significantly more time, especially since this phase is executed multiple times after previous original code executions. A very short time is spent in the optimized phase since, as soon as it starts executing, a misspeculation is detected.

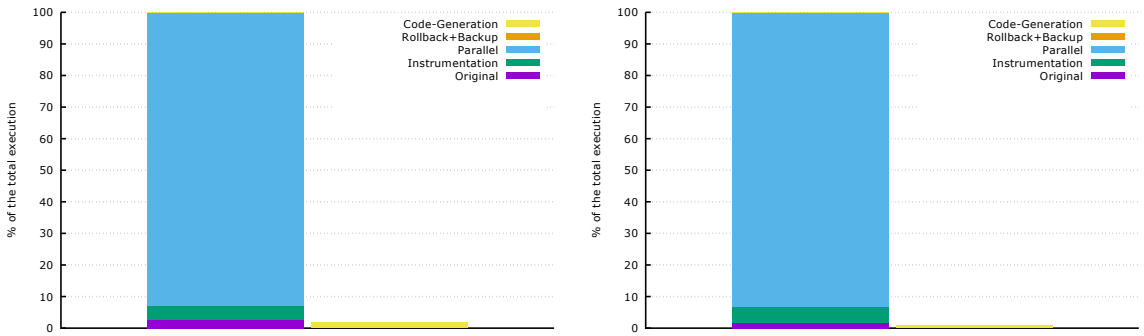

Figure 6.16: Mri-q: Percentage of the total execution time spent in each phase of Apollo, with 8 threads, for Lemans (left) and Armonique (right).

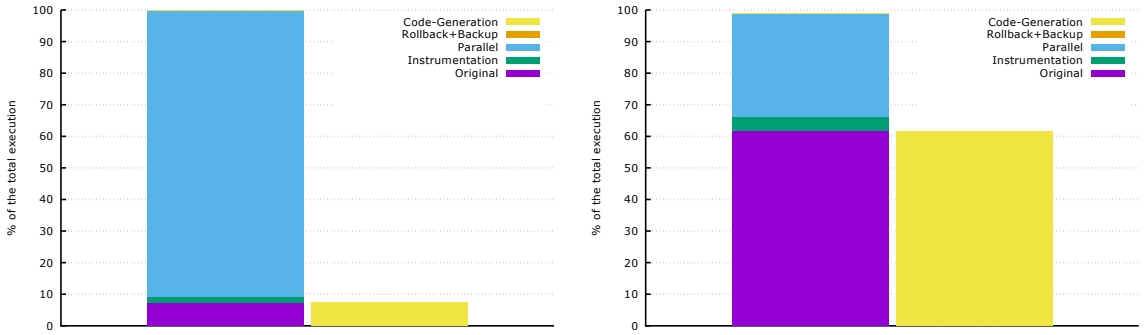

Figure 6.17: Needle: Percentage of the total execution time spent in each phase of Apollo, with 8 threads, for Lemans (left) and Armonique (right).

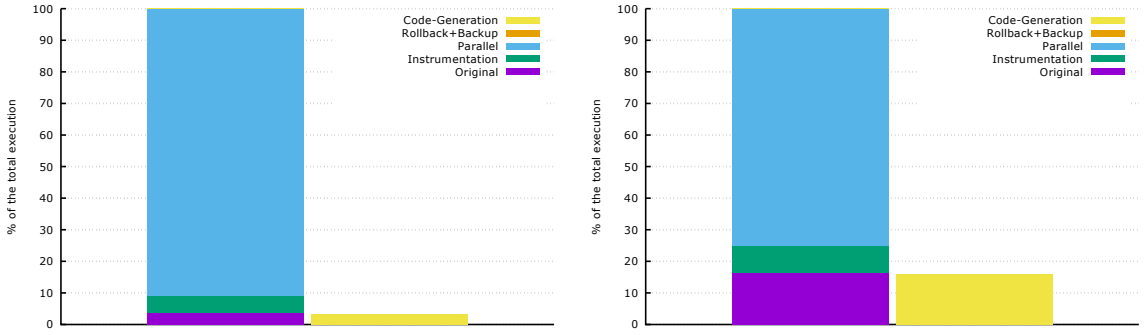

Figure 6.18: SOR: Percentage of the total execution time spent in each phase of Apollo, with 8 threads, for Lemans (left) and Armonique (right).

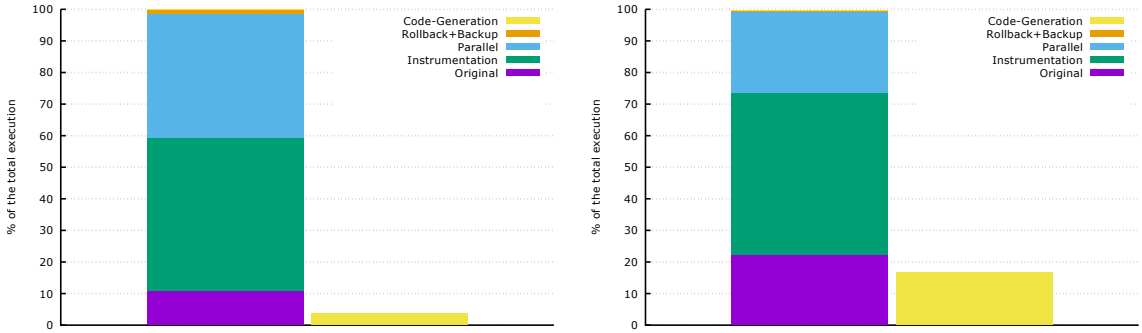

Figure 6.19: Backprop: Percentage of the total execution time spent in each phase of Apollo, with 8 threads, for Lemans (left) and Armonique (right).

