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 :

**Aravind SUKUMARAN RAJAM**

Soutenue le : **05 Novembre 2015**

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

# Beyond the Realm of the Polyhedral Model: Combining Speculative Program Parallelization with Polyhedral Compilation

| | |
|---|---|
| **Thèse dirigée par :** | |
| **M. Philippe CLAUSS** | Professeur, Université de Strasbourg |
| **Rapporteurs :** | |
| **M. P. Sadayappan** | Professor, Ohio State University, USA |
| **M. Erven ROHOU** | Directeur de Recherche Inria, Équipe Alf, Rennes, France |
| **Examinateurs :** | |
| **M. Henri Pierre CHARLES** | Directeur de Recherche informatique au CEA, Grenoble, France |

This page intentionally left blank

# APOLLO

*Automatic Speculative Polyhedral Loop Optimizer*

This page intentionally left blank

This page intentionally left blank

This work is dedicated to my parents and Little God. Thanks for always being there for me, supporting and guiding me through each and every step of my life.

This page intentionally left blank

# Contents

# Listing of figures

This page intentionally left blank

# Listing of tables

This page intentionally left blank

# 0

# Résumé en Français

## 0.1 Contexte

La demande en puissance de calcul est de plus en plus importante. Afin d'y répondre dans le passé, l'industrie des semi-conducteurs s'appuyait sur une élévation des fréquences d'horloge des micro-processeurs jusqu'à être bloqué par la limite de puissance. En effet, pour faire fonctionner les cœurs de calcul à très haute fréquence, un voltage élevé est nécessaire, qui augmente la consommation dynamique de puissance de la puce. Pour surmonter ce problème, la taille des transistors a été réduite, et nous atteignons désormais les limites de miniaturisation à cause de contraintes physiques. La solution économique qui a été trouvée est d'assembler de plus en plus de cœurs sur la même puce. Ainsi, l'industrie est passée d'un cœur à plusieurs cœurs, et tend désormais vers l'ère des puces à cœurs multiples. Cependant, le matériel n'est pas capable seul d'exploiter tous ces cœurs, qui ont introduit un nouveau challenge: la parallélisation. Pour qu'un même programme utilise de multiples cœurs, celui-ci doit exprimer d'une manière ou d'une autre ses régions parallèle inhérentes. Celles-ci peuvent être soit indiquées explicitement par le programmeur, en utilisant des structures syntaxiques telles que les pragmas, soit détectées automatiquement par un compilateur. L'aspect difficile de la programmation parallèle correspond aux dépendances. Un programme parallèle doit respecter la sémantique du programme séquentiel original ou en d'autres termes, il ne doit pas violer les dépendances. La parallélisation manuelle nécessite de

la part du programmeur d'identifier les dépendances, et de trouver une transformation permettant la parallélisation, qui respecte ces dépendances. Non seulement cela est difficile et est une cause importante d'erreurs, mais est en plus impossible dans certains cas où les dépendances dépendent de facteurs dynamiques telles que les données d'entrée, ou lorsque l'allocation dynamique de mémoire est utilisée, etc.

Les compilateurs peuvent alléger ce problème en détectant automatiquement les dépendances et en produisant une transformation de code valide qui respecte toutes les dépendances. Le développeur peut ainsi se concentrer sur la logique du programme, pendant que le compilateur se charge des optimisations. Les compilateurs de parallélisation automatique peuvent résoudre cela efficacement lorsque le code peut être analysé statiquement. Cependant, en présence de dépendances qui ne peuvent pas être résolues statiquement, le compilateur est obligé de prendre des décisions conservatives afin d'assurer la correction du programme. Ces décisions conservatives limitent très fortement la capacité à paralléliser ou à appliquer d'autres transformations d'optimisation. Une technique permettant de surmonter cette limite, c'est-à-dire de paralléliser ou d'optimiser du code non-analysable statiquement, est d'utiliser la parallélisation spéculative.

## 0.2 Définition du problème

Grâce à des décennies de travaux de recherche, plusieurs compilateurs automatiques peuvent effectuer des optimisations efficaces et agressives sur des codes statiquement analysables. Néanmoins, les compilateurs actuels prennent des décisions conservatives en présence d'entités dynamiques, et cela les empêche d'appliquer des optimisations avancées. Les codes comportant des accès mémoire dynamiques (accès indirects ou par pointeurs), un flot de contrôle et des bornes de boucle dynamiques, sont monnaie courante. Une stratégie prometteuse pour surmonter ces difficultés est d'appliquer des optimisations de code pendant l'exécution, lorsque le programme cible est en train de s'exécuter, afin d'utiliser avantageusement les informations accessibles dynamiquement.

Les optimiseurs dynamiques ont deux avantages majeurs : le premier est la disponibilité de plus d'informations, les valeurs des variables étant connues ; le deuxième est la possibilité de raisonner à partir du comportement courant du code en l'observant. L'objectif de ce travail est d'optimiser des codes dynamiques en utilisant le modèle polyédrique et la parallélisation spéculative.

Le modèle polyédrique est un cadriciel bien connu et largement utilisé d'optimisation de boucles, et à été étudié intensivement dans le contexte statique. Le modèle est capable de proposer des optimisations agressives de boucles, telles que la parallélisation, le pavage de boucles, la torsion de boucles, l'échange de boucles, etc. Cependant, l'unique présence d'une entité non linéaire dans le code empêche le compilateur d'appliquer des optimisations polyédriques. Actuellement, les compilateurs automatiques appliquent uniquement des optimisations polyédriques lorsque le code est entièrement analysable statiquement. L'application du modèle polyédrique à des codes dynamiques nécessite un profilage, un ordonnancement, une génération de code et une vérification sémantique dynamiques. Le profilage du code à l'exécution peut révéler s'il est, entièrement ou sur certaines phases, conforme au modèle polyédrique. Si oui, le code résultant de la transformation doit être généré. Cela peut être accompli (i) en générant statiquement un ensemble de versions de code, l'une d'elle étant sélectionnée à l'exécution, (ii) en générant tout le code optimisé à l'exécution, ou (iii) en générant des squelettes de code et en les complétant à l'exécution. L'instrumentation pour le profilage n'étant appliquée que sur une petite partie de l'exécution, le compilateur doit s'assurer que la suite de l'exécution suit le comportement observé, qui sert de prédiction. Cela peut être accompli en utilisant une vérification à l'exécution. Si celle-ci échoue, le système doit être capable de restaurer l'état du programme à un état consistant, de telle sorte que la contrainte sémantique ne soit pas violée. C'est pourquoi, un système spéculatif est nécessaire. La spéculation de threads (Thread Level Speculation, TLS) est une technique spéculative utilisée pour paralléliser les boucles. Dans les systèmes TLS, des threads sont exécutés spéculativement, et le système surveille continuelle-

ment l'exécution du programme afin de détecter tout risque d'erreur sémantique. Si un tel risque survient, le système stoppe le thread fautif et corrige les effets de son exécution. Pour cela, un état cohérent du système est créé avant le démarrage de la région spéculative. La région où le risque a été détecté est alors ré-exécutée soit dans l'ordre séquentiel original, soit en utilisant une autre stratégie spéculative.

Cependant, une utilisation plus large et plus efficace de la parallélisation spéculative est freinée par le surcoût temporel prohibitif induit par la détection centralisée des conflits mémoire, la modélisation dynamique du code et la génération de code. La plupart des systèmes TLS existants se basent sur un découpage naïf de la boucle cible en tranches, et l'exécution parallèle de ces tranches, avec le support d'un module de vérification centralisé très pénalisant en temps. Car n'utilisant pas de modèle de dépendances, ces systèmes spéculatifs sont incapables d'appliquer des transformations avancées et, de manière encore plus importante, la probabilité d'erreur et de nécessité de restauration à un état cohérent est très élevée. La maintenance d'un état cohérent dans les systèmes TLS implique la conservation de copies multiples des données. Les systèmes TLS classiques effectue de telles copies à chaque itération de boucle, provoquant une croissance exponentielle du besoin en espace mémoire.

## 0.3 Solutions proposées

La solution proposée consiste à utiliser le modèle polyédrique au cours de l'exécution du programme cible, via une parallélisation spéculative. Apollo, pour Automatic POLyhedral Loop Optimizer, est un cadriciel qui va plus loin que les compilateurs actuels, et qui applique le modèle polyédrique dynamiquement par une technique TLS. Il utilise l'instrumentation partielle pour identifier le comportement du code à l'exécution. A partir de l'information de profil, il détermine si le comportement est compatible avec le modèle polyédrique. Si oui, Apollo fait appel au compilateur polyédrique Pluto pour calculer les optimisations de boucle à appliquer, puis il instancie un des squelettes de code générés statiquement. Il surveille continuellement l'exécution pour détecter tout

risque d'erreur de la spéculation de manière décentralisée. Si un problème est détecté, un état cohérent antérieur du programme est restauré et le code séquentiel original est exécuté.

Grâce à une utilisation dynamique le modèle polyédrique, Apollo est capable d'appliquer des transformations de boucle avancées à des codes dynamiques. Le système TLS interne à Apollo l'utilise également pour son système de vérification décentralisé, ainsi que pour son système de sauvegarde mémoire pour réduire le volume de donnée à copier pour construire un état consistant. L'approche combinée modèle polyédrique-TLS renforce ainsi un compilateur qui peut optimiser spéculativement des codes dynamiques pour la performance. Nous exposons dans cette thèse notre contribution au développement d'Apollo. Mais cette approche ouvre une voie d'extension peut-être encore plus impressionnante : l'application du modèle polyédrique à des codes a priori incompatibles, c'est-à-dire présentant des comportements non-linéaires.

## 0.4 Présentation générale de l'architecture d'Apollo

Une vision globale d'Apollo est montrée à la figure 1. Son fonctionnement est résumé ci-dessous.

À la compilation, la phase statique d'Apollo: (1) analyse précisément les instructions mémoire qui peuvent être désambiguïsé à la compilation; (2) génère une version instrumentée pour observer les instructions mémoire qui ne peuvent pas être analysées à la compilation. La version instrumentée s'exécutera sur un échantillon des itérations de la boucle la plus externe et les informations acquises dynamiquement seront utilisées à construire un modèle de prédiction des accès mémoire non-analysables statiquement; (3) génère des squelettes de code parallèle qui sont des versions incomplètes du nid de boucle original et qui requièrent une instanciation dynamique pour générer le code final. Chaque instanciation représente une nouvelle optimisation, ainsi, les squelettes de code peuvent être vus comme des motifs hautement génériques qui supportent un large ensemble de transformations optimisantes et parallélisantes. De plus, les squelettes em-

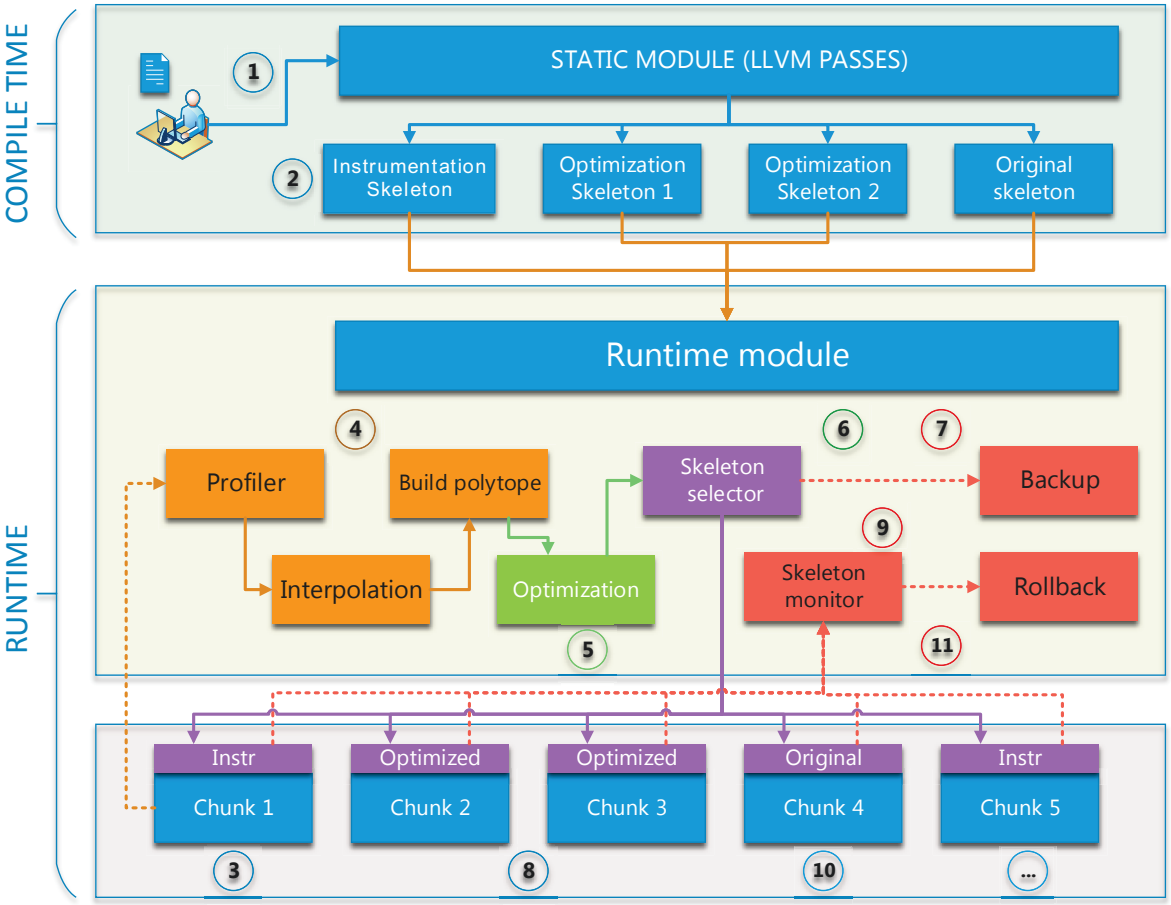



**Figure 1:** Présentation générale d'Apollo

barquent le support nécessaire aux spéculations (c'est-à-dire du code de vérification et de recouvrement d'erreurs). Ces étapes sont décrites dans la partie étiquetée "Compile time" dans la figure 1.

À l'exécution, la phase dynamique d'Apollo: (1) exécute la version instrumentée sur un échantillon d'itérations successives de la boucle la plus externe; (2) construit un modèle de prédiction linéaire pour les bornes de boucle et les accès mémoire; (3) calcule les dépendances entre ces accès; (4) instancie un squelette de code, et génère une version optimisée parallèle du code séquentiel original, qui est sémantiquement correct au regard du modèle de prédiction; (5) pendant l'exécution du code multi-threadé, chaque thread vérifie indépendamment si la prédiction est toujours correcte. Sinon, un retour-arrière (*rollback*) est initié et le système tente de construire un nouveau modèle

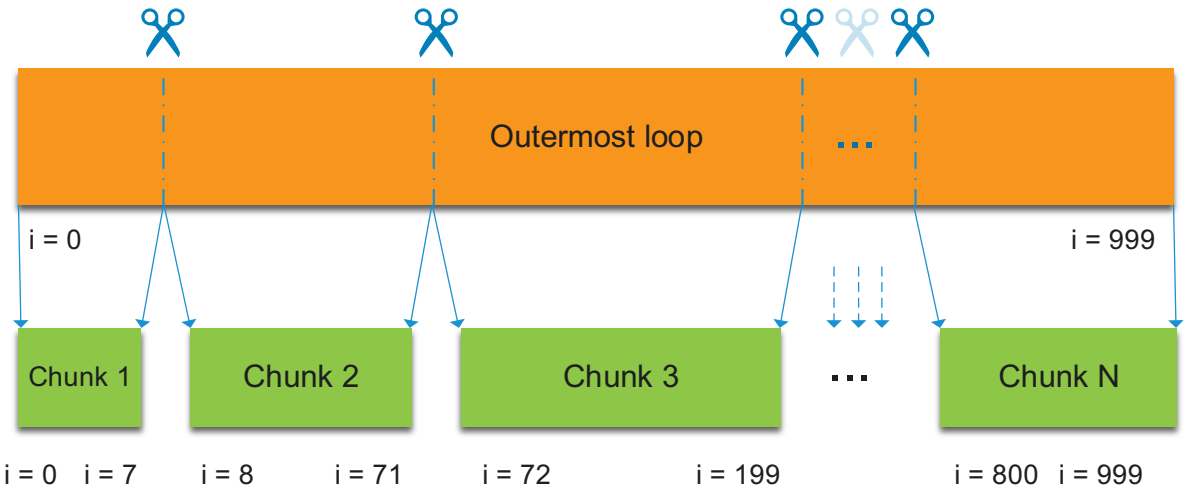



**Figure 2:** Mécanisme de chunking de la boucle externe

de prédiction. Ces étapes sont décrites dans la partie "Runtime" de la figure 1.

### 0.4.1 Exécution par tranches

Apollo divise la boucle la plus externe en petites tranches appelées *chunks*. L'instrumentation de code, les optimisation, etc., sont appliquées sur chaque chunk. Le premier chunk est toujours utilisé pour l'instrumentation du code. Les autres chunks peuvent être assignés à l'exécution des versions optimisées, de la version originale, ou de la version instrumentée du code, selon le comportement du programme. Ce mécanisme est décrit à la figure 2. Sans ce système de chunking, en cas de mauvaise spéculation, la re-exécution devrait démarrer du tout début du nid de boucles. Il permet également de détecter des phases de comportements différents du programme.

## 0.5 Le module statique

L'objectif principal du module statique est de préparer le code utilisateur à la spéculation. Le module statique est constitué d'un ensemble de passes LLVM et opère sur la représentation intermédiaire de LLVM (LLVM IR). Il démarre ses opérations en extrayant chaque nid de boucle qui a été marqué par le pragma 'apollo dcop', en une fonction propre.

### 0.5.1 Analyse statique

Le module statique exécute les passes d'analyse standards telles que l'analyse d'alias (AA) ou l'évolution scalaire (scev), afin de collecter les informations d'alias et les fonctions linéaires d'accès mémoire, de bornes de boucles et de scalaires, qui peuvent être déterminées statiquement. Les informations ainsi collectées sont injectées dans le binaire lui-même et seront utilisées ensuite par le module dynamique.

### 0.5.2 Itérateurs virtuels

Les boucles while et do-while ne possèdent pas la notion d'itérateur dans leur syntaxe. Cependant, Apollo repose sur les itérateurs de boucles pour presque tous ses mécanismes, comme la construction de fonctions linéaires, le calcul des dépendances, le calcul des transformations, ainsi que la vérification. C'est pour cela que pour chaque boucle d'un nid de boucle ciblé, une variable supplémentaire, appelée 'itérateur virtuel' est insérée. Ces itérateurs virtuels se comportent comme des itérateurs de boucle, et essentiellement convertissent tout type de boucle en boucle for. Ils démarrent toujours à la valeur '0' et sont incrémentés de '1' à chaque itération: ils sont normalisés. Le raisonnement sur les fonctions d'accès et donc les dépendances est basé sur ces itérateurs virtuels.

### 0.5.3 Le versioning de code et les squelettes

La fonction la plus importante du module statique est la création de différentes versions du code d'entrée, appelées squelettes. Les squelettes peuvent être vus comme des codes paramétrés où différentes instanciations de leurs paramètres résultent en différentes transformations du code. La structure générale des squelettes d'optimisation est représentée à la figure 3.

Il y a trois types de squelettes de code: (i) les squelettes d'optimisation, (ii) les squelettes d'instrumentation, et (iii) les squelettes originaux. Les squelettes

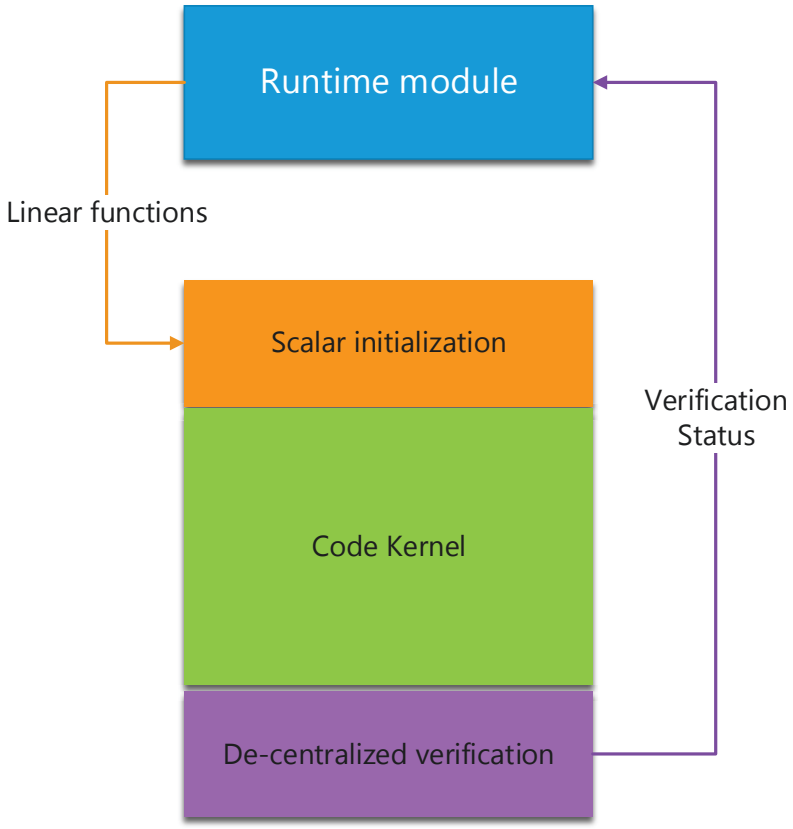



**Figure 3:** Structure générale d'un squelette de code

d'optimisation sont des motifs de code, chacun représentant un ensemble de transformations de code. Ces squelettes sont des programmes incomplets qui doivent être instanciés à l'exécution à l'aide de paramètres associés à la transformation appliquée. Un squelette d'instrumentation est constitué de parties de code spécifiquement dédiées au profilage de code et à la communication avec le module dynamique. Les squelettes originaux sont essentiellement constitués du code original séquentiel, sans transformation.

## 0.6 Le module dynamique

Le module dynamique d'Apollo contrôle l'orchestration de l'exécution du code cible. Il est écrit en *C++* et est lié au binaire en tant qu'objet partagé (*Shared Object*, SO). Ses fonctions principales incluent la sélection des squelettes de code, la construction des fonctions linéaires et du modèle de dépendances, l'invocation de l'ordonnanceur pour

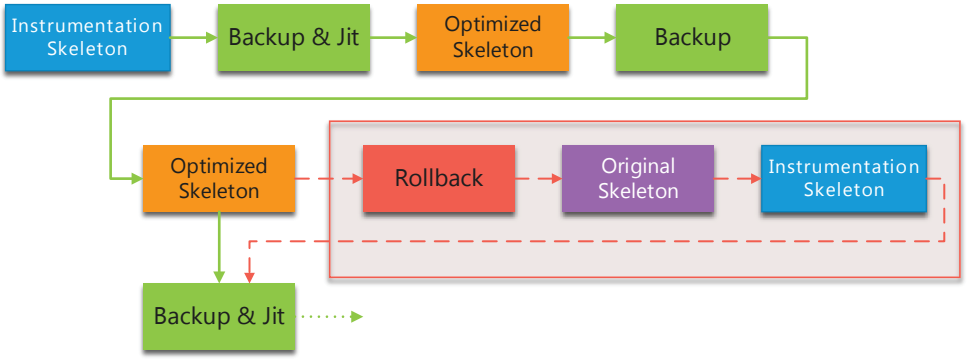



**Figure 4:** Flot d'exécution du module dynamique

calculer la transformation, la mise en oeuvre de la parallélisation, des sauvegardes et du retour-arrière, ainsi que le management global du système. Le flot d'exécution complet du module dynamique est schématisé à la figure 4.

### 0.6.1 Profilage de code

Le cycle de vie d'Apollo commence lorsque le contrôle atteint un nid de boucles ciblé. Le module dynamique sélectionne et instancie d'abord le squelette d'instrumentation, puis lui passe le contrôle. Pendant l'instrumentation, c'est-à-dire pendant que le squelette d'instrumentation s'exécute, les accès mémoire, les valeurs de scalaires, les bornes de boucles, ainsi que leurs itérateurs englobants sont retournés au module dynamique à travers des appels de fonction. Le module dynamique mémorise ces données dans des structures spécialisées pour un traitement ultérieur.

### 0.6.2 Calcul du polyèdre de dépendances

Lorsque le squelette d'instrumentation a terminé son exécution, le module dynamique tente de construire des fonctions linéaires pour (i) les accès mémoire, (2) les scalaires, et (ii) les bornes de boucles. Les dépendances sont ensuite déduites des données issues de l'instrumentation. Lorsque les dépendances ont été identifiées, le polyèdre de dépendances est construit en utilisant les conditions de dépendance et les domaines des instructions.

### 0.6.3 Sélection de l'ordonnancement et du squelette

Apollo utilise Pluto pour calculer un ordonnancement valide et optimal. Pour cela, Pluto est utilisé comme une bibliothèque et prend en entrée le polyèdre de dépendance précédemment calculé. Lorsque l'ordonnancement a été déterminé, Apollo détecte les boucles parallèles, en plus d'autres propriétés de la transformation. De là, le squelette de code approprié est sélectionné.

### 0.6.4 JIT

Apollo utilise la compilation juste-à-temps (JIT) [1] pour optimiser encore plus le code. En plus du bénéfice habituel du JIT, il est particulièrement important dans Apollo à cause de l'usage dynamique du modèle polyédrique, et des nombreuses variables constantes des fonctions linéaires qui peuvent être optimlisées. Le code résultant du JIT est mis en cache et est réutilisé si la même transformation est re-appliquée.

### 0.6.5 Sauvegarde et point d'exécution sûr

Pour prendre en compte le risque d'échec de la spéculation, le système doit préserver un état intermédiaire d'exécution valide, appelé *état sûr*. Grâce aux fonctions linéaires, Apollo peut déterminer les zones mémoire d'écriture, même avant que le chunk soit exécuté. Dès que la zone a été calculée, une copie mémoire propre de type 'memcpy' est exécutée afin de sauvegarder la zone d'écriture, avant de lancer le chunk.

### 0.6.6 Parallélisation

Apollo utilise son propre méchanisme intrinsèque, appelé *dispatcher manager*, pour la parallélisation, reposant sur OpenMP [2]. À l'intérieur de chaque squelette d'optimisation, chaque boucle est représentée par une fonction, qui est paramétrée par les bornes de boucle inférieure et supérieure. Si la boucle est marquée comme étant parallèle, le domaine d'itération est alors divisé en petites tranches, et la fonction

est appelée en parallèle pour chacune de ces tranches. Si la boucle est séquentielle, tout le domaine d'itération est exécuté en une seule grande tranche séquentielle.

### 0.6.7 Vérification dynamique

Alors que le squelette parallèle instancié s'exécute dans un chunk, le module dynamique doit continuellement s'assurer de la correction du code transformé qui s'exécute [3], en vérifiant (1) l'adhérence du modèle de prédiction aux locations mémoire qui sont réellement accédées et (2) que les occurrences d'accès mémoire non-prédits n'invalident pas la transformation de code qui a été appliquée, au regard de la sémantique du programme. Apollo vérifie les fonctions linéaires en les comparant aux adresses, ou valeurs, observées.

### 0.6.8 Retour arrière (rollback)

Si un thread détecte une mauvaise spéculation, un indicateur est activé. Lorsque tous les threads terminent, l'indicateur est consulté pour vérifier si la spéculation a réussi. Si ce n'est pas le cas, la sauvegarde est restaurée au point sûr. Puis, le squelette original est sélectionné pour s'exécuter dans le chunk mal prédit, qui est suivi par un squelette d'instrumentation. Si l'indicateur n'est pas activé, alors la spéculation a réussi et la sauvegarde est supprimée.

### 0.6.9 Terminaison

Apollo exécute les squelettes de code jusqu'à atteindre la borne de la boucle la plus externe. Si celle-ci est découverte à la compilation, alors avant de lancer chaque chunk, elle est comparée à la borne supérieure du chunk pour vérifier si le prochain chunk est le dernier. Si ce n'est pas le cas, ou si la borne n'est pas connue, alors le prochain chunk est lancé spéculativement avec la même transformation, en assumant que le programme n'a pas modifié son comportement. Mais il est possible que la condition de sortie soit atteinte au cours du chunk. Si la borne du chunk n'est pas égale à la borne réelle, alors

un retour-arrière est lancé et le chunk est re-exécuté avec le squelette original. Lorsque la boucle externe termine son exécution, le module dynamique met à jour les variables vivantes en sortie et rend le contrôle au programme utilisateur.

## 0.7 Resultats

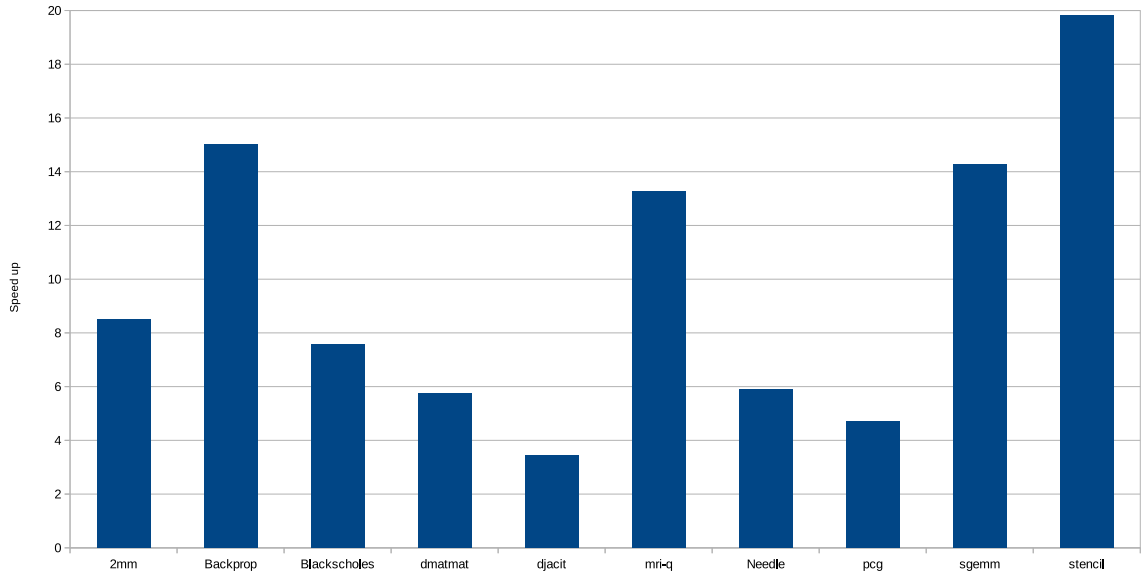



**Figure 5:** Speedup of APOLLO, using 24 threads.

Les expérimentations ont été effectuées sur une plate-forme comprenant deux processeurs AMD Opteron 6172 de 12 coeurs chacun, à 2.1 Ghz, exécutant Linux 3.11.0-17-generic x86_64. Les mesures rapportées ont été obtenues en exécutant chaque programme test cinq fois, et en prenant la moyenne des mesures. Les accélérations sont par rapport aux meilleures version séquentielles obtenues avec les compilateurs clang et gcc et l'option '-O3'.

L'ensemble des programmes tests a été construit à partir de bancs d'essai, de telle sorte que les programmes sélectionnés mette en lumière les capacités d'Apollo. Les programmes mri-q, sgemm et stencil proviennent de la suite Parboil [4], blackscholes de la suite Parsec[5], backprop et needle de la suite Rodinia [6], dmatmat, djacit et pcg de la suite SPARK00 [7], et enfin 2mm de la suite Polybench [8]. Le programme

2mm a été re-écrit afin que ces accès mémoire se fassent via des pointeurs, et qu'il ne soit pas analysable statiquement.

La figure 5 monttre les accélérations obtenues grâce à Apollo.

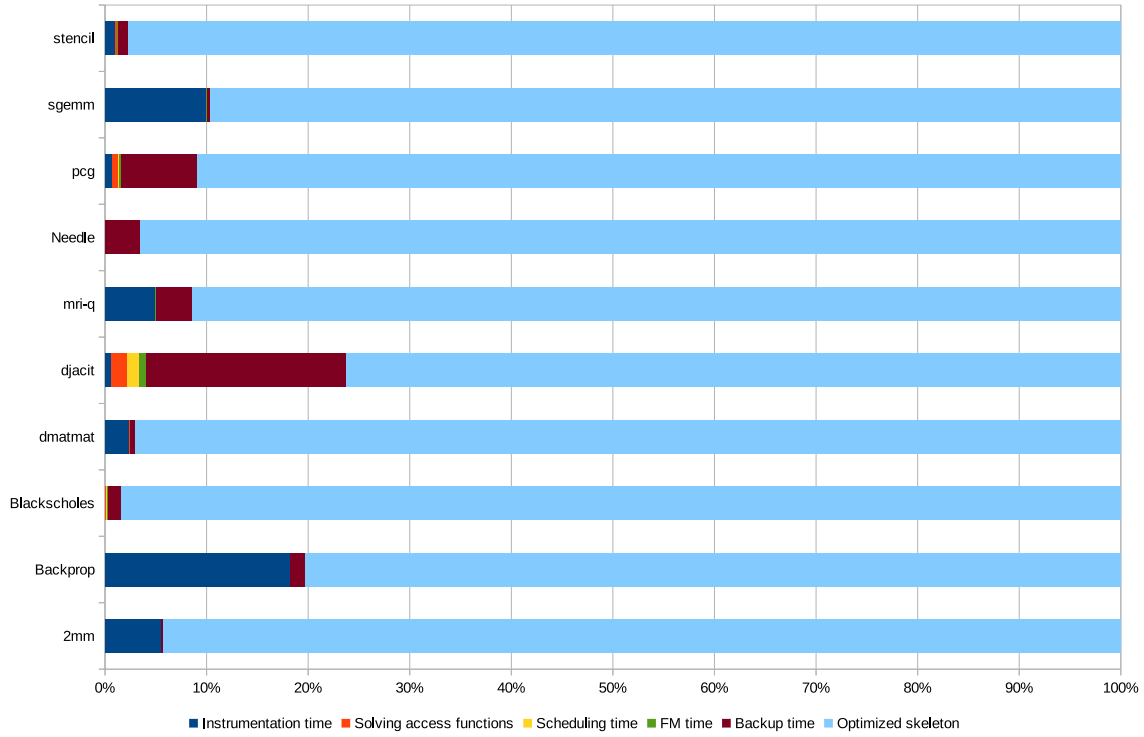



**Figure 6:** Comparaison des sur-coûts temporels

La figure 6 montre les différents sur-coûts temporels associés à chaque module d'Apollo. Seuls les sur-coûts principaux sont considérés. Le temps d'instrumentation (*instrumentation time*) comprend le temps de profilage du code. Il inclut le temps d'exécution d'un petit chunk, et le temps de communication des accès mémoire et des bornes de boucles au module dynamique. La légende *solving access functions* comprend le temps passé à calculer les fonctions linéaires d'interpolation des accès mémoire et des bornes de boucles. La légende *scheduling time* comprend le temps pris par l'ordonnanceur, c'est-à-dire le temps d'instanciation de l'ordonnanceur, et le temps pris par l'ordonnanceur de Pluto pour déterminer un ordonnancement optimal. La légende *FM time* concerne le temps nécessaire à l'élimination de Fourier-Motzkin, qui est utilisée pour calculer les bornes de boucles dans l'espace d'itérations transformé.

Si la transformation reste la même entre chunks contigus, le solveur FM n'est invoqué qu'une seule fois, mais les fonctions résultats réutilisées pour chaque chunk. Cependant, si la transformation change, alors le solveur est invoqué à nouveau. La légende *Backup time* se réfère au temps passé par le système de sauvegarde à calculer la zone mémoire à sauvegarder et à effectuer effectivement la sauvegarde. La légende *optimized skeleton* porte sur le temps pris par l'exécution parallèle effective, la vérification décentralisée et la sauvegarde à la volée.

## 0.8 Apollo non-linéaire

Apollo peut traiter efficacement des classes de codes où les accès mémoire et les bornes de boucles sont linéaires, au moins par phases. Si le comportement du code est compatible avec le modèle polyédrique, la région concernée est assujettie à des optimisations polyédriques de manière spéculative. Par contre, lorsque le comportement du code dévie d'un comportement affine, alors cette région est exécutée avec le code original. Pourtant, beaucoup de codes exhibent des comportement non-linéaires, et particulièrement les codes comportant des accès mémoire indirects, des pointeurs, *etc.*

Les références mémoire indirectes et les pointeurs sont très communs dans les codes, c'est pourquoi la capacité à les traiter est un challenge majeur en parallélisation automatique. La plupart des codes exhibent un comportement mémoire non-linéaire ou des bornes de boucles non-linéaires. La présence d'une seule entité non-linéaire dans un programme empêche l'application du modèle polyédrique: aucune transformation polyédrique ne peut être appliquée, ni la parallélisation d'aucune boucle. L'impact de cette limitation empêche l'optimisation de beaucoup de codes, même lorsque la plupart de leurs accès mémoire sont linéaires.