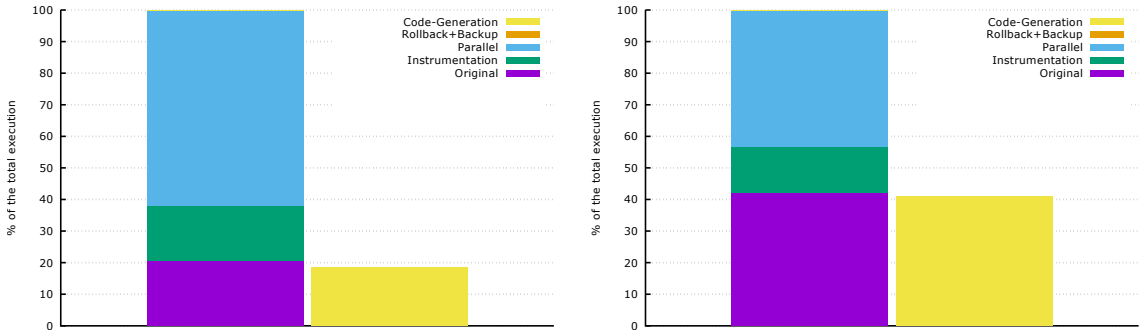

Figure 6.20: PCG: Percentage of the total execution time spent in each phase of Apollo, with 8 threads, for Lemans (left) and Armonique (right).

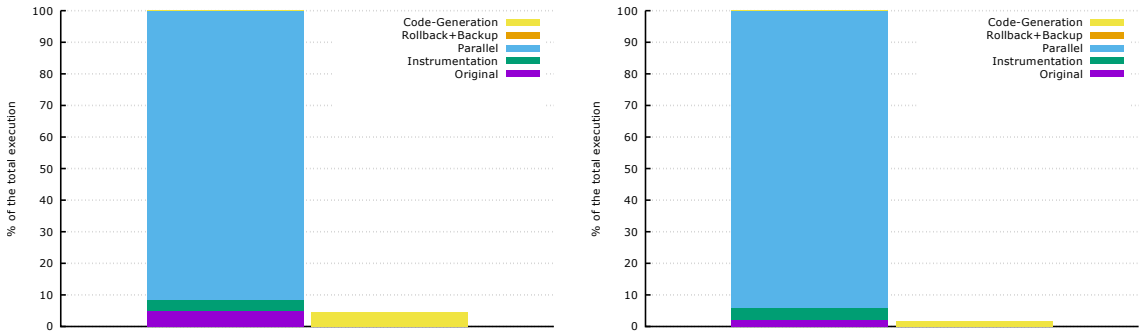

Figure 6.21: DMatmat: Percentage of the total execution time spent in each phase of Apollo, with 8 threads, for Lemans (left) and Armonique (right).

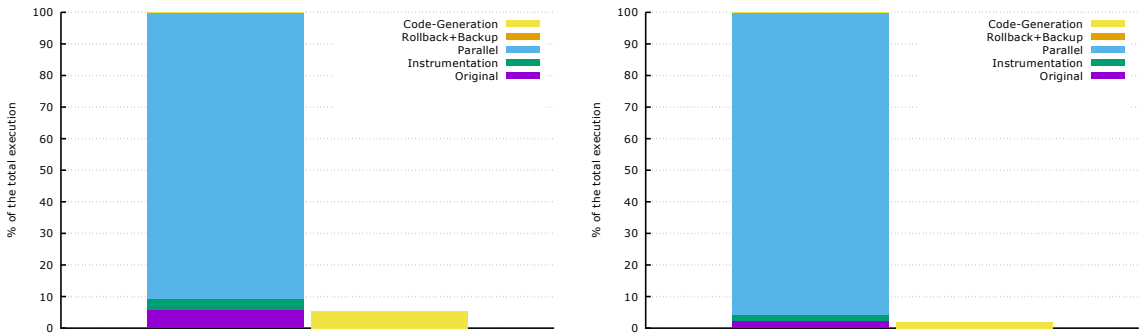

Figure 6.22: ISPMatmat: Percentage of the total execution time spent in each phase of Apollo, with 8 threads, for Lemans (left) and Armonique (right).

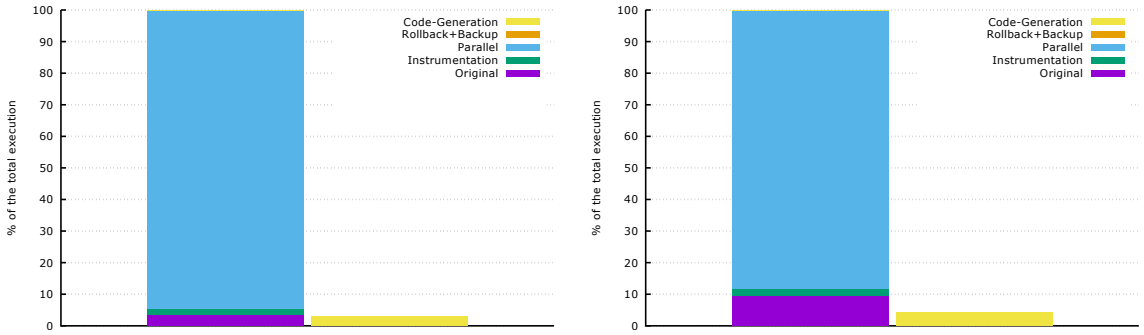

Figure 6.23: SPMatmat Square: Percentage of the total execution time spent in each phase of Apollo, with 8 threads, for Lemans (left) and Armonique (right).

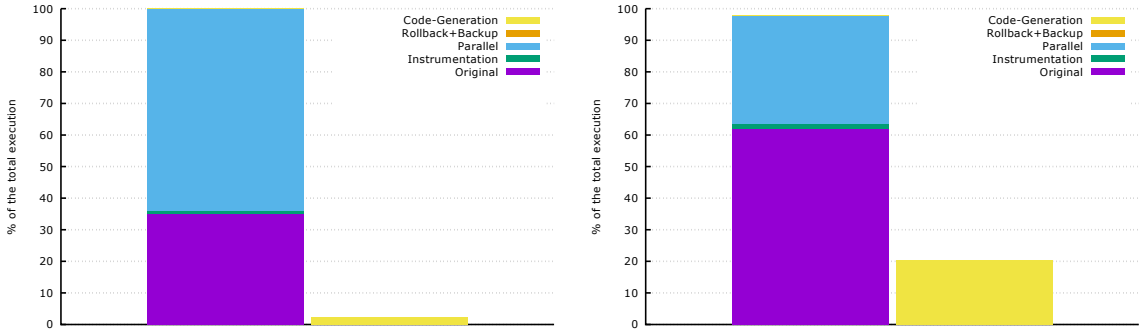

Figure 6.24: SPMatmat Diagonal: Percentage of the total execution time spent in each phase of Apollo, with 8 threads, for Lemans (left) and Armonique (right).

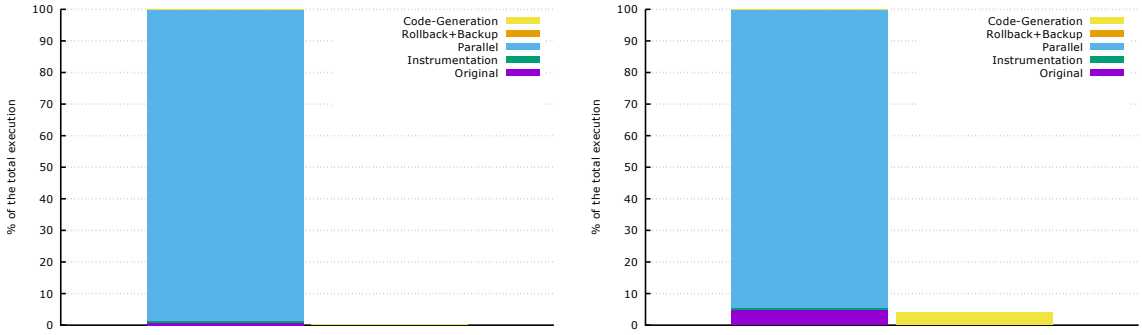

Figure 6.25: SPMatmat Tube: Percentage of the total execution time spent in each phase of Apollo, with 8 threads, for Lemans (left) and Armonique (right).

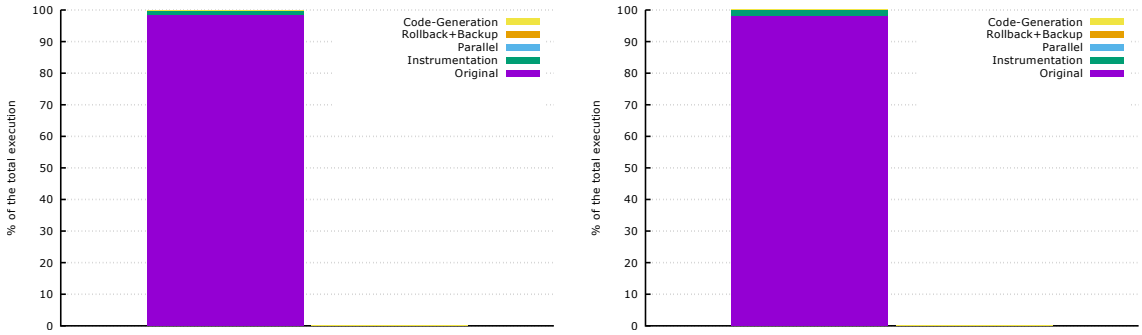

Figure 6.26: SPMatmat Random: Percentage of the total execution time spent in each phase of Apollo, with 8 threads, for Lemans (left) and Armonique (right).

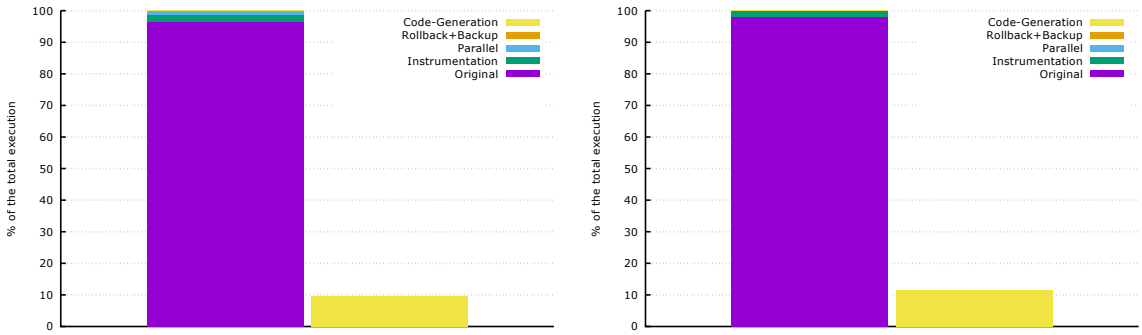

Figure 6.27: SPMatmat Worst Case: Percentage of the total execution time spent in each phase of Apollo, with 8 threads, for Lemans (left) and Armonique (right).

6.3 Code Generation Time Overhead

In this Section, we analyze the time overhead consumed by the different code generation phases of the Code-Bones approach. These results correspond to executions with 8 threads.

We consider four different phases:

- Interpolation and Encoding: the time spent while building the prediction model, reconstructing the original loop nest using the Code-Bones, performing delinearization, until finally generating an OpenScop representation of the target loop nest.

- Scheduling: the time spent performing dependency analysis, with Candl, and choosing an optimizing transformation, with Pluto.

- Scan: the time spent by CLooG generating the scan of the statements.

- Just-In-Time compilation: the time spent performing classical optimizations (constant propagation, dead code elimination, vectorization, loop unswiching), and generating the binary executable code.

For most benchmarks, the time spent in the Interpolation and Encoding phase is significantly shorter than for the rest of the phases.

The times spent by the different code generation phases remains acceptable for most experiments, which are: Mri-q (Figure 6.28), Needle (Figure 6.29), SOR on Lemans (Figure 6.30), Backprop (Figure 6.31), DMatmat (Figure 6.33), ISPMatmat (Figure 6.34), SPMatmat Square (Figure 6.35) and SPMatmat Diagonal and Tube on Lemans (Figures 6.36 and 6.37 respectively).

For SOR on Armonique, Just-In-Time compilation takes above 2 seconds to complete. This can be explained by the tiling and skewing transformations that are applied for this benchmark. Typically, the more complex a code is, the more time is required to optimize it and to generate native code. A solution could be to perform some basic optimizations on the whole loop nest, and to perform more aggressive optimizations only on the innermost loops, since it is in the innermost loops where most of the computations take place.