Notre objectif est de rendre Apollo capable d'optimiser les accès non-linéaires et de le valider, afin de traiter plus de codes efficacement [9]. Pour traiter des entités non-linéaires, une approche générale est de relâcher le modèle de dépendances. S'il est relâché, alors il ne préserve plus la garantie originale concernant la validité de la

transformation d'optimisation. Cela nous amène à un *validateur dynamique*, en plus du système dynamique de vérification linéaire.

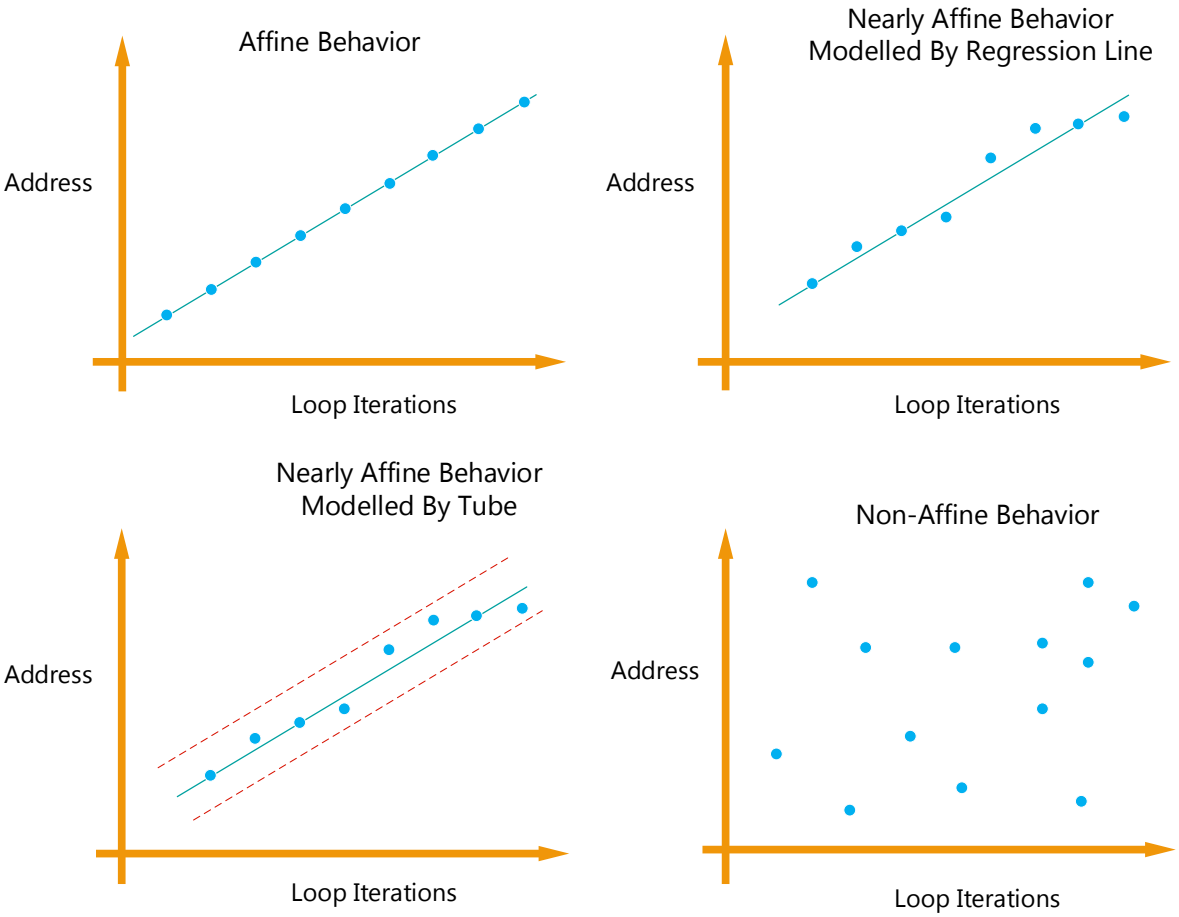



**Figure 7:** Représentation graphique des comportements *affines*, *quasi-affines* et *non-affines*.

### 0.8.1 Construction du polyèdre de dépendances

Le processus d'instrumentation est similaire à celui décrit précédemment. Une fonction de régression est construite pour chaque accès mémoire, chaque scalaire et borne de boucle. L'adhérence entre les valeurs observées d'adresses mémoire accédées (ou compteurs de boucles) est utilisée pour classifier les accès comme étant soit *affines*, *quasi-affines* ou *non-affines*. Un accès mémoire (ou borne de boucle) est classifié comme:

- *affine*, si toutes les observations adhèrent parfaitement à une équation linéaire, et si tous les coefficients de la fonction sont des entiers;

- *quasi-affine*, si la plupart des observations restent “proches” d’une équation linéaire;
- *non-affine*, si les observations ne correspondent à aucune des deux autres catégories.

Une représentation graphique en dimension 2 de ces trois cas est donnée à la figure 7.

## 0.9 La modélisation des accès mémoire non-linéaires

Les accès mémoire non-linéaires correspondent aux accès mémoire (lecture ou écriture) dont les instances ne correspondent à aucune fonction affine. C’est pourquoi une fonction de régression est calculée pour chaque instruction mémoire de ce type, ainsi que le coefficient de corrélation associé. Selon ses valeurs, les accès mémoire non-linéaires sont traités comme suit.

**Le coefficient de corrélation se situe entre 0.9 et 1.**

Un coefficient de corrélation entre 0.9 et 1 indique une bonne adhérence des valeurs observées avec l’équation de régression, et suggère donc qu’elle approxime bien les accès mémoire effectifs. It est donc raisonnable d’inclure cette équation de régression dans le modèle de dépendance, l’idée étant que l’optimiseur polyédrique (l’ordonnanceur) procure une transformation qui optimise les accès non-affines conjointement avec les accès affines, optimisant ainsi la localité des données et le parallélisme pour les accès affines et non-affines. Chaque hyperplan de régression est calculé via une régression multivariée utilisant la méthode des moindres carrés [10]. L’hyperplan de régression devrait donc être encodé dans le polyèdre de dépendances. S’il est directement encodé, tel une fonction linéaire, l’ordonnanceur polyédrique considérera uniquement les points situés exactement sur l’hyperplan de régression lorsqu’il calculera la solution. Par conséquent, l’utilisation seule de l’hyperplan de régression n’est pas suffisant, car il ignore les points situés à côté de l’hyperplan. Ainsi, pour chaque accès quasi-affine, deux hyperplans qui sont parallèles à l’hyperplan de régression sont construits, en ajoutant un déplacement positif et négatif à l’hyperplan original. De manière informelle, un

hyperplan est “au dessus” et l’autre “en dessous” de l’hyperplan original, formant tous les deux un *tube* autour de l’hyperplan de régression (voir la partie inférieure gauche de la figure 7). La distance entre les hyperplans tubulaires (le déplacement) est choisi de telle sorte que tous les points raffinés se trouvent à l’intérieur du tube. Ce tube forme une enveloppe convexe des points observés et est encodé dans le modèle de dépendance.

**Le coefficient de corrélation est inférieur à 0.9.**

Si le coefficient de corrélation est inférieur à 0.9, alors les accès mémoire sont caractérisés comme étant non-affines et sont exclus de la représentation des dépendances par régression. Pour chacune des instructions mémoire correspondantes, les accès sont alors approximés en utilisant l’intervalle des valeurs minimum et maximum des adresses accédées durant l’instrumentation. Les chevauchements entre de tels intervalles, les tubes de régression et les fonctions linéaires, sont testés afin de déterminer si une dépendance est possible, auquel cas aucune transformation de code ne sera effectuée. L’inclusion d’accès quasi-affines et non-affines relâche le modèle polyédrique, mais ajoute le challenge d’une vérification supplémentaire pour les accès mémoire qui ont lieu en dehors des régions prédites.

### 0.9.1 La modélisation des bornes de boucles non-linéaires

En présence de bornes de boucles non-linéaires, l’intervalle des adresses mémoire qui seront accédées ne peut pas être déterminé, même avec une instrumentation dynamique. Ainsi, nous ne pouvons pas construire le polyèdre de dépendances. Cela est même le cas lorsque tous les accès mémoire sont linéaires.

La modélisation d’une boucle dans le modèle polyédrique nécessite:

1. Une fonction linéaire pour la borne inférieure, paramétrée par ses itérateurs de boucles englobantes, ses paramètres de boucle et des constantes;

2. Une fonction linéaire pour l’itérateur, paramétrée par ses itérateurs de boucles englobantes, ses paramètres de boucle et des constantes;

3. Une fonction linéaire pour sa borne supérieure, paramétrée par ses itérateurs de boucles englobantes, ses paramètres de boucle et des constantes.

En calculant un hyperplan minimal pour la borne de boucle supérieure, on s'attend à ce que toute la boucle s'exécute au moins jusqu'au point où l'itérateur rencontre cet hyperplan. Si l'hyperplan minimal conserve sa validité pendant l'exécution, la boucle non-linéaire peut être parallélisée jusqu'à cet hyperplan minimal. Notons que la parallélisation doit malgré tout être valide, c'est-à-dire que les dépendances ne doivent pas empêcher la parallélisation. Le reste des itérations de la boucle non-linéaire doit être exécuté séquentiellement, jusqu'à atteindre la condition de sortie de boucle originale. L'avantage de cette approche est d'être capable de paralléliser et d'optimiser la partie prédite, avec un très faible surcoût temporel. Cependant, cette approche ne garantit pas la correction du programme car la partie séquentielle, qui n'a pas été prédite, peut introduire une dépendance non prévue, et ainsi violer le modèle de prédiction des dépendances. Les accès mémoire qui ont lieu dans l'espace non prédit sont par conséquent considérés de manière équivalente aux accès mémoire non-linéaires. Comme il a été expliqué dans la sous-section précédente, le système de traitement des accès non-linéaires d'Apollo peut détecter toute violation sur des régions non-prédites, et garantit donc la validité du programme. Si pendant l'exécution effective, la boucle exécute des itérations non prédites par l'hyperplan minimal, alors que la boucle est marquée pour une exécution parallèle, le système effectue un retour-arrière.

Une approche similaire au cas de la borne supérieure peut être appliquée pour la borne inférieure. Les deux techniques peuvent être combinées pour produire trois vagues d'exécutions: une partie séquentielle de tête, une partie "cœur" au milieu d'exécution parallèle, et une partie séquentielle de queue. La gestion de toute transformation telle que l'échange de boucles, le pavage de boucles, etc. suit les mêmes principes.

**Le coefficient de corrélation se situe entre 0.9 et 1.**

Une valeur haute du coefficient de corrélation indique que la borne de boucle oscille autour d'un hyperplan. À la différence des hyperplans de régression des accès mémoire, l'hyperplan des bornes de boucle ne peut pas être utilisé directement. L'hyperplan de régression, par construction, minimise la moyenne des carrés des distances entre les points observés. Cela implique qu'il peut y avoir des points au-dessus et/ou en-dessous de l'hyperplan. Afin de gérer cela, pour l'hyperplan bornant inférieurement, l'hyperplan est glissé de la largeur du tube, l'idée étant que l'hyperplan glissé se comporte comme un minimum attendu. Notons que cela ne garantit toujours pas que tous les points sont dans le domaine.

**Le coefficient de corrélation est inférieur à 0.9.**

Comme pour les hyperplans de régression des accès mémoire, un coefficient de corrélation inférieur à 0.9 indique que la régression présente une faible adhérence aux bornes de boucle observées. Cependant, des transformations optimisantes peuvent malgré tout être appliquées, si une valeur seule représentant la borne minimale peut être déterminée. Ainsi, une valeur seule (plutôt qu'un hyperplan), indiquant la borne minimale prédite est utilisées à partir des observations. Dès que le polyèdre de dépendances a été construit, un ordonnancement et un squelette sont déterminés de manière similaire au cas purement linéaire d'Apollo.

### 0.9.2 Sauvegarde et point d'exécution sûr

Pour les accès non-affines en écriture, il n'y a pas d'information dans le modèle permettant de prédire les zones mémoire qui seront écrites pendant l'exécution du prochain chunk. C'est pourquoi une sauvegarde à la volée est effectuée pendant l'exécution parallèle spéculative du prochain chunk, où chaque location mémoire est sauvegardée juste avant d'être mise à jour. Pour les accès quasi-affines en écriture, toutes les locations mémoire à l'intérieur du tube de régression sont sauvegardées avant de lancer le prochain chunk. Cependant, il peut toujours y avoir des accès en dehors du tube, mais

qui ne corrompent pas la sémantique de la boucle transformée. Mais puisque de tels accès ne peuvent seulement être connus pendant l'exécution du chunk, ils sont traités de la même manière que les accès non-affines: ils sont sauvegardés à la volée pendans l'exécution du chunk. Pour les boucles avec des bornes non-linéaires, la sauvegarde est calculée en se basant sur les hyperplans minimum et maximum. Chaque location pouvant être mise à jour entre les hyperplans est sauvegardée avant le lancement du chunk et ceux qui sont en dehors sont sauvegardés à la volée par chaque thread, de manière similaire aux accès non-affines en écriture.

### 0.9.3 Vérification dynamique

La conformité des accès mémoire prédits par rapport aux fonctions linéaires de prédiction, et l'adhérence des accès non-prédits au modèle polyédrique spéculatif, peuvent seulement être vérifiées pendant l'exécution effective du code optimisé. Chaque thread vérifie la validité de chacun de ses accès mémoire par rapport à tous les accès mémoire prédits, et effectue ainsi une vérification immédiate et décentralisée. Selon la précision du modèle, les opérations de vérification suivantes sont effectuées:

- Si l'instruction mémoire a été modélisée exactement par une fonction affine: le thread vérifie l'égalité entre l'adresse accédée et celle prédite en instanciant la fonction affine de prédiction. En cas de mauvaise prédiction, un risque potentiel de dépendance non-prédite est détecté;

- Si l'instruction mémoire a été modélisée par un tube de régression (ou par un intervalle d'adresses): le thread vérifie que l'adresse accédée se situe à l'intérieur du tube (ou à l'intérieur de l'intervalle). Sinon, l'adresse est comparée aux adresses prédites comme étant touchées par les autres instructions mémoire pendant l'exécution du chunk courant. Si aucun risque d'interaction a été détecté, l'adresse est conservée dans une table locale, afin d'être examinée plus tard, après la fin du chunk courant, en utilisant la vérification centralisée. Sinon, un risque

potentiel de dépendance non-prédite est détecté.

S'il y a un risque potentiel de dépendance non-prédite, un retour-arrière est lancé par le thread fautif. Sinon, à la fin du chunk, une vérification entre threads est effectuée afin d'assurer la consistance entre threads, en comparant les adresses qui ont été conservées dans leurs tables locales respectives.

## 0.10 Résultats

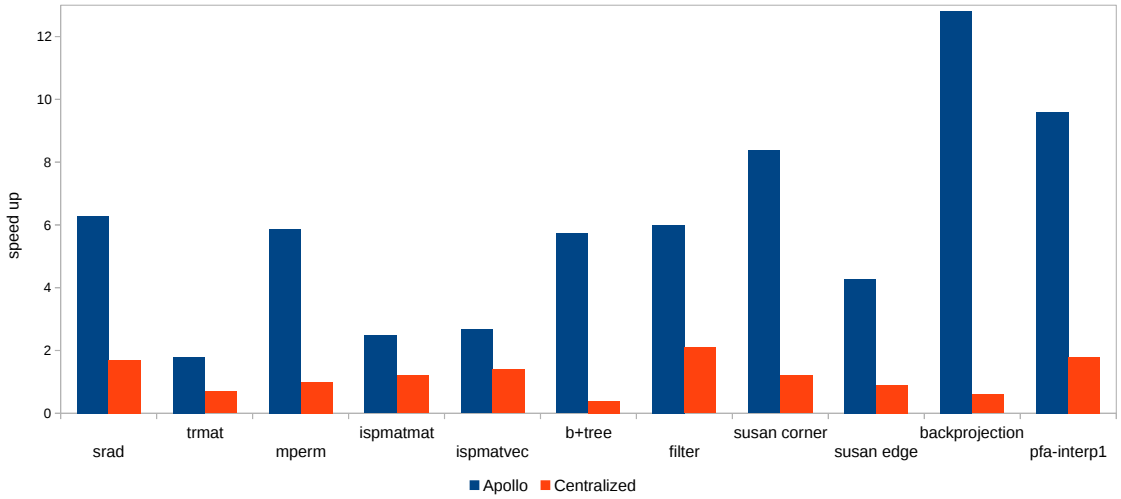



**Figure 8:** Accélération d'Apollo non-linéaire et du système centralisé, avec 24 threads

Les programmes de test ont été sélectionné parmi plusieurs bancs d'essai tels que SPARK00 [7], Rodinia [11], Cbench [12] et Perfect [13]. Il faut noter que ces programmes ne peuvent pas être optimisés par des outils tels que Pluto, ou la version précédente d'Apollo, à cause de la non-linéarité des accès mémoire et des bornes de boucles. En plus de ces programmes, nous avons inclus `Filter`*, qui est une application réelle de traitement d'images qui exhibe des références non-linéaires à des tableaux. Chaque programme a été écrit en *C/C++*.

À la figure 8, nous montrons les accélérations d'Apollo avec la modélisation non-linéaire et le système de vérification qui ont été présentés, et qui est comparé à une

*http://lodev.org/cgtutor/filtering.html

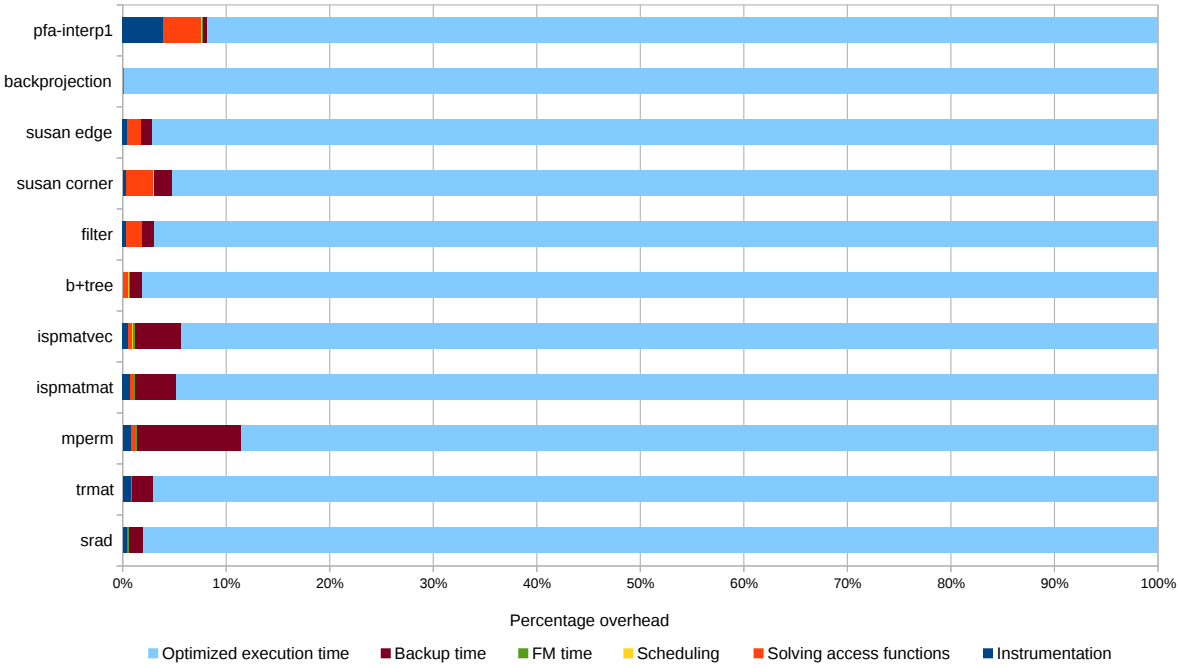



**Figure 9:** Classification des surcoûts temporels en pourcentages

version complètement centralisée de vérification. Pour tous les programmes de test, Apollo surpasse la version centralisée, tout en procurant des accélérations significatives par rapport aux codes séquentiels originaux.

La figure 9 montre une classification des surcoûts temporels du système.

## 0.11 Conclusion

Dans cette thèse, nous présentons nos contributions à Apollo (Automatic speculative POLyhedral Loop Optimizer), qui est un compilateur automatique combinant la parallélisation spéculative et le modèle polyédrique, afin d'optimiser les codes à la volée. En effectuant une instrumentation partielle au cours de l'exécution, et en la soumettant à une interpolation, Apollo est capable de construire un modèle polyédrique spéculatif dynamiquement. Ce modèle spéculatif est ensuite transmis à Pluto, qui est un ordonnanceur polyédrique statique. Apollo sélectionne ensuite un des squelettes d'optimisation de code générés statiquement, et l'instancie. La partie dynamique d'Apollo surveille continuellement l'exécution du code afin de détecter de manière dé-

centralisée toute violation de dépendance. Une autre contribution importante de cette thèse est notre extension du modèle polyédrique aux codes exhibant un comportement non-linéaire. Grâce au contexte dynamique et spéculatif d'Apollo, les comportements non-linéaires sont soit modélisés par des hyperplans de régression linéaire formant des tubes, soit par des intervalles de valeurs atteintes. Notre approche permet l'application de transformations polyédriques à des codes non-linéaires grâce à un système de vérification de la spéculation hybride, combinant vérifications centralisées et décentralisées.

*The beginning is the most important part of the work.*

Plato, The Republic

*The secret to getting ahead is getting started.*

Mark Twain

*We all have to start somewhere...*

Asif

# 1 Introduction

## 1.1 Context

The demand for computing power is increasing at an unprecedented rate. In order to meet this vast requirement, the computing industry relied upon increasing the clock speeds till it was hindered by the power wall. To operate the computing cores at high frequencies, a high voltage is required, which increases the dynamic power consumption of the chip. In order to overcome this, the size of transistor was reduced and we are nearing the limits of shrinking due to physical constraints. The economic solution to this problem is to pack more and more cores to the chip. Thus, the industry moved from single core to multi core and is moving towards the many core era. However, mere hardware is not capable of taking advantage of the latter as it introduced a new challenge, program *parallelization*.

To utilize multiple cores by the same program, the program's code should somehow express its inherent parallel regions. Parallel regions could be either indicated explicitly

by a programmer using syntax structures like *pragma* or they could be automatically detected by a compiler. The challenging aspect of parallel programming is *dependencies* between instructions. A parallel program should respect the semantics of the original sequential program or in other words, it should not violate any program dependence. Manual parallelization requires the programmer to identify the dependencies and find a transformation which allows parallelization, but still respecting all the dependencies. Not only is this process difficult and error prone, but also it is even impossible in some cases where the dependencies are related to dynamic factors such as input data, or when dynamic memory allocation is used etc.

Compilers can alleviate this problem by automatically detecting the dependencies and producing a valid code transformation respecting them. The developer can thus concentrate on the program logic while the compiler takes care of code optimization. Current automatic parallelization compilers can solve this efficiently when the code can be statically analyzed. However, in the presence of non statically resolvable dependencies, the compiler is forced to take conservative decisions in order to ensure program correctness. These conservative decisions limit to a huge extent the ability to parallelize or to perform other optimizing code transformations. One technique to overcome this limitation, i.e. to parallelize or to optimize non statically analyzable code, is to use Thread Level Speculation (TLS).

## 1.2 Problem definition

Thanks to decades of research, many automated compilers can perform effective and aggressive optimizations on statically analyzable codes. However, the current compilers take conservative assumptions in presence of dynamic entities, and this prevents them to perform aggressive optimizations on dynamic codes. Codes with dynamic memory accesses (indirect accesses, pointers), dynamic control flow and dynamic loop bounds are quite common. A promising strategy for overcoming such limitations is to perform code optimization at runtime – while the target code is running – in order to take

advantage of the dynamically available information. Runtime optimizers have two major advantages: one is that more information is available at runtime as the values of variables are resolved; the other is the opportunity to reason about the code behavior by observing it. The objective of this work is to optimize dynamic codes using the polyhedral model and thread level speculation (TLS).

The polyhedral model is a well known and widely used loop optimization framework, and has been studied extensively in the static context. The model is able to propose aggressive loop optimizations such as parallelization, loop tiling, loop skewing, loop interchange *etc.*. However, the presence of a single non affine entity in the code prevents the compilers from applying polyhedral optimizations. Currently, automated compilers only apply polyhedral optimizations if the code is entirely statically analyzable. Applying the polyhedral model to dynamic codes requires dynamic monitoring, dynamic code scheduling and code generation, and runtime verification. Dynamically profiling the code can reveal whether the code, or at least some phases of the code are amendable for polyhedral optimizations. If compatible, the code representing the transformation has to be generated. This can be done by (i) statically generating a set of code versions and choosing one at runtime, (ii) or by generating the entire code at runtime, (iii) or by generating some code skeletons statically and completing them at runtime. Since only a sample of the code execution is instrumented, the compiler should ensure that the rest of the code phase follows the predicted behaviour. This can be done by using runtime verification. If the verification fails, the system should be able to restore the program state to a consistent one so that the semantic constraint is not violated. Thus, a speculative system is required.

Thread Level Speculation (TLS) is a speculative technique to parallelize loops. In TLS, some threads are executed speculatively, and the runtime system continually monitors the program execution to detect any data race. If a data race occurs, the system squashes the faulty speculative thread, and recovers the effects of its execution (rollback). In order to recover, a consistent state of the system is created before the

speculative execution begins. The violated region is either executed in the original sequential order, or using some other speculative strategy. However, a wider and more efficient use of TLS is mainly hampered by the prohibitive time overhead induced by centralized data race detection among the parallel threads, dynamic code behavior modeling and code generation. Most of the existing TLS systems rely on naively slicing the target loops into chunks, and trying to execute the chunks in parallel with the help of a centralized performance-penalizing verification module that takes care of data races. Due to the lack of a data dependence model, these speculative systems are not capable of doing advanced transformations and, more importantly, the chances of rollback are high. Maintaining consistent state in TLS system, involves keeping multiple copies of data. Typical TLS systems choose to make copies at every loop iteration, thus exponentially increasing the memory requirement.

## 1.3 Proposed solution

The proposed solution consists of using the polyhedral model at runtime, aided by thread level speculation. Apollo (Automatic speculative POLyhedral Loop Optimizer) is a framework that goes one step beyond the current automated compilers, and applies the polyhedral model dynamically by using TLS. Apollo employs partial instrumentation to detect the code behaviour at runtime. Based on the profiling information, Apollo determines if the code behaviour is compatible with the polyhedral model. If compatible, Apollo uses the static polyhedral compiler Pluto to compute the set of loop optimizations to apply, and then instantiates one of the statically generated code skeletons. It continually monitors the program execution to detect any dependence violation in a decentralized manner. If a violation is detected, the program state is restored to a consistent state and the original sequential code is executed. We present our contributions in the development of Apollo in this thesis.

But this dynamic and speculative approach opens to further extensions that are even more important: the application of the polyhedral model to codes which are not

a priori, compatible with it, *i.e.* which exhibit non-linear behaviors. In this thesis, we develop and implement the idea of a prediction model based on approximation, and which allows the application of advanced polyhedral transformations on such codes. By using the polyhedral model dynamically, Apollo is able to propose advanced loop transformations for dynamic codes. The TLS system inside Apollo also employs it to construct a de-centralized verification system, and for the backup system to reduce the amount of data which needs to be copied for constructing the consistent state. The combined polyhedral model - TLS approach thus powers an automated compiler which can speculatively optimize dynamic codes for performance.

The rest of thesis is organized as follows. Chapter 2 recalls the polyhedral model of nested loops. Chapter 3 addresses the thread level speculation and the related state-of-the-art. Details on the Apollo framework, and how dynamically polyhedral codes are optimized are presented in Chapter 4. Chapter 5 details our extension of handling codes which are exhibiting non-linear behaviors. Chapter 6 concludes the manuscript and briefs the possible future work.

This page intentionally left blank

*God used beautiful mathematics in creating the world.*

Paul Dirac

*Mathematics is a game played according to certain simple rules with meaningless marks on paper.*

David Hilbert

*The beating of earth's heart is in mathematics.*

Asif

# 2

# The Polyhedral Model

## 2.1 Introduction

The polyhedral model or the polytope model [14, 15] is a powerful mathematical and geometrical framework for analyzing and optimizing for-loop nests. It captures the program execution in a concise and precise manner. The model treats each loop iteration inside a nested for-loop as lattice points inside a rational polyhedron. Thus, the model looks at each instance of a statement rather than the statement itself. Traditional program representations such as Abstract Syntax Tree (AST), Control Flow Graph (CFG) etc. suffice for simple optimizations such as constant folding or scalar replacement, however, they are not powerful enough to perform complex transformations such as loop skewing, tiling etc. Representing a loop nest and their dependencies in a mathematical model allows to formulate the loop optimization task as an optimization problem using a set of linear equalities and inequalities. The solution to this

optimization problem can be found using linear programming techniques.

The necessity of the polyhedral model arise from the limitations of the Internal Representation (IR) used in compilers. In traditional intermediate program representations such as abstract syntax trees, each statement appears only once, even if executed multiple times (for e.g. statements inside a loop). One of the problems in such representations is that they are not sufficiently precise. Two statements inside a loop may be dependent only for some iterations of the statements; this fact cannot be represented in traditional representations and these statements are conservatively treated as a single entity. One other limitation is that iteration based transformations of a statement are impossible, thus limiting the possible schedules.

The origin of the polyhedral model can be traced back to the work of Karp, Miller and Winograd [16]. This work consisted of determining the schedule for SUREs (System of Uniform Recurrence Equations). In [17], Quinton studied a (restricted) form of SUREs, defined over a polyhedral space. The space of valid schedules were obtained as the space of solutions of an integer linear programming problem. Identifying dependencies automatically from array subscript equations is an essential part of polyhedral model based optimizations. This was studied by Banerjee in [18], which proposes a test called the Banerjee test which can be used to prove (or disprove) whether a real solution exists to a single linear equation subjected to loop bounds. The test is an approximate test and only considers one array subscript of a multi dimensional array at a time. Also, the test requires the loops to have constant bounds and ignores the constraints on loop bounds expressed using other loop indices. Typically, if the Banerjee test fails to prove the non existence of dependencies, the GCD (Greatest Common Divisor) test can be used. However, the GCD test ignores the loop bounds and in many cases the GCD is 1, which ends up being very conservative. The I-Test [19] is based on the Banerjee test, the major difference being that I-Test can prove (or disprove) the existence of integer solutions in many cases, whereas Banerjee test deals with real solutions. The Omega test [20] and Power test [21] are both dependence tests based on

the Fourier-Motzkin variable elimination. The Omega test is an exact dependence test and it consists of a general purpose integer constraint satisfaction algorithm. As the polyhedral model evolved, most tools relied on checking for emptiness of a polyhedron (Omega test, simplex test) to detect dependencies. In [22], Feautrier introduced one dimensional affine schedules based on Farkas' lemma. This was extended to incorporate multi dimensional schedules in [23]. From there, many tools and libraries were developed to support the polyhedral model. Currently production level compilers such as GCC (Graphite) and R-stream use the polyhedral model to optimize loops.

This chapter focuses on the polyhedral model and its associated mathematical background. The following sections describe the building blocks of the polyhedral model such as iteration spaces, access functions, dependence analysis, loop transformations etc., and how the polyhedral model can be used to model loop nests. This chapter also describes a set of existing polyhedral tools, such as polyhedral compilers and a set of libraries that helps in analysis, scheduling or generating code.

## 2.2 Mathematical background

In this chapter, the set of all real numbers, and the set of integers are represented by $\mathbb{R}$ and $\mathbb{Z}$, respectively.

**Definition 1** (Affine function)**.** A Function $f : \mathbb{R}^m \Leftrightarrow \mathbb{R}^n$ is affine if it can be defined as:

$$f(\vec{x}) = A\vec{x} + \vec{b}$$

Where matrix $A$ / $\mathbb{R}^{m\times n}$ and vector $\vec{b}$ / $\mathbb{R}^n$.

In our context, $A$ is an integer matrix. i.e. $A$ / $\mathbb{Z}^{m\times n}$. Informally, an affine function is a linear function plus a translation. All linear functions are affine, but all affine functions are not linear.

**Definition 2** (Affine hyperplane)**.** An affine hyperplane is an affine ($n$ 1) dimensional subspace of an $n$-dimensional space. In our context, for $\vec{c}$ / $\mathbb{Z}^n$ with $\vec{c} \nvDash \vec{0}$ and a scalar,

$b \ / \ \mathbb{Z}$, an affine hyperplane is the set of all vectors $\vec{x} \ / \ \mathbb{Z}^n$, such that:

$$\vec{c} * \vec{x} = b$$

Two hyperplanes are parallel when they are defined with two different values of $b$, but with the same vector $\vec{c}$, normal to each hyperplane. Two vectors $\vec{x_1}$ and $\vec{x_2}$ are on the same hyperplane iff $\vec{c} * \vec{x_1} = \vec{c} * \vec{x_2}$. It generalizes the notion of planes: for instance, a point, a line and a plane are hyperplanes in 1, 2 and 3 dimensional spaces respectively.

**Definition 3** (Affine half-space)**.** A hyperplane divides the space into two half-spaces, the positive half-space $H_1$ and the negative half-space $H_2$, such that:

$$H_1 = \}\vec{x} \ / \ \mathbb{R}^n \ \vec{c} * \vec{x} \approx b|$$

and

$$H_2 = \}\vec{x} \ / \ \mathbb{R}^n \ \vec{c} * \vec{x} \geq b|$$

$\vec{c} \ / \ \mathbb{R}^n$ with $\vec{c} \not\equiv \vec{0}$ and $b \ / \ \mathbb{R}$.

**Definition 4** (Convex polyhedron)**.** A convex polyhedron is the intersection of a finite number of half-spaces. We denote $A \ / \ \mathbb{R}^{m \times n}$ a constraint matrix, $b \ / \ \mathbb{R}^n$ a constraint vector and $P$ a convex polytope so that $P \ll \mathbb{R}^n$:

$$P = \}\vec{x} \ / \ \mathbb{R}^n \ A\vec{x} + \vec{b} \approx 0|$$

**Definition 5** (Parametric polyhedron)**.** A parametric polyhedron denoted $P(\vec{p})$ is a polyhedron parametrized by a vector of symbols denoted ny $\vec{p}$, such that:

$$P(\vec{p}) = \}\vec{x} \ / \ \mathbb{R}^n \ A\vec{x} + B\vec{p} + \vec{b} \approx 0|$$

Where $A \ / \ \mathbb{R}^{m \times n}$ is a constraint matrix, $B \ / \ \mathbb{R}^{m \times p}$ is a coefficient matrix and $\vec{b} \ / \ \mathbb{R}^m$.

**Definition 6** (Polytope)**.** A polytope is a bounded polyhedron.

**Definition 7** (Iteration vector)**.** An iteration vector of a statement $S$ is the vector containing all the values of the iterators of the loops surrounding the statement. It can be represented by $\vec{i_S}$ / $\mathbb{Z}^n$ where $n$ is the depth of the loop nest enclosing the statement $S$. It represents the dynamic instance of the statement.

**Definition 8** (Perfect loop nest, Imperfect loop nest)**.** A set of nested loops is called a perfect loop nest iff all statements appearing in the nest appear inside the body of a unique innermost loop. Otherwise, the loop nest is called an imperfect loop nest

## 2.3 Static Control Parts (SCoP)

In order for a code to be compatible with the polyhedral model, the requirements are that (i) each loop iterates according to a unique index variable whose bounds are affine expressions of the enclosing loop indices, and (ii) the memory instructions that are handled are limited to accesses to simple scalar variables or to multi-dimensional array elements referenced using affine expressions on the enclosing loop indices. Such loop nests are analyzed precisely with respect to data dependencies that occur among the statements and across iterations.

Static Control Parts (SCoP) [24, 25] are sections of code composed of sequence of loops (possibly imperfect) which are amendable for polyhedral transformations. The *static control* refers to the fact that the data dependencies and control flow can be determined at compile time and can be expressed in terms of a finite number of expressions. A static control nest may embed several simple, multi-level statements. More specifically, a statement is a source code instruction, assigning a value, potentially the result of arithmetic operations, into a memory location. Breaking the control flow with instructions such as break, goto, or return is illegal inside a SCoP. In a SCoP, the data access functions, the loop bounds and the conditionals are expressed as affine functions of the enclosing loop iterators and loop parameters. Though it is not nec-

Listing 2.1: Example of a valid SCoP: *jacobi-2b* kernel

```
#pragma scop
  for (t = 0; t < tsteps; t++)
    {
      for (i = 1; i < n   1; i++)
        for (j = 1; j < n   1; j++)
          S0: B[i][j] = 0.2 * (A[i][j] + A[i][j  1] +
                               A[i][1+j] + A[1+i][j] + A[i  1][j]);
      for (i = 1; i < n  1; i++)
        for (j = 1; j < n  1; j++)
          S1: A[i][j] = B[i][j];
    }
#pragma endscop
```

essary to know the value of the parameters at compile-time, they must remain fixed during the execution of the SCoP.

An example of a valid SCoP is shown in Listing 2.1. The polyhedral representation of a SCoP requires three components: (i) A context, which encodes the loop parameters and is used to prune the resulting space, (ii) A constraint matrix encoding the iteration domain and dependence constraints and (iii) A scattering function, encoding the schedule or order in which the statements should be executed. For each statement there is an associated constraint matrix and a scattering function.

## 2.4 DCoP

A Dynamic Control Parts (DCoP) is the dynamic counterpart of a SCoP. Many programs, especially the ones using while loops, indirect accesses and pointers cannot be analyzed statically. However, these programs can possibly exhibit an affine behavior at runtime, at least in phases. If the program behavior can be predicted as affine, then the program can be optimized using dynamic and speculative techniques using polyhedral transformations. In order to handle a DCoP region, the following mechanisms are required: (i) A mechanism to express DCoP regions, or to automatically identify the DCoP regions, (ii) A mechanism to dynamically observe the code behavior, (iii) A

Listing 2.2: Example to illustrate DCoP

```
1 for(i=0; i<NUM_NODES; i++)
2 {
3     Node *neighbours = nodes_list[i];
4     while(neighbours)
5     {
6       neighbours->value++;
7       neighbours = neighbours->next;
8     }
9 }
```

mechanism to check whether the observed behavior fits the affine model, (iv) A mechanism to propose an optimizing transformation, (v) A mechanism to generate/select the code representing the code transformations, (vi) A mechanism to facilitate speculation, (vii) A mechanism to facilitate backup and rollback and (ix) A mechanism to verify the speculation.

Consider the code in Listing 2.2. The DCoP of this example is valid in program phases where the following conditions hold: (i) The loop bounds of the while loop should be expressible as affine functions, (ii) The memory accesses in lines 4, 6 and 7 should be expressible as affine functions.

## 2.5 Loop nest modelling

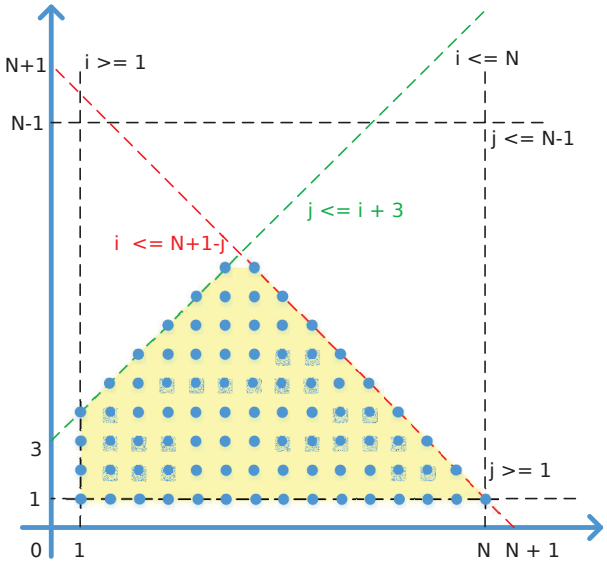
The polyhedral model is a mathematical model to represent the loop nest. It represents each instance of a statement as an integer points inside a convex polyhedron. This section explains the details of the representation and its building blocks.

### 2.5.1 Iteration space

**Definition 9** (Iteration space or Iteration domain)**.** The set of integer vectors corresponding to the actual executions of a statement is called the iteration domain of that statement. The set of all iteration vectors corresponding to a given statement will form its iteration domain.

```
for(i = 1; i <= N; i++)
  for(j=1; j<=N 1; j++)
    if(i<=N+1 j && j<=i+3){
      S0(i, j);
    }
  }
}
```

**(a)** A sample *for*-loop




**(b)** polyhedron

**Figure 2.1:** Illustration of a loop nest and its associated domain

$$\begin{cases} i \quad 1 \approx 0 \\ j \quad 1 \approx 0 \\ i+N \approx 0 \\ j+N \quad 1 \approx 0 \\ i \quad j+N+1 \approx 0 \\ i \quad j+3 \approx 0 \end{cases} \qquad \mathcal{P}^{S_0}(M) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} / \mathbb{Z}^n \middle| \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 3 \end{pmatrix} \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} \approx 0 \right\}$$

**Figure 2.2:** Domain constraints and the corresponding constraint matrix

Figure 2.1 shows an affine loop nest and its corresponding domain. The statement $S_0$ is enclosed by a loop nest of depth two and guarded by an *if* condition. Thus, the domain constraints for statement $S_0$ includes the constraints on loop iterators $i$ and $j$ and the constraints of the conditional *if*. This is graphically shown in Figure 2.1b. The iteration domain of the loop is expressed as set of inequalities on Figure 2.2. Each constraint is expressed as an inequality. Each integer point inside the resulting polyhedron represents an instance of $S_0$.

The iteration domain of a statement S can be modelled by an $n$ dimensional polytope, $\mathcal{P}^S(\vec{p}) \ll \mathbb{Z}^n$, such that:

Listing 2.3: Example to illustrate memory access functions

```
for (i = 1; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < i; k++)
      B[i][j] += alpha * A[i][k] * B[j][k];
```

$$\mathcal{P}^S(\vec{p}) = \}\vec{x} \ / \ \mathbb{Z}^n \ A\vec{x} + B\vec{p} + \vec{b} \approx 0|$$

**Definition 10** (Lexicographical order)**.** The lexicographical order, denoted by $\rightarrow$, which represents the total order or sequential execution of the statements is defined by:

$$(a_1, \ldots, a_n) \rightarrow (b_1, \ldots, b_n) \in \ \mathcal{D}\mathcal{d} : 1 \geq i \geq n, \exists m : 1 \geq m < i, \ a_m = b_m \ \{ \ a_i < b_i$$

### 2.5.2 Access functions

The access functions of a statement is the set of affine functions which captures the memory locations on which the statement operates (reads or writes). The access functions are parametrized by the enclosing loop iterators and loop parameters. They play a key role in precise memory analysis and thus for determining the dependencies. They can be represented as

$$f(\vec{x}) = F\vec{x} + \vec{f}$$

Where $F \ / \ \mathbb{Z}^{l \times d}$ is a coefficient matrix of the array subscripts, $l$ is the dimensionality of the array, $d$ is the loop depth, $\vec{x}$ is an iteration vector and $\vec{f}$ is a constant vector.

In Listing 2.3, there are five memory accesses. $B[i][j]$, $A[i][k]$, $B[j][k]$ and *alpha* are read from memory and $B[i][j]$ is written to memory. Since all the arrays are two dimensional, $l = 2$ and since the statement is surrounded by three loops, $d = 3$, the access function for each access is as follows:

$$f_{R^B}(\vec{x}) = f_{W^B}(\vec{x}) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \\ N \\ 1 \end{pmatrix} \in B[i][j]$$

$$f_{R^A}(\vec{x}) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \\ N \\ 1 \end{pmatrix} \in A[i][k]$$

$$f_{R^B}(\vec{x}) \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \\ N \\ 1 \end{pmatrix} \in B[j][K]$$

$$f_{R^{alpha}}(\vec{x}) \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \\ N \\ 1 \end{pmatrix} \in \&(alpha)[0]$$

## 2.6 Dependence analysis

The loop transformations performed on a loop must preserve the original semantics of a program *i.e.* the output produced by the transformed program should be the same as the one produced by the original program. Dependence analysis is the key to ensure this criterion.

**Definition 11** (Data Dependence)**.** Two statements $S$ and $T$ are said to be dependent,

iff there exists an instance of $S$, $S(\vec{x_i})$, and an instance of $T$, $T(\vec{x_j})$ such that, both $S(\vec{x_i})$ and $T(\vec{x_j})$ access the same memory location and at least one of them is a write access.

If in the original sequential execution order $S(\vec{x_i})$ occurs before $T(\vec{x_j})$ ($\vec{x_i} \rightarrow \vec{x_j}$), then $T$ is said to be dependent on $S$. $S$ is called the source statement and $T$ is called the target, or the sink, or the destination statement. A loop carried dependence arises when two different iterations of the loop access the same memory location:

Based on the kind of memory accesses the dependences are classified as follows:

- RAW: read-after-write, or *flow dependence*
- WAR: write-after-read, or *anti-dependence*
- WAW: write-after-write, or *output dependence*

RAW is a true dependence and cannot be removed. WAR and WAW dependencies can be eliminated using techniques like renaming, scalar expansion [26] or array expansion [27], but these techniques are not generally considered inside the polyhedral model. RAR (read-after-read or input dependence) dependencies are not considered as dependencies, since the memory is not altered. However, RAR dependencies are useful for some optimizations to improve data locality by reducing the data reuse distance.

There are two popular dependence representations which are widely used: (i) The Distance Vectors and (ii) The Dependence polyhedron.

### 2.6.1 Distance vector representation

Distance vectors indicate the distance between the source and the target statements. If the statements $S(\vec{x_i})$ (source) and $T(\vec{x_j})$ (target) are dependent, then, the distance vector is:

$$\vec{d} = \vec{x_j} \text{ - } \vec{x_i}$$

Listing 2.4: Example for illustrating distance vector and direction vector kernel

```
for (i = 0; i < N; i++)
{
  S0: C[i] = C[0] + i;
  for (j = 0; j < N; j++)
  {
    S1: A[i][j] = A[i][j  1] + B[i][j];
  }
}
```

The direction vector shows the direction of the dependence. A Direction vector is more concise than a distance vector, but less precise. The direction vector is represented by:

$$\vec{\sigma} = \text{sign}(\vec{d})$$

The sign of $\vec{d} = (d_1, d_2, ..., d_h)$ is given by $\text{sig}(\vec{d})$ = (sig($d_1$), sig($d_2$), ..., sig($d_h$)), where $h$ is the loop depth and each component

$$sign(d_i) = \begin{cases} 1, \ if \ i \ > \ 0, \\ 1, \ if \ i \ < \ 0, \\ 0, \ if \ i \ = \ 0. \end{cases}$$

The distance vector and the direction vector should always be lexicographically non negative. A vector is lexicographically positive if the first non zero component is positive. Otherwise, it is negative or null. The first non zero element is also known as the leading element denoted by $d_l$. The dependence is said to be carried by the loop at depth $l$, if $1 \geq l \geq h$. The dependence is not carried by any loop if $l = h + 1$, which is equivalent to $\vec{d} = \vec{0}$. Only loop carried dependencies are considered for dependence analysis.

Consider Listing 2.4. The dependence from statement $S_1$ to itself can be represented by:

- The distance vector $\vec{d} = (i, j) \quad (i, j \quad 1) = (0, 1)$

- The direction vector $\sigma = (0, 1)$ or $(\leq +)$

For statement $S_0$, there are multiple distance vectors. However, there is only one direction vector:

- The distance vectors $\vec{d} = (x, 0)$ where $1 <= x <= N \quad 1$

- The direction vector $\sigma = (1, 0)$ or $(+, \leq)$

### 2.6.2 Dependence polyhedron

A data dependence graph can be used to represent the data dependencies. Each vertex of the graph corresponds to a statement and each edge corresponds to an inter or intra statement dependence. For each edge $e$, the relation between the dynamic instances of $S$ and $T$ that are dependent can be captured by a dependence polyhedron [28]. The dimensionality of the dependence polyhedron is equal to the sum of the source and the target instructions dimensions along with additional dimensions for the loop parameters and constants. The exact conditions of the existence of a dependence is derived through linear equalities and inequalities based on the affine relation between the iterations and the data access functions. Since the equalities can be replaced by two inequalities ('$\geq$ 0' and '$\approx$ 0'), all the conditions can be expressed in terms of inequalities. While representations like distance vector or direction vector are associated with a particular syntactic nesting, the dependence polyhedron is more general and captures the exact relation between the integer points of the polyhedra. The dependence from statement $S_1$ to itself, in Listing 2.4, can be represented by a dependence polyhedron defined by the following equalities and inequalities:

$$
\begin{array}{ll}
i >= 0; & i >= N; \\
i' >= 0; & i' >= N; \\
j >= 0; & j >= N; \\
j' >= 0; & j' >= N; \\
i' = i; & j' = j \quad 1; \\
j' >= j; &
\end{array}
$$

Where $i$ and $j$ are the source iterators and $i'$ and $j'$ are the target iterators.

### 2.6.3 Loop scheduling

**Definition 12** (Scheduling function)**.** The scheduling function of a statement $S$, or simply the schedule of $S$, is a function that maps each dynamic instance of statement $S$ to a logical date, expressing the execution order between statements:

$$\exists \vec{x} \ / \ \mathcal{P}^S, \theta^S(\vec{x}) = \Theta^S \vec{x} + \vec{t}$$

Where $\Theta^S \vec{x}$ is a matrix representing the schedule and $\vec{t}$ is a constant vector.

A normalized representation of scheduling matrices was proposed by Cohen *et al.* [29, 30]. This representation encodes the relative order of each statement within a loop by using one additional extra row per loop. The representation also encodes both static and dynamic information:

$$
\Theta^S = \left[\begin{array}{ccc|ccc|c}
0 & *** & 0 & 0 & *** & 0 & \beta_0^S \\
A_{1,1}^S & *** & A_{1,d}^S & \Gamma_{1,1}^S & *** & \Gamma_{1,p}^S & \Gamma_{1,p+1}^S \\
0 & *** & 0 & 0 & *** & 0 & \beta_1^S \\
A_{2,1}^S & *** & A_{2,d}^S & \Gamma_{2,1}^S & *** & \Gamma_{2,p}^S & \Gamma_{2,p+1}^S \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\
A_{d,1}^S & *** & A_{d,d}^S & \Gamma_{d,1}^S & *** & \Gamma_{d,p}^S & \Gamma_{d,p+1}^S \\
0 & *** & 0 & 0 & *** & 0 & \beta_d^S
\end{array}\right]
$$

Listing 2.5: Example to illustrate scheduling (cholesky from polybench)

```
for (i = 0; i < n; ++i)
{
  S0: x = A[i][i];
 for (j = 0; j <= i    1; ++j)
    S1: x = x    A[i][j] * A[i][j];
  S2: p[i] = 1.0 / sqrt(x);
 for (j = i + 1; j < n; ++j)
   {
      S3: x = A[i][j];
     for (k = 0; k <= i    1; ++k)
        S4: x = x    A[j][k] * A[i][k];
      S5: A[j][i] = x * p[i];
   }
}
```

This encoding is suitable for expressing composition of transformations. It is composed by three sub matrices:

- $A^S$ / $\mathbb{Z}^{d^S \times d^S}$ represents the iteration vector where $d^S$ is the loop depth of statement $S$.
- $\Gamma^S$ / $\mathbb{Z}^{d^S \times (d^p+1)}$ represents the global parameters where $d^p$ is the number of global parameters.
- $\beta^S$ / $\mathbb{Z}^{d^S+1}$ represents the scheduling order of the statements which are scheduled in the same iteration. Informally this represents the textual order.

Consider the code in Listing 2.5. The scheduling functions of the statements are:

$$\theta^{S_0}(\vec{x}) = (0, i, 0, 0, 0, 0, 0)$$

$$\theta^{S_1}(\vec{x}) = (0, i, 1, j, 0, 0, 0)$$

$$\theta^{S_2}(\vec{x}) = (0, i, 2, 0, 0, 0, 0)$$

Listing 2.6: Example to illustrate scheduling (parallel)

```
for (i = 1; i < N; i++)
{
  #pragma omp parallel section
  {
    #pragma omp section
     {
        S0 :
     }
     #pragma omp section
     {
        S1 :
     }
  }
  S2
}
```

$$\theta^{S_3}(\vec{x}) = (0, i, 3, j, 0, 0, 0)$$

$$\theta^{S_4}(\vec{x}) = (0, i, 3, j, 1, k, 0)$$

$$\theta^{S_5}(\vec{x}) = (0, i, 3, j, 2, 0, 0)$$

Since the scheduling represents the order in which the statements are going to be executed, it should also be able to express parallelization. For a detailed illustration, consider Listing 2.6. The corresponding scheduling functions are:

$$\theta^{S_0}(\vec{x}) = (0, i, 0)$$

$$\theta^{S_1}(\vec{x}) = (0, i, 0)$$

$$\theta^{S_2}(\vec{x}) = (0, i, 1)$$

Statement $S_2$ should execute after both statements $S_0$ and $S_1$ as its timestamp is higher. However, the timestamp for both $S_0$ and $S_1$ are the same, implying that they

could be executed in any order relative to themselves. Thus, this execution order also covers parallelization.

### 2.6.4 Loop Transformation

Loop transformations are aimed at re-ordering or re-structuring loops so that the program exhibits desirable qualities regarding optimization issues like data locality and parallelism.

**Definition 13** (Loop transformation)**.** A transformation maps each instance of a statement's to a logical date in the transformed space, which respects the original program semantics. Transformations are typically applied to enhance data locality, reduce control code and expose or improve the degree of parallelism. The new schedule might express a new execution order of the statements.

**Lemma 1** (Affine form of the Farkas lemma/ Alternative Theorem)**.** Let $\mathcal{P}$ be a non-empty polyhedron, defined by inequalities of the form $A\vec{x} + \vec{b} \approx 0$. Then, any affine function $f(\vec{x})$ is non negative everywhere in $\mathcal{P}$ iff it is a non negative affine combination of the faces of $\mathcal{P}$:

$$f(\vec{x}) = \lambda_0 + \vec{\lambda}(A\vec{x} + \vec{b}), \text{ where } \lambda_0 \approx 0 \text{ and } \lambda \approx 0$$

$\lambda_0$ and $\vec{\lambda}$ are called the Farkas multipliers.

Most of the widely used polyhedral tools such as Pluto [31] find an optimizing transformation by restricting the search space. An alternative approach would be to generate a set of possible optimizing transformations and to check the validity of each transformation. Yet another approach would be to choose a single preferred transformation and find the closet legal transformation, possibly itself. Multi-dimensional affine transformations capture a sequence of simpler transformations such as loop fission, loop fusion, loop interchange, loop skewing, loop reversal, tiling etc. Tiling cannot

Listing 2.7: Example to illustrate Fourier elimination

```
for (i = 0; i <=N; i++)
{
  for (j = i + 1 ; j <=N; j++)
  {
     for (k = j; k <=i + j; k++)
     {
        S1(i, j, k);
     }
  }
}
```

be directly expressed as an affine function of the original iterators as it requires increasing the dimensions of the domain. Polyhedral transformations can be used to apply transformations on any kind of nesting structure. More formally, for two dependent iterations $\vec{x_i}$ and $\vec{x_j}$, the difference of the schedules $\Theta^S\vec{x_i}$ and $\Theta^T\vec{x_j}$ must be positive, such that, $\exists\vec{x_i},\vec{x_j}\ /\wedge_e,\ \Theta^T\vec{x_j} \quad \Theta^S\vec{x_i} \approx 0$ [32]. The entries of the iteration vectors $\vec{x_j}$ and $\vec{x_i}$ have unknown coefficients which are the components of the generic scheduling matrices $\Theta^S$ and $\Theta^T$. The problem is reformulated to involve the dependence polyhedron and is linearized with the affine form of the Farkas lemma. According to the Farkas lemma, the inequality can be described by an equality involving the Farkas multipliers, such that:

$$\exists\vec{x_i},\vec{x_j}\ /\wedge_e,\Theta^T\vec{x_j} \quad \Theta^S\vec{x_i} = \lambda_0 + \vec{\lambda}(A(\vec{x_i}\ \vec{x_j}) + \vec{b}), \text{ with } \lambda_0 \approx 0 \text{ and } \vec{\lambda} \approx \vec{0}$$

The Fourier-Motzkin algorithm is then applied to eliminate the Farkas multipliers. The set of solutions describes the space of the legal schedules.

However, any transformation picked from the resulting legal transformation space must be legal regarding all the remaining statement dependencies as well.

To illustrate the usefulness of the Fourier-Motzkin elimination, consider the code

in Listing 2.7. Assume that a loop interchange is applied such that the transformed loop iterators $(p, q, r) = (k, i, j)$. The original loop satisfies the following constraints:

$$\begin{cases} 1 \geq i \geq N \\ i + 1 \geq j \geq N \\ j \geq k \geq i + j \end{cases}$$

Expressing the above in the transformed iteration space results in:

$$\begin{cases} q \geq 0 \\ q \geq N \\ r \geq N \\ p + r \geq 0 \\ q \quad r \geq \quad 1 \\ p \quad q \quad r \geq 0 \end{cases}$$

By considering the equations which only contain $r$, the bounds for $r$ can be expressed as follows:

$$max(q + 1, p \quad q) \geq r \geq min(N, p)$$

By eliminating $r$, the following is obtained:

$$\begin{cases} q \geq 0 \\ q \geq N \\ q \geq N \quad 1 \\ p \quad q \geq N \\ p + q \geq \quad 1 \end{cases}$$

This system gives the bounds for $q$ as being:

$$max(0, p \quad 1, p \quad N) \geq q \geq min(N, N \quad 1, p \quad N)$$

Listing 2.8: Example to illustrate Fourier elimination (final code)

```
for (p = 1; p <= 2 * N     1; p++)
{
  for (q = max(0, p    N) ; q <= min(N    1, p    N); q++)
  {
    for (r = max(q + 1, p    q); r <= min(N,p); r++)
    {
      S1(q, r, p);
    }
  }
}
```

By assuming $N \approx 1$ and by simplification, we get:

$$max(0, p \quad N) \geq q \geq min(N \quad 1, p \quad N)$$

By eliminating $q$, the following is obtained:

$$\begin{cases} 1 \geq N \\ p \geq 2N \quad 1 \\ 1 \geq p \end{cases}$$

The range of $p$ is then:

$$1 \geq p \geq 2N \quad 1$$

The resulting program is shown in Listing 2.8.

In a loop nest, for a loop level $l$ to be parallel, none of the statement instance dependencies must cross iterations. Consequently, a polyhedron defined by $\{ \exists \vec{x_i}, \vec{x_j} \ / \ \wedge_e, \Theta^S[l..\trianglelefteq] < \Theta^T[l..\trianglelefteq] \}$ must be empty, *i.e.* the loop must not carry any dependence. Informally this means that there must not be two different iterations accessing the same memory location at the same loop level, where at least one access is a write. A dependence polyhedron can be augmented with this condition to check for parallelism.

Polyhedral optimizations aim at improving data locality and exploiting parallelization opportunities. Any polyhedral transformation is essentially a re-ordering of state-

Listing 2.9: Example to illustrate scheduling matrix (doitgen)

```
for (r = 0; r < N; r++)
  for (q = 0; q < N; q++) {
    for (p = 0; p < N; p++) {
      S0: sum[r][q][p] = 0;
      for (s = 0; s < N; s++)
       S1: sum[r][q][p] = sum[r][q][p]
            + A[r][q][s] * C4[s][p];
    }
    for (p = 0; p < N; p++)
      S2: A[r][q][p] = sum[r][q][p];
  }
```

ment instances such that spatial and temporal data localities are improved, and parallelization opportunities are exposed. The *schedule* represents the ordering of statements and iterations which map iterations in the domain of a statement ($D^S$) to a timestamp. A timestamp is a logical date, similar to the conventional date, e.g. date:month:year:hour:minute:second.

Different transformations can be easily represented by changing each of the scheduling sub-matrices. Loop transformations such as loop interchange, skewing, reversal can be achieved by modifying $A^S$. Loop shifting can be achieved by modifying $\Gamma^S$. Loop fission and loop fusion can be achieved by modifying $\beta^S$. $\beta^S$ is called the static component as it refers to the textual order of the statements, which is statically fixed at code generation. $A^S$ and $\Gamma^S$ are related to the iterators, and hence are called the dynamic components.

For a better understanding, the scheduling sub matrices for the statements in Listing 2.9 are listed below:

$$A^{S_0} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \Gamma^{S_0} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \beta^{S_0} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$A^{S_1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \Gamma^{S_1} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \beta^{S_1} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$A^{S_2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \Gamma^{S_2} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad \beta^{S_2} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

## 2.7 Polyhedral Tools

This section describes the most commonly used polyhedral tools and libraries.

### 2.7.1 Math libraries

Piblib (Parametric Integer Linear programming solver) [33] is a widely used library to find the lexicographical minimum (or maximum) point from the set of *integer* points belonging to a convex polyhedron. Piblib can also be used to check whether a given polyhedron is empty in order to validate whether a dependence exists or not. Polylib (Polyhedral Library) [34] can be used to compute intersections, differences, unions, convex hulls, simplifications, images and pre-images on both parametrized and non parametrized polyhedra. It can also be used to find the vertices of a polyhedron and to compute its Ehrhart polynomials. VisualPolylib is a visualization environment for Polylib, which allows the interactive use of Polylib. Barvinok [35] can be used to count the number of integer points inside both parametric and non parametric polyhedrons. FM (Fourier-Motzkin library) [36] is a library dedicated to rational polyhedra and is widely used for projecting out certain dimensions from a given system of inequalities. Along with projection, it also provides many functions to manipulate rational polyhedra, including the computation of the lexicographical maximum and minimum. ISL

(Integer Set Library) [37] is a library for manipulating integer sets and relations. It also supports parameters. ISL has specialized modules to compute dependencies, to generate schedules and to generate code. The code generation outputs an AST (Abstract Syntax Tree).

### 2.7.2 Code analyzers

Clan (Chunky Loop ANalyzer) [38] is a polyhedral input parser written in *C*. Clan can parse programs written in *C, C++, C# and Java* and extracts the polyhedral loop nests in the *OpenScop* format [39]. The output can then be used for dependence analysis, program transformation etc. Candl (Chunky ANalyzer for Dependences in Loops) [40] is capable of computing dependencies. It is also capable of checking whether a given transformation respects the program dependencies. In order to remove some dependencies, Candl uses techniques such as array expansion or array privatization.

### 2.7.3 DSL Languages

Xfor [41] provides a general purpose syntax for expressing an affine loop nest. The syntax is aimed at closely representing a loop nest in such a manner that the polyhedral model's key optimization parameters can be directly expressed in the syntax. It supports semi-automatic parallelization along with manual tuning or complete manual optimization, and is designed to bridge the gap between automatic and manual optimization. By semi-automatically optimizing the code, overhead of finding a suitable schedule can be avoided and by manual tuning, higher performance can possibly be obtained. Along with a graphical interface to specify the transformation, an automatic dependence analyzer is provided to check the validity of the manual transformations. The IBB (Iterate But Better) [42] compiler automatically converts *xfor* code to *C* code.

Polymage [43] is a domain specific language and compiler for image processing pipelines. The work is based on the fact that the typical image processing operations can be represented as pipelines consisting of interconnected stages. The work tries

to extract parallelism within the stages and to improve the data locality. The work proposes an overlapped tiling method for heterogeneous processing and a heuristic model to trade off between data locality and redundant computations.

### 2.7.4 Compilers and related

ClooG (Chunky Loop Generator) [44] can generate efficient code for a given $\mathbb{Z}$ polyhedron. CLooG has special flags (switches) to simplify the loop control. Users are allowed to control the code generation mechanism for efficiency or for readability. Major projects like Pluto and Graphite makes use of ClooG to generate the output code.

PLuTo (parallelization and locality optimization tool) [31] is a widely used automatic polyhedral source-to-source optimizer. It targets arbitrarily nested, static affine loop nests for simultaneous optimizations of locality and parallelism. It uses Clan, Candl/ISL and Cloog, for scanning *C* code, computing dependencies and to generate the transformed code. Pluto supports a wide range of loop optimizations such as parallelization, tiling, diamond tiling, fusion, fission, interchange, skewing, reversal etc. It is specifically known for its tiling capabilities. The tool aims to provide synchronization free parallelism, improved data locality and pipelined parallelism.

Polly [45] is a high level loop and data locality optimizer for LLVM infrastructures. It is implemented as a set of LLVM passes and works on the LLVM Internal Representation (IR). Polly relies on ISL for dependence analysis and transformation. Since Polly works on the LLVM IR, which is in SSA form, it has better granularity for each statement when compared to tools such as Pluto. Along with other optimizations Polly is able to generate SIMD and OpenMP code.

Graphite [46] is a high level memory optimization tool based on the polytope model and is integrated in the GCC framework. The *SCoP* regions are automatically detected. The framework is built on top of the scalar evolution, array and pointer analyses, data dependence analysis and scalar range estimate modules of gcc. Code generation was achieved using ClooG. In newer versions (gcc $\approx$ 5.0), it is done using ISL.

Loopo [47] is yet another automatic polyhedral loop optimizer. The project embeds a code parser, dependence analyzer, automatic scheduler and code generator. It also embeds dispo, a polyhedral visualization tool which is capable of visualizing the polyhedron along with the dependencies. Loopo also allows manual scheduling. The project is no more under active development.

LeTseE (the LEgal Transformation SpacE Explorator) [48] is an iterative polyhedral optimization framework focusing on static affine parts of the program. The tool aims to improve the optimization by measuring machine characteristics and by looking at the data to break some conservative assumptions made by the compiler. From static analysis, the framework determines the space where the transformations are legal, and is then fed to a feedback directed space exploration algorithm. While running each version, the hardware counters are monitored to measure performance. The best performing version is output as *C* code.

CHiLL (Composing High-Level Loop Transformations) [49] is an optimization framework with composable code transformations and empirical optimizations. Instead of statically predicting the optimization, the framework runs the code on the target platform with representative input to determine the set of transformations that should be applied. A decision algorithm generates a set of transformation scripts each of which expresses a set of valid high level transformations. A search engine is then used to bound the parameters of the transformation scripts. From the original code and the bounded transformation script, the code generator produces the final code.

PPCG (Polyhedral Parallel Code Generation for CUDA) [50] is an automatic polyhedral compiler targeting GPU's. PPCG targets to offload any data parallel affine loop execution to the GPU. The tool offers a multi level tiling strategy tailored for GPU memory hierarchies. The memory transfer is automatically controlled by the compiler. PPCG uses *pet* to extract the polytope model. It uses *ISL* to compute the dependencies and to generate the schedule. The scheduler in ISL closely resembles to the one of Pluto. The scheduler also takes care of partitioning the section into host and GPU

sections. This is followed by memory allocation. Code generation is handled by PPCG itself, and uses a custom code generator similar to the one of *Cloog*. PPCG is able to produce CUDA, OpenMP and OpenCL codes.

R-Stream [51, 52] is a commercially available polyhedral compiler from Reservoir Labs. It is capable of producing OpenMP code as well as CUDA codes. The target machine information is given in XML format to the compiler. From the input code, the compiler automatically extracts the polyhedral regions. From the polyhedral representation it schedule and group operations into tasks, such that the required communication is minimized. Unlike PPCG, R-Stream can automatically determine the tile sizes.

In [53], Jean-François Dollinger *et al.* proposed an adaptive runtime compiler for hybrid CPU-GPU system. PPCG is used to generate different code versions of the input statically. The generated code versions differ in terms of transformations, such as the CUDA block size. Each such version is parametrized by loop parameters which can vary at runtime. During installation, a profiler simulates the different code versions of a micro benchmark, to characterize the bandwidth available between CPU and GPU. Using this prediction metadata, during actual runtime, a time estimate is built for each code version and the best one is selected and ran on both CPU and GPU. The first one to finish is committed. For CPU version, OpenMP code generated from Pluto is used. In [54], both CPU and GPU are jointly used. The scheduler, based on the separate time estimate for CPU and GPU, schedules different task units to CPU and GPU in such a manner that the load is balanced.

## 2.8 Limitations of the polyhedral model

The polyhedral model requires the memory accesses, the loop bounds and the control to be expressed as affine combinations of enclosing loop iterators, loop parameters and constants. The model has been extensively studied in the static context. However, the static approach strictly limits the class of codes that can be handled by the polyhedral

Listing 2.10: spmatmat (SPARK00)

```
1  for( row = 1; row <= left >Size; row++ )
2  {
3      pElement = left >FirstInRow[row];
4      while( pElement )
5      {
6          for( col = 1; col <= cols; col++ )
7          {
8              result[row][col] += pElement >Real *
9                                  right[pElement >Col][col];
10         }
11         pElement = pElement >NextInRow;
12     }
13 }
```

model. The current polyhedral model tools are restricted to a small class of compute-intensive codes that can be analyzed accurately and transformed only at compile-time. However, most legacy codes are not amenable to this model due to indirect or pointer based accesses to static or dynamic data structures, which prevent a precise dependence analysis to be performed statically. Another common reason is non-affine loop bounds. Function calls inside loop nests set another challenge to the polyhedral model, as the function definition may not be available, which is essential to build the dependence polyhedra. Informally, it should be able to statically determine the exact memory accesses that will occur inside the function call.

For a better understanding, consider the spmatmat code from SPARK00 [7] in Listing 2.10. The code multiplies two matrices and saves the result in another matrix. Assume that the input matrices are full matrices. Note that there is a *while loop* in Line 4, pointer based accesses in Lines 3, 8, 9 and 11. The presence of anyone of the latter will force the current polyhedral model based approaches to take the conservative approach. However, at runtime, by partial instrumentation, one can determine whether the *while loop* in Line 4 follows an affine function. This can be done by adding a loop counter variable, initialized to zero and incremented once per loop iteration and reset

to zero on loop exit. By collecting this counter value on loop exit (before resetting), and the enclosing iterators, and subjecting this data to interpolation, one can determine whether the behavior of the while loop can be precisely captured by the interpolating function. The same holds for the pointer based memory accesses. Now that the code behavior is determined as affine, the problem at hand is equivalent to a statically analyzable code, with two exceptions. One exception is that, in statically analyzable codes, the transformation itself is known at compile time, so the transformed code can be statically produced. However, in the case of dynamic transformations, one has to generate the transformed code at runtime. The second, and more profound problem is that, dynamic codes can change their behavior. For example, consider the memory read ($pElement \Leftrightarrow NextInRow$) in Line number 11. The memory allocation to the rows need not be affine. It may happen that the program behavior observed during the instrumentation phase is affine but not for the rest of the execution. This requires runtime verification. The verification system should ensure that the behavior observed during the instrumentation remains consistent with that of the rest of the execution. In cases where the prediction does not hold, the system should rollback and re-execute. Then the execution is thus speculative. To support this speculative execution and to facilitate the rollback, a backup system may also be required.

Another set of limitations that arises from the foundations of the polyhedral model is the rigorous adherence to affine functions. Typical dynamic codes may deviate from the affine behavior. It may exhibit piecewise affine behaviors, near affine behaviors or completely random behaviors. This type of memory behavior is usually the result of indirect accesses and pointer based memory accesses. One way to handle codes with piecewise affine behavior would be to split the outermost loop into small slices and then speculate on these slices. Only the chunks where the affine functions break needs to be rollbacked, and linear functions need to be readjusted to characterize the next phase. Near affine behavior is an interesting problem with respect to polyhedral optimizations. For the moment, let us consider the case where an access is near affine

Listing 2.11: Example to illustrate non linear accesses (trmat - SPARK00)

```
for( i = 1; i <= n; i++ )
{
    k1 = ia[i];
    k2 = ia[i + 1]    1;
    for( k = k1; k <= k2; k++ )
    {
        j = ja[k];
        next = iao[j];
        jao[next] = i;
        if( job == 1 )
            ao[next] = a[k];
        iao[j] = next + 1;
    }
}
```

and never causes any dependence violation. In this case, the latter access can be approximated by an affine function and this affine function can also be considered for computing the polyhedral transformation so that, even this memory access is optimized along with others. The challenging problem here is how to handle cases when there is an actual dependence and how to verify that non predicted accesses do not cause a dependence violation. For the random memory accesses, it is better to ignore them while computing the polyhedral optimization, but , then ensure that these accesses do not cause any dependence violation. Consider the trmat code from the SPARK00 benchmark suite shown in Listing 2.11. The code contains single and double indirect memory accesses, and thus static analysis is impossible. The memory read and write on array *iao* depends on value $ja[k]$, which is only resolved at runtime. Similarly, the actual accesses to arrays *jao* and *ao* can only be known at runtime as they are data dependent. However, if the accesses are performed on unique memory locations, the loops can be parallelized.

Non linear loops are also incompatible with the polyhedral model. A loop is non linear if (i) the loop bounds cannot be represented by affine functions or (ii) if the loop increment is non affine. Non linear bounds are quite common in codes, when compared

to the non affine loop increments. The upper bound of a loop can be determined from the lower bound and the number of loop iterations. Thus, if the latter are affine expressions, one can apply polyhedral optimizations. Note that the loop increment and the number of loop iterations alone are not sufficient to be compatible with the polyhedral model. A trivial solution to this problem would be subsume the non linear loop to a single statement. This approach has a serious limitation which is that the non linear loop cannot be parallelized along with the fact that the memory accesses inside the non linear loop are not optimized. Consider the code in 2.11 and assume that the lower bound of the loop is an affine expression. In this case, we just need to ensure that no value of $k$ would create a new dependency. For example, consider a situation where $\&jao[0]$ is equal to $\&ao[100]$. As long as the value of $k$ is less than 100, there is no dependence between these accesses. The challenging problem is when the lower bound of the loop is not an affine expression. Just ensuring that dependencies are not violated is not enough in this case: there should be a dynamic mechanism which ensures that the predicted lower bound and the actual value are the same.