In Figure 6.32, the time overheads for PCG are shown. A large number of Code-Bones participates in this benchmark, slowing down our code generation approach.

For SPMatmat Diagonal and Tube, on Armonique, the high overhead for the Interpolation and Encoding phase is related to the algorithm for obtaining a regression line

describing the memory accesses. This algorithm has an algorithmic time complexity in the order of $\mathcal{O}(N^2(d+1))$ operations, where d is the loop depth (typically small, in this case equal to 3), and N is the number of instrumented samples (typically big, in this case around 16^d). Finally, for both cases, the time overhead imposed by the Just-In-Time compiler remains high due to the complexity of the final optimized code.

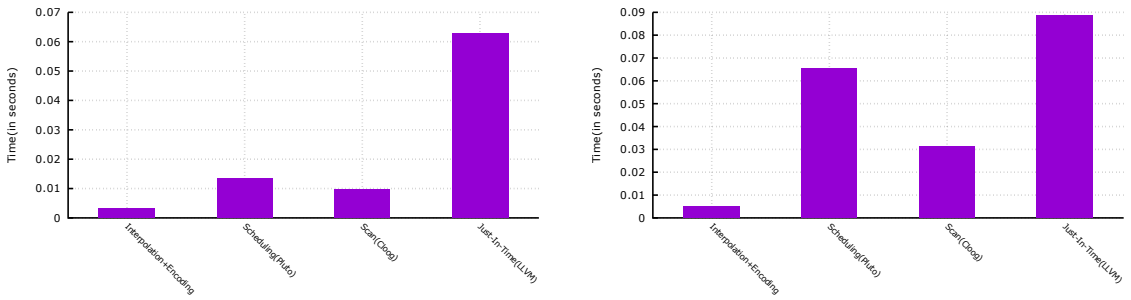

Figure 6.28: Mri-q: Time spent in each phase of code generation in Apollo, with 8 threads, for Lemans (left) and Armonique (right).

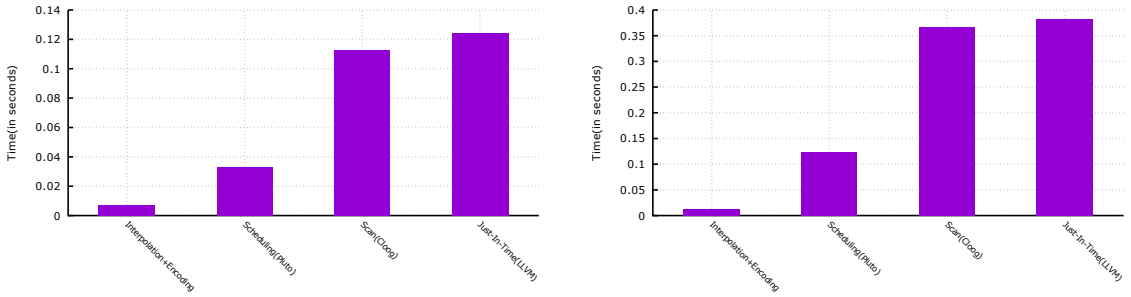

Figure 6.29: Needle: Time spent in each phase of code generation in Apollo, with 8 threads, for Lemans (left) and Armonique (right).

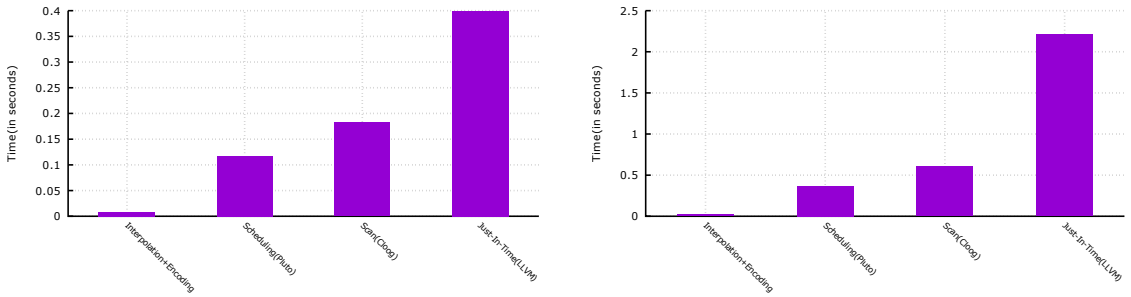

Figure 6.30: SOR: Time spent in each phase of code generation in Apollo, with 8 threads, for Lemans (left) and Armonique (right).

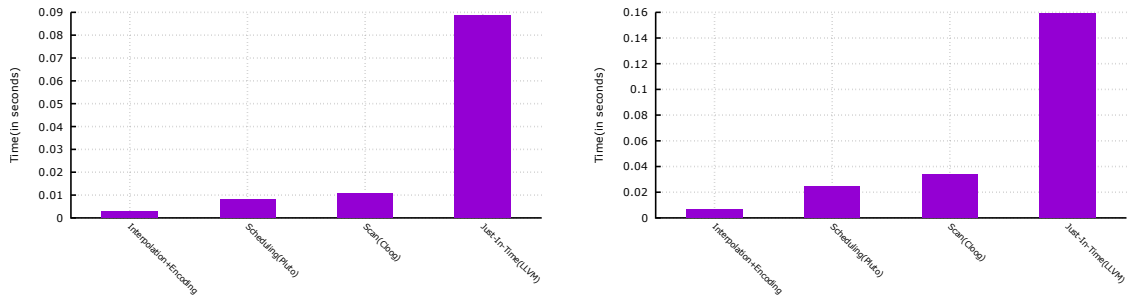

Figure 6.31: Backprop: Time spent in each phase of code generation in Apollo, with 8 threads, for Lemans (left) and Armonique (right).