Our solution to the above problems is APOLLO, a tool which applies the polyhedral model dynamically. Its core principles rely on the polyhedral model and dynamic speculation. By profiling a sample of the outermost loop, Apollo predicts the affine functions and uses them to apply polyhedral model optimizations dynamically. The next chapter details the architecture and functioning of Apollo.

*A ship is always safe at the shore - but that is NOT what it is built for.*

Albert Einstein

*In practical life we are compelled to follow what is most probable; in speculative thought we are compelled to follow truth.*

Baruch Spinoza

*Chance, too, which seems to rush along with slack reins, is bridled and governed by law.*

Boethius

# 3

# Thread Level Speculation

## 3.1 Introduction

Thread Level Speculation or TLS refers to a system in which different threads process data simultaneously, possibly performing some unsafe operation, and temporarily buffering the result on an individual basis. In order to ensure program correctness, at some point in time, the threads are validated or discarded by checking for data conflicts between parallel segments. If the thread is validated, the temporary state of the thread is committed, thus merging the speculative updates to the globally valid state, called *safe state*. If the thread is discarded, speculative updates are discarded and possibly some cleanup operations are performed. The execution is then resumed from the last safe state. Thanks to the safe state, the whole re-execution of the program can thus be prevented.

TLS systems are mainly used in dynamic environments to parallelize tasks where

Listing 3.1: Example to illustrate TLS system

```
void vector_sum(int *dest, int *source_1, int *source_2)
{
    for(i = 0; i < N    1; i++)
    {
        dest[i] = source_1[i    1] + source_2[i    1];
    }
}
```

the static analysis is not sufficient enough to determine the validity of parallelization at compile time. In general, TLS systems extract parallelism from either (i) a loop or (ii) a set of tasks. Thread level speculation has two forms, (i) Software based and (ii) Hardware based. In software based TLS systems, the dependence violation, and possible backup and rollback, *etc.* are handled by the software itself, whereas, in hardware based TLS systems, these are handled by dedicated hardware.

### 3.1.1 Importance of thread level speculation

Thread level speculation is used in situations where, even dynamically, the code behaviour cannot be predicted. Simple dynamic optimizations do not always need to rely on on speculative techniques. For example, consider the code in Listing 3.1 which adds two arrays and saves the result in the third array. Assuming that the objective is the straightforward parallelization of the *for loop*, the only condition that needs to be checked at runtime is to check whether any of the source arrays ($source\_1 and source\_2$) overlaps with the target array (*dest*). A simple condition checking for the latter can be inserted before the loop entry and the parallel and sequential versions can be selected accordingly; hence, no speculation is required.

However, there are codes for which, even at a given sequence point in the runtime, the code behavior for the rest of the execution cannot be guaranteed. In order to illustrate the need for dynamic analysis, consider the example in Listing 3.2. The statement $S_0$ and $S_1$ are enclosed by two loops $i$ and $j$. Static dependence analysis

Listing 3.2: Example to illustrate TLS system

```
int A[N][N];
int selector[N][N];
for (i = 1; i <= N; i++)
{
    for (j = 1; j <= N; j++)
    {
        S0: int c = selector[i][j];
        S1: A[i][c] = A[i    1][j] + i + j;
    }
}
```

can help in determining that the straightforward parallelization of the outermost loop is not valid. However, since the value of array *selector* (and hence the variable $c$) is unknown at compile time, dynamic analysis is required for determining whether the inner loop can be parallelized. However, by just observing a few iterations of the loop, it cannot be guaranteed that the rest of the program execution would behave in a similar manner. Thus, speculation is required to handle this code dynamically.

At runtime, the $j$ loop may or may not carry a dependence based on the input data. Assume that the data values of $selector[i][j] = j$. Also assume that the system decided to speculate that loop $j$ is parallel. One possible execution scenario is depicted in Figure 3.1. Since all the threads access different memory addresses, the speculation is valid. Now assume another scenario where the data values of $selector[i][j] = j \,\%\, 3 + 1$. As shown in Figure 3.2, thread 0 and thread 3 executing in time step 0 access the same memory location, thus causing a data race. Hence, the parallel execution is not valid. Note that the execution would be valid if the thread 0 was executed in time step 0 and the thread 3 was executed in time step 1. A similar situation can be observed for thread 4 and thread 7. Since the system predicted that the parallel execution is valid and a memory dependence violation occurred, the system should rollback.

This chapter is focused on Thread Level Speculation (TLS). The *Overview* section gives the general principles and architecture of existing TLS systems. The applications

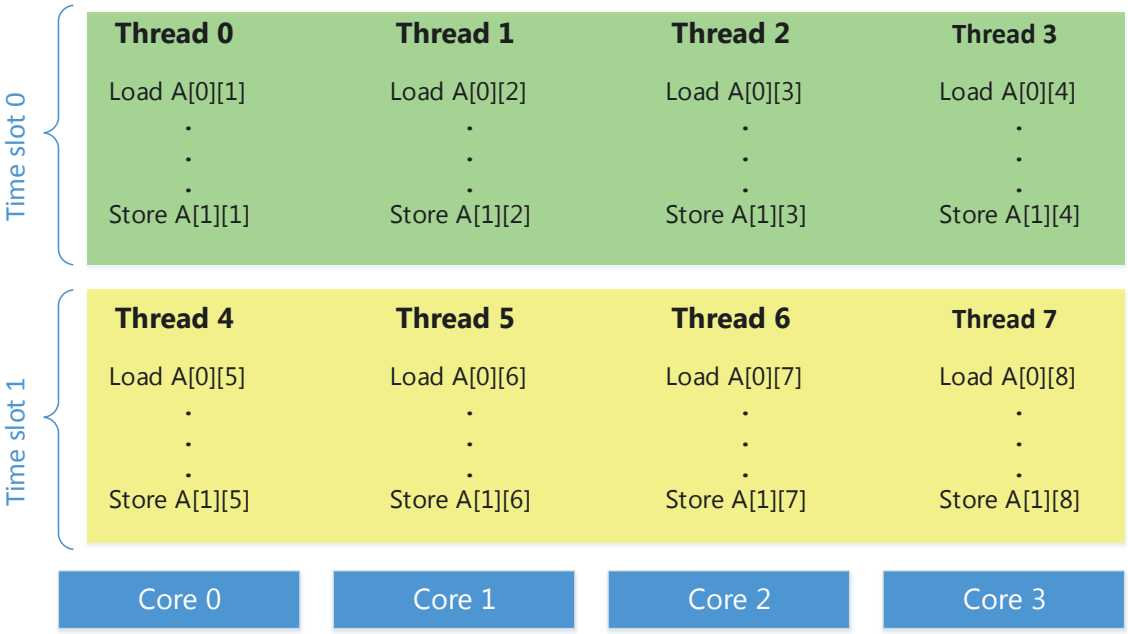



**Figure 3.1:** TLS: Safe scenario

of the TLS systems are detailed in the *Applications of TLS system* section. The current state of art is described in the *State of the art* section, where both hardware and software based TLS approaches are described. The *Limitations* section describes the current TLS system's limitations. The last section concludes the chapter.

## 3.2 Overview

This section details the working of a typical TLS system. The outline of such a system is shown in Figure 3.3. A typical TLS system begins by performing static analysis on the code. The static analysis information helps in detecting dependencies which can be statically resolved at compile time. Any code transformation that invalidates any of these static dependencies will surely violate the program semantics and thus should be avoided. Some systems also try to generate statically expressible conditionals for checking the existence of dependencies at runtime. The static conditions are computed and injected statically, and are evaluated dynamically to detect the presence of dependencies.

The next task is to identify the blocks of code, called execution units. Each exe-

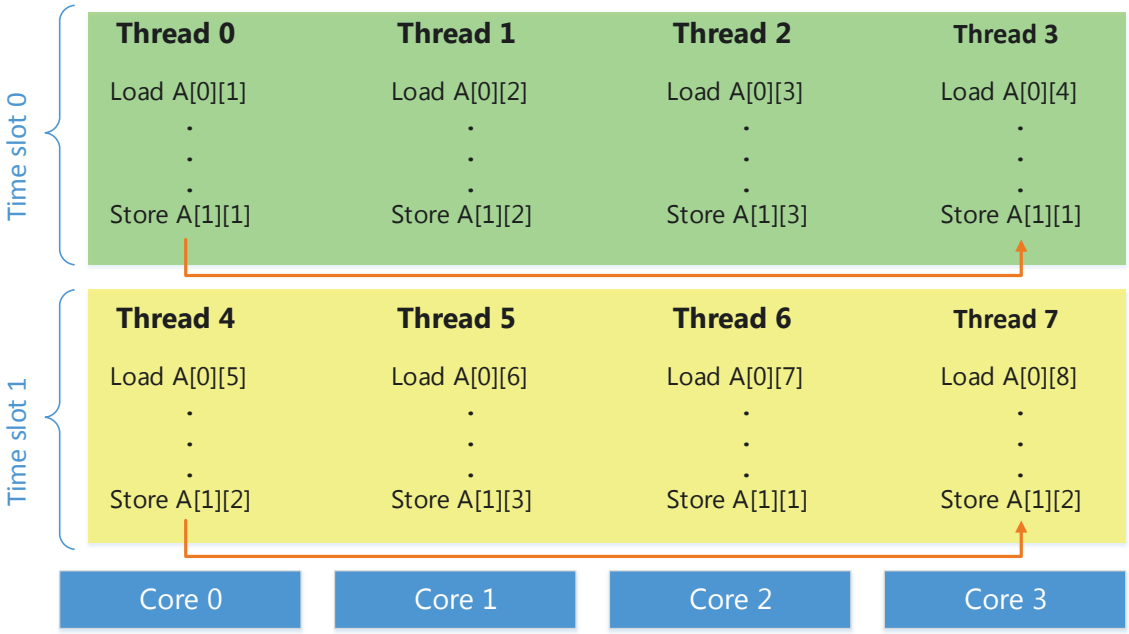



**Figure 3.2:** TLS: Unsafe scenario

cution unit represents a unit of computation. The instructions in an execution unit will be executed sequentially inside the same thread, while different execution units could be run in parallel. Execution units could be chunks of loops or a collection of tasks. Based on the execution units, some systems statically generate code snippets required for supporting parallelization. A partial order between the execution units can be constructed, so as to identify which of them can be run in parallel. However, this partial order needs to be augmented with runtime information for including dynamic dependencies.

At runtime, some systems simply launch optimistically the tasks in parallel, while others try to do runtime profiling of the code to determine the dynamic code characteristics, especially the dynamic dependencies. Based on the dynamic information and the static information, a transformation is predicted. In most TLS systems, the only transformation performed is parallelization, i.e. loops are cut into slices which are then executed in parallel. Next, the code representing the transformation has to be generated. This could be done either by selecting one of the statically generated code snippets, or by generating the code entirely at runtime, or by augmenting a statically

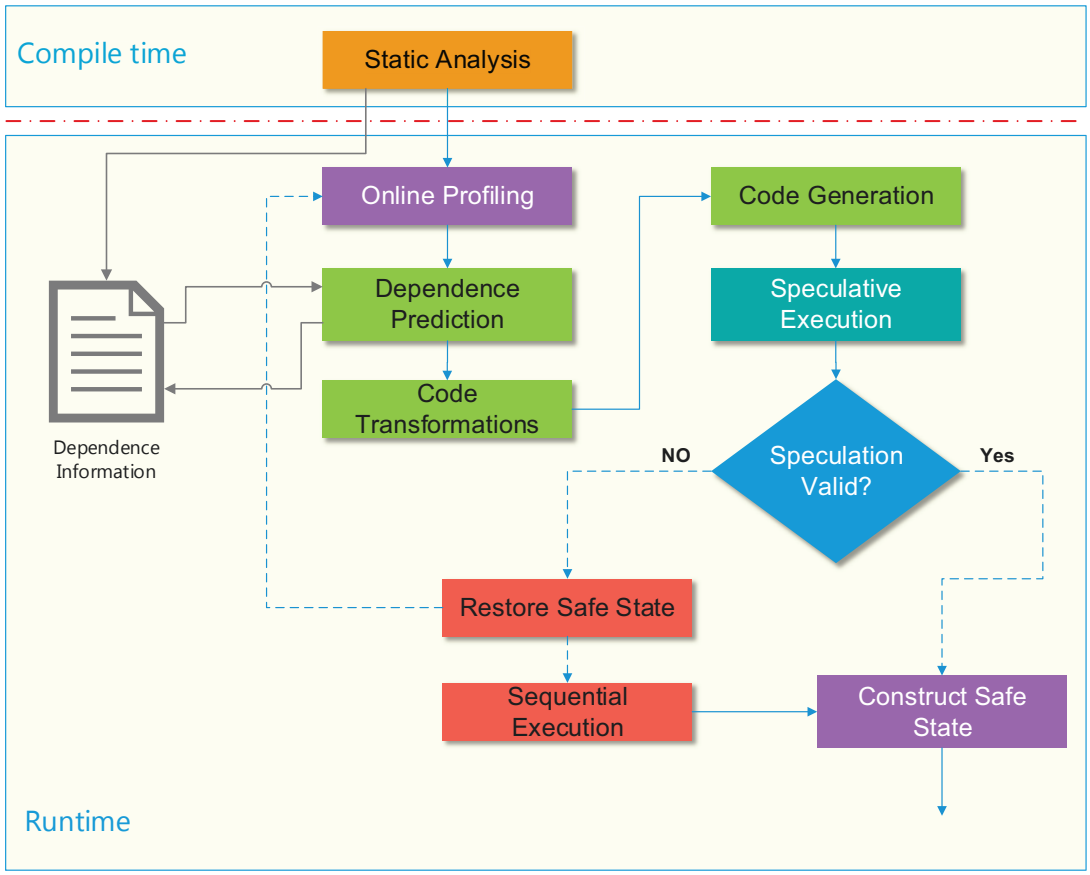



**Figure 3.3:** TLS: overview

generated snippet with runtime information. A monitoring system keeps tracks of the memory addresses accessed to detect any possible violation. In order to recover from a possible rollback, the speculative data is written to a temporary buffer. An alternative to temporary buffers is to use *in-place* updates, in which a backup of the original data is made before modifying the actual locations, and the computation is then performed on the actual locations. If the thread is successfully validated, the temporary buffer is merged to the global state or if the in-place update strategy is used, the backed up location is simply freed. On the other hand, if the thread is invalidated, the temporary buffer is discarded, or if the in-place update strategy is used, the backed up region is restored. The re-execution then begins with either a profiling phase, or with a new transformation, or with the original code.

In order for the TLS system to be efficient, the number of mis-speculations and thus

the number of rollbacks must be as small as possible. A crucial aspect affecting this is whether the *in-place* memory update strategy or whether the temporary buffering strategy was used. In-place updates typically perform well when there is a low number of rollbacks. In this scenario, as restores are rare, the only overhead is in backing up data, and the backup is discarded when the threads commit. However, if the number of rollbacks are higher, then in-place updates are costlier as the rate of both backups and restores are high. In this latter scenario, it is better to use a temporary buffer, as the only action required is to discard it. Thus, the overhead can be considerably reduced.

TLS systems are not necessarily high energy consumers. It can be argued that, in general, optimizations that improve performance, also reduce the energy consumption. As with performance, energy consumption is also adversely affected if there are frequent rollbacks. Energy consumption overhead in TLS system is mainly caused by the back-up and restore mechanisms and from executing the speculative threads themselves. However, even if the threads are discarded, not all the energy spent is wasted, as they could have brought some data into caches which could be used in the following phase(s), and also could have trained the branch predictor.

## 3.3 Applications of TLS system

Though mainly and most widely used for parallelization, TLS is also used to attain other objectives. In [55], TLS principles are applied for speculative synchronization of explicitly parallel programs. The goal is to remove speculatively placed synchronization constraints by executing the threads past the active barriers, the busy locks, and the unset flags. Every such construct is associated to one or more threads called *safe threads* which cannot be rollbacked, and thus ensuring progress. Some restrictions are present in the system, such as, there is no ordering between the threads and even in the presence of anti-dependence, a rollback is triggered. Dedicated hardware checks for cross-thread dependencies between safe threads and speculative threads. If a violation

is detected between a speculative thread and the safe thread, the speculative thread is rollbacked. If a violation is detected between two speculative threads, then one of them is rollbacked as there is no ordering between them.

In [56] TLS is used for profiling and debugging. The system is used to monitor memory locations and is capable also of breaking the code execution or rolling back when a program error occurs. TLS is used to execute the main thread and the monitoring threads in parallel, thus reducing the overhead of program monitoring. It is also used to buffer the states so that a safe state can be constructed for rolling back. In [57], TLS mechanisms are used for debugging data races in multi-threaded programs. Program states are constructed, while the program is executing. Unlike traditional TLS, only when the maximum epochs are reached or when the cache lines are full, the *commit* is invoked. This is because, in traditional TLS systems, the commit is invoked as soon as it can be guaranteed that the data dependencies are not violated. However, for bug detection, the safe states should be kept alive as far as possible. Among the threads, a partial order is constructed to determine the hierarchy for rolling back.

## 3.4 State of the art TLS systems

Existing TLS systems can be broadly classified into two categories (i) Hardware based and (ii) Software based.

### 3.4.1 Hardware based

Hardware based approaches use dedicated hardware to support TLS whereas software based approaches do not require any special support from the hardware. Typically, due to the additional hardware, Hardware Transactional Memory (HTM) systems have lower overhead and are less reliant on compiler based optimizations. Typically, they also have better power and energy efficiency. However, the publicly available HTMs are not fully evolved for general purpose programs.

Transactional Synchronization Extensions (TSE) [58] from *Intel* is an extension

to the x86 instruction set which enables TLS support in hardware. Hardware Lock Elision (HLE) allows easy conversion of lock based programs to transactional programs. Restricted Transactional Memory (RTM) in TSE supports instructions to begin, end or abort a transaction. In addition to this, an additional instruction is provided to check if the processor is executing a transactional region.

IBM's Blue Gene/Q chip [59] provides some hardware support for speculative states in memory. If required, it is the responsibility of the software to save and restore the state of the registers. Two modes of operations are supported. (i) Ordered mode: In this mode the final value of multiple writes to the same location is determined by the thread ordering. The younger threads can watch older threads; (ii) Unordered mode: In this mode, access to any speculative region by multiple transactions is considered invalid if at least one of them is a write. Thus, the former supports speculative execution while the latter supports transactional memory.

Rock [60] from *SUN* uses a checkpoint based architecture in-order to support speculation. In the Execute Ahead (EA) mode, a single core can run two application threads. When the system encounters a long latency instruction (Cache miss, micro DTLB miss etc.), it creates a check point and EA mode is activated. The destination register of the long latency instruction is marked as *not available* (NA) and is sent to the *Delayed execution Queue* (DQ). For any following instruction which has at least one source operand tagged as NA, the corresponding destination register is also marked as NA and the instruction is sent to DQ. The instructions which do not have NA tagged source operands are executed, and the results are saved as a speculative copy in a destination register and the NA tag on the destination register is cleared. When the long latency instruction completes its operation, the system switches from EA phase to replay phase, where the deferred instructions from DQ are read and executed. Simultaneous speculative threading (SST) offers a more aggressive form of speculation. In contrast to EA, in SST, two instruction queues are dedicated to a single software thread. When a long latency instruction is completed, one of them continues to re-

ceive new instructions from the fetch unit, while the other replays instructions from DQ. Thus, with SST, it is possible to start a new EA phase in parallel with the replay phase.

In [61] J.G. Steffan *et al.* propose a TLS system which uses a cache invalidation protocol to detect dependence violations. The cache lines are extended to add two new states, speculative-exclusive (SpE) and speculative-shared (SpS). The speculative modifications are not propagated to the rest of the memory hierarchy and the cache lines that are speculatively loaded are tracked to detect any violations. If a speculative cache line has to be replaced, it is treated as a violation and the epoch is re-executed. Three coherence messages are also added: read-exclusive-speculative, invalidation-speculative, and upgrade-request-speculative. They are appended with the epoch number in order to determine the logical order between itself and the requester. When the speculation fails, speculatively modified lines are invalidated, and speculatively loaded lines change their state to either exclusive or shared, depending on the speculative counterpart.

### 3.4.2 Software based

Software based approaches do not require any special hardware support for thread level speculation. They provide more flexibility than the hardware based approaches and are easier to modify. They can be easily integrated to existing systems and even languages. Software Transactional Memory (STM) systems have fewer intrinsic limitations when compared to the hardware as there is no limit of hardware structures such as caches. Certain hybrid approaches have also been proposed to take advantage of both software and hardware together.

In [62], J. Renua *et al.* study Chip Multiprocessor (CMP) TLS systems with the perspective of power and energy. The bulk overhead of energy consumption in a hardware TLS system steams from thread squashing, having hardware structures in the cache hierarchy for data versioning and dependence checking, extra traffic due to data

versioning and dependence checking, and finally the additional dynamic instructions induced by TLS. The proposed solution, *POSH*, is a compiler based approach, where the compiler automatically generates TLS code from sequential integer applications. POSH generates task modules from subroutines of any nesting level or from loop nests of any depth. It has two task spawning modes (i) Out of order: Select all subroutines and loops larger than a certain size, and (ii) In order: Heuristics about task size and overheads are used to select only one task. Once the tasks are identified, the system inserts spawn instructions and tries to hoist them up as far as possible without violating any data or control dependence. The framework also includes a profiler, which tries to identify the tasks that are not beneficial due to thread squashing. When a task is invalidated or committed, all the corresponding cache lines needs to be updated to mark it as killed or committed. For the latter a tag walk is required which is expensive in terms of energy. The proposed system only does a lazy tag walk, when the cache is near full, thus saving energy. Typical TLS system sends many messages to check the version ID, which itself consumes a lot of energy. In the proposed system, the cache lines are extended with two bits, *newest* and *oldest* and are updated when the cache line is loaded. Thus, if a write occurs on the newest line, there is no need to check other cache lines for exposed reads. Similarly, when the processor wants to displace a line with the oldest bit set, it has to check for older versions. A huge number of thread squashing will obviously hurt power and performance. In order to limit the number of squashing, each task is only allowed to restart a limited number of times after squashing. After reaching the limit, the task is only allowed to execute when it becomes non speculative. The system also proposes to use task aware pruning, which eliminates potential tasks that are not beneficial by using profiling.

In [63], J.G. Steffan *et al.* propose a combined software and hardware based approach in a framework called STAMPede. The system divides the entire work into small work units each associated with its own speculative thread. The relative order among the work units is expressed by assigning a time stamp to each work unit. The

works unit are committed in the order of original the sequential execution. In order to fork a new thread, a lightweight spawn instruction is implemented. Rather than making the software aware of the number of speculative contexts, the semantics of the spawn instruction allows it to fail. When a spawn fails, the speculative thread which tried to spawn, executes that particular work unit itself. Thus, the number of speculative threads can be controlled so that all the resources can be consumed, while preventing unmanageable explosion of the number of threads. Once the threads finish their speculative execution, they are stalled till they receive a token from its predecessor. Once the token is received, each thread commits or discards its operations and passes the token to the next thread. A special portion of the stack is used to forward and synchronize variables, especially the scalar variables so that some dependencies can be avoided. The consumer threads are responsible for detecting any violation. In order to support this, each producer reports the locations that it produces and the consumers track all the speculatively consumed locations. The hardware mechanism used is the same in [61].

In [64], Lawrence Rauchwerger *et al.* propose the concept of inspector/executor: a runtime technique to parallelize loops at runtime. At its core, there are two processes: (i) inspector and (ii) executor. The inspector monitors the code and collects the memory addresses that were accessed. From the accessed addresses, data dependencies are computed. The inspector is also able to identify privatizable and reduction variables. The scheduler then finds iteration wise dependence 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. This estimate is based on the previous runs of the program. An expected speedup is computed based on this. The inspector can be automatically generated, if the source loop can be distributed into a loop computing the addresses of the array and another loop which uses these addresses (i.e., when the address computation and data computation are not contained in the same strongly

connected component of the dependence graph). The data from the instrumentation phase is used to determine whether it is profitable to perform runtime parallelization. If possible and profitable, the scheduler then parallelize the code. The executor, then executes the code.

Using transactional memory with TLS has been studied in the work of Joao Barreto *et al.* [65]. Transactional Memory (TM) requires the programmer to explicitly fork the program into multiple threads. Though not trivial, this in theory can attain higher levels of parallelism. On the other hand, TLS systems may encounter multiple rollbacks if the data dependencies severely restrict the number of tasks that can be parallelized. The proposed method, TLSTM (STM + STLS), uses compile time analysis to break each transaction into multi threaded STM programs, which will run in parallel. If there are no conflicts, then the transaction can commit earlier.

A seminal work on speculative runtime parallelization of loops is the work of Lawrence Rauchwerger *et al.* [66]. The presented system tries to parallelize loops speculatively, by detecting if the loop has any cross iteration dependencies. It tries to avoid the use of an inspector executor phase because the address computation of an array may actually depend on the data computation. This would introduce a cycle between the data and address computations, which is essentially a cycle between the inspector and the executor. The loops are speculatively parallelized in a doall fashion, while the runtime checks for any violation. If a violation is detected, the loop is re-executed sequentially. The exceptions are treated similarly to mispredictions: The parallel execution is abandoned, the execution flags are cleared, followed by the rollback and re-execution. For each array whose dependencies cannot be resolved statically, special data structures called *shadow arrays* are created for indicating whether a location was read or written, and whether the read of a memory location in an iteration was preceded by a write in the same iteration. By using shadow array, it can be determined whether the loop can be executed in a doall manner and also whether privatization is required. Reduction is also performed by looking at certain known pat-

terns, but using runtime checks to confirm its validity. The work does not try to find a schedule which is valid for all cross iteration dependencies. This work is extended in [67] to add support for partial parallelism. The loop is transformed into a sequence of partially parallel loop. The privatization condition is replaced by a copy-in condition. Since no transformation other than parallelization is applied, in the case of a single mis-speculation, the computation is valid till the beginning of the mis-speculated iteration. The re-execution only needs to be started from the mis-speculated iteration. In the case of multiple mis-predictions, it is valid till the iteration which is lexicographical less than all the mis-speculated iterations. Hence, the results of the valid iterations are committed and the mis-predicted parts are re-executed.

In [68, 69], Alexandra Jimborean *et al.* present a STL framework called VMAD (a Virtual Machine for Advanced Dynamic Analysis of Programs). VMAD is a speculative loop optimization framework 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 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 versions, 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 addresses accessed and the enclosing iterator values. At runtime, the outermost loop is split into chunks and the instrumentation version is selected for profiling the first chunk. Based on the dynamic code behaviour, the system selects a polyhedral transformation and instantiates one of the statically generated code versions. 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 re-instrumentation. 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 a successor of VMAD. One among the major differences between APOLLO and VMAD is how the transformation is selected. In VMAD, the user proposes a set of code transformations statically. At runtime, when the dependencies are resolved, the first valid transformation from the user proposed list is selected. This approach has two major downsides. The first is that the system is not fully automatic: the user needs to be aware of the program structure to propose transformations and should enlist the transformations in the order of potential performance. The second downside is that, since the system is dynamic and is targeting dynamic codes, the exact code behaviour can only be known at runtime; hence proposing a set of transformations statically is hard and moreover, the number of transformations that needs to be listed may itself be high. In addition to this, selecting the first valid transformation strategy, is a sub optimal solution; there may exist another valid transformations in the proposed list of transformations 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 the dependencies which are identified during instrumentation, the dependence polyhedron is constructed, and is fed to Pluto to get a valid optimizing transformation. Thus, in our system, the process is fully automatic. Our system also utilizes JIT (Just In Time compilation) to fine tune the code versions attaining higher levels of performance. In terms of code compatibility, APOLLO is also capable of handling non linear memory accesses and loop bounds using a generic extension of the polyhedral model. Design and implementation details on Apollo are presented in Chapter 4.

Speculation support in OpenMP has been studied in [70]. The work proposes a software only TLS system to support parallel execution of codes that cannot be ana-

Listing 3.3: Example to illustrate limitations of TLS system (seidel 2d)

```
for (t = 0; t <= tsteps - 1; t++)
  for (i = 1; i <= n - 2; i++)
    for (j = 1; j <= n - 2; j++)
      A[i][j] =
      (A[i - 1][j - 1] + A[i - 1][j] + A[i - 1][j + 1]
       + A[i][j - 1] + A[i][j] + A[i][j + 1]
       + A[i + 1][j - 1] + A[i + 1][j] + A[i + 1][j + 1])
       / 9.0;
```

lyzed at compile time. The speculative region is marked by the user, using a special syntax which enlists the speculative variables. The reads and writes inside the speculative region are replaced by their corresponding speculative versions of loads and stores. The speculative load obtains the most up-to-date value and the speculative store writes the result in the version copy of the current processor and ensures that no subsequent thread uses an outdated value. A 'commit or discard' function is called once the threads have finished their execution and the data is committed or discarded accordingly. The scheduling method of OpenMP is also changed as the original schedulers present in OpenMP always assumes that the tasks will never fail. The new scheduler assigns a new chunk to each free thread. If the tread succeeds it will receive a new chunk, if not, the scheduler may assign the same thread to re-execute the failed chunk to improve locality and cache utilization.

## 3.5 Limitations of existing TLS systems

The current TLS system's performance is limited by the factors that are detailed in the following sections.

### 3.5.1 Missed parallelization opportunities

TLS systems mainly try to execute loop nests in parallel by simply cutting the outermost loop into slices and executing them in parallel. However, no transformation is

considered to make a loop parallel. To illustrate this, consider the example in Listing 3.3. Due to the dependencies from reads and writes on array $A$, none of the loops can be parallelized.

However, by applying loop skewing ((t, i, j) -> (t, t+i, 2t+i+j)), the inner loop can be parallelized. In some cases, the inner loop may be trivially parallel, but the outer loop is not parallel unless a transformation has been applied. Thus, the coarseness of the selected parallelization may be sub optimal, resulting in poor performance. Using a profiling mechanism, some existing systems can determine whether it is profitable to parallelize. However, if parallelization is not profitable, the only option for them is to run the sequential version. In addition to a transformation, dynamic codes may require the TLS system to support changing the transformation on the fly. The program, depending on the input data, may exhibit a different behaviour in different phases of the execution, and thus different optimizing transformations may be required.

### 3.5.2 Data locality is not considered

Efficient utilization of cache memories is a key factor for performance, especially since the gap between the processing power and memory speeds are increasing. Traditional TLS systems are focused only on parallelization and data locality is ignored. Our system considers both data locality and parallelization simultaneously by taking advantage of polyhedral transformations.

### 3.5.3 Backup

Since TLS systems are by definition speculative, hence they should have a mechanism to restore their state in case of a mis-speculation. This is done by backing up the memory. The different types of backup mechanisms available are briefly described in Section 3.1. The amount of backup required, and the backup mechanism itself, has a huge impact on the scalability of the TLS system. Versioning at each iteration can increase the memory requirement to exponential levels, which will have adverse effects

on the cache and memory bus. The empirical evidence suggests that backing up each memory location one by one is costlier than backing up a block of memory. However, most TLS systems perform their backup when a write happens on the fly, i.e. backing up one memory location at a time. The memory bus has to support the load of both the actual computations and the backup, resulting in an increased cache pressure. Our system uses the polyhedral model to overcome this limitation as well. By using the access functions, the speculative write region can be estimated and backed-up at once. More details on our system is given in Section 4.6.7. However, if the accesses do not fit the polyhedral model, then live backup is an essential requirement. Our system is able to handle this case by backing up the predicted write region as a block, and then only backing up the non predicted regions on the fly, thus minimizing the number of backup operations. More details on this extension are given in Section 5.8.

### 3.5.4 Verification

Verification is another important factor that affects scalability. Traditional verification mechanisms require the threads to communicate with each other. This requires synchronization and acts as a bottleneck. If no transformation other than parallelization is applied, the verifications can be limited to thread boundaries with reduced impact on performance. However, when a transformation is applied, each iteration needs to be verified, which increases the centralization load, even to extents where the parallel code is consequently almost running sequentially, if not worse. The typical structure of a verification system associated with a TLS system is shown in Figure 3.4.

An issue associated with this model of verification is the amount of data that needs to be communicated for verification. Like online backup, the huge data exchange required for verification is competing with the actual data access on the memory bus which leads to processor stalls. A suitable solution to this verification problem is to formulate the memory accesses into a model and then use this model for verification. The constructed model should facilitate a verification mechanism which will only use

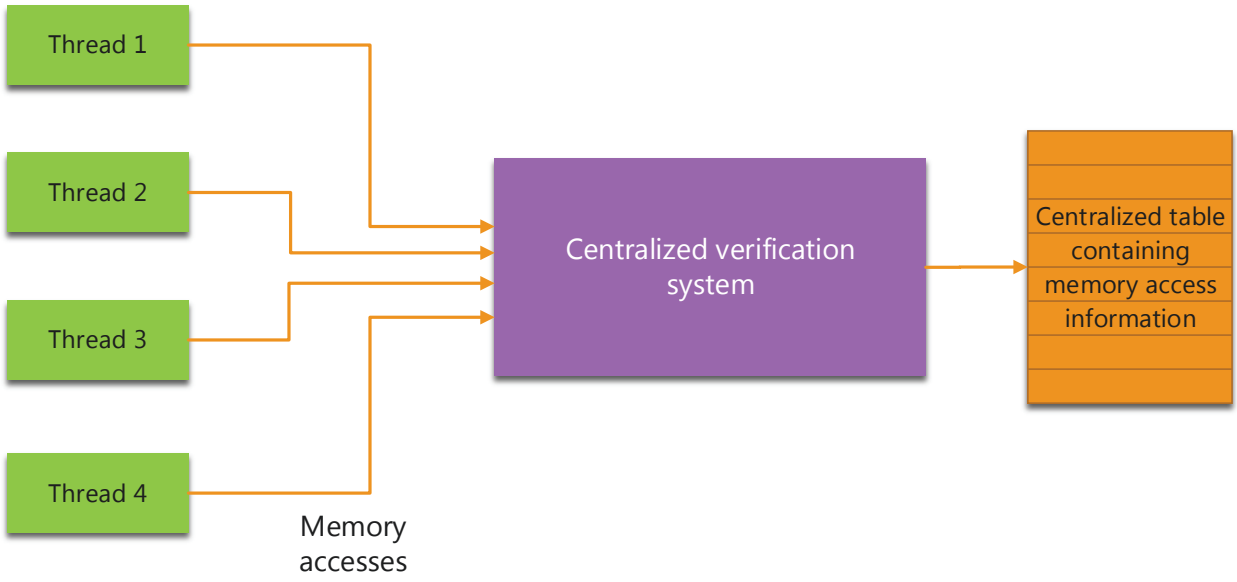



**Figure 3.4:** TLS: verification

locally available data (constants, local variables etc.) and should not require inter thread communication. Thus, the model should provide a function $verif\_S_i(\vec{v})$, for each memory access statement $S_i$, such that the parameter vector $\vec{v}$ only contain elements which does not require inter thread communication. In essence each thread verifies its own data by checking the adherence to the model without any central communication. Our system uses the polyhedral model for this purpose. By comparing the access functions, each thread can verify whether the reads and writes it is performing consistent with the speculation. However, if the accesses do not fit the polyhedral model, our system employs a centralized system requiring minimal communication. More details on this extension is given in Section 5.9.

Early detection of any dependence violation is crucial for performance. In general, the earlier the system detects a violation, the lesser the price it has to pay. For example, if the system can detect a possible violation even before launching the speculative code, then the cost associated with backup, running the speculative code and rollback can be avoided. For this purpose, our system employs a pipelined verification approach, which is capable of detecting possible violations as early as possible.

Thread Level Speculation itself is not sufficient for optimal performance. Most

TLS systems only try to parallelize loops. However, some codes may require some code transformations to be performed to make the loop parallel. In addition to this, most existing systems are not capable of performing advanced code transformations which aim both at parallelism and data locality. Centralized verification is another bottleneck. The problems can be solved by modeling the code behavior and then using the model for code optimization and verification. The polyhedral model is one widely used loop optimization model. The next chapter details our adaptation and usage of the polyhedral model at runtime, for efficient speculative loop parallelization and optimization, implemented in our Apollo framework.

*You can never know how far you can go until you attempt a great task.*

Lailah Gifty Akita

*If you are opportunist, then you are already one step closer to achieve your target!*

Priyanshu Joshi

*You can't cross the sea merely by standing and staring at the water.*

Rabindranath Tagore

# 4

# APOLLO

## 4.1 Introduction

Typical automatic optimizers perform static analysis of the code at compile time to resolve the dependencies and to compute a transformation which will optimize the code. However, not all codes can be statically analyzed. The presence of while loops, pointers, indirect accesses, etc. limit the scope of static optimizers. When a statically unresolvable memory access is encountered, these compilers have to opt for the conservative strategy and this results in suboptimal code. Dynamic optimizers can overcome this limitation by using dynamic analysis and runtime optimizations. They can monitor the runtime behavior of the program and can make use of the dynamically resolved values to apply a better code optimization. The drawback of this approach is the time overhead introduced at runtime, due to code profiling, code analysis and code generation. For such a system to be efficient, in terms of performance, it should be able to mask the overheads with the performance gained from optimization. Dynamic optimiz-

ers often rely on speculative techniques for achieving maximum performance. Thread Level Speculation (TLS) [55, 56, 57, 58, 59, 60, 61, 62, 63, 61, 64, 65, 66, 67, 69, 70] can be used to handle dynamic codes speculatively. However, in the traditional TLS systems, the lack of a prediction model limits the performance gains. A novel approach would be to club the polyhedral model with thread level speculation. The polyhedral model is traditionally only applied to static codes (SCoP) Our system, Apollo, combine the TLS strategy and the polyhedral model to optimize dynamic codes.

## 4.2 Overview of Apollo

Apollo stand for *Automatic speculative POLyhedral Loop Optimizer*. It is an automatic and dynamic loop optimization framework based on the polyhedral model. Apollo targets loop nests of arbitrary depth, possibly imperfect, and which cannot be analyzed statically at compile time. Apollo's core concept is built around the fact that, even though statically not analyzable as being linear, many programs exhibit a linear behavior at runtime, at least in some phases. We call these codes DCoP, as defined in Chapter 2. The system targets to optimize these codes efficiently by using the polyhedral model on the fly. Currently, we only target 'C' and 'C++' codes. However, all the processing is done in LLVM IR and hence, extending Apollo to any other language is as simple as extending the corresponding LLVM language parser to recognize our pragma, which is used to specify the target loop nests.

The global overview of Apollo's architecture is shown in Figure 4.1. The working of Apollo is summarized below.

The user marks the target loop nest(s) of interest with a pragma (#pragma apollo dcop). The pragma attributed code is passed to the Apollo compiler (apollocc for 'C' programs and apollo++ for 'C++' programs). At compile-time, Apollo's static phase: (1) precisely analyses memory instructions that can be disambiguated at compile-time; (2) generates an instrumented version to track the memory accesses that cannot be

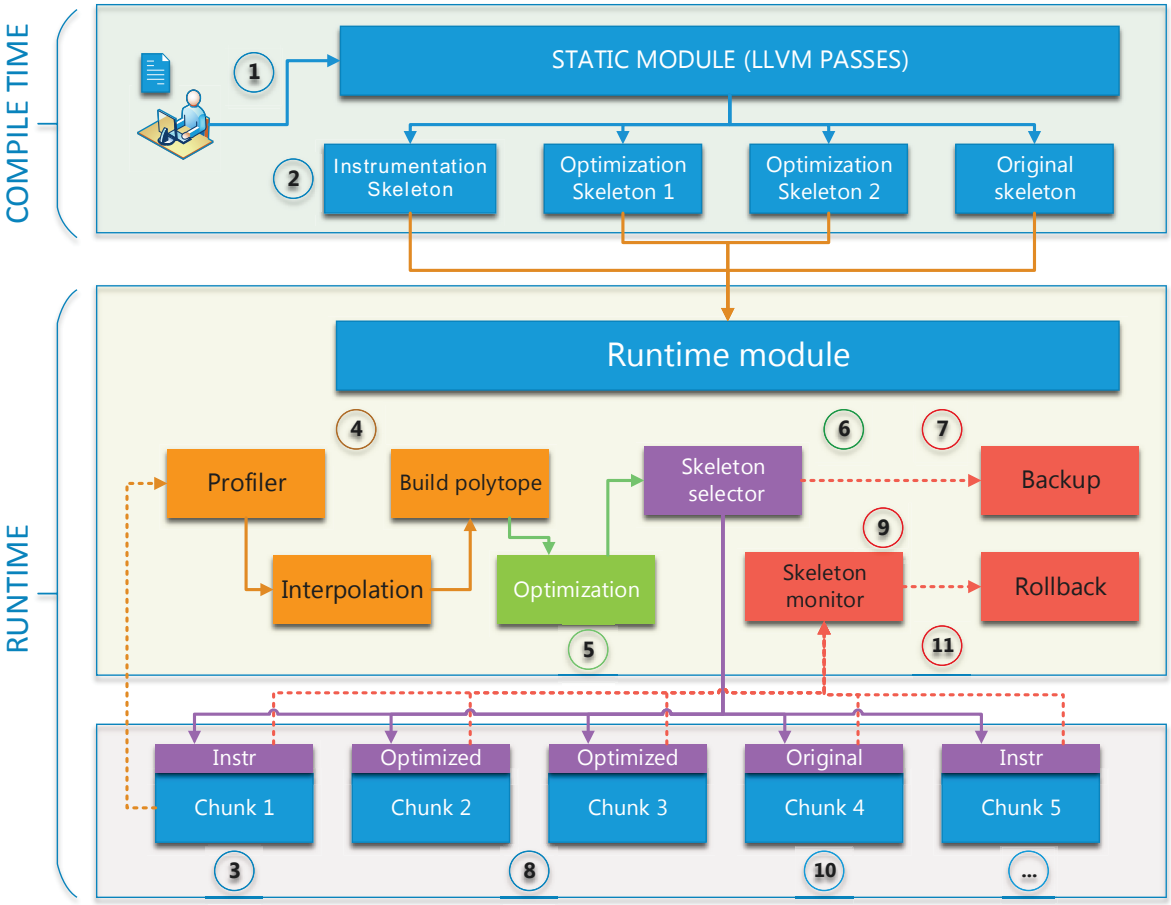



**Figure 4.1:** Global overview of apollo

disambiguated at compile-time. The instrumented version will run on a sample of the outermost loop iterations and the information acquired dynamically will be used to build a prediction model of non statically analyzable memory accesses; (3) generates parallel code skeletons which are incomplete versions of the original loop nest and require runtime instantiation to generate the final code. Each instantiation represents a new optimization, therefore the code skeletons can be seen as highly generic templates that support a large set of optimizing and parallelizing transformations. Additionally, the skeletons embed the required support for speculations (e.g. verification and recovery code). These stages are depicted in the region labeled "compile time" in Figure 4.1.

At runtime, Apollo's dynamic phase: (1) runs the instrumented version on a sample

of consecutive outermost loop iterations; (2) builds a linear prediction model for the loop bounds and memory accesses; (3) computes dependencies between these accesses; (4) instantiates a code skeleton, and generates an optimized, parallel version of the original sequential code, semantically correct with respect to the prediction model; (5) during the execution of the multi-threaded code, each thread verifies independently whether the prediction still holds. If not, a rollback is initiated and the system attempts to build a new prediction model. These stages are depicted in the region labeled "compile time" in Figure 4.1.

The rest of the chapter is organized as follows. Section 4.3 describes the basic requirements to apply the polyhedral optimizations dynamically and speculatively. Section 4.4 describes how Apollo identifies the loop nests to be optimized. The Section 4.5 contains detailed information on the static part of Apollo. This includes how the target loop nest is extracted (Section 4.5.1), the static analysis phase (Section 4.5.2) and the code versioning (Section 4.5.4). The runtime part is detailed in Section 4.6. This includes the working of instrumentation (Section 4.6.1), interpolation of the linear functions (Section 4.6.2), dependence analysis (Section 4.6.3 and 4.6.4) and the code scheduling and the skeleton instantiation (Section 4.6.5). The chapter concludes with the summary of Apollo.

Apollo divides the outermost loop into small slices called *chunks*. Each chunk represents an execution unit of Apollo. Code instrumentation, optimization *etc.* is applied on each chunk(see Section 4.5.4). The first chunk is always used to instrument the code. The rest of the chunks may be assigned to run the optimized code versions, the original code version, or the instrumented code version, depending on the program behavior. The size of each chunk is determined by the runtime. Each chunk boundary marks a consistent state of the program. If a speculatively executed chunk fails, the re-execution needs to begin from the end of last committed chunk. The chunking mechanism is depicted in 4.2.

Without the chunking system, whenever a mis-speculation occurs, the re-execution

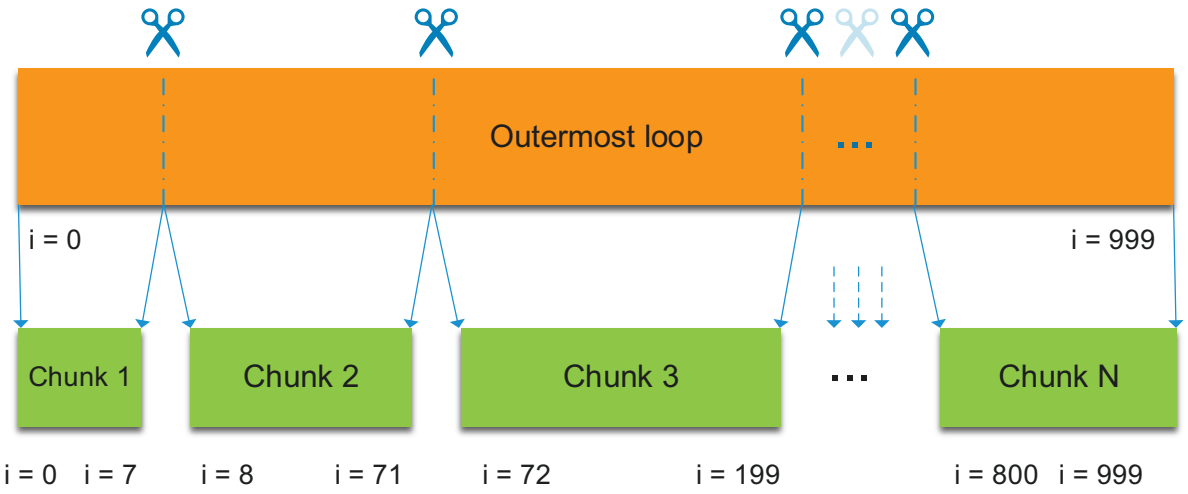



**Figure 4.2:** Chunking of outer most loop

should begin from the very beginning of the loop nest. Chunking also helps in capturing different program phases. The optimizations are applied to each chunk, thus, when the program behavior changes, the set of optimizations to be applied can be changed accordingly.

## 4.3 Applying polyhedral optimizations dynamically

For the loop nests that cannot be analyzed statically but which exhibit a polyhedral behavior at runtime, our strategy for enabling the polyhedral model at runtime relies on speculation coupled with runtime verification. For a code to be optimized using the polyhedral model, the following conditions should be met:

- All the memory accesses can be represented by affine functions of enclosing loop iterators, loop parameters and constants.
- The loop bounds can be represented by affine functions of enclosing loop iterators, loop parameters and constants.
- The control flow of the code can be determined by affine functions of enclosing loop iterators, loop parameters and constants.

Thus, for dynamic optimizations using the polyhedral model, the system should dynamically ensure that all the above conditions are met. In general static analysis fails to handle codes, where indirect accesses, pointers or while loops are present. At runtime, for the latter class of codes, even though many variables are resolved, it cannot be guaranteed that the polyhedral model holds without executing the code, and hence it requires speculative execution. A dynamic system thus should:

1. Construct affine functions for loop bounds and memory accesses. (See subsection 4.6.2)
2. Perform dependence analysis on the fly. (See subsection 4.6.3)
3. Propose a polyhedral transformation. (See subsection 4.6.5)
4. Generate the code representing the transformation. (See subsection 4.6.5)
5. Provide a mechanism to recover the program state in case of a mis-speculation (See subsection 4.6.7)
6. In order to validate the speculative optimization ensure that the predicted code behavior is the same as the observed code behavior. This can be done by verifying the predicted linear functions. (See subsection 4.6.10)

In addition to the above, dynamic optimizers should handle scalar variables which carry cross-iteration dependencies. Predicting the scalar values can help in breaking some dependencies. To illustrate this consider a simple loop nest in Listing 4.1 and the corresponding LLVM IR in Listing 4.2. Consider the update of *ptr* in line number 3 and 9 of Listing 4.1. The corresponding update, represented by a phi node (%*ptr*.011) is highlighted by red color in Listing 4.2. %*ptr*.011 depends on %4 which is the load instruction updating the value of *ptr*, inducing a dependence carried by the inner while loop. However, if this scalar can be represented by a linear function parametrized only by enclosing loop iterators and constants, this linear function can then be used to

Listing 4.1: Example to illustrate the handling of scalars

```
1  for(i = 0; i < size; i++)
2  {
3      struct NODE *ptr = head;
4      int key = key_list[i];
5      while(ptr)
6      {
7          if(ptr >val == key)
8              found[i]++;
9          ptr = ptr >next;
10     }
11 }
```

initialize %*ptr*.011, instead of updating it from %4. The latter breaks the dependence carried by the while loop. Another common case where Apollo can effectively break the dependence by scalar prediction is the adaptation of the loop iterators. The update of the 'for loop' iterator, *i* (*%indvars.iv*), is highlighted by blue color in Listing 4.2. *%indvars.iv* depends on *%indvars.iv.next*, introducing a dependence carried by the outermost loop. Just like the former, Apollo can effectively predict the value of this scalar, thus breaking the dependence.

The details on how the scalar prediction works is presented in Section 4.6.2.

## 4.4 Identifying the target loop nest(s)

Apollo can identify the loop nests to optimize in two ways. The user can mark the loop nests of interest in the source code with the following pragma '*#pragma apollo dcop*'. The dcop stands for dynamic control part (emphasizing that we are targeting dynamic codes). The Clang parser is modified to recognize our pragma and to add metadata to the statements in the loop for later processing. If the user does not wish to mark the loop nests, then Apollo can automatically identify all the loops and apply the metadata to all loops.

Listing 4.2: Example to illustrate the handling of scalars (LLVM IR)

```
for.body:
  %indvars.iv = phi i64
    [0, %for.body.lr.ph ], [ %indvars.iv.next, %for.inc ]
  %arrayidx = getelementptr inbounds i32* %key_list,
                  i64 %indvars.iv
  %1 = load i32* %arrayidx, align 4, tbaa 1
  br i1 %tobool10, label %for.inc, label %while.body.lr.ph
while.body.lr.ph:
  %arrayidx3 = getelementptr inbounds i32* %found,
                  i64 %indvars.iv
  br label %while.body
while.body:
  %ptr.011 = phi %struct.NODE*
    [%head, %while.body.lr.ph], [ %4, %if.end ]
  %val = getelementptr inbounds %struct.NODE* %ptr.011,
                  i64 0, i32 0
  %2 = load i32* %val, align 4, tbaa 5
  %cmp1 = icmp eq i32 %2, %1
  br i1 %cmp1, label %if.then, label %if.end
if.then:
  %3 = load i32* %arrayidx3, align 4, tbaa 1
  %inc = add nsw i32 %3, 1
  store i32 %inc, i32* %arrayidx3, align 4, tbaa 1
  br label %if.end
if.end:
  %next = getelementptr inbounds %struct.NODE* %ptr.011,
                  i64 0, i32 1
  %4 = load %struct.NODE** %next, align 8, tbaa 8
  %tobool = icmp eq %struct.NODE* %4, null
  br i1 %tobool, label %for.inc.loopexit, label %while.body
for.inc.loopexit:
  br label %for.inc
for.inc:
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %lftr.wideiv = trunc i64 %indvars.iv to i32
  %exitcond = icmp eq i32 %lftr.wideiv, %0
  br i1 %exitcond, label %for.end.loopexit, label %for.body
for.end.loopexit:
  br label %for.end
for.end:
  ret void
```

## 4.5 Static module

The main objective of the static module is to prepare the user code for runtime speculation. The static module is built as set of LLVM passes and operates on the LLVM IR (LLVM Intermediate Representation). LLVM IR is a SSA (Single Static Assignment form) based intermediate code representation. It is specially designed for type safety, low level operations and flexibility. LLVM IR has 3 representations. First is the human readable form, where the IR is represented in 'text' form. This form is particularly useful for debugging. The second is 'on-disk' binary representation, where the LLVM IR is dumped in binary form on the disk. This form is particularly useful to inspect actions performed by each pass and can be used by the JIT compiler. The third form is 'in-memory' representation, in which the bit code is maintained in the memory itself. This form is mainly used to pass the LLVM IR between the LLVM passes so that the overhead of dumping to disk and loading can be avoided. The following sections explain the functions of the static module in detail.

### 4.5.1 Extracting the loop nest

The static module begins its operations by extracting each loop nest marked by the 'apollo dcop' pragma into its own function. The target loops are identified by looking for the Apollo specific metadata of the statements, which would have been appended by the modified clang parser. Extracting each marked loop nest to its own function helps in keeping tight control over the loops that will be optimized by Apollo. For example, consider a target loop nest which is nested inside another loop. Assume the user does not wish to optimize the outer loop (due to IO operations, or low iteration count etc.). If we run LLVM optimization passes, without extracting the loop to a function, some statements may be hoisted or moved from one loop to another. Since the target loop nest is identified by checking for the metadata of the statements, hoisting a statement annotated with Apollo metadata to a loop which was not selected for optimization,

Listing 4.3: Example to illustrate Virtual iterators

```
while(ptr)
{
    neigh_ptr = ptr >neighbours;
    while(neigh_ptr)
    {
          S1(vi,vj);
          S2(vi,vj);
          neigh_ptr = neigh_ptr >next;
    }

    ptr = ptr >next;
}
```

would result in considering the non selected loop for optimization.

### 4.5.2 Static analysis

The static module runs the standard LLVM analysis passes such as Alias Analysis (AA) and Scalar Evolution (scev) to collect the alias information and statically determinable linear functions for memory accesses, scalars, and loop bounds. The collected information is injected to the binary itself and is later used by the runtime. The static information helps in reducing the required instrumentation and there by reduces the runtime overhead. For example, if the memory access function of a statement is statically determinable, the runtime instrumentation for that statement can be avoided, only the base address needs to be collected at runtime. It also helps in determining some program characteristics such as the total number of loops, the number of statements inside each loop etc. which are later used by the runtime.

### 4.5.3 Virtual iterators

While loops and do while loops do not have the notion of an iterator in their syntax. However, Apollo relies on the loop iterators for almost everything such as constructing the linear functions, to compute dependencies, to compute transformation and also

Listing 4.4: Example to illustrate Virtual iterators

```
for(vi = 0; vi < UPPER_1 && ptr ; i++)
{
    for(vj = 0; vj < UPPER_2 && neigh_ptr; vj++)
    {
          S1(vi,vj, neigh_ptr);
          S2(vi,vj, neigh_ptr);
          //may be replaced by scalar prediction
          neigh_ptr = neigh_ptr >next;
    }
    //may be replaced by scalar prediction
    ptr=ptr >next;
}
```

to perform verification. In order to overcome this, for each loop inside the loop nest, an additional variable, called 'virtual iterator' is inserted. These virtual iterators act as loop iterators, essentially converting all types of loops to 'for loops'. The virtual iterators always start at '0' and are incremented by '1', or in other words, they are normalized. The reasoning about the access functions and hence dependencies are based on these virtual iterators. As an example consider the code in Listing 4.3. The code has two *while loops*. The 'C' version of the code skeleton, after Apollo has inserted the virtual iterator is shown in Listing 4.4. Variables $vi$ and $vj$ represent the virtual iterator for the outer loop and the inner loop respectively.

### 4.5.4 Code versioning and Skeletons

The most important function of the static module is to create different code versions of the input code, called skeletons. Each skeleton supports a particular combination of loop transformations. In addition to this, an instrumentation skeleton is built for profiling the code. Skeletons can be seen as parametrized codes where different instantiation of their parameters result in different code transformations.

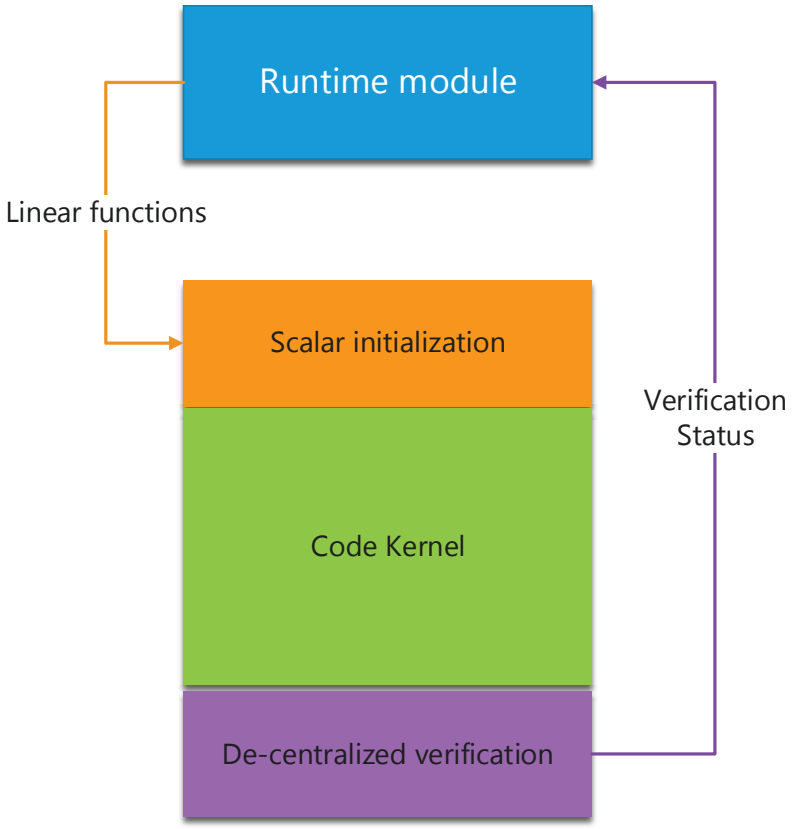



**Figure 4.3:** General structure of skeleton

### Skeletons representing code transformations: Optimization skeletons

Optimization skeletons are code templates, each of which represent a set of code transformations. Each such skeleton supports a certain combination of loop transformations such as skewing, interchange etc. These skeletons are incomplete frameworks which should be instantiated at runtime with parameters representing the actual transformation. Note that when the skeletons are generated, the transformation itself is not fixed; the exact transformation is later determined by the runtime during the program execution.

These skeletons are converted to binary format during compilation. However, the LLVM IR form of the optimization skeletons are kept inside the binary, producing a fat binary. The embedded LLVM IR is later to be used by JIT for online optimization. Our choice of fat binary instead of LLVM IR disk binary representation, is based on

the empirical fact that in Apollo, embedding the LLVM IR gives better performance.

Each optimization skeleton has three functions: the first is to apply the transformation given by the runtime; the second is to perform the original computations on the transformed iteration domain; and the third is to verify the speculation.

The general structure of the optimization skeleton is represented in the Figure 4.3. They can be abstracted to three main parts as follows:

**Scalar Initialization**: Apollo tries to predict the value of each scalar. The first section of the skeleton corresponds to the initialization of the scalar values from the prediction. This is essentially required to break some dependencies, especially if the code contains pointers.

**Code Kernel**: This section essentially contains the actual computations performed by the original code. The code kernel is governed by virtual iterators. Since the virtual iterators are iterating over the transformed space, the code kernel will also be executed in the transformed space. All the memory accesses are performed based on the virtual iterators. The scalars, for which the linear functions were constructed during instrumentation, are replaced by the corresponding computations in the scalar initialization section.

**De-centralized verification**: For each statically unresolvable memory access, loop bound or scalar, a linear prediction function is built during the instrumentation phase. The de-centralized verification, ensures that the program behavior observed during the instrumentation holds during the optimized run. Thus, the verification system ensures that the predicted linear functions are valid, which in turn ensures that the transformation itself is valid. In order to attain better performance, the verification should verify the linear functions without central communication. Inter thread communication requires synchronization which induces huge overhead and will act as a bottleneck. Apollo's de-centralized verification system verifies the memory accesses, scalars and loop bounds by checking the equality between the predicted function and the actual value. If the verification succeeds, the execution continues, otherwise, the

thread signals a rollback by setting the rollback flag.

Just like the transformation, the linear functions for verifying the memory accesses, the loop bounds and the scalars are passed from the runtime to the skeletons. Generic linear functions are placed when the skeletons are constructed. It is the coefficient of these generic linear functions which are patched during the runtime.

In addition to the above mechanisms, each optimized skeleton contains specialized mechanisms to communicate with the runtime (for eg. to get the loop bounds using Fourier Motzkin elimination). Even though the skeletons contain mechanisms to execute the user code in the transformed space, they do not incorporate mechanisms to parallelize the code. Parallelization is controlled and managed solely by the runtime as explained in Section 4.6.8.

#### Skeletons representing the original code

This skeleton essentially represents the original code kernel without any code transformation. The only modification done to the original code is to allow chunked execution i.e. the runtime is able to instruct this skeleton to run from a specific outermost loop iteration to another specific outermost loop iteration (see section 4.6.10). Original skeleton is mainly used to execute the mis-predicted regions, or in cases where the polyhedral model itself could not be constructed. If the original skeleton is selected for execution due to a rollback, it is followed by an instrumented skeleton, hoping to capture another polyhedral phase of the program.

#### Instrumentation skeleton

The instrumentation skeleton is used for code profiling and is the first skeleton launched by the runtime. The instrumentation skeleton is injected with special code snippets for code profiling and communication with the runtime. The injected code serves to report the memory addresses accessed, the scalar values and loop exit values along with the enclosing loop virtual iterator values. Several techniques are applied to reduce the

Listing 4.5: Example to illustrate instrumentation

```
1  while(ptr)
2  {
3      Node *neighbours = ptr->neighbours;
4      while(neighbours)
5      {
6        neighbours->value += ptr->value;
7        neighbours = neighbours->next;
8      }
9      ptr = ptr->next;
10 }
```

overhead of instrumentation. As prescribed in sub section 4.5.2, one of them is to use static analysis, to resolve the memory access function, loop bound function or scalar access functions; if found statically, these statements are not instrumented. One of the most important overhead reduction mechanisms is to instrument by samples, i.e. to instrument only parts of each loop rather than the full loop.

For better understanding, consider the example in Listing 4.5. We have to resolve the read memory accesses in line numbers 3, 6, 7 and 9 along with the write memory access on line number 6. Assume that the write accesses on line numbers 3, 7 and 9 are linear and thus are treated as scalars. In addition to the memory accesses, we have to resolve the loop bound for the inner *while loop*. Note that the outer *while loop* is chunked, and hence its not necessary to find its loop bound; however if statically determinable, it can potentially avoid one rollback of the last chunk (see subsection 4.6.12). Each instrumented access reports to the runtime, the instrumented value along with the enclosing virtual iterator values. A *C* style version of instrumentation for the code in Listing 4.5 is shown in Listing 4.6. The actual instrumentation skeleton is in LLVM IR form, which is cumbersome to read.

Only a sample of each loop is selected for profiling. The rest of the iterations inside the instrumentation skeleton simply executes in the original order without any profiling. The general structure of partial instrumentation of the loops is represented

Listing 4.6: Example to illustrate instrumentation

```
for(vi = chunk_lower; vi!= chunk_upper; vi++) // loop id = 0
{
    apollo_run_read_mem(
          &ptr >neighbours,      // memory address
          vi);                   // enclosing iterators'
    Node *neighbours = ptr >neighbours;
    vj = 0;
    while(neighbours)            // loop id = 1
    {
      apollo_run_read_mem(
            &ptr >value,         // memory address
            vi, vj);             // enclosing iterators'
      apollo_run_write_mem(
            &neighbours >value, // memory address
            vi, vj);             // enclosing iterators'
      neighbours >value += ptr >value;
      apollo_run_read_mem(
            &neighbours >next,   // memory address
            vi, vj);             // enclosing iterators'
      neighbours = neighbours >next;
       vj++;
    }
    apollo_run_reg_bound( 1   ,  // loop id
                          vj  ,  // loop exit value
                          vi  ); // enclosing iterators'
    apollo_run_read_mem(
          &ptr >next,            // memory address
          vi);                   // enclosing iterators'
    ptr = ptr >next;
}
```

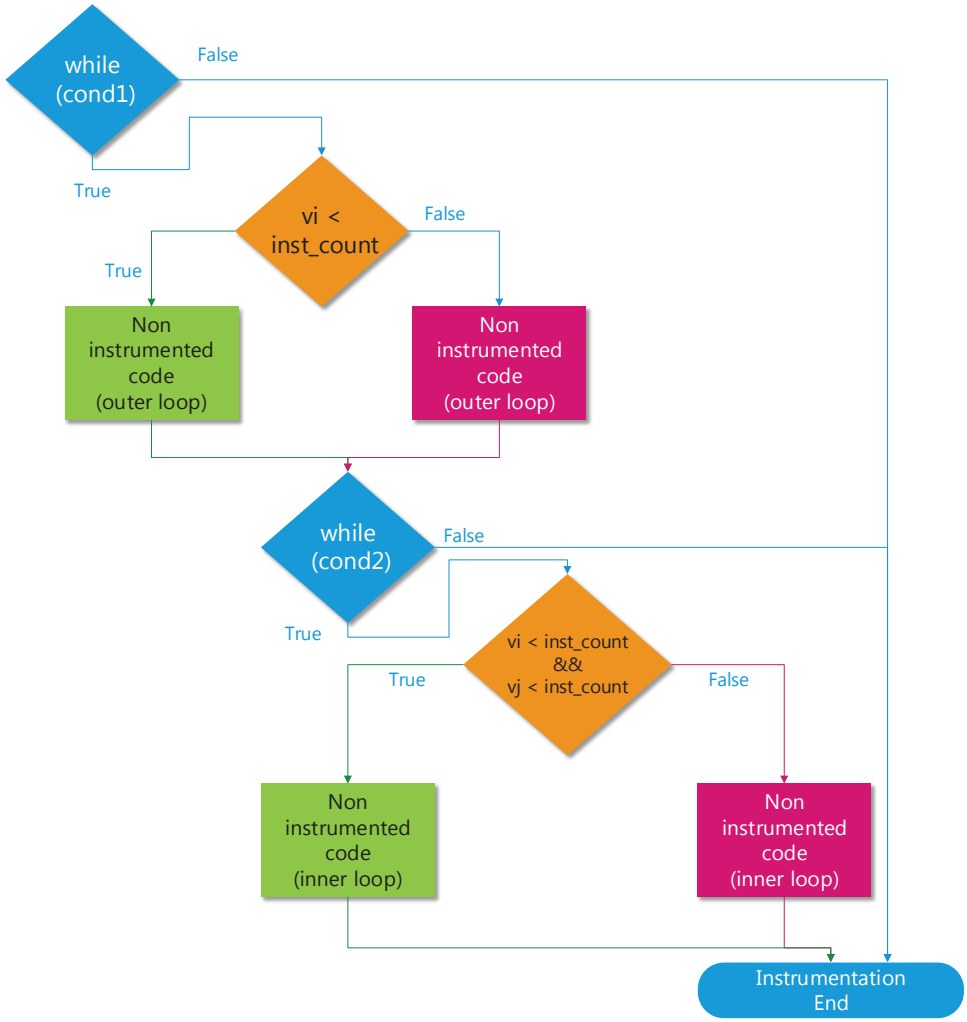



**Figure 4.4:** Partial instrumentation of loops

in Figure 4.4. As shown in the figure, each statement is only instrumented if all of its enclosing loop iterators are below a certain limit. This limit can be varied by the runtime. Though Apollo focuses on loop optimization, it can also be used as a profiler, even for non polyhedral codes. The limit can be changed to control the amount of instrumentation. Lower limit would give less profiling data but faster execution and vice-versa.

## 4.6 Runtime module

Apollo's runtime module controls the entire code execution and orchestration. It is written in *C++* and is attached to the binary as a Shared Object (SO). The major

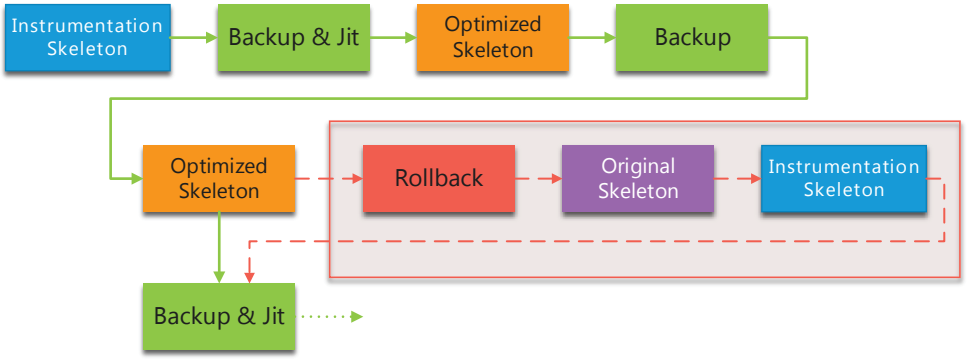



**Figure 4.5:** Execution flow

functions of the runtime include the selection of the code skeletons, the building of the linear functions and dependence model, the invocation of the scheduler to compute the transformation, facilitating parallelization, the backup and rollback mechanisms, and the overall monitoring and management of the system. The entire execution flow of the runtime is sketched in Figure 4.5.

### 4.6.1 Code profiling

The life cycle of Apollo begins when the control reaches the target loop nest. The runtime initially selects and instantiates the instrumentation skeleton and transfers control to it.

During the instrumentation, i.e. when the instrumentation skeleton is running, memory accesses, scalar values and loop bounds and their enclosing loop iterators are reported to the runtime via function calls. The runtime simply buffers this data into specialized data structures for later processing.

### 4.6.2 Building the linear functions

Once the instrumentation skeleton finishes its execution, the runtime tries to build the linear functions for (i) memory accesses, (ii) scalars and (iii) loop bounds. Computing the linear functions for each of them follows the same principle, and hence only computing the linear functions for the memory accesses is explained.

A linear function corresponding to a memory access statement $S_i$ will have $N$ variables, where $N$ is the loop depth. To solve a linear equation with $N$ unknowns, $N$ linearly independent equations are required. By plugging the values obtained during instrumentation, we can construct linear equations. The set of linear equations thus obtained, can be filtered to select $N$ linearly independent equations. They can then be solved to obtain the coefficients of the linear functions using the matrix method. We need to represent the linear functions in the following form

$$A \leq X = B$$

Where $A$ corresponds to the virtual loop iterator values and the constant 1, $B$ corresponds to the observed memory addresses and $X$ corresponds to the coefficient of the iterators. $X$ can be found by

$$X = A^{-1} \leq B$$

The whole process is illustrated in Figure 4.6. If building the linear function for the scalars fails, they are treated as memory accesses, so that dependence induced by them are appropriately handled. This behavior is usually observed in the case of reduction computations, where the values cannot be predicted and thus should be treated as memory accesses.

### 4.6.3 Computing the dependencies

A dependence occurs when at least two statements access the same memory location, and at least one of them performs a write operation. In Apollo, the statements which accessed the same memory location can be identified by scanning the instrumentation data [71]. The dependence type and conditions are then inferred from it. Based on the order of read and write statements, the dependencies are classified into Read After Write (RAW), Write After Read (WAR) and Write after Write (WAW).

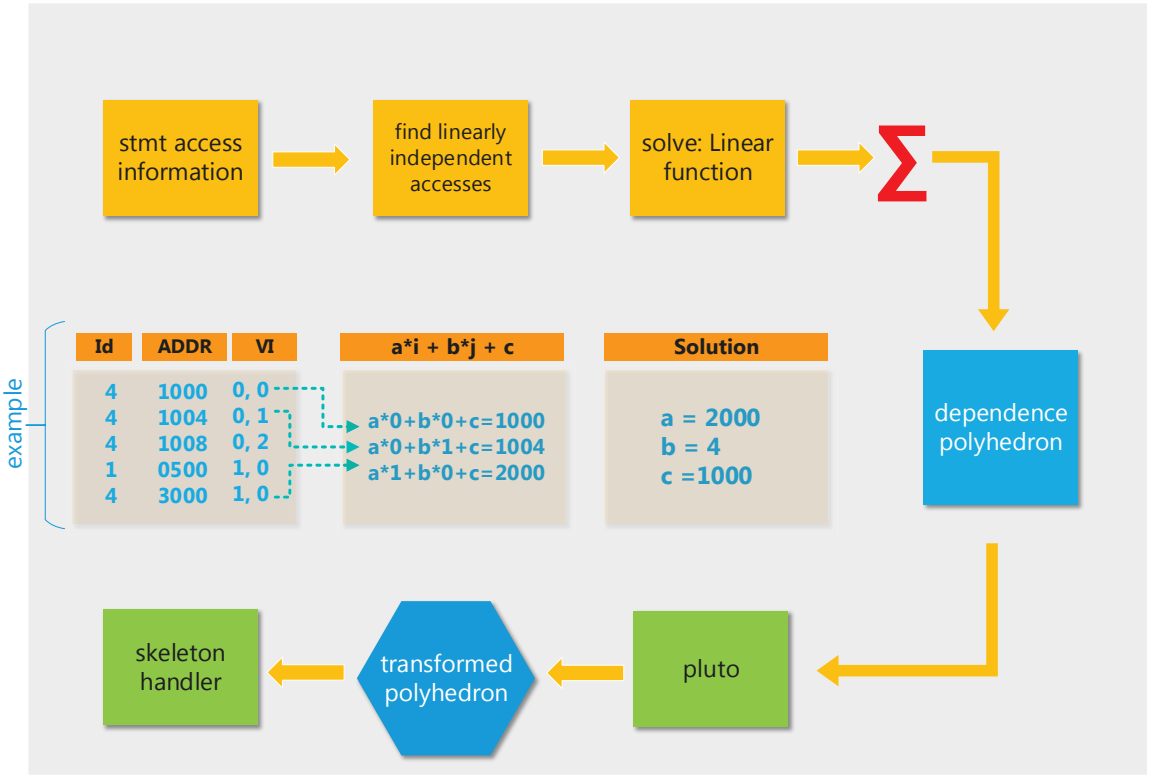



**Figure 4.6:** Building linear functions

### 4.6.4 Computing the dependence polyhedron

Once the dependencies are identified, the dependence polyhedron can be constructed, by using the dependence conditions and the domain of the statements. Since Apollo uses Pluto as the scheduler, the dependence polyhedron is directly populated into Pluto's structures. Note that this step is decoupled from computing the dependencies itself; i.e. the dependence detection is separated from populating the dependence which enables Apollo to use any polyhedral scheduler. The domain part of the dependence polyhedron is populated by the equations corresponding to the range of virtual iterators associated with both the source and the target statements. An equation which expresses that both the source and the target statements are accessing the same memory location is injected. This is done by equating the access function of source and target statements with the help of an additional parameter. Then the lexicographical constraints are populated. The added parameter is then removed through Fourier

Motzkin elimination.