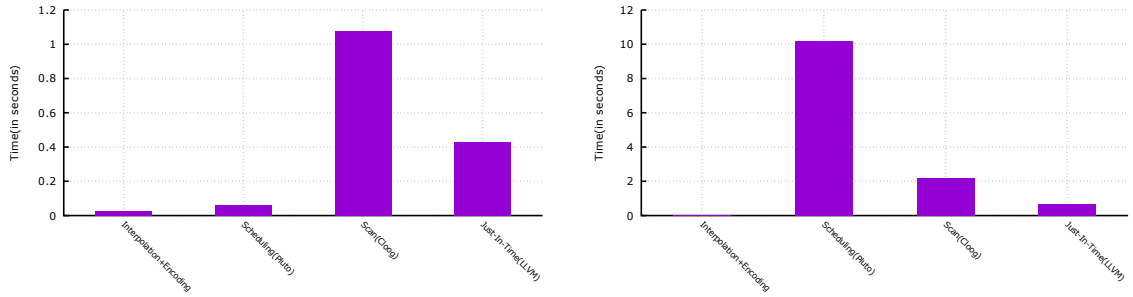

Figure 6.32: PCG: Time spent in each phase of code generation in Apollo, with 8 threads, for Lemans (left) and Armonique (right).

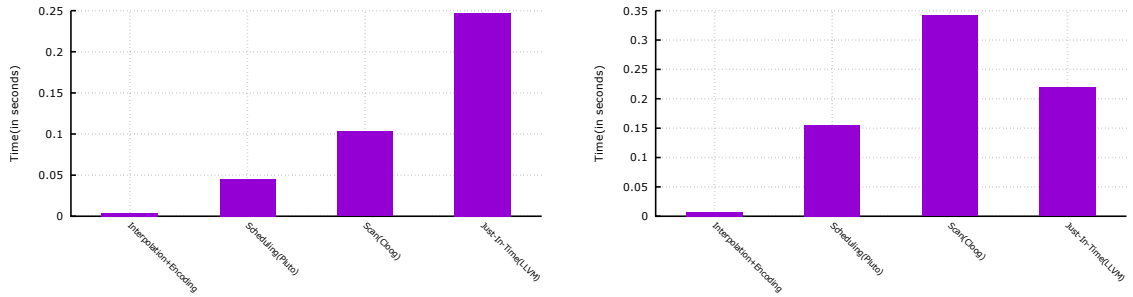

Figure 6.33: DMatmat: Time spent in each phase of code generation in Apollo, with 8 threads, for Lemans (left) and Armonique (right).

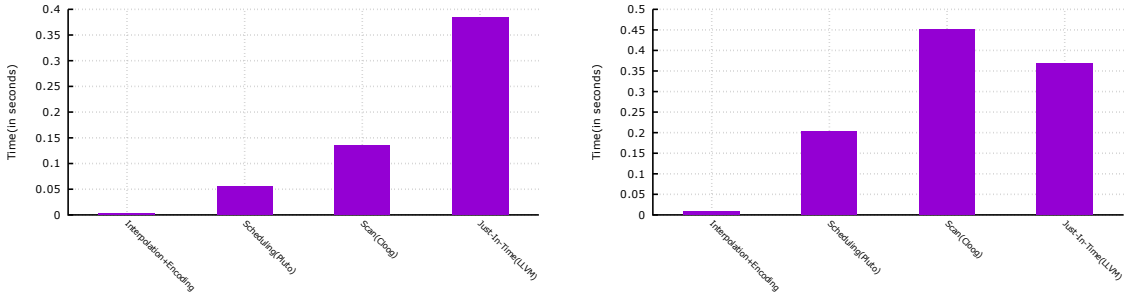

Figure 6.34: ISPMatmat: Time spent in each phase of code generation in Apollo, with 8 threads, for Lemans (left) and Armonique (right).

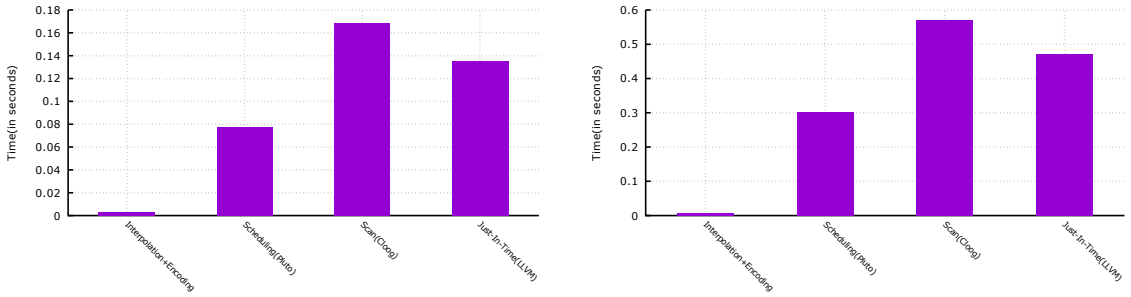

Figure 6.35: SPMatmat Square: Time spent in each phase of code generation in Apollo, with 8 threads, for Lemans (left) and Armonique (right).

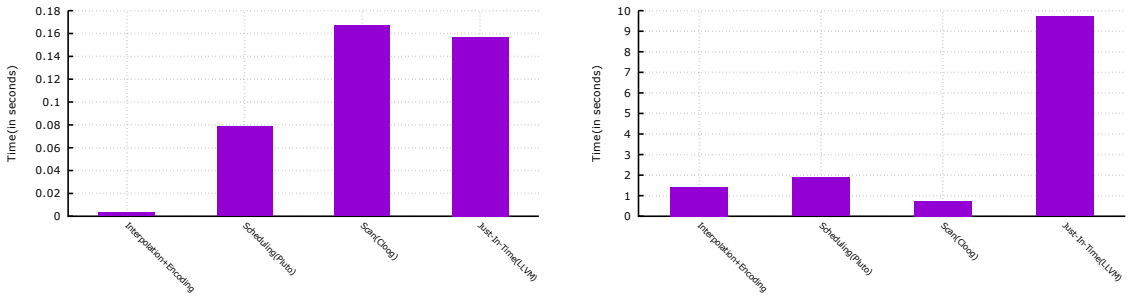

Figure 6.36: SPMatmat Diagonal: Time spent in each phase of code generation in Apollo, with 8 threads, for Lemans (left) and Armonique (right).