For illustration assume that we have two statements: $S_0$, which updates A[i][j] and $S_1$, which reads A[i][j+2], inside the same loop nest. Assume that the loops $i$ and $j$ iterate from 0 to 1000 and that the access function of A[i][j] is $i \leq 4000 + j \leq 4 + 10000$. Let $i'$ and $j'$ represent the target iterators and $c$ be the added parameter for equating the linear functions. The write-after-read dependence polyhedron is shown in Figure 4.7 (left) and the dependence polyhedron after Fourier Motzkin elimination is shown in Figure 4.7 (right).

$$\begin{pmatrix} i & j & i' & j' & 1 & c & >= & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & >= & 0 \\ 1 & 0 & 0 & 0 & 999 & 0 & >= & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & >= & 0 \\ 0 & 0 & 1 & 0 & 999 & 0 & >= & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & >= & 0 \\ 0 & 1 & 0 & 0 & 999 & 0 & >= & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & >= & 0 \\ 0 & 0 & 0 & 1 & 999 & 0 & >= & 0 \\ 4000 & 4 & 0 & 0 & 10008 & 1 & == & 0 \\ 0 & 0 & 4000 & 4 & 10000 & 1 & == & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & == & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & >= & 0 \end{pmatrix} \begin{pmatrix} i & j & i' & j' & 1 & >= & 0 \\ 1 & 0 & 0 & 0 & 0 & >= & 0 \\ 1 & 0 & 0 & 0 & 999 & >= & 0 \\ 0 & 0 & 1 & 0 & 0 & >= & 0 \\ 0 & 0 & 1 & 0 & 999 & >= & 0 \\ 0 & 1 & 0 & 0 & 0 & >= & 0 \\ 0 & 1 & 0 & 0 & 999 & >= & 0 \\ 0 & 0 & 0 & 1 & 0 & >= & 0 \\ 0 & 0 & 0 & 1 & 999 & >= & 0 \\ 1 & 0 & 1 & 0 & 0 & == & 0 \\ 0 & 1 & 1 & 0 & 2 & == & 0 \\ 1 & 0 & 1 & 0 & 0 & == & 0 \\ 0 & 1 & 0 & 1 & 1 & >= & 0 \end{pmatrix}$$

**Figure 4.7:** Populating the dependence polyhedron

### 4.6.5 Scheduling and Skeleton selection

Apollo uses Pluto for computing a valid and optimal schedule. For this purpose, Pluto is used as a library and it takes as input the computed dependence polyhedron. Once the scheduling is finished, Apollo detects the parallel loops and other transformation properties. Based on the transformation properties, an appropriate code skeleton is selected. The loop transformation and the predicted linear functions are then passed as arguments to the selected skeleton. As explained in Section 4.5.4, the optimization skeletons have embedded generic linear functions. The coefficients of these linear functions are then instantiated from the arguments The loop bounds over the transformed

space are then computed using Fourier Motzkin elimination. It is worth noting that Apollo does not invoke the 'C' code generator of Pluto.

### 4.6.6 JIT

Apollo uses Just In Time (JIT) [1] compilation to further optimize the code. In addition to the general benefits of using JIT compilation, JIT usage inside Apollo is particularly important due the dynamic use of the polyhedral model. As explained earlier, the optimizations skeletons have linear functions corresponding to the transformation, loop bounds and verification functions. These values are constant per phase and are resolved only when the skeletons are instantiated. Note that each skeleton has two versions: binary and LLVM IR. The JIT operates on LLVM IR version of the skeleton. When the JIT is invoked, these values are treated as immediate values, which helps to obtain better performance by using constant folding, common sub expression limitation and dead code elimination. The resulting assembly code from JIT is saved in a code cache and is reused if the transformation is reused. In general, applying JIT gives performance benefits and thus is enabled by default in Apollo. However, it can be disabled by specifying some environment variables.

### 4.6.7 Backup and Safe execution point

The optimizing transformations are deduced from the speculative prediction model that was built. To accommodate the possibility that the speculation may fail, the system must preserve a valid intermediate execution state, called *safe state*. In the case of mis-speculation, the safe state can be restored and the original code execution can be restarted from this point. Without any safe intermediate state, the entire original code execution would have to be restarted from the very beginning of the target loop-nest, which would induce a prohibitive time overhead and also, would eliminate the opportunity of handling different dynamic phases of the same program.

One way to construct a safe point would be to backup each memory location just

before it is updated. This strategy can be implemented as (i) each thread backs up every location it is going to update or (ii) each thread communicates with others to know whether the location has already been backed-up and the current thread backs-up only if the other threads have not yet backed up the current location. However, (i) suffers from a huge amount of required memory because of the possible multiple copies. It also requires a mechanism to detect which backup occurred first among the multiple backups of the same location, in order to be restored properly. On the other hand, (ii) requires thread synchronization and should be avoided for performance. Finally, both strategies involve numerous 'malloc' calls which obviously hurts performance.

With the help of the linear functions characterizing the write memory accesses, and with the values of the loop bounds of the next launched chunk, Apollo can determine the memory write regions even before the chunk is launched. Once the range is computed, a custom 'memcpy' is executed to backup the write regions, before launching the chunks. The custom 'memcpy' is designed to take advantage of vectorization in modern processors and is more efficient than the default 'memcpy'. While designing the backup system, one alternative strategy which was considered was to backup per thread; i.e. each thread backups its own data before updating. The rational was that, since the backup is done by the same thread which is going to compute that data, and the computation is performed back- to-back to that of backup, the cache locality will be improved. However, this strategy was slower than backing up all memory locations together (before launching the chunk), due to the additional number of backups and possibly because the huge data prevented any data reuse.

The region to be backed up for each statement is determined by the lowest and highest access address, which in turn is obtained by substituting the lower and upper bounds of each enclosing loop in the access function of that statement. If the access is not dense, this will result in backing up a lot of data which will never be used. For e.g., consider an access like $A[i \leq 4][j]$ inside a loop nest of depth 2 with $i$ and $j$ as iterators. The computed range includes a lot of memory that will never be used, as the access to

the first dimension is sparse (for e.g. $A[1][\leq], A[2][\leq$ etc. is not used). To compute the exact range, the entire loop nest has to be simulated, which is considerably more costly. Our solution is to find the dimension which causes the highest stride in memory; this can be found by looking at the access function and by selecting the iterator with the highest coefficient. By simulating just this loop, the amount of backed up memory can be reduced, without affecting performance.

### 4.6.8 Parallelization

The LLVM IR representation is originally designed for sequential programs and does not have direct support for expressing parallelism. Most of the frameworks which use LLVM and do parallelization, relies on the GNU OpenMP Library (libgomp). Certain functions such as 'GOMP parallel loop runtime start' and 'GOMP parallel end' can be used to mark parallel regions. However, this process is difficult and is not flexible.

Apollo uses its own intrinsic mechanism, called dispatcher manager, for parallelization and it relies on OpenMP [2]. Inside each optimization skeleton, each loop is represented by a function, which is parameterized by the lower bound and upper bound of the loop. When the optimized skeleton is running and when the control reaches a loop, the dispatcher manager in the runtime is invoked with the loop id and loop function pointer as parameters. The dispatcher manager checks whether the loop can be parallelized by checking the transformation properties (obtained from Pluto). If the loop is not parallelizable, the loop function is simply called with the lower and upper bounds of the loop as parameters, which results in sequential execution. If the loop is parallel, the dispatcher manager, divides the iteration space of the loop into small chunks, called *dispatcher chunks* and the loop function is invoked in parallel with the dispatcher chunk boundaries as parameters.

For better understanding consider the Listing 4.7. The 'C' version of optimized skeleton to illustrate parallelization is given in Listing 4.8; other details are abstracted. Listing 4.9, shows the working of the dispatcher manager.

Listing 4.7: Sample code to illustrate various runtime mechanisms

```
1 for(i = 1; i < 1000; i++)
2 {
3     while(ptr)
4     {
5         ptr = ptr >next;
6         compute_statments(ptr);
7     }
8 }
```

Each loop is exacted to its own function. This is shown in Listing 4.8. The loops are then executed by calling the dispatcher, with the loop function, loop id, the loop bounds and other parameters. As shown in Listing 4.9, the dispatcher manager checks if the loop is parallel (line numbers 5-6). If parallel, the loop function is called, multiple times in parallel, each invocation with different dispatcher chunk bounds, and thus orchestrating parallelization (line numbers 17 - 24). Otherwise, the loop function is simply invoked with the loop bounds, thus orchestrating sequential execution (line numbers 29-29).

### 4.6.9 Optimized skeleton execution

The optimized skeleton is instantiated with the transformation and the linear functions for memory accesses, loop bounds and scalar values. If the binary version of the skeleton was selected, it patches the generic transformation functions and other linear functions. The JIT-ed skeletons already have the transformation and linear functions patched. In either case it is followed by execution of the code in the transformed space.

Listing 4.10 shows an abstracted view of optimized skeleton with the perspective of initializing the loop bounds, and initializing the linear functions of binary skeletons. The loop bounds are computed in Line 5 and 9. The original iterators are recovered in Line number 12. In Line number 17 (and in 21), the scalar linear function for *ptr* is instantiated with the parameters from the runtime. In Line numbers 18 and 19, the

Listing 4.8: Abstracted view of optimized skeleton to illustrate parallelization

```
void apollo_loopnest_optimized_skelton_1(int chunk_lb,
        int chunk_ub, ...)
{
    int loop_id = 1;
    //calling dispatcher corresponding to the 'for loop'
    dispatcher_manager(&loop_1, chunk_lb, chunk_ub, ...);
}

void loop_1(int lb, int ub, ...)
{
    int i;
    int loop_id = 2;
    for(i = lb, i < ub; i++)
    {
        //calling dispatcher corresponding
        //to the 'while loop'
        dispatcher_manager(&loop_2, loop_id, lb, ub, ...);
    }
}

void loop_2(int lb, int ub, ...)
{
    int i;
    for(i = lb, i < ub; i++)
    {
        ptr = ptr >next;
        compute_statments(ptr);
    }
}
```

memory access linear function is instantiated with the parameters from the runtime.

### 4.6.10 Runtime verification

While the instantiated parallel skeleton runs inside a chunk, the runtime system must continuously ensure the correctness of the transformed code that is run [3] by verifying (1) the adherence of the prediction model to the memory locations that are actually accessed and (2) that occurrences of any unpredicted memory accesses do not invalidate the code transformation that has been applied, regarding the code semantics.

Listing 4.9: Dispatcher Manager

```
apollo_run_dispatcher((void*)loop_function(int, int, ...),
        int loop_id, int loop_lower, int loop_upper, ...)
{

  bool loop_is_parallel =
          trasformation_properties.is_loop_parallel(loop_id);

  if (loop_is_parallel)
  {
     //decide number of threads to use
     num_threads = apollo_run_dispatcher_get_num_threads
             (loop_id, loop_lower, loop_upper);
     //calculate dispatcher chunk_size
     dispatcher_chunk_size = apollo_run_dispatcher_get_size
             (loop_id, loop_lower, loop_upper, num_threads);

      //run_in parallel
     #pragma omp parallel for num_threads(num_threads)
     for (int slice_lower = loop_lower; slice_lower <
          loop_upper; slice_lower += dispatcher_chunk_size)
     {
         loop_function(param, slice_lower, slice_upper,
                       loop_upper, ...);
     }
  }
  else
  {
      //run sequentially
     loop_function(loop_lower, loop_upper, ...);
  }

}
```

The speculation verification system must be efficient to accommodate the fact that the verification is potentially associated with each memory instruction, while at the same time, it must be general enough to handle all the different types of required verifications. The most common practice, especially in the field of thread level speculation, is to use a centralized verification system [72, 73, 74]. Such a system does not scale as

Listing 4.10: An abstracted view of optimized skeleton to illustrate transformation

```
1  void apollo_loopnest(int chunk_lb, int chunk_ub,
2              TX_TYPE TX_inv, LN_FN_TYPE linear_fn ...)
3  {
4   loop_id = 0;
5   FM_compute_bounds(loop_id, &vi_lower, &vi_upper, ...);
6   for(vi = vi_lower; vi < vi_upper; vi++)
7   {
8     loop_id = 1;
9     FM_compute_bounds(loop_id, &vj_lower, &vj_upper, vi, ...);
10    for(vj = vj_lower; vj < vj_upper; vj++)
11    {
12      [i,j] = TX_inv * [vi,vj]^T;
13
14      auto ln_fn_ptr = linear_fn.scalar[ptr_id]);
15      auto ln_fn_ptr_next = linear_fn.mem[ptr_next_id]);
16
17      *(ln_fn_ptr[0] * i + ln_fn_ptr[0] * i + ln_fn_ptr[0]) =
18          *(ln_fn_ptr_next[0] * i +  ln_fn_ptr_next[0] * j +
19          ln_fn_ptr_next[0]);
20
21      compute_statments(ln_fn_ptr[0] * i + ln_fn_ptr[0] * i
22          + ln_fn_ptr[0]);
23    }
24  }
25 }
```

each memory access may be subject to verification, which implies a huge time overhead induced by thread communication and synchronization.

Apollo verifies the linear functions for memory accesses, loop bounds and scalars, by equating the predicted linear functions and the observed address (or value). For better understanding consider the code in Listing 4.7. In order to illustrate verification (other mechanisms are omitted for clarity), a 'C' version of the optimized skeleton is shown in Listing 4.11.

In Listing 4.7 assume that the loop bound and the linear function corresponding to the *i loop* iterator can be determined statically. We have to dynamically verify the loop bound of the *while loop* in line number 3, the memory access function corresponding

Listing 4.11: 'C' version of the verification system corresponding to Listing 4.7

```
1  for(vi = 0; vi < 1000; vi++)
2  {
3       vj = 0;
4       while(vj < α4 * vi + β4 * vj + γ4) && ptr)
5       {
6            ptr = α1 * vi + β1 * vj + γ1;
7            if(&ptr >next != α2 * vi + β2 * vj + γ2)
8            {
9                 set_rollback_flag();
10                return;
11           }
12           compute_statments(ptr);
13           ptr = ptr >next;
14           vi_next = getnext_VI(vi);
15           vj_next = getnext_VI(vj);
16           if(ptr != α3 * vi_next + β3 * vj_next + γ3)
17           {
18                set_rollback_flag();
19                return;
20           }
21      }
22      if(vj != α4 * vi + β4 * vj + γ4)
23      {
24           set_rollback_flag();
25           return;
26      }
27 }
```

to *ptr* $\Leftrightarrow$ *next* in line number 5 and the scalar value of *ptr* in line 5. The verification function for the memory access and the loop bound are shown in line number 7 and line number 22 in Listing 4.11. As illustrated, if the linear functions are satisfied, the execution continues, otherwise, the rollback flag is set and the execution is squashed.

Unlike the memory accesses and the loop bounds, for the scalars, each iteration verifies the next sequential iteration. Consider the scalar value of *ptr* in line 5 in Listing 4.7 and the corresponding verification statement in line number 16 in Listing 4.11. Note that the iterator values used are corresponding to the next sequential iteration.

The state of a scalar variable at the end of one iteration is the same as the state when it enters the next sequential loop iteration. The rational for our scalar verification follows the same principle. The state of scalar variables in the iteration corresponding to the very first sequential iteration of any chunk is trivially valid, as they are initialized from a chunk boundary which is a consistent state. For the rest of iterations, the previous iteration in the sequential order, verifies the current iteration, even if the iterations are run in a different, and probably parallel, order.

### 4.6.11 Rollback

If any thread detects a mis-peculation a rollback flag is set. When all the threads finish their executions, the rollback flag is checked to see if the speculation was successful or not. If the speculation failed, the backup is restored to the safe point. Then the original chunk is selected to run the mis-predicted chunk which is then followed by the instrumentation skeleton. If the flag is not set, then the speculation succeeded and all the backup is released.

### 4.6.12 Termination

Apollo continues to execute the code skeletons till the outermost loop bound is reached. If the outermost loop bound is known at compile time, before launching each chunk, the upper bound of the chunk is checked to see if the current chunk is the last chunk. If the current chunk is not the last chunk, or if the loop bound is not known, then the next chunk is speculatively launched with the same transformation assuming that the program did not change its behavior. Eventually, the outermost loop bound is reached. If the upper bound of the last chunk is not equal to the actual loop bound, a rollback is triggered and the chunk is the re executed with the original skeleton.

Additionally, Apollo needs to update the state of 'live out' variables. The scalar variables used inside the skeletons may be used outside the loop nest. These variables are thus 'live' on loop nest exit. However, each skeleton has its own copy of scalars.

Hence on exit, scalar values which are live, should be copied to the original scalars. Once the outermost loop successfully completes the execution, Apollo cleans its data structures and relinquishes the control back to the user code.

## 4.7 Results

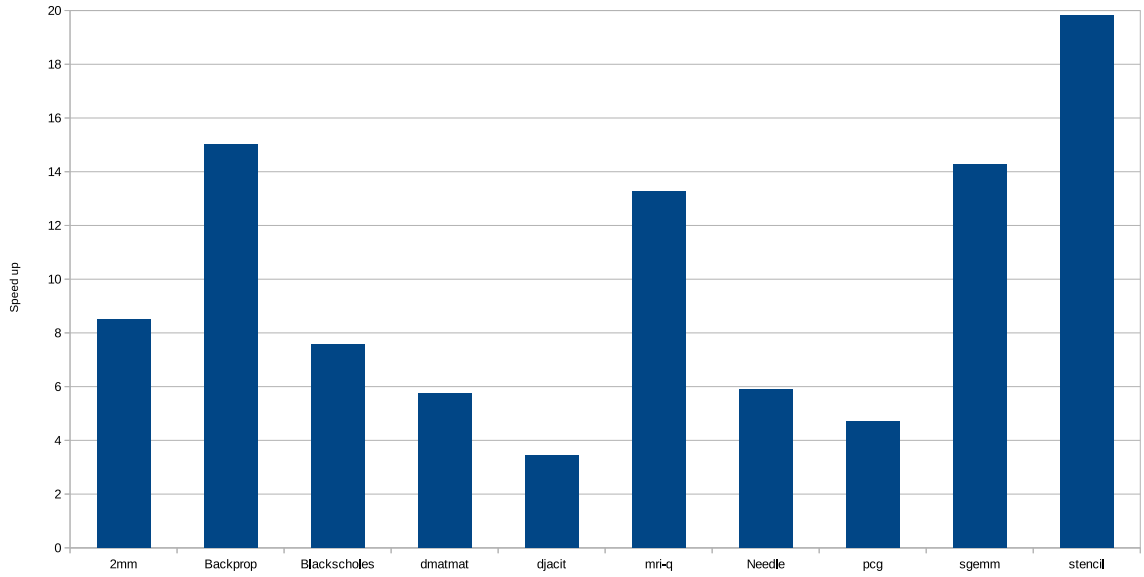



**Figure 4.8:** Speedup of APOLLO, using 24 threads.

The experiments were run on a platform embedding two AMD Opteron 6172 processors of 12 cores each, at 2.1 Ghz, running Linux 3.11.0-17-generic x86_64. The reported measurements were obtained by running each benchmark five times, and taking their average. The speed-ups are reported against the best performing serial codes among clang and gcc-compiled binaries with flag '-O3'.

The set of benchmarks has been built from a collection of benchmark suites, such that the selected codes highlights Apollo's capabilities. The benchmarks mri-q, sgemm and stencil are from the Parboil benchmark suite [4], blackscholes from the Parsec benchmark suite [5], backprop and needle from the Rodinia benchmark suite [6], dmatmat, djacit and pcg from the SPARK00 benchmark suite [7] and finally 2mm from the Polyhedral benchmark suite polybench. The 2mm benchmark was rewritten to have pointer based memory accesses, and hence is not analyzable by static tools.

Figure 4.8 shows the speedup obtained using Apollo.

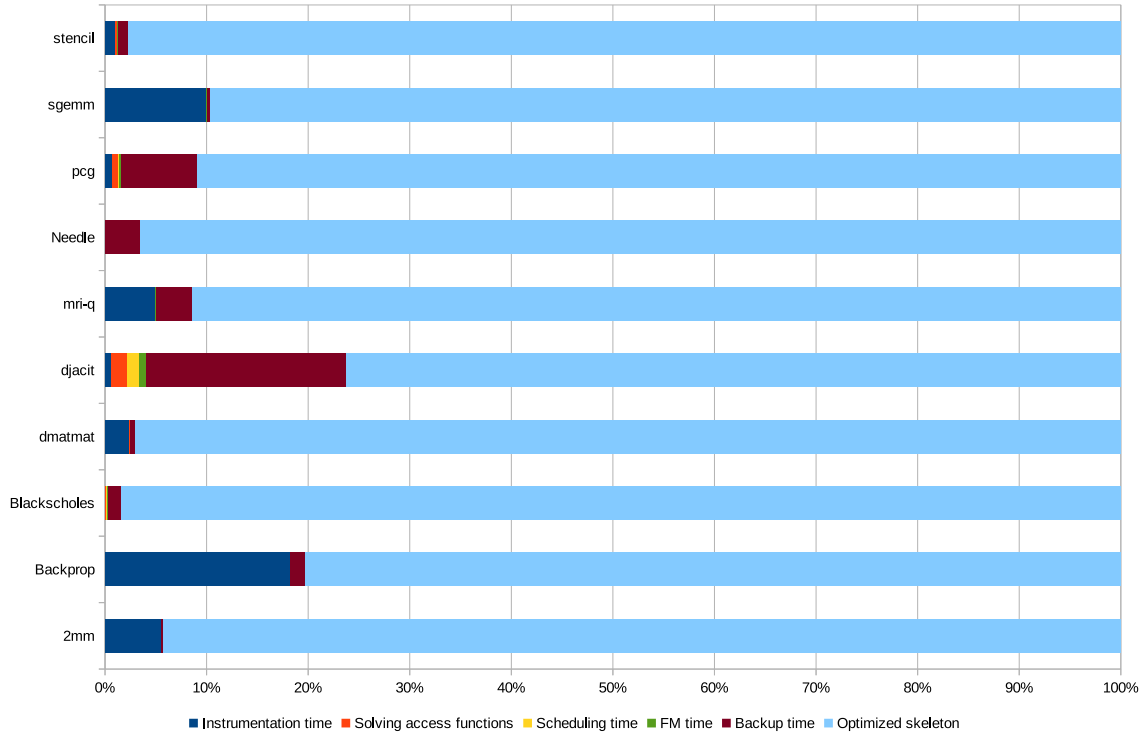



**Figure 4.9:** Overhead comparison

Figure 4.9 shows the various overheads associated with each Apollo module. Only the major overheads are considered. The *instrumentation time*, refers to the time taken for code profiling. It includes the time for running a small chunk, and the time to register the memory accesses and loop bounds information to the runtime. The *solving access functions* refers to the time spent on resolving the memory accesses and loop bounds interpolating functions. The *scheduling time* refers to the time taken by the scheduler, *i.e.*, the time taken to instantiate the scheduler and the time taken by the scheduler of Pluto to determine an optimal schedule. The *FM time* refers to the time required for Fourier-Motzkin elimination. The Fourier-Motzkin elimination is used to determine the loop bounds in the transformed iteration space. If the transformation remains the same across contiguous chunks, the FM solver is only invoked once, but the resulting functions are reused for each chunk. However, if the transformation is changed, the FM solver is invoked again. *Backup time* refers to the time taken by the

backup system to calculate the memory area that needs to be backed up and the time for actual backup. The *optimized skeleton* refers to the time spent on actual parallel execution, decentralized verification and live backup.

As Figure 4.9 shows, most of the execution time is spent in running the optimized skeleton. The overheads on average is very less (around 4.35 %). In general, instrumentation and backup accounts for the major overheads. It is worth noting that the instrumentation skeleton is slower not just due to code profiling, but also due to the fact that this skeleton follows the original sequential schedule and thus no optimizations are applied yet. The latter accounts for the higher instrumentation time of the benchmark *backprop*, where the optimization consists of parallelization and loop interchange. Even though instrumentation is costly, since it is rarely used, it can efficiently be masked by the execution time of the optimized skeleton, especially if the number of a rollbacks are less. If the execution time is less, the overheads are more pronounced. This can be seen in the case of the benchmarks *djacit* and *pcg* which have low execution times and thus the overheads are higher than the average (19.67 % for *djacit* and 9.05 % for *pcg*). In general, the overhead due to backup is unavoidable in Apollo. One strategy which was tried to reduce the backup time was to verify all the accesses first in an inspector-executor fashion. Once the entire chunk is verified then the computations are performed. Since the computation is performed only on verified accesses, the backup is not required. However, due to data locality issues, this approach did not yield much benefit. Also, the latter approach can only be applied to codes where the verification can be split from the computation. Speedups could be further improved by adding more transformations, especially tiling.

## 4.8 Summary

Apollo is an automatic loop optimizer which is based on Thread Level Speculation and the Polyhedral model. By combining TLS with the Polyhedral model, codes exhibiting dynamic polyhedral behavior, at least in phases, can take advantage of the advanced

optimizing transformations offered by the polyhedral model. The overview of the whole system is depicted in the Figure 4.1

Apollo consists of two main modules. The static module statically analyses the code to determine the compile time dependencies, alias information and access functions. It constructs a set of skeletons, each supporting a set of polyhedral transformations. The skeletons are incomplete code snippets which needs to be filled with runtime parameters for execution. The instrumentation skeleton has additional instructions injected to profile the code. All the skeletons and static analysis information is included in the final binary.

During the program run, the runtime first selects the instrumentation skeleton to profile the code. By interpolating the instrumentation data, linear functions are built for memory accesses, scalars and loop bounds. Then the dependencies are identified and the dependence polyhedron is constructed. Pluto is then invoked to compute a valid and optimizing transformation. From the transformation computed by Pluto, the parallel loops are identified and based on the transformation properties, one of the code optimization skeletons is selected. By using the linear functions, the write region of the chunk which is going to be executed is computed, and that region is backed up. The selected skeleton is then instantiated with transformation information and verification information During the execution of the optimized skeleton, each thread verifies its accesses in a de-centralized manner. In case of mis-speculation, the rollback flag is set. If the speculation succeeds, the next chunk is launched with the same transformation; only the loop bounds are updated using Fourier Motzkin elimination. If the speculation does not succeed, a rollback is performed by restoring the backed up data and the chunk is re-launched with the original skeleton, which is then followed by the instrumentation skeleton. The whole process is repeated till the outermost loop bound is reached. As it was shown in the result section, very good performance can be achieved with our approach.

Even though Apollo can handle codes exhibiting a runtime polyhedral behavior, it

is not able to handle codes when the memory accesses are not linear. Especially when dynamic codes with indirect memory accesses or pointers are considered, it is seldom the case that, all the accesses fits a linear model. The limitation in Apollo arises from the fact that the polyhedral model is only able to handle affine codes. One possible solution is not to use the polyhedral model and replace it with some other model. However, to our knowledge, no other model is as effective as the polyhedral model in proposing advanced code transformations. Hence, our solution consists of extending the polyhedral model to handle non-linear codes and is made possible by our dynamic framework. This approach is described in the next chapter.

This page intentionally left blank

*All birds find shelter during a rain. But eagle avoids rain by flying above the clouds.*

A. P. J. Abdul Kalam

*Once we accept our limits, we go beyond them.*

Albert Einstein

*The only way to discover the limits of the possible is to go beyond them into the impossible.*

Arthur C. Clarke

# 5

# Non affine Apollo

## 5.1 Introduction

Apollo can efficiently handle class of codes where the memory accesses and loop bounds are linear; at least in phases. If the code behavior is compatible with the polyhedral model, that particular region is subjected to polyhedral optimizations speculatively. On the other hand, when the code behavior deviates from affine behavior, then that particular region is executed using the original code. However, many codes exhibit non linear behaviors, especially codes with indirect memory accesses, pointers *etc.*. In addition to this, for a speculative system, such as Apollo, to be efficient in terms of performance, the number of rollbacks should be small. The version of Apollo, described in Chapter 4, would rollback whenever it encounters a non linear entity.

Indirect memory references and pointers are quite common in codes and hence, ability to handle them is a prime challenge in automatic parallelization. Most of the codes exhibit non-linear memory behavior or non-linear loop bounds. Traditional

Thread-Level Speculation (TLS) systems which *may* handle such codes require a huge amount of time for backing up memory, while also incurring a huge time overhead from the centralized verification of the speculation. The lack of a prediction model results in higher chances for wrong parallelization and rollback. Moreover, additional loop transformations may be required to make the loop parallel and fast. Hence, a prediction model that can accurately represent the program behavior is essential for better performance.

However, even the presence of a single non-affine entity inside a program invalidates the use of the polyhedral model, *i.e.*, no polyhedral code transformation shall be performed nor parallelization of any loop. The impact of this limitation prevents a lot of codes to be optimized, even when *most* of their memory accesses are linear. This limitation arises because the dependence model cannot be constructed. When at least one non-affine entity is present, the system should make conservative assumptions in order to guarantee correctness. The most conservative approach is obviously to execute the original code itself. In presence of a non-linear entity, most polyhedral optimizers do so. In some very specific cases, some static checks can be encoded to still allow some optimizations, however this is not enough in general [75].

The objective is to enable Apollo to optimize the non linear accesses and validate them, so that it could handle more codes efficiently [9]. This chapter details the challenges and our solutions to this problem. We show that a dynamic and speculative strategy provides new opportunities to enlarge the scope of polyhedral optimizations.

The rest of the chapter is organized as follows. Section 5.2 gives the motivation for handling non linear memory accesses using the polyhedral model. Section 5.3 describes the class of non linear codes which is targeted by Apollo. The non linear dependence polyhedron construction is detailed in section 5.4. Sections 5.5 and 5.6 detail the modeling of non linear memory accesses and non linear loop bounds respectively. Section 5.7 details the scheduling. The modified safe point construction is detailed in Section 5.8. Speculation overhead management is detailed in Section 5.10. Related works and

Listing 5.1: A simple non linear code

```
int A[N], B[N], C[N];
init_array(B, C);
for (i = 0; i < N; i++)
{
    A[i] = B[C[i]];
}
```

their evaluation and comparison are detailed in Section 5.12. Section 5.13 shows the performance gained using our approach.

## 5.2 Motivation

Non linear memory accesses introduce an interesting challenge to polyhedral optimizations or speculative optimizations in general. A lot of code kernels exhibit this behavior. For example, consider a code which allocates a linked list of dynamic structures and assume that each of the structures has a single dynamic field (say char* var_string). Assume that, successive 'malloc' calls allocates contiguous memory locations. If all the elements of the array have the same size, then accesses to these elements are dynamically linear. However, more often than not, the dynamic field varies in size, and the memory accesses are no more linear. It is worth noting that, if the initial assumption of 'malloc' giving contiguous memory locations is broken, even if the dynamic field has the same size across the linked list, the memory accesses are non linear.

For ease of understanding, consider a very simple code given in Listing 5.1. This code is not compatible for polyhedral optimizations. The *i loop* can be parallelized if array $A$ do not overlap with either $B$ or $C$. Since the definitions of the arrays are available, alias analysis can successfully determine that these arrays do not overlap. However, assuming that the values of array $C$ are not linear, the memory access corresponding to array $B$ is not linear and hence not amendable for polyhedral optimization.

For a concrete and real world example, consider the loop kernel of the *trmat* code

Listing 5.2: Example to illustrate non linear accesses (trmat - SPARK00)

```
1  for( i = 1; i <= n; i++ )
2  {
3      k1 = ia[i];
4      k2 = ia[i + 1]    1;
5      for( k = k1; k <= k2; k++ )
6      {
7          j = ja[k];
8          next = iao[j];
9          jao[next] = i;
10         if( job == 1 )
11             ao[next] = a[k];
12         iao[j] = next + 1;
13     }
14 }
```

from the *SPARK00* benchmark suite [7], listed in Listing 5.2. This kernel computes the transpose of a sparse matrix. Since the matrix is sparse, the actual number of elements in each column is unknown. Hence, before executing this kernel, the benchmark code traverses the old index structure and populates the number of elements per column into array $ia$ (not shown in Listing 5.2). It is worth noting that this code cannot be handled statically using the polyhedral model for several reasons:

- the for-loop in line number 5 has bounds defined by values $k1$ and $k2$, that may change randomly during the execution of the loop nest, since these are elements of array $ia$;
- a direct consequence is that the memory read $ja[k]$ in line number 7 cannot be disambiguated, as values of loop iterator $k$ are unknown. The same holds for access $a[k]$ in line number 12;
- variable $next$ holds the result of a single indirect access $iao[j]$ in line 8, which is actually $iao[ja[k]]$. At runtime, this latter memory access may be linear, or quasi-linear, or of any other kind. Similarly, the memory writes in lines 10 and 12 are both using double indirect accesses and hence may also be of any kind:

reference $jao[next]$ is actually $jao[iao[ja[k]]]$, and reference $ao[next]$ is actually $ao[iao[ja[k]]]$.

To summarize, the code in Listing 5.2 has one loop with non-linear loop bounds, and 6 memory accesses of unknown kinds; thus 7 unknown entities in total. It is worth noting that no existing polyhedral static tools can handle this code. The presence of just one unknown entity would force them to choose the conservative approach of not optimizing this code. By using the techniques described in this chapter, we can successfully optimize such codes, even if they exhibit non-linear memory behaviors and loop trip counts.

A part from the obvious challenge of optimizing the codes with non linear accesses, there is an additional challenge regarding the verification system. The de-centralized verification system presented in Chapter 4 relies on predicting affine functions. In addition to this, the initialization of scalar values also depends on affine functions. However, since these functions may not be affine anymore, the whole mechanism has to be redefined. The rest of the chapter shows our approach for tackling these problems.

## 5.3 Non-linear codes

Even if applied dynamically, the polyhedral model still limits the domain of codes that can be handled to codes which exhibit a linear behavior, at least in some phases. This single factor eliminates a large class of codes, as most of the codes have non-affine loop bounds and non-affine memory accesses. However, in this chapter we show that the applicability of the model can be extended by amending certain runtime policies to accommodate these code classes, and thanks to the speculative and dynamic context of application. Depending on the code kernel, the whole kernel or a part of it can be subjected to the effective optimizations provided by the polyhedral model, even if it does not exhibit a linear behavior.

*DCoP* defined in Section 2.4 corresponds to codes which exhibit polyhedral behavior

| **Code behavior** | **Pluto** | **Apollo** | **NL Apollo** |
|---|---|---|---|
| Linear memory accesses, which are an affine function of loop iterators, with constants and unknown loop parameters (e.g. a[i + N + c]) | ✔ | ✔ | ✔ |
| Linear memory accesses, for which the loop iterator coefficients depend on unknown loop parameters (e.g. a[i * N + j]) | ✗ | ✔ | ✔ |
| Linear loop bounds, which are affine functions of loop iterators, constants and unknown loop parameters (e.g. j < i + N + c) | ✔ | ✔ | ✔ |
| Linear loop bounds, for which the loop iterator coefficients depend on unknown loop parameters (e.g. j < i * N) | ✗ | ✔ | ✔ |
| Statically unresolvable, dynamically linear memory accesses (e.g. a[b[i]], where b[i] is linear at runtime) | ✗ | ✔ | ✔ |
| Statically unresolvable, dynamically linear memory accesses through pointers (e.g. *p, where p is linear at runtime) | ✗ | ✔ | ✔ |
| Statically unresolvable, dynamically linear loop bounds (e.g. j < b[i], where b[i] is linear at runtime) | ✗ | ✔ | ✔ |
| Non-linear read memory accesses | ✗ | ✗ | ✔ |
| Non-linear write memory accesses | ✗ | ✗ | ✔ |
| Non linear loop bounds | ✗ | ✗ | ✔ |
| Support for function calls (including recursion) inside the loop nest | ✗ | ✗ | ✔ |

**Table 5.1:** Comparison of different polyhedral code optimization approaches
'a[]' and 'b[]' represent arrays, '*p' represents a memory access through a pointer, 'i' and 'j' are the loop iterators, 'c', a compile-time constant and 'N' an unknown compile-time loop parameter.

dynamically. For the rest of the chapter, we re-define *DCoP* as follows. *DCoP* are sections of code composed of sequence of loops, possibly imperfect, possibly with non affine accesses, non affine conditionals and non affine loops, which are amendable to polyhedral transformations.

This chapter emphasizes on techniques implemented in Apollo to handle such non-linear behaviors. Table 5.1 shows a comparison between Pluto that applies the polyhedral model statically, Apollo, and the new Apollo with its extensions.

In order to handle non-affine entities, a general approach is to relax the dependence

model. This can be done either by approximating the non-affine accesses with affine functions, which can be computed by using linear regression; or by simply ignoring these accesses while constructing the dependence polyhedron, and accounting for them later during verification. If the model is relaxed, then it no more preserves the original guarantee on the validity of the optimizing transformation. This calls for a *runtime validator*, in addition to the runtime linear verification system. Whether to populate the relaxed dependence model with regression equations, or to ignore the non-affine accesses, depends on the quality of the regression hyperplanes: regarding how they model the non-affine accesses. The *fitness* of the regression hyperplanes regarding the original points can be expressed in terms of *correlation coefficients* [10]. The correlation coefficient essentially measures the fitness of the predicted hyperplane and the observed points. If the correlation coefficient is high, then including the regression equation in the dependence model should provide a parallel schedule which optimizes the non-linear accesses conjointly to the linear accesses. On the other hand, if the regression hyperplane has a low correlation coefficient, adding it to the dependence model may not provide any benefit with respect to optimizations; it could even be harmful to do so. In this latter case, it is better to optimize the schedule by initially ignoring the associated memory accesses. Additional runtime tests must then be performed in order to ensure subsequently the validity of the optimizing transformation.

In the following, we give a detailed description on how loops with non-affine memory accesses and loop bounds are handled for speculative optimization and parallelization.

## 5.4 Constructing the dependence polyhedron

The instrumentation process is similar to the one described in Section 4.6.4. During the actual execution, a small slice of each loop of the target loop nest is profiled. The collected profile consists of the memory addresses that are accessed – possibly through successive function calls which may be recursive –, the iteration counts and the enclosing loop iterators. By analyzing the loop trip count of each loop, and subjecting

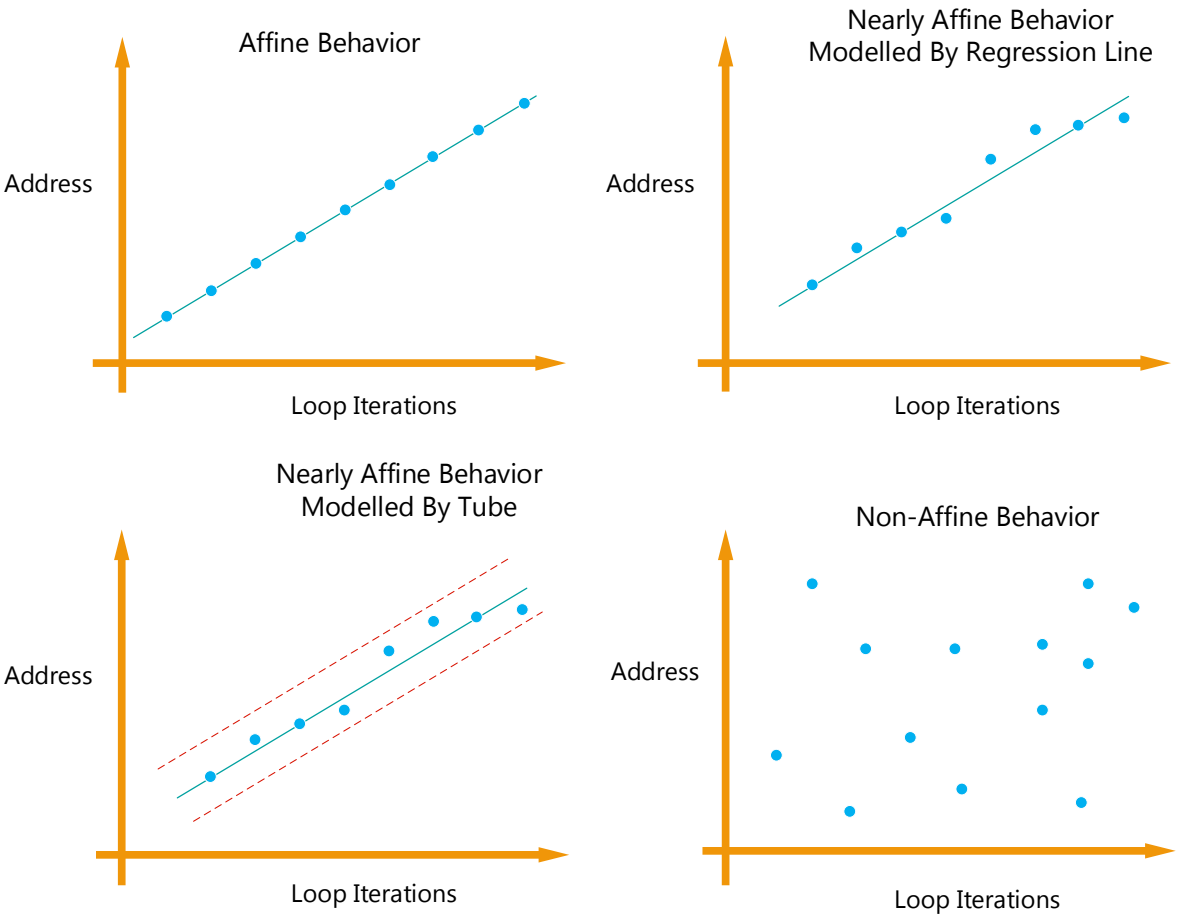



**Figure 5.1:** A graphical representation of *affine*, *nearly-affine* and *non-affine* behaviors.

this information to linear interpolation, and if necessary, to linear regression, we can obtain the linear functions characterizing the loop bounds. Similarly, by analyzing the memory access information, for each memory instruction, we can interpolate the linear functions characterizing the sequence of memory addresses that are accessed. The memory accesses occurring inside a function, possibly recursive and called from within the loop nest, are handled similarly to other memory accesses, although only the non recursive iterators are considered while constructing the linear functions. Essentially, all the memory accesses occurring inside a recursive function are treated as non-linear memory accesses.

The fitness between the observed values of accessed memory addresses (or loop trip counts) is used to classify the accesses as either *affine*, *nearly-affine* or *non-affine*. A

memory access (or loop bound) is classified as:

- *affine*, if all the observations fall perfectly into a linear equation and all the coefficients of the linear functions are integers;
- *nearly-affine*, if most of the observations stay "close" to a linear equation;
- *non-affine*, if the observations do not fall into the other two categories.

A 2-dimensional graphical representation of these three cases is depicted in Figure 5.1. Accesses which can be completely modeled using linear equations couple perfectly with the polyhedral model. However, since the dependencies are calculated on the basis of a small instrumented sample, a runtime verification system is still required to validate the conformance of the prediction model to the rest of the code execution. The details of this verification system are presented in the previous chapter.

The nearly-affine behaviors are common in C/C++ programs which use `malloc` for allocating memory regions. Accesses to malloc-ed regions may not be affine due to various reasons such as unavailability of contiguous space, differences between the access pattern and the allocation pattern, variation of the size for each allocation request either due to the program logic or due to security reasons: malloc may actually allocate more size than requested, so that buffer zones can be created, which greatly helps in avoiding buffer overflow attacks [76, 77]. Non-affine accesses are usually the result of indirect accesses, or of accesses through pointers where the program logic dictates accesses to memory locations in a random fashion.

As opposed to affine accesses, neither nearly-affine nor non-affine accesses can be naturally handled by the polyhedral model. In order to extend the scope of the model, the regression information must be expressed in a way that suits the model. The choice of how memory accesses are characterized is based on the correlation coefficients of the regression hyperplanes. We distinguish two cases that are explained below.

## 5.5 Non-linear memory accesses modeling

Non-linear memory accesses refer to the memory accesses (read or write) whose instances do not correspond to any affine function. Hence, a regression function is computed for such memory statement and the associated correlation coefficient is computed. Based on its value, the non-linear memory access is handled as follows.

**The correlation coefficient is between 0.9 and 1.**

A correlation coefficient between 0.9 and 1 indicates a good fit of the observed values with the regression equation and thus suggests that it approximates well the actual memory accesses. Hence, it is reasonable to include this regression equation into the dependence model, the rationale being that the polyhedral optimizer (scheduler) gives a transformation which optimizes the non-affine accesses conjointly with the affine accesses, thus optimizing data locality and parallelism for the affine and non-affine accesses. The value 0.9 was selected as an optimal value due to the following reasons. If a higher value is selected, it will put more restrictions on the amount of non-linearity. With a lower value, the amount of required backup will increase accordingly, resulting in less performance gains. Thus, 0.9 was selected since it provides a good balance that was observed through numerous experiments. It is worth noting that the correlation coefficient also represents the confidence of the linear function against the memory accesses. Thus, if the value is lower, it is better to treat the corresponding access using the range mechanism explained below.

Each regression hyperplane is computed by multivariate linear regression using the least-squares estimation [10]. However, the presence of outlier points can sway the regression hyperplane away from the majority of the access points. In-order to avoid this, we build the regression hyperplane in two steps: from an initial computed regression equation, we compute the distance of each point to the resulting regression hyperplane. Then, a set of points which are close to the median point is selected and the regression hyperplane is recomputed by just considering these points. When compared

to the original regression hyperplane, the newly constructed regression hyperplane provides a better fit to the memory accesses as the outliers were eliminated while computing it.

The polyhedral model requires the dependence constraints to be expressed in integer $\mathbb{Z}$-domains, whereas the regression hyperplanes lie in real $\mathbb{R}$-domains. For each regression hyperplane, the closest regression hyperplane in the $\mathbb{Z}$-domain is constructed by approximating the coefficients along each dimension to their closest integers. This approximation does not alter much the accuracy since the polyhedral model only considers the *integer* points that lie in the convex hull defined by the constraints.

The regression hyperplane should then be encoded into the dependence polyhedron. If encoded directly, like a linear function, the polyhedral scheduler only considers the points falling exactly on the regression hyperplane when computing the solution. Therefore, using only the regression equations is not sufficient as it ignores the points falling near the hyperplane. Hence, for each nearly-affine access, two hyperplanes that are parallel to the regression hyperplane are constructed by adding positive and negative offsets to the original hyperplane. Informally one hyperplane is "above" and the other is "below" the original regression hyperplane, both forming a *tube* around the regression hyperplane (see the lower left of Figure 5.1). The distance between the tubular hyperplanes (offset) is selected such that all the refined points fall inside the tube. This tube forming a convex hull of the observed points is then encoded into the dependence model.

**The correlation is less than 0.9.**

If the correlation coefficient is less than 0.9, then the memory accesses are characterized as non-affine and excluded from the dependence representation using regression. For each corresponding memory instruction, the accesses are then approximated using the range of the minimum and maximum values of the addresses that were accessed during instrumentation. Notice that this computed range may be very large. For ex-

ample, consider a case in which the first loop iteration accesses the first element of a large array and the next one accesses the last element. In this case, it is better not to perform a range-based backup as the access is very sparse and the range size is very large (in this example, the whole array should be backed up). Apollo only preforms range-based backup if the backup size is less than three times the *expected backup size*. The expected backup size is computed from the number of statement instances, which in turn can be deduced using the enclosing loop bounds. If the range-based backup is not performed, then all the accesses of the corresponding statement will be treated using a "live backup" mechanism (see subsection 5.8).

Overlaps between such ranges, the regression tubes and the linear functions, are tested to know whether a dependence may occur, in which case no code transformation will be performed. This eager early detection of possible dependencies saves the time spent in backing up data, executing a code with an invalid transformation and finally rolling back.

The inclusion of nearly-affine and non-affine accesses relaxes the polyhedral model, but appends the challenge of handling additional verification of the memory accesses occurring outside the predicted regions. This is explained in subsection 5.9.

## 5.6 Non-linear loop bounds modeling

In the presence of non-linear loop bounds, the range of memory addresses that will be accessed cannot be determined even with dynamic instrumentation. Thus, we cannot build the dependence polyhedron. This is even true when all the memory accesses are linear.

To illustrate this issue, consider the code in Listing 5.3. The non-linear inner `j`-loop has a bound based on the value of `C[i]`. At runtime, if the successive values of `C[i]` can be expressed as an affine function of iterator `i` and loop parameter `N`, then no extension is required for Apollo to handle this loop. Otherwise, by applying regression techniques and measuring the correlation coefficient, we can predict the values and

Listing 5.3: A sample code to illustrate the limitations imposed on polyhedral model by non-linear loop bounds

```
function fun_nl_loop(int *A, int *B, int *C)
{
    for(int i = 0; i < N; i++){
            A[i] = i;
        for(int j=0; j < C[i]; j++){
            B[i] += i*j;
        }
    }
}
```

build the model as explained below. It is worth noting that even though the memory access functions for all three arrays `A`, `B`, and `C` are linear, it is unknown at compile-time whether their address ranges overlap, or in other words, whether they alias (for example, if the function is invoked with $(array1, array2, array1)$ as arguments, then pointers A and C will alias). Compilers such as Pluto *cannot* handle this case due to the presence of non-linear loop bounds. It is also worth noting that, even in the absence of non-linear loop bounds, Pluto will assume that arrays `A`, `B`, and `C` do not alias, since it is the programmer's responsibility to ensure so; whereas in Apollo, this is automatically verified.

Modeling a loop in the polyhedral model requires:

1. A linear function for the lower bound of a loop, parametrized by its outer loop iterators, loop parameter and constants;

2. A linear function for the loop iterator, parametrized by its outer loop iterators, loop parameter and constants;

3. A linear function for the upper bound of a loop, parametrized by its outer loop iterators, loop parameter and constants.

Items (1), (2) and (3) are determined simultaneously by instrumenting the scalar values characterizing the loop iterator. (4) is determined by instrumenting the loop trip

count. For a given vector representing the values of the enclosing loop iterators, (1) and (2) are used to compute the current value of the iterator of the loop in consideration and (4) is used to compute until when the loop should iterate. The transformation framework relies on these linear functions to define the domain of the statements and the exact instances of the statements that are scanned.

For simplicity, consider only loop parallelization as being the optimizing transformation. Also assume that (1), (2) and (3) are at least dynamically linear, *i.e.*, all the linear functions characterizing the loop, except the upper bound, were obtained by interpolation. Direct usage of a regression hyperplane may lead to an under or over approximated domain. This implies that a loop with non-linear bounds cannot be parallelized directly. One approach to solve this issue would be to backup the addresses accessed by each loop just prior to its execution (note that Apollo does not do this. It only backups regions based on the outermost loop, before the optimized skeleton is invoked); parallelize code in chunks; execute the chunks one after the other, until a rollback occurs. Then, restore the state and execute the last chunk sequentially. The immediate problem with this approach is the backup. Since each loop backs up data, it would increase the storage requirement exponentially, and there may be a lot of overlaps among backed up data. Moreover, the control of the loop would be very complex, as well as the mechanism to restore memory.

We go one step further to enable parallelization of such loops. By computing a minimal hyperplane for the loop upper bounds, we expect all the loops to execute at least until the point where the iterator meets this hyperplane. If the minimal hyperplane holds its validity during execution, the non-linear loop can be parallelized until this minimal hyperplane. Note that, the parallelization should still be legal, *i.e.*, the dependencies should not prevent parallelization. The rest of the iterations of the non-linear loop should be executed sequentially until it reaches the original loop exit condition. The advantage of this approach is that we are able to parallelize and optimize the predicted part, with a very low time overhead. However, this does not guarantee the

Listing 5.4: A sample code to illustrate the limitations imposed by non-linear lower loop bound

```
function fun_nl_loop(int *A, int *B, int *C)
{
    for(int i = 0; i < N; i++){
            A[i] = i;                          //S_1
            for(int j = B[i]; j < N; j++){
            A[j] += i*j;                       //S_2
        }
    }
}
```

program correctness as the sequential part, which was not predicted, could introduce an unseen dependence, thus violating the dependence prediction model. The memory accesses occurring in the unpredicted space are therefore considered to be equivalent to non-linear memory accesses. As explained in the previous subsection, Apollo's non-linear memory access handling system can detect any violation on unpredicted memory regions and thus guarantees the program validity. If during the actual execution, the loop runs iterations that are not predicted using the minimum hyperplane, while the loop is marked for parallel execution, the system performs a rollback.

A similar approach to that of the upper bound can be followed for the lower bound. By using the current instance of the enclosing loop iterators and using regression, one can predict the lower loop bound. To handle non-linearity, the loop can be run sequentially using the actual values until the predicted lower bound, and then continue in parallel manner. Both the upper and lower bounds non-linear techniques can be combined to produce three waves of computation, a head sequential part, a middle –core– parallel part and a tail sequential part. Handling any transformation such as loop interchange, tiling, etc. follows the same principles.

All the loop iterators appear as scalars in the code. The scalar prediction is thus responsible for the initialization of these values. If the loop lower bound cannot be predicted, the initialization of these values is impossible. Thus, these loops cannot be

optimized using the polyhedral model. This is illustrated by the code example in listing 5.4. The lower bound of the `j`-loop is determined by `B[i]`. If the successive values of `B[i]` do not follow a linear function, there is no way to initialize `j` at each `i`-iteration. The non-linear lower bound for loop `j` can be treated as a memory address and a related dependence can be added to the model. Effectively, for each `i`-loop iteration, the lower bound of the `j`-loop is loaded, not predicted. The immediate consequence is that many optimizations are prevented. Hence, we choose another approach. Based on the correlation coefficient of the loop trip counts, the loops are processed as follows.

**The correlation coefficient is between 0.9 and 1.**

A high value of the correlation coefficient indicates that the loop bounds tend to swing around a hyperplane. Unlike the regression hyperplanes of memory accesses, the loop bounds hyperplane cannot be used directly. The regression hyperplane, by construction, minimizes the average square distance between the observed points. It implies that there could be points below and/or above the hyperplane. Consider the lower bound of a loop. Following the principles of construction of the lower bound regression hyperplane, there may be bounds which are actually above the computed hyperplane. If the loop is parallelized, due to over approximation, the generated code could scan points outside the actual domain. A similar situation occurs when the actual loop bounds are greater than the predicted hyperplane, as this indicates that the generated code could skip some iterations which should be present in the domain. In order to handle this, for the lower bounding hyperplane, the hyperplane is shifted up by the width of the tube. The rationale being that, the shifted hyperplane acts as an expected minimum. Note that this still does not ensure that all the points are in the domain.

**The correlation is less than 0.9.**

As with memory access regression hyperplanes, a correlation coefficient lower than 0.9 indicates that the regression is a poor fit for the observed loop bounds. However,

some optimizing transformations can still be applied if a single value representing the lower bound can be found out. Hence, a single value (rather than a hyperplane), indicating the predicted lower bound is selected from the observations.

Note that, unlike some existing schemes which handles a very simple outermost while loop [78, 79, 80, 81], our system is capable of handling complex while loops appearing anywhere in the loop-nest. Also, in most of the existing schemes, only the upper limit is assumed to be unknown, whereas our system allows either upper bounds or lower bounds, or even both of them, to be unknown. There is also an additional challenge of handling non-linear memory accesses along with non-linear loop bounds.

If the non-linear loop bound cannot be characterized by regression, and a minimal or maximal hyperplane cannot be constructed, or if after constructing the cutting hyperplanes the optimized part will only have a very low number of iterations, then the whole loop is subsumed to a statement such that the scheduler sees the whole loop as a single statement and processes accordingly.

## 5.7 Scheduling and skeleton selection

The scheduling and skeleton selection is similar to the one described in Section 4.6.5. Once the dependence polyhedron has been constructed as mentioned in the above sections, Pluto is used for scheduling. Traditionally, Pluto takes as input a 'C' language-based code and produces a 'C' language-based output. However, in Apollo, Pluto is used as a library. Apollo directly encodes the dependence information into Pluto's internal structure and invokes its scheduler. The scheduler outputs the optimized schedule (in a matrix format). Note that, the Pluto's 'C' code generator (CLOOG) is not invoked by Apollo. In addition to the transformation, Apollo also identifies the loops which can be parallelized and uses this information to parallelize these loops. Based on the transformation properties, Apollo selects an instantiates an appropriate code skeleton.

## 5.8 Backup and Safe execution point

The backup for the affine write accesses are handled in a similar manner as explained in Section 4.6.7.

For non-affine write accesses, there is no information in the model enabling prediction of the memory regions that will be read or written during the run of the next chunk. Hence, a "live" backup is performed during the speculative parallel execution of the next chunk, where each memory location is backed up just before being updated. For nearly-affine write accesses, all the memory locations occurring inside the regression tube are backed-up before launching the chunk. However, there still could be some outlier accesses occurring outside the predicting regression tubes, which however do not corrupt the semantics of the transformed loop nest. Since such accesses can only be known during the chunk execution, these accesses are handled similarly to the non-affine case: they are backed up "live" during the chunk execution. As the live backup is performed by each thread, in a distributed manner, one thread is not aware of the locations which may have been already backed up by another thread. Nevertheless, by using the prediction model, the amount of live backup is significantly reduced and therefore saves memory and execution time.

For loops with non-linear bounds, the backup is computed based on the predicted minimum bound hyperplane and maximal bound hyperplane. Every location that may be updated between the hyperplanes is backed-up before launching the chunk and the ones lying outside are backed-up live by each thread, similarly to non-affine write accesses.

Typical runtime speculation systems back up the data on the fly (live backup). Our experiments show that, individually backing up memory locations is costly. This is shown in Figure 5.2 where different backup strategies are compared. An image processing kernel is studied to see the effect of different backup strategies while varying the number of non-linear writes. Each non-linear write corresponds to a write operation

on one of the RGB components of the image. Each code version is named $\alpha$___strategy where $\alpha$ represents the number of non-linear writes. In the "Runtime only" strategy, the whole backup is performed live, *i.e.*, just before the write operation is performed. "Runtime only optimized" is an optimized version of "Runtime only" strategy, where the memory management is partially achieved by Apollo, thus reducing the number of required memory allocation calls. "Apollo combined" uses a two pronged approach by using the prediction model. As explained above, by using the linear functions, the regression functions and the range information, Apollo tries to predict the set of memory locations that is going to be written during the run of the next chunk, and backs-up this region before the chunk is launched. Only the unpredicted accesses are backed up live. For the live backup, the memory management is also partially handled by Apollo, as in the "Runtime only optimized" strategy.

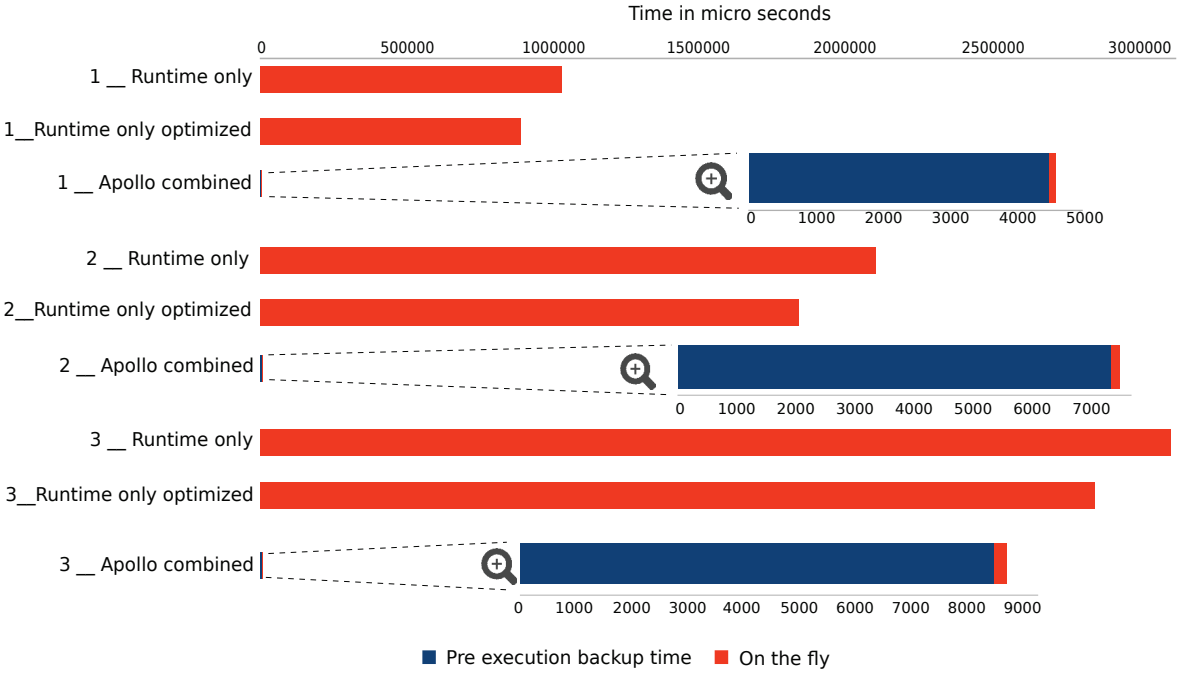



**Figure 5.2:** Comparison of various backup strategies

Each code version is named $\alpha$___strategy where $\alpha$ represents the number of non-linear writes

## 5.9 Runtime verification

The affine memory accesses, scalars and loop bounds are verified in a similar manner as described in Section 4.6.10. For the affine accesses the centralized communication in not required. However, centralized verification, eager or lazy, cannot be avoided in all cases, especially when there are memory accesses which cannot be predicted. Nevertheless, the volume and frequency of communication to the central verification system can be greatly minimized by using an approximate prediction model. In Apollo, each thread verifies all its memory accesses in a decentralized manner, while only the unpredicted accesses whose impact cannot be disambiguated using the prediction model, and which potentially occurs rarely, are verified using a centralized mechanism. Moreover, notice that the centralized verification inside Apollo only occurs at the frontiers between consecutive chunks. A comparison between centralized and decentralized verification approaches within Apollo is provided in Section 5.13

The conformity of the predicted memory accesses regarding the predicted linear functions, and the compliance of the unpredicted accesses to the speculative polyhedral model can only be verified during the actual execution of the optimized code. Each thread verifies the validity of each of its memory accesses against all the predicted memory accesses, and thus performs an eager and decentralized verification. Depending on the accuracy of the model, the following verification operations are performed:

- If the memory instruction has been modeled exactly by an affine function: the thread checks the equality between the actual accessed address and the one predicted by instantiating the predicting affine functions. In case of mis-prediction, a potential risk of unpredicted dependence is detected;

- If the memory instruction has been modeled by a regression tube (or by a range of addresses): the thread verifies that the actual accessed address lies inside the tube (or inside the range). If not, the address is compared to the addresses that

are predicted to be touched by all the other memory instructions during the execution of the current chunk. If no risk of interaction has been detected, the address is registered in a local table in order to be examined after the current chunk completion using the centralized verification. Otherwise, a potential risk of unpredicted dependence is detected.

If there is any potential risk of unpredicted dependence, a rollback is triggered by the faulty thread. Otherwise, at the end of the chunk execution, a cross thread verification is performed to ensure cross thread consistency by comparing the addresses that were registered in the respective local tables. The chances of this lazy verification to fail is very small, as the eager verifications performed by each thread detect, in most cases, any potential risk much earlier. However, if it fails, a rollback is triggered.

Detecting the possibility of a rollback as early as possible is crucial for performance. For the dynamically linear memory accesses and bounds, as soon as the interpolating linear functions have been constructed, the rest of the instrumented points are used for checking their correctness. If this fails, it is inferred that the program is not amenable to any transformation, at-least in the current phase. For the nearly-affine linear functions, the same principles are followed. For the non-affine accesses and bounds, a simple and a fast check is done to verify if these accesses may interfere with any of the speculated affine or nearly-affine accesses. Thus, this checking decreases the probability of rollback. Note that these checks are done before even launching any parallel chunk, thus saving the time overhead of memory backup, speculative execution and rollback.

## 5.10 Speculation overhead management

Apollo evaluates the benefit of speculation by itself during the program run and thus is able to predict whether the speculation will provide any performance improvement. This self-check is very important when the number of speculative chunks that succeed is small, in which case the system should appropriately select the original code or

the optimized one based on the predicted benefit. When a speculative chunk fails, it induces the following overheads: (i) The cost of backup, (ii) the time taken for running the (invalid) optimized skeleton, (iii) the time to rollback and (iv) the time required to run the original code to overcome the faulty chunk. In order to analyze the cost of rollbacks, consider Figure 5.3 which profiles an image processing code kernel where the number of non-linear write accesses is controlled. One non-linear write represents a write to one of the three RGB components of the image. In addition to the number of non-linear writes, the number of rollbacks are also controlled by forcing Apollo to rollback.

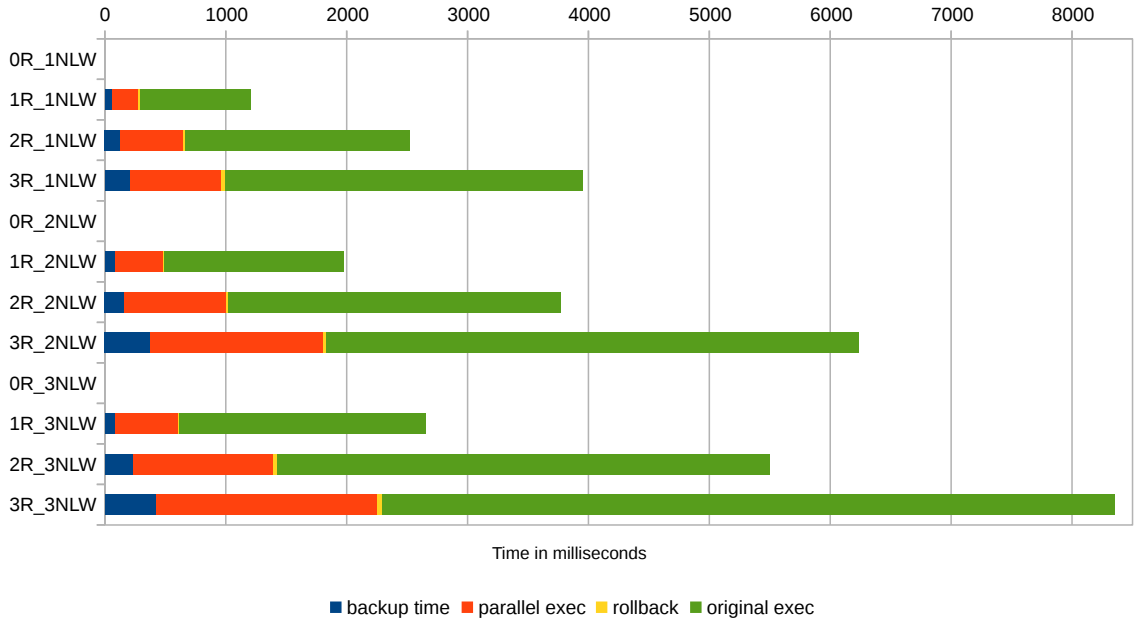



**Figure 5.3:** Overhead of mis-speculation

Each code version is name $\alpha$R_$\beta$NLW where $\alpha$ represents the number of rollbacks and $\beta$ represents the number of non-linear writes

In the figure, only the time lost by mis-speculation is measured. The backup time represents the time required to compute the backup region, as well as the cost of memory allocation and data copying. It also includes the cost of live backup. The parallel execution time represents the time spent by running the optimized skeleton. The rollback represents the time required to restore memory, while the original execution

time represents the time taken by the original skeleton to re-run the mis-speculated chunk. Note that since the figure only represents the time lost in the case of a mis-speculation, the values are zero when there is no mis-speculation. As expected, the total cost of the rollbacks approximately follows a linear relationship with that of a single rollback. Furthermore, the cost obviously increases as the number of non-linear writes are increased.

It is obvious that performance degradation occurs when the system encounters multiple rollbacks. In Apollo, if the speculation fails consecutively for the first three chunks, the rest of the execution completes using the original code. However, if the system encounters a mix of speculation successes and failures, it predicts the possible performance gain of a single chunk and following this prediction, the original or parallel skeleton is selected. This is done as follows. When a speculative chunk fails, Apollo measures the individual time for backup, parallel execution, rollback and original execution. Note that even though the computation was wrong (as speculation failed), the parallel execution time approximately remains the same as that of a correct one. In order to obtain an interesting speedup, the ratio

$$\frac{original\ sequential\ execution\ time(TO)}{apollo\ optimized\ execution\ time(TA)}$$

should be greater than 1.

Let $f$ denote the number of chunks that failed and $p$ denote the number of chunks that succeeded. Let $F$ denote the total time required for executing and recovering a failed chunk, $P$ denote the time required for executing a parallel chunk along with the associated backup, $O$ denote the execution time of an original chunk, and $R$ denote the restore time. Let $\gamma$ denote the expected speed up of an optimized chunk when compared to the original chunk, *i.e.*, $\gamma = O/P$. We want the execution time of Apollo to be less than the execution time of the original serial code, *i.e.*:

$$TO/TA \approx 1 \tag{5.1}$$

where:

$$TA = p \leq P + f \leq F \tag{5.2}$$

$$TO \subset (p + f) \leq O \tag{5.3}$$

The execution time of a failed chunk consists of the parallel execution time (which includes the backup time), the rollback time and the original serial execution time. The rollback time is significantly less than the parallel execution time and the original time, and thus can be ignored:

$$F = P + R + O \subset P + O \tag{5.4}$$

By substituting 5.10, 5.10 and 5.10 in 5.10 and re-arranging we get:

$$p \leq O + f \leq O \approx p \leq P + f \leq P + f \leq O \tag{5.5}$$

By substituting $\gamma = O/P$ in 5.10, removing $f \leq O$ on both sides and re-arranging, we obtain:

$$(\gamma \quad 1) \approx f/p \tag{5.6}$$

Equation 5.10 relates the expected speedup to the number of chunks that can speculatively succeed or fail. Apollo uses the latter to infer the effectiveness of speculating in cases where there is a mix of succeeding and failing chunks, and selects the skeletons accordingly.

## 5.11 Putting it all together

Given an input program, and target loop nests marked by the "apollo pragma", Apollo statically analyzes the loop-nest and produces a set of generic code skeletons, which represent some generic combinations of polyhedral transformations. At runtime, Apollo

first profiles the code by employing instrumentation by sampling. Based on the profile, Apollo computes the linear equations, regression equations and ranges suitable for the modeling of each memory access and loop bound as explained in subsection 5.4, which are then encoded into a relaxed polyhedral model. Once the dynamic polyhedral model has been constructed, the polyhedral scheduler, Pluto, is invoked to produce an optimizing parallelizing transformation. Based on the dynamic polyhedral model, a safe point is computed by backing up the predicted memory write regions, and the speculative chunk is launched. The speculative chunk is monitored constantly, in a decentralized manner, to detect any potential dependence violation. If an unpredicted memory access occurs inside a speculative chunk, live data backup is performed. If the speculative chunk completes successfully, a cross thread verification, which validates the program consistency across threads is launched. The probability of this verification to fail is low, as the distributed verification, in most cases, detects any potential violation much earlier. If the speculative execution succeeds, the speculative transformation is re-applied for the next chunk and so on. On the other hand, if the speculation fails, a rollback is performed and a chunk executing the original serial code is launched, which is then followed again by an instrumentation phase if speculation has still been evaluated as beneficial. The whole process is continued until the entire target loop nest has been executed.

## 5.12 Related work

This section discusses existing work on handling non linear memory accesses using speculative techniques and their comparison with Apollo.

Some previous work looks at handling while-loops in a limited setting. The work from Martin Griebl *et al.* [78] proposes a method to handle while-loops using the polyhedral model. However, the approach does not allow pointers or pointer based accesses and requires all the accesses to be affine. It also requires a centralization mechanism which needs synchronization in each and every iteration of the loop nest.

The impact on performance is not studied. Our comparison with a centralized system in Section 5.13 shows that our system performs better even with the added support for non-affine accesses. Another work by Jean-Francois Collard [79] adds a very basic speculative support to the latter, however, the work further restricts the scope to programs with a single outermost while loop. The work requires huge amounts of memory and all the latter restrictions apply. The work [81] follows a similar approach to that of [79] but choses a conservative approach instead of speculation; no information regarding the applicability or the performance impact is available.

In [80], Stefan J. Geuns *et al.* propose a framework to extract task based parallelism. Polyhedral transformations cannot be applied using this approach and the system is focused on a very specific application. The work tries to parallelize while loops by extracting task based parallelization. Each task is represented by a node in the task graph, and the shared variables form the edges. Special shared data structures are created to share variables. The proposed approach requires the memory accesses to follow a dynamic single assignment per section (In a code section, the memory writes to a given location must be unique) which limits the scope of the work. Our system is able to parallelize and apply advanced loop transformations, and we do not rely on dynamic single assignment. Moreover, the shared data structures proposed in the work has limited size, there by limiting parallelism, whereas we have no such restrictions.

PPCG [50] is a polyhedral compiler targeting statically analyzable code for GPU's. In PPCG dynamic controls statements are supported by subsuming the whole loop which contains the non-affine control (break, continue, etc), to a single statement. This clearly limits the optimization opportunities, for example, in this scheme, a while loop or a for loop containing a dynamic control flow can never be parallelized. Moreover, the memory accesses should still be affine.

In [82], Barthou *et al.* present a static analysis procedure to find approximate solutions to dataflow problems involving unknowns such as the iteration count of a while statement or non-affine array subscripts. Pointers and aliasing are not allowed

in the handled programs. Moreover, since it is static, the approach is necessarily conservative, while our runtime approach handles the actual memory references and iteration counts, that yield obviously more accurate approximations.

In [83], Anand Venkat *et al.* propose a method to handle non-linear codes using the polyhedral model. It only handles sparse matrix computations where the non-linear memory accesses that are considered are exclusively read operations through indirections. It is based on an inspector-executor mechanism. During the inspector phase, it collects all the non-linear memory reads and creates a mapping function which maps the iteration space to the non-linear memory access addresses. The executor then uses the inverse of the created function to re-map the addresses in the transformed space. When compared to our approach, this approach has the following major limitations: (i) The work only supports non-linear reads, whereas our work also supports writes which are crucial regarding data dependences. (ii) The work relies on an inspector-executor mechanism. Thus, as with any inspector-executor system, it can only handle codes where the computation of the target memory addresses can be split from the accesses. Many codes violate this assumption, especially when using pointers. (iii) It is the responsibility of the user to ensure that the non-linear reads are disjoint from the write regions. Hence, no related dependence analysis is performed by the proposed system. (iv) Since the inspector code is sequential, its associated overhead has to be amortized over many calls to the target loop nest. In our proposal, each run of a target loop nest is handled and optimized. Thus, when compared to our system, the proposed system has less automation.

In comparison, Apollo: features a linear prediction model using interpolation and regression; applies optimizing polyhedral transformation on-the-fly, aiming for automatic data locality optimization and parallelization; and takes advantage of a hybrid centralized/decentralized speculation verification system, which significantly reduces the time overhead of data race detection. Another key aspect of the proposed system is to handle pointers and non-affine accesses along with non-affine loop bounds.

## 5.13 Results

The experiments were run on a platform embedding two AMD Opteron 6172 processors of 12 cores each, at 2.1 Ghz, running Linux 3.11.0-17-generic x86_64. The reported measurements were obtained by running each benchmark five times, and taking their average. The speed-ups are reported against the best performing serial codes among clang and gcc-compiled binaries with flag '-O3'.

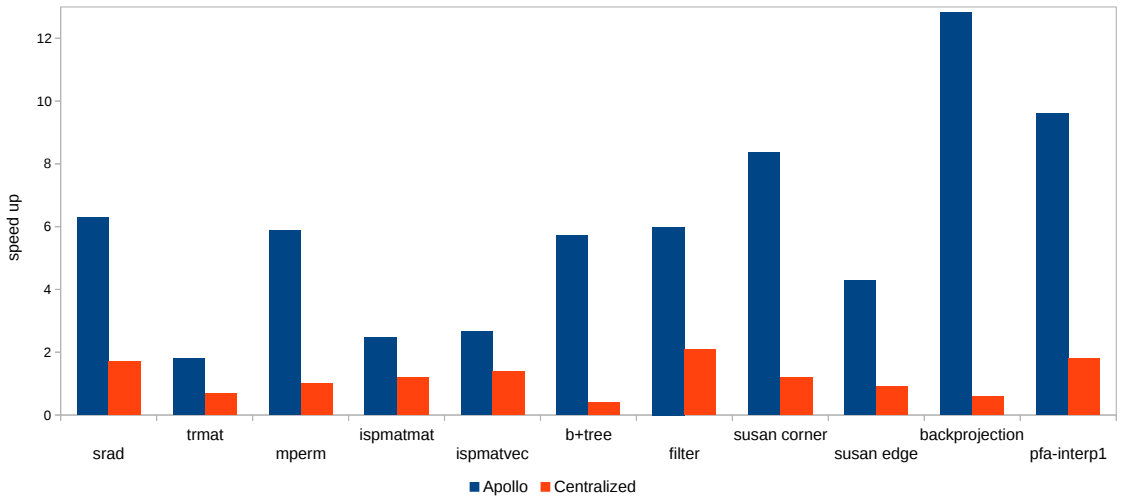



**Figure 5.4:** Speedup of non linear Apollo and the centralized system, using 24 threads.

The benchmarks were selected from various benchmark suites such as the SPARK00 benchmark suite [7], the Rodinia benchmark suite [11], Cbench [12] and the Perfect benchmark suite [13]. They were selected based on their loop properties and memory access patterns. It is worth noting that these benchmarks cannot be optimized by tools like Pluto or the previous version of Apollo, due to the non-linearity of the memory accesses and loop bounds. In addition to the above benchmarks, we included `Filter`*, a real world image processing application which exhibits non-linear array references. Every benchmark source code is written in *C/C++*.

To show the gains provided by Apollo compared to a standard TLS system with centralized speculation verification, we created a modified version of our framework

*http://lodev.org/cgtutor/filtering.html

to simulate such a centralized mechanism. In the centralized verification system, each thread has to communicate with each other. Our modification only consists of a call to a runtime function for each memory access that the loop attempts to perform, thus simulating such a centralized invocation. Inside the centralized runtime verification function, two values are added and the result is written to a memory location. No other operation are performed. Note that these operations are the bare minimum for a centralized system. Furthermore, in our centralized version, the outermost loop is parallelized and no memory backup is performed. We would like to emphasize the fact that in this version, since the runtime verification simply performs a write, not even the overhead of dependence verification is present and the dependencies are always assumed to be respected. Thus, all actions are chosen in favor of the centralized mechanism (no backup, no dependence checking, no central checks). The addresses of the write locations were carefully chosen such that they reside in different cache lines, to avoid the penalty of cache coherence protocols. This measurement serves as a base for comparing against centralized systems, since this modified version only performs a minimum amount of operations per memory access.

A real centralized system would have to perform costlier operations and would rely on a central algorithm. It should be emphasized that our system is based on a weakly centralized model, having a very small centralized verification in addition to possible live backups. However, the weakly centralized verification is not live and hence does not cause bottlenecks. Also, as mentioned earlier, the chances for a rollback to occur at this step is very low, thanks to our prediction model, non-linear analysis and eager verification system.

In figure 5.4, we show the speedups of Apollo using the non-linear modeling and verification system that were presented. As mentioned before, we compare our framework to the centralized verification system. For all the benchmarks, Apollo outperforms the centralized version, while providing significant speed-ups against the original serial codes.

Listing 5.5: susan corner (cbench)

```
1  for (i=5;i<y_size  5;i++)
2      for (j=5;j<x_size  5;j++) {
3          n=100;
4          p=in + (i  3)*x_size + j     1;
5          cp=bp + in[i*x_size+j];
6
7          n+=*(cp  *p++);
8          n+=*(cp  *p++);
9          n+=*(cp  *p);
10         ...
```

Benchmark `trmat` contains non-linear loop bounds and non-linear memory accesses (as explained in subsection 5.3) that are modeled using regression tubes. Listing 5.5 shows an extract of the susan corner detection kernel from the Cbench benchmark suite. Lines 6, 7, 8 (and a long list of lines which are not shown here) access a memory location whose address is the result of a pointer computation depending on 'cp', which is updated in line 5. The computation of 'cp' only uses linear memory accesses. However, its successive values are not linear. Since it is used for an address computation, the resulting address is not linear. Range modeling was used for this code as the regression coefficient was low. Benchmarks `ispmatmat` and `ispmatvec` are both modeled using regression tubes. Benchmark `pfa-interp1`, shown in Listing 5.6 (only some parts are shown), has non-linear conditionals, non-linear loops and non-linear memory accesses. Line 5 assigns the result of a function to variable 'nearest', which is then used in the *if*-condition in line number 7, which leads to a dynamic non-linear conditional. Lines 13, 15, 16 and 18 update variables *rmin* and *rmax*, which are then used as loop bounds for the *for*-loop in line 19. Since these values are not linear, the loop bounds are also not linear, and consequently, so are the memory accesses in lines 20, 24 and 25. This benchmark is handled using the range based mechanism.

Figure 5.5 shows a classification regarding the time-overhead of the system. Only the major overheads are considered. The *instrumentation time*, refers to the time taken

Listing 5.6: pfa-interp1 (Perfect)

```
for (p = 0; p < N_PULSES; ++p){
  ...
  for (r = 0; r < PFA_NOUT_RANGE; ++r){
    ...
    int nearest = find_nearest_range_coord(output_coords[r],
          input_start, input_spacing, input_spacing_inv);
    if (nearest < 0){
        resampled[p][r].re = 0.0f;
        resampled[p][r].im = 0.0f;
        continue;
    }
    ...
    rmin = nearest    PFA_N_TSINC_POINTS_PER_SIDE;
    if (rmin < 0)
        rmin = 0;
    rmax = nearest + PFA_N_TSINC_POINTS_PER_SIDE;
    if (rmax >= N_RANGE)
        rmax = N_RANGE    1;
    for (k = rmin; k <= rmax; ++k){
        win_val = window[window_offset + (k    rmin)];
        sinc_arg = (out_coord    (input_start + k *
                input_spacing)) * input_spacing_inv;
        sinc_val = sinc(sinc_arg);
        accum.re += sinc_val * win_val * data[p][k].re;
        accum.im += sinc_val * win_val * data[p][k].im;
    }
    ...
```

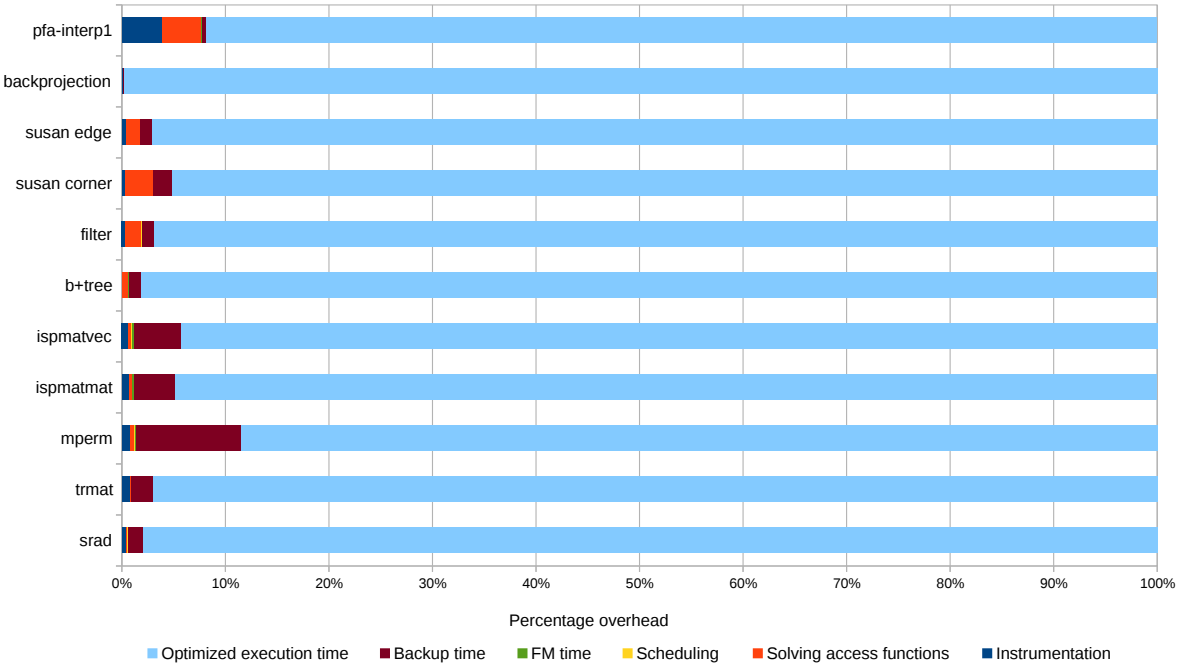



**Figure 5.5:** Overhead classification in percentage

for code profiling. It includes the time for running a small chunk, and the time to register the memory accesses and loop bounds information to the runtime. The *solving access functions* refers to the time spent on resolving the memory accesses and loop bounds interpolating functions. It also includes the time for finding the regression functions, the range functions, and the regression coefficients – which classifies the access as affine, nearly-affine or non-affine –. The *scheduling time* refers to the time taken by the scheduler, *i.e.*, the time taken to instantiate the scheduler and the time taken by the scheduler of Pluto to determine an optimal schedule. The *FM time* refers to the time required for Fourier-Motzkin elimination. The Fourier-Motzkin elimination is used to determine the loop bounds in the transformed iteration space. If the transformation remains the same across contiguous chunks, the FM solver is only invoked once, but the resulting functions are reused for each chunk. However, if the transformation is changed, the FM solver is invoked again. *Backup time* refers to the time taken by the backup system to calculate the memory area that needs to be backed up and the time for actual backup. The backup consists of copying regions of (1) affine memory

accesses –computed using affine functions– (2) nearly-affine accesses – computed using the tube formulation of regression functions, and (3) non-affine memory accesses –predicted using range information –. The *optimized skeleton* refers to the time spent on actual parallel execution, decentralized verification and live backup.

**Table 5.2:** Comparison on binary size

| Benchmark | Apollo (in KB) | | clang 3.4 (in KB) | gcc 4.8.2 (in KB) |
|---|---|---|---|---|
| | Binary size | Runtime (shared lib) | | |
| trmat | 140.7 | 566.7 | 23.6 | 28.1 |
| mperm | 153 | 566.7 | 27.8 | 28.1 |
| ispmatmat | 177.6 | 566.7 | 23.7 | 28.1 |
| ispmatvec | 140.7 | 566.7 | 23.7 | 23.9 |
| b+tree | 228.9 | 566.7 | 38.3 | 42.4 |
| filter | 200 | 566.7 | 13.2 | 9 |
| srad | 152.2 | 566.7 | 17.6 | 17.5 |
| susan corner | 342.6 | 566.7 | 43.2 | 43.2 |
| susan edge | 395.8 | 566.7 | 43.2 | 43.2 |
| backprojection | 145.5 | 566.7 | 18.2 | 18.4 |
| pfa-interp1 | 160.8 | 566.7 | 18.1 | 18.6 |

Table 5.2 compares the binary size of each benchmark generated by Apollo, against the sizes of the binaries generated by compilers gcc and clang. On average, the binary size produced by Apollo is 6.7 times larger than the one of clang and 6.4 times larger than the one of gcc. The increase in size of the binary is due to the presence of multiple skeletons. Also, each skeleton in LLVM-IR is encoded inside the binary, which is later compiled at runtime using the LLVM JIT compiler. The runtime is compiled as a shared object (SO) file. Apollo also requires some standard LLVM libraries for the JIT.

## 5.14 Summary

Until now, the applicability of the polyhedral model on codes with dynamically-polyhedral or non-affine behavior was impossible due to unavailable crucial information during static compilation. Though the version of Apollo presented in Chapter 4 was able to optimize dynamically affine codes, it was still not able to handle non linear codes. This

Chapter detailed the extensions to our Apollo framework, which enables it to handle non linear codes. During instrumentation Apollo classifies each access as either affine, nearly-affine or non-affine. The classification is based on the correlation factor of the interpolated access function. The accesses which strictly follow the affine function are classified as affine. The accesses where the correlation factor is greater than 0.9 are classified as nearly-affine and the rest of the accesses, where the correlation factor is less than 0.9 are classified as non-affine.

When constructing the speculative polyhedral model, the affine memory accesses and scalar accesses are treated in a similar manner as explained in Chapter 4. For the nearly-affine memory accesses, a regression tube is computed by adding a positive offset and negative offset to the corresponding regression function. This regression tube is then encoded to the dependence polyhedron using linear inequalities. The non-linear memory accesses are ignored while computing the dependence polyhedron. However, additional checks are done to ensure that these accesses do not violate any dependence. A range information is computed for each non-linear memory access, which later will be used by the backup module and the verification module. For the non-linear scalars and nearly-linear scalars, additional dependence constraints are added to the dependence polyhedron to ensure that the scalar variables are loaded accordingly. For the nearly-linear loops, speculative bounding hyperplanes are constructed, such that the region between the bounding planes can be optimized using the polyhedral model. The rest of the iterations follow the original sequential iteration. A range information is computed for the non-linear loops, which is then used as a substitute for the bounding hyperplanes.

The backup module uses the affine functions, regression functions and the range information to perform the backup. For affine write accesses, the backed-up region is computed based on the corresponding affine function and the chunk bounds. For the nearly affine write accesses, the backed-up region is computed based on the regression tube and the chunk bounds. For non affine accesses, the backup is only performed

if the range is less than three times the expected backup. The latter heuristic helps Apollo to avoid backing up huge regions of memory which will be sparsely used by the optimized chunk, thereby reducing memory overheads.

During the optimized execution, all the non predicted accesses are backed-up live per thread in a de-centralized manner. Additional checks are also done to ensure that these accesses do not violate any predicted access. If a violation occurs, the rollback flag is set and the thread execution is squashed. At the end of the optimized chunk execution, a centralized verification system checks for cross thread dependence violations. Thanks to eager verification mechanisms employed in Apollo, the chance of a dependence violation at this stage is very less. If the rollback flag is set, the restore module revives the safe state of the system, and employs the original skeleton to overcome the faulty region. This is then followed by the instrumentation skeleton, and the whole cycle is continued. The speculative execution is then continued till the outermost loop bound is reached, at which point, the runtime relinquishes the control of the loop and transfers the control back to the program.

Using the non linear approach Apollo is able to extend the scope of codes which can be handled using the polyhedral model. Thanks to the dynamic and speculative approach, and by relaxing the required constraints, we show that the applicability of the polyhedral model can be extended to codes with a dynamic behavior that do not naturally fit the model. Thus, this work effectively increases the domain of codes that can take advantage of complex optimizing transformations at runtime to include codes which exhibit non-linear memory behaviors. As shown in the experiments, aggressive speculative polyhedral transformations, including parallelization, along with our verification system which handles most of the verification in a distributed manner, can yield very interesting speed ups. Moreover, Apollo also highlights the fact that codes may exhibit interesting optimization opportunities depending on the processed input. This work opens to investigations related to new memory allocation and access strategies that may be better adapted to code parallelization and optimization, either in software

or hardware.

The next chapter concludes the manuscript and points to the possible future work on the framework.

*There is no real ending. It's just the place where you stop the story.*

Frank Herbert

*I do the very best I know how - the very best I can; and I mean to keep on doing so until the end.*

Abraham Lincoln

*Lessons gained, ideas secured, is mankind's eternal nourishment.*

Asif

# 6 Conclusion

## 6.1 Contributions

The efficient utilization of multi core and many core architectures is still an open problem. Parallelization and data locality are the key factors to be targeted for yielding high performance. Parallelization allows one to explore the processing power of all the cores within a single program. Data locality optimizations will help in better using the memory hierarchy, which is critical for performance since the gap between the processing power and the memory speed is growing at an exponential rate.

Automatic compiler can solve this problem efficiently. They can abstract the optimization task from the developer, allowing him/her to concentrate on the program logic. The polyhedral model is a well known mathematical model to optimize loop nests. However, current compilers are limited to apply it on statically analyzable codes. Moreover, due the limitation imposed by the model – the strict adherence to affine functions –, the application of this model is limited to a small set of compute

intensive codes.

The main objective of this dissertation is to extend the applicability of the polyhedral model to a wider a class of codes. The ideas are implemented in an indigenous compiler framework called APOLLO (Automatic speculative POLyhedral Loop Optimizer). Apollo combines the power of the polyhedral model and Thread Level Speculation (TLS).

The basic version of Apollo extends the applicability of the polyhedral model to dynamic codes. To our knowledge, Apollo is the first framework which is capable of dynamically predicting a polyhedral optimization (backed by Pluto), based on the dynamic code characteristics. Apollo is able to target any kind of loop nest, possibly imperfect, and can apply the polyhedral optimizations dynamically. In addition to applying the polyhedral model dynamically, Apollo contains various mechanisms to reduce the runtime overheads. Partial instrumentation allows Apollo to reduce the instrumentation overhead. By using Pluto as a dynamic scheduler, Apollo is able to predict an optimizing transformation purely based on the runtime behavior of the program. The backup mechanism is fine tuned to reduce the memory consumption when the memory accesses are sparse. The de-centralized verification promotes code scaling in addition to reducing the verification overhead. A Just-In-Time (JIT) compiler further optimizes the code, by extracting the invariants representing the code transformation and linear functions.

The non linear version of Apollo adapts the polyhedral model to handle non linear codes. To our knowledge, this is the first framework which uses the polyhedral model on non-trivial codes with generic non affine memory accesses and loop bounds. Handling non linear entities allows Apollo to optimize function calls, possibly recursive occurring inside the target loop nest. The framework is capable of detecting and classifying non linear memory accesses and loop bounds by instrumenting the code. The instrumentation information, including the non linear entities, are encoded in the polyhedral model with the help of regression. Additional runtime checks are used to

validate the model in addition to the verification of linear functions. A smart backup system is able to speculatively approximate the write regions of a given chunk, even before the chunk execution begins. Based on the estimation, the backup is performed before the chunk is launched. For the non predicted accesses, the backup is performed individually at runtime. Live backups are much more expressive than backups before launching the chunk. The ability of the system to approximately predict the write regions beforehand, helps Apollo to avoid huge runtime overheads. A novel two pronged verification system employs both eager and lazy verification methodologies in the form of a de-centralized and centralized verification system. The de-centralized system is used to verify the predicted accesses when the target code is running and does not induce the overhead of thread synchronizations. Inter thread dependencies are resolved using the centralized verification system, which is executed after the speculative chunk finishes its execution. It is worth nothing that only accesses which cannot be verified by the de-centralized system are subjected to central verification. Thanks to the pre-chunk execution verifications and eager verifications, the chance of a rollback being triggered in the lazy verification system is very small.

## 6.2 Future work

One promising future direction of Apollo is to extend it to multiple platforms. Currently the framework is only evaluated on X86_64 platforms. ARM based processors are interesting targets due to their wide use. The challenges in this direction involves not only improving the performance but also respecting the power constraints. A mechanism estimating the amount of available parallel work can be employed to determine the number of threads to be used and to decide whether to power up a dormant core. The speculative performance gain prediction mechanism described in Chapter 5 can be fine tuned to control the amount of speculation itself.

Many-core architecture is another interesting target for Apollo. If there is not enough work to occupy all the computing cores, Apollo could utilize these cores for

running code versions where the scalar values are also predicted. In the current version of Apollo, if the scalar value cannot be predicted, it is marked as a dependence. However, if it fits a nearly affine model, each version corresponding to one scalar value along the direction of the regression tube can be launched. When the correct scalar value is determined, the modifications of the corresponding thread can be committed to the global state. In order avoid thread explosion and memory explosion, this strategy should only be applied if the regression co-efficient is very high. Another possible improvement targeting many-core architectures is to explore the transformation space in Apollo. Automated schedulers typically use heuristics to determine the optimal transformation, but the resulting transformation may not be the best for a given architecture and environment. Instead of selecting one possible transformation, Apollo could select multiple transformations and then apply each such transformation, possibly on different chunks. Then, based on the chunk execution time, Apollo can be used to determine the best version, and use it for the rest of the chunks. The architecture of Apollo can be extended easily to support execution of multiple chunk simultaneously. The selected transformation can be dumped to a persistent transformation cache for reuse across different program executions. Instead of using heavy techniques like machine learning to match the program to the transformation cache, one could simply walk this transformation cache and find a match based on a dependence encoding which is independent of the transient program characteristics such as base address of arrays.

Apollo can be extended to support data re-mapping for non-linear codes. This can be done either virtually or by physical remapping. Remapping memory may require one to observe the code regions outside the target loop nest to determine the remapping order. This technique is useful when the mapping is reused. Data locality can be improved on non linear codes with this approach, as the data is remapped in the order of computations. However, one should take into consideration the cost of mapping, copying in and copying out data.

Although current Apollo can handle codes with recursion, the recursive functions are treated as a single block of statement. However, recursive patterns such as tail recursion can be converted to a while loop. Apollo can then handle this code, even if the loop bounds are non linear.

GP-GPUs is a challenging target for any speculative system. Typically, the conditionals inside the code kernels should be less for these systems to obtain optimal performance. The major challenge of Apollo here is to redesign the verification system, which induces a lot of conditionals into the code kernels. One possibility is to handle codes where the verification can be split from the computation. The host device can perform verifications in this case, and only the verified computations are performed in GPU's. Remapping is also interesting for GP-GPUs as the system inherently requires to copy in and copy out data.

Heterogeneous computing can be facilitated by allocating threads to different work units based on their processing power. The processing power can be estimated empirically either statically or dynamically. The dispatcher manager of Apollo can be used to map the threads to processors and to control the amount of work. This can also be used to share work between CPU's and GP-GPU's or integrated GPU's. Distributed computing is another interesting target for Apollo. The dispatcher manager could assign the big slices (typically the outermost loop slices) to distributed nodes, while still parallelizing the slices inside each distributed node. Thanks to the de-centralized verification, the affine accesses do not require any communication. For the non-affine accesses, the multi staged verification should be extended by adding a layer which would act as central verification system per distributed node. Once each node is verified, the verification status, along with the summarized information of the accesses should be sent to the master node, which would then perform the final global verification.

The code generation part of Apollo can be improved to provide support to a wider class of transformations. For example, as of now, Apollo is not able to perform tiling. Even though this can be done by adding support to the skeletons, a generic alternative

approach can be employed. One such possibility is to generate a set of code snippets representing each statement and loops, and then re-arranging and merging them at runtime based on the transformation. An alternative approach would be to entirely use dynamic code generation and to optimize it using JIT.

The architecture of Apollo is highly modular, which enables reusing different modules of Apollo elsewhere. The instrumentation module for example, could be used to study the characteristics of any code. Apollo can dump the instrumentation data to disk (using a library DLOG [84]), and thus can even be used for off-line analysis.

The regression model used in this work can be used in other fields. One such possible field is value prediction. The value prediction can be used at a very low level to predict the value of a load instruction before it is completed, or at a high level to speculatively execute a function by predicting its input values. It can also be used for branch prediction. Speculatively executing a function by offloading it to a thread is particularly interesting for the regression tubes. If the function has enough computations and is safe, then the tube based approach can almost entirely hide the cost of the function call, subjected to the condition that there are enough available cores.

The non-linear multi staged backup system can be applied to other TLS systems. Typical TLS systems perform live backup, resulting in multiple copies of the same location. The other option is to backup with central communication. The former requires much more memory and the latter requires inter thread synchronizations. Both of these are performance penalizing. Using the multi staged backup can reduce both the memory requirement and the synchronization requirement.

# References


[1] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part i," *Commun. ACM*, vol. 3, pp. 184–195, Apr. 1960.

[2] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, pp. 46–55, Jan. 1998.

[3] A. Sukumaran-Rajam, J. M. Martinez, W. Wolff, A. Jimborean, and P. Clauss, "Speculative Program Parallelization with Scalable and Decentralized Runtime Verification," in *Runtime Verification* (B. Bonakdarpour and S. A. Smolka, eds.), vol. 8734 of *LNCS*, (Toronto, Canada), pp. 124–139, Springer, Sept. 2014.

[4] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu, "The Parboil technical report," tech. rep., IMPACT Technical Report, IMPACT-12-01, University of Illinois, at Urbana-Champaign, 2012.

[5] C. Bienia and K. Li, "Parsec 2.0: A new benchmark suite for chip-multiprocessors," in *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[6] 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 *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, (Washington, DC, USA), pp. 44–54, IEEE Computer Society, 2009.

[7] H. L. A. van der Spek, E. M. Bakker, and H. A. G. Wijshoff, "SPARK00: A benchmark package for the compiler evaluation of irregular/sparse codes," *CoRR*, vol. abs/0805.3897, 2008.

[8] L. N. Pouchet, "Polybench: The polyhedral benchmark suite," 2010.

[9] A. Sukumaran-Rajam, L. E. Campostrini, M. Juan Manuel, and P. Clauss, "Speculative Runtime Parallelization of Loop Nests: Towards Greater Scope and Efficiency," in *20th International Workshop on High-level Parallel Programming Models and Supportive Environments, held in conjunction with 29th IEEE International Parallel & Distributed Processing Symposium*, (Hyderabad, India), May 2015.

[10] J. Cohen, P. Cohen, S. G. West, and L. S. Aiken, *Applied Multiple Regression/-Correlation Analysis for the Behavioral Sciences*. Routledge, Aug. 2002.

[11] 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 *IISWC*, pp. 44–54, IEEE, 2009.

[12] G. Fursin and O. Temam, "Collective optimization: A practical collaborative approach," *ACM Trans. Archit. Code Optim.*, vol. 7, pp. 20:1–20:29, Dec. 2010.

[13] K. Barker, T. Benson, D. Campbell, D. Ediger, R. Gioiosa, A. Hoisie, D. Kerbyson, J. Manzano, A. Marquez, L. Song, N. Tallent, and A. Tumeo, *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual*. Pacific Northwest National Laboratory and Georgia Tech Research Institute, December 2013. http://hpc.pnnl.gov/projects/PERFECT/.

[14] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *PLDI '08*, ACM, 2008.

[15] P. Feautrier and C. Lengauer, "Polyhedron model," in *Encyclopedia of Parallel Computing*, pp. 1581–1592, Springer US, 2011.

[16] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *J. ACM*, vol. 14, pp. 563–590, July 1967.

[17] P. Quinton, "The systematic design of systolic arrays," in *Centre National De Recherche Scientifique on Automata Networks in Computer Science: Theory and Applications*, (Princeton, NJ, USA), pp. 229–260, Princeton University Press, 1987.

[18] U. K. Banerjee, *Dependence Analysis for Supercomputing*. Norwell, MA, USA: Kluwer Academic Publishers, 1988.

[19] X. Kong, D. Klappholz, and K. Psarris, "The i test: An improved dependence test for automatic parallelization and vectorization," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, pp. 342–349, July 1991.

[20] W. Pugh, "A practical algorithm for exact array dependence analysis," *Commun. ACM*, vol. 35, pp. 102–114, Aug. 1992.

[21] M. Wolfe and C. W. Tseng, "The power test for data dependence," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, pp. 591–601, Sept. 1992.

[22] P. Feautrier, "Some efficient solutions to the affine scheduling problem, part 2 : multidimensional time," *International Journal of Parallel Programming*, vol. 21, no. 6, 1992.

[23] P. Feautrier, "Some efficient solutions to the affine scheduling problem. part ii. multidimensional time.," *International Journal of Parallel Programming*, vol. 21, no. 6, pp. 389–420, 1992.

[24] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991.

[25] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam, "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies," *International Journal of Parallel Programming*, vol. 34, pp. 261–317, June 2006. Classement CORE : A.

[26] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Commun. ACM*, vol. 29, pp. 1184–1201, Dec. 1986.

[27] P. Feautrier, "Array expansion," in *Proceedings of the 2nd International Conference on Supercomputing*, ICS '88, (New York, NY, USA), pp. 429–441, ACM, 1988.

[28] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, 1991.

[29] A. Cohen, S. Girbal, and O. Temam, "A polyhedral approach to ease the composition of program transformations," in *Euro-Par 2004 Parallel Processing*, pp. 292–303, Springer, 2004.

[30] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam, "Putting polyhedral transformations to work," in *LCPC'16 Intl. Workshop on Languages and Compilers for Parallel Computers, LNCS 2958*, (College Station), pp. 209–225, Oct. 2003.

[31] Pluto, "An automatic parallelizer and locality optimizer for multicores." http://pluto-compiler.sourceforge.net.

[32] P. Feautrier, "Toward automatic partitioning of arrays on distributed memory computers," in *Proceedings of the 7th international conference on Supercomputing*, pp. 175–184, ACM, 1993.

[33] P. Feautrier, "Parametric integer programming," *RAIRO Recherche Opérationnelle*, vol. 22, no. 3, pp. 243–268, 1988.

[34] P. Clauss and V. Loechner, "Parametric analysis of polyhedral iteration spaces," *JOURNAL OF VLSI SIGNAL PROCESSING*, vol. 19, pp. 179–194, 1998.

[35] A. I. Barvinok, "Computing the volume, counting integral points, and exponential sums," *Discrete & Computational Geometry*, vol. 10, pp. 123–141, 1993.

[36] "PoCC, the Polyhedral Compiler Collection, version 1.3." http://pocc.sourceforge.net.

[37] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *Mathematical Software - ICMS 2010* (K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, eds.), vol. 6327 of *Lecture Notes in Computer Science*, pp. 299–302, Springer, 2010.

[38] C. Bastoul, "Extracting polyhedral representation from high level languages," tech. rep., LRI, Paris-Sud University, 2008. Related to the Clan tool.

[39] C. Bastoul, "Openscop: A specification and a library for data exchange in polyhedral compilation tools," tech. rep., Paris-Sud University, France, September 2011.

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

[41] I. Fassi and P. Clauss, "XFOR: Filling the Gap between Automatic Loop Optimization and Peak Performance," in *14th International Symposium on Parallel and Distributed Computing* (IEEE, ed.), (Limassol, Cyprus), June 2015.

[42] P. Clauss, I. Fassi, and A. Jimborean, "Software-controlled processor stalls for time and energy efficient data locality optimization," in *XIVth International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS 2014, Agios Konstantinos, Samos, Greece, July 14-17, 2014*, pp. 199–206, 2014.

[43] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "Polymage: Automatic optimization for image processing pipelines," *SIGARCH Comput. Archit. News*, vol. 43, pp. 429–443, Mar. 2015.

[44] C. Bastoul, "Generating loops for scanning polyhedra," Tech. Rep. 2002/23, PRiSM, Versailles University, 2002. Related to the CLooG tool.

[45] T. Grosser, A. Größlinger, and C. Lengauer, "Polly - performing polyhedral optimizations on a low-level intermediate representation.," *Parallel Processing Letters*, vol. 22, no. 4, 2012.

[46] S. Pop, G.-A. Silber, A. Cohen, C. Bastoul, S. Girbal, and N. Vasilache, "GRAPHITE: Polyhedral analyses and optimizations for GCC," Tech. Rep. A/378/CRI, Centre de Recherche en Informatique, École des Mines de Paris, Fontainebleau, France, 2006. Contribution to the GNU Compilers Collection Developers Summit 2006 (GCC Summit 06), Ottawa, Canada, June 28–30, 2006.

[47] M. Griebl and C. Lengauer, "The loop parallelizer loopo," in *Proc. Sixth Workshop on Compilers for Parallel Computers, volume 21 of Konferenzen des Forschungszentrums Jülich*, pp. 311–320, Forschungszentrum, 1996.

[48] L.-N. Pouchet, C. Bastoul, and A. Cohen, "LetSee: the LEgal Transformation SpacE Explorator." Third International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES'07), L'Aquila, Italia, July 2007. Extended abstract, pp 247–251.

[49] C. Chen, J. Chame, and M. Hall, "Chill: A framework for composing high-level loop transformations," tech. rep., 2008.

[50] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for cuda," *ACM Trans. Archit. Code Optim.*, vol. 9, pp. 54:1–54:23, Jan. 2013.

[51] R. Lethin, P. Mattson, E. Shweitz, A. Leung, V. Litvinov, M. Engle, and C. Garrett, "R-stream 3.0: Technologies for high level embedded application mapping," in *Proceedings of the 8th Annual High Performance Embedded Computing (HPEC) Workshops, Lexington, MA, USA, September*, Sep 2004.

[52] E. Schweitz, R. Lethin, A. Leung, and B. Meister, "A parametric high level compiler," in *Proceedings of the High Performance Embedded Computing Workshop (HPEC), Lexington, MA, USA, September*, Sep 2006.

[53] J. Dollinger and V. Loechner, "Adaptive runtime selection for GPU," in *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*, pp. 70–79, 2013.

[54] J.-F. Dollinger and V. Loechner, "CPU+GPU Load Balance Guided by Execution Time Prediction," in *Fifth International Workshop on Polyhedral Compilation Techniques (IMPACT 2015)*, (Amsterdam, Netherlands), Jan. 2015.

[55] J. F. Martínez and J. Torrellas, "Speculative synchronization: Applying thread-level speculation to explicitly parallel applications," in *In Proceedings of the Tenth Symposium on Architectural Support for Programming Languages and Operating Systems*, 2002.

[56] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas, "iwatcher: Efficient architectural support for software debugging," in *31st Int. Symp. on Computer Architecture (ISCA)*, pp. 224–237, 2004.

[57] M. Prvulovic and J. Torrellas, "ReEnact: using thread-level speculation mechanisms to debug data races in multithreaded codes," *Computer Architecture News*, vol. 31, pp. 110–121, May 2003.

[58] Intel, "Intel® 64 and ia-32 architectures software developer's manual." https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set\-reference-manual-325383.pdf, 2015.

[59] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, a. gara, G. Chiu, P. Boyle, N. Chist, and