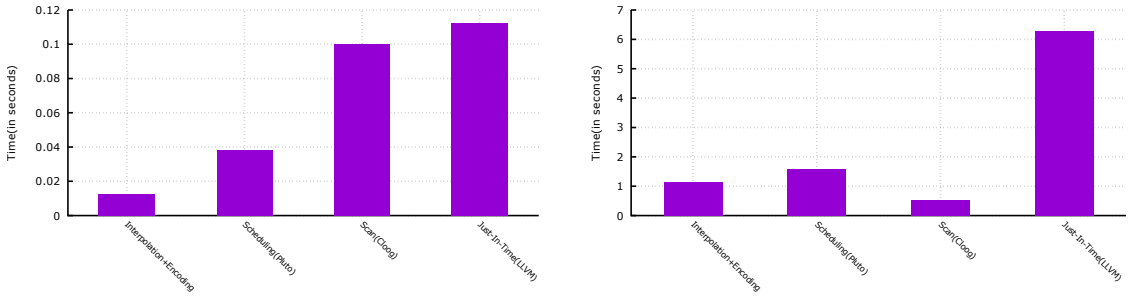

Figure 6.37: SPMatmat Tube: Time spent in each phase of code generation in Apollo, with 8 threads, for Lemans (left) and Armonique (right).

6.4 Final remarks

The code generation mechanism presented in this thesis succeeds in dynamically optimizing several codes. The new supported transformations allow the Code-Bones code generation strategy to outperform the previous Code-Skeletons strategy.

In general, our approach is successful in keeping a small time-overhead. However, there are some cases where our approach faces problems: when the number of Code-Bones participating in the target code is high and when the generated optimized code is complex. These issues are mostly due to tools that are external to Apollo: Pluto, CLooG and the LLVM JIT compiler. This gives rise to the need for compilation tools that are better adapted to dynamic parallelization and optimization. For example, a polyhedral runtime scheduler could provide solutions that are incrementally improved, while already running some initial parallel solutions.

Finally, the optimizations devoted to the verification of the speculation have been proven to be efficient.

Chapter 7

Conclusions and Perspectives

7.1 Conclusions

In this thesis, we have presented our contributions to the field of speculative parallelization and dynamic code generation. Our objective is to dynamically optimize any loop nest, as soon as it exhibits a runtime behavior that is compliant with the polyhedral model, and also to cope with the possible changes of this behavior.

For these purposes, we have implemented a speculative parallelization framework called Apollo. Our framework is able to detect such polyhedral phases, select a polyhedral optimizing and parallelizing transformation, and generate optimized code on-the-fly. In a dynamic context, performing all these operations efficiently is a difficult challenge.

The main contribution of this thesis is a code generation strategy that enables fast and flexible code generation in Apollo. It also enlarges significantly the possible optimizations that Apollo may apply. We called our strategy *Code-Bones*.

Apollo is a framework that extends the applicability of the polyhedral model to codes that cannot be handled at compile-time. From an on-line profiling phase, Apollo builds a prediction model that describes the future expected behavior of the loop nest. Some entities may not exhibit a behavior that is fully compatible with the polyhedral model. However, in many cases, it is possible to model such behaviors as *tubes* by enclosing them using linear inequalities.

Then, from the prediction, a polyhedral representation is obtained for the loop nest. Delinearization is used to improve this representation with multi-dimensional array accesses. This representation is used to select a polyhedral transformation by invoking the well-known polyhedral scheduler: Pluto.

Once the transformation has been selected, it is time to generate the optimized code.

The previous code generation strategy used in Apollo, named *Code-Skeletons*, traded performance for flexibility. Therefore, only a small subset of polyhedral optimizations were supported at runtime. In this thesis, we propose a strategy that expands this set to potentially any polyhedral optimization, while still being efficient.

The new available optimizations achieve better performance than their counterpart using the previous code generation mechanism. Additionally, we proposed some specific optimizations that target the code devoted to the verification of the speculation. These optimizations have been made possible thanks to the Code-Bones strategy. Even more, this strategy provides an Inspector-Executor mechanism in Apollo, that enables early detection of misspeculations, reducing the time overhead in case of failure.

We evaluated our proposal with several kernels on a x86 general purpose server and on an ARM64 embedded multi-core chip. We have presented performance and time overhead measurements for each benchmark, for both machines. With these results, we show that the Code-Bones strategy outperforms or equals its main competitor, the Code-Skeletons strategy. In general, the time-overheads remain small. However, when there is a large number of Code-Bones participating in the target loop nest, our framework faces some limitations. If this is the case, the amount of time taken by the Polyhedral Model tools is unsustainable for a runtime usage, and the time required to instantiate the optimized code using the LLVM Just-In-Time compiler increases. Beside these limitations, the outcome of this work is promising. With Apollo, dynamic codes can now fully benefit from the polyhedral model.

Although we consider Apollo to be already successful, we are aware of some limitations. For example, Apollo blindly optimizes a target code, without ensuring that the selected transformation actually results in better performance. Furthermore, by tweaking some parameters, as the tile sizes or the number of threads, better performance could be achieved. In the following Section, we address these issues by providing a road-map to enhance our framework.

7.2 Perspectives

In our opinion, Apollo stands as a major evolution from VMAD, its predecessor, where many aspects were revisited and enhanced. However, there are still some ideas to revisit in order to excel in handling dynamic codes.

Among the short term goals, the priority is to fully support behaviors modeled as tubes. As it was shown in the experiments, the algorithm used to compute the regression hyperplane does not fit a runtime usage, because of its high algorithmic complexity.

```
for (i = 0; i < N; ++i) {
  j = A[i]
  while (j != 0x0) {
    for (k = 0; k < N; ++k)
      S(i, j, k)
    j = j->next
  }
}
```

Figure 7.1: Original code with loop "j" modeled as tube.

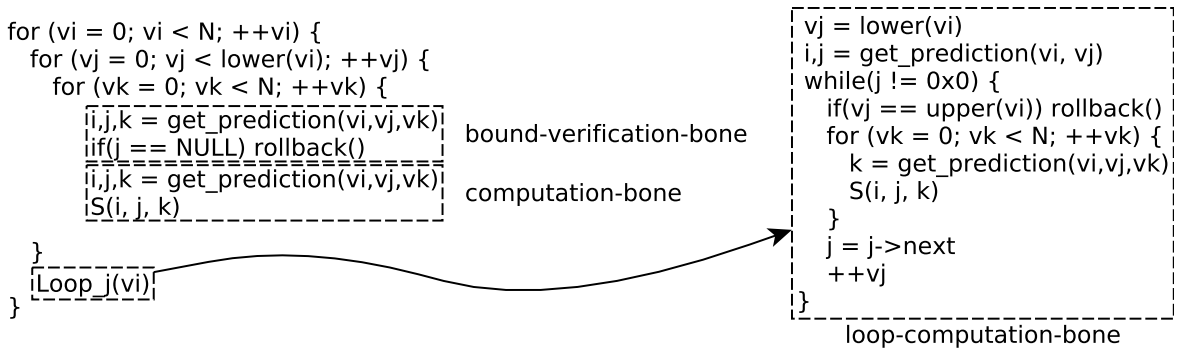

Figure 7.2: Loop's equivalent representation with Code-Bones.

A trivial solution would be to simply consider a subset from all the registered accesses, even if this may lead to a suboptimal regression hyperplane. Another solution would be to switch the algorithm for another one better suited for larger datasets.

Loops whose bounds are modeled as tubes are not yet supported by the Code-Bones approach. To support them, new Code-Bones have to be developed. These bones require to extract entire subloops of the target code, and to encapsulate them into Code-Bones. In Figure 7.1, we show a small kernel whose j-loop bound is modeled as a tube, while in Figure 7.2 we show the version reconstructed using Code-Bones. lower(vi) and upper(vi) stand for the minimum and maximum number of iterations to be executed by the j-loop. get_prediction(vi, ...) stands for the computation of the predictions, using the prediction model and the virtual iterators. To handle such behaviors, we need to extract one additional code bone per loop. These new bones contain entire subloops, with all their contained instructions and required verification code. Notice that the new bone recall the structure of a Code-Skeleton, but its outermost loop iterates according to the original iterator of the code. The execution of this loop completes either when the original exit is reached, or when a rollback is signaled because the upper limit of iterations has been reached.

The time required by the code generation phase of Apollo is a major concern as we would like to handle even more complex codes. A solution would be to store the selected transformation and the assembled Code-Bones as a Code-Skeleton, keeping

some coefficients of the prediction model as parameters, for example, coefficients for the predictions of loop bounds and base addresses of memory accesses. In future executions, it would be enough to verify the validity of the stored transformation to reuse the associated Code-Skeleton. This would cut the time required by Pluto, CLooG, and the LLVM JIT compiler.

For more complex codes, where it is impossible to dynamically obtain a transformation, some computations of the code generation phase could be postponed to an offline phase. After each execution of the target code, Apollo could build a representation (encoded using OpenScop) that combines information about the loop nest and the prediction. Notice that each of these representations would be specific to a particular prediction model. We call them *specific representations*. In between executions, a dedicated offline service could group specific representations per common transformation proposed by Pluto. Then, for each group, a *parametrized representation* could be derived, such that, by assigning values to the parameters, all the specific representations within the group would be instantiated. The parametrized versions and the associated transformations could be encoded in a small database. For future executions, Apollo could inspect if there is a parametrized version that corresponds to the current behavior of the code, and use its associated transformation.

Even if the mentioned approach can help to overcome some of the limitations, we still consider that a gap exists between the current polyhedral model tools and the dynamic usage provided by Apollo. In particular, a precise control of the execution is required, to be able to stop or pause it. In our dynamic context, a tool that purposes sub-optimal solutions, but that guarantees a smaller time overhead, could be advantageous. With faster polyhedral tools, Apollo could generate multiple optimized versions, instead of only one. Then, it could be possible to execute some iterations using each optimized version, choose the one that performs the best, and use it during the rest of the execution.

In the long term, we argue that there are two main research directions that should be followed. The first is to extend Apollo to support new behaviors that are not currently handled by our prediction model. For memory accesses that cannot be modeled as tubes, one idea is to map each accessed address as an entry in a table. Accesses to the same memory address would be associated with the same entry in the table. Each entry would be associated with a unique integer value, that corresponds to its position in the table. Then, we could apply our linear interpolation or regression techniques on these integer values to obtain a prediction.

The second axis of research is to dynamically adjust some parameters in Apollo. Currently, Apollo is blind about the effectiveness of the selected transformation, which

can even perform poorly compared to the original code. Apollo could use timers and hardware performance counters to monitor the execution of the target code, and to adjust runtime parameters during execution. It would be also interesting to monitor other properties, as power consumption, to try to target other objectives apart from execution time-performance. The number of threads used or the CPU frequency could be dynamically adapted to achieve an optimal computation per watt factor.

Bibliography

[1] C. Bastoul. Generating loops for scanning polyhedra. Technical Report 2002/23, PRiSM, University of Versailles, France, 2002.

[2] Cédric Bastoul. Openscop: A specification and a library for data exchange in polyhedral compilation tools. Technical report, University of Paris-Sud, France, September 2011.

[3] Cédric Bastoul. *Contributions to High-Level Program Optimization*. Habilitation thesis, University of Paris-Sud, France, 2012.

[4] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. Putting polyhedral loop transformations to work. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, LNCS, pages 23–30, College Station, Texas, October 2003. Springer-Verlag.

[5] Uday Kumar Reddy Bondhugula. *Effective automatic parallelization and locality optimization using the polyhedral model*. PhD thesis, Ohio State University, 2008.

[6] Derek Bruening, Srikrishna Devabhaktuni, and Saman Amarasinghe. Softspec: Software-based speculative parallelism. In *ACM Workshop on Feedback-Directed and Dynamic Optimization*, Monterey, California, Dec 2000.

[7] Juan Manuel Martinez Caamaño, Willy Wollf, and Philippe Clauss. Code bones: Fast and flexible code generation for dynamic and speculative polyhedral optimization. In *Euro-Par 2016 Parallel Processing - 22th International Conference, Grenoble, France. Proceedings. (Best paper mention)*, 2016.

[8] Henri-Pierre Charles, Damien Couroussé, Victor Lomüller, Fernando A. Endo, and Rémy Gauguey. *deGoal a Tool to Embed Dynamic Code Generators into Applications*, pages 107–112. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[9] Henri-Pierre Charles and Victor Lomüller. Is dynamic compilation possible for embedded systems? In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '15, pages 80–83, New York, NY, USA, 2015. ACM.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.

[11] Philippe Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *Proceedings of the 10th International Conference on Supercomputing*, ICS '96, pages 278–285, New York, NY, USA, 1996. ACM.

[12] Albert Cohen, Sylvain Girbal, and Olivier Temam. *A Polyhedral Approach to Ease the Composition of Program Transformations*, pages 292–303. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[13] Paul Feautrier. Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[14] Tobias Grosser, Armin Größlinger, and Christian Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(4), 2012.

[15] Alexandra Jimborean. *Adapting the polytope model for dynamic and speculative parallelization*. PhD thesis, Université de Strasbourg, September 2012.

[16] Alexandra Jimborean, Philippe Clauss, Jean-François Dollinger, Vincent Loechner, and Juan Manuel Martinez Caamaño. Dynamic and speculative polyhedral parallelization of loop nests using binary code patterns. In *Proceedings of the International Conference on Computational Science, ICCS 2013, Barcelona, Spain, 5-7 June, 2013*, pages 2575–2578, 2013.

[17] Alexandra Jimborean, Philippe Clauss, Jean-François Dollinger, Vincent Loechner, and Juan Manuel Martinez Caamaño. Dynamic and speculative polyhedral parallelization using compiler-generated skeletons. *International Journal of Parallel Programming*, 42(4):529–545, 2014.

[18] Alexandra Jimborean, Luis Mastrangelo, Vincent Loechner, and Philippe Clauss. *VMAD: An Advanced Dynamic Program Analysis and Instrumentation Framework*, pages 220–239. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[19] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. *SIGPLAN Not.*, 29(6):171–185, June 1994.

[20] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. Posh: A tls compiler that exploits program structure. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 158–167, New York, NY, USA, 2006. ACM.

[21] Vadim Maslov. Delinearization: An efficient way to break multiloop dependence equations. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 152–161, New York, NY, USA, 1992. ACM.

[22] 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.

[23] Dorit Nuzman, Revital Eres, Sergei Dyshel, Marcel Zalmanovici, and Jose Castanos. Jit technology with c/c++: Feedback-directed dynamic recompilation for statically compiled languages. *ACM Trans. Archit. Code Optim.*, 10(4):59:1–59:25, December 2013.

[24] The PolyLib polyhedral library. http://icps.u-strasbg.fr/PolyLib/.

[25] Lawrence Rauchwerger, Nancy M. Amato, and David A. Padua. A scalable method for run-time loop parallelization. *Int. J. Parallel Program.*, 23(6):537–576, December 1995.

[26] Lawrence Rauchwerger and David Padua. The lrpd test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 218–232, New York, NY, USA, 1995. ACM.

[27] Scimark benchmark suite. `http://math.nist.gov/scimark2/`.

[28] John A. Stratton, Christopher Rodrigrues, I-Jui Sung, Nady Obeid, Liwen Chang, Geng Liu, and Wen-Mei W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Urbana, March 2012.

[29] Aravind Sukumaran-Rajam. *Beyond the Realm of the Polyhedral Model: Combining Speculative Program Parallelization with Polyhedral Compilation.* PhD thesis, Université de Strasbourg, November 2015.

[30] Aravind Sukumaran-Rajam, Juan Manuel Martinez Caamaño, Willy Wolff, Alexandra Jimborean, and Philippe Clauss. Speculative program parallelization with scalable and decentralized runtime verification. In *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, pages 124–139, 2014.

[31] Aravind Sukumaran-Rajam, Luis Esteban Campostrini, Juan Manuel Martinez Caamaño, and Philippe Clauss. Speculative runtime parallelization of loop nests: Towards greater scope and efficiency. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 245–254, 2015.

[32] Ramakrishna Upadrasta and Albert Cohen. Sub-polyhedral scheduling using (unit-)two-variable-per-inequality polyhedra. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 483–496, New York, NY, USA, 2013. ACM.

[33] Harmen L. A. van der Spek, Erwin M. Bakker, and Harry A. G. Wijshoff. SPARK00: A benchmark package for the compiler evaluation of irregular/sparse codes. *CoRR*, abs/0805.3897, 2008.

[34] Sven Verdoolaege. Isl: An integer set library for the polyhedral model. In *Proceedings of the Third International Congress Conference on Mathematical Software*, ICMS'10, pages 299–302, Berlin, Heidelberg, 2010. Springer-Verlag.

[35] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

Juan Manuel MARTINEZ CAAMAÑO

Fast and Flexible Compilation Techniques for Effective Speculative Polyhedral Parallelization

Summary

In this thesis, we present our contributions to APOLLO: an automatic parallelization compiler that combines polyhedral optimization with Thread-Level-Speculation, to optimize dynamic codes on-the-fly. Thanks to an online profiling phase and a speculation model about the target's code behavior, Apollo is able to select an optimization and to generate code based on it. During optimized code execution, Apollo constantly verifies the validity of the speculation model. The main contribution of this thesis is a code generation mechanism that is able to instantiate any polyhedral transformation, at runtime, without incurring a major time-overhead. This mechanism is currently in use inside Apollo. We called it Code-Bones. It provides significant performance benefits when compared to other approaches.

Résumé

Dans cette thèse, nous présentons nos contributions à APOLLO: un compilateur de parallélisation automatique qui combine l'optimisation polyédrique et la parallélisation spéculative, afin d'optimiser des programmes dynamiques à la volée. Grâce à une phase de profilage en ligne et un modèle spéculatif du comportement mémoire du programme cible, Apollo est capable de sélectionner une optimisation et de générer le code résultant. Pendant l'exécution du programme optimisé, Apollo vérifie constamment la validité du modèle spéculatif. La contribution principale de cette thèse est un mécanisme de génération de code qui permet d'instancier toute transformation polyédrique, au cours de l'exécution du programme cible, sans engendrer de surcoût temporel majeur. Ce procédé est désormais utilisé dans Apollo. Nous l'appelons *Code-Bones*. Il procure des gains de performance significatifs par comparaison aux autres approches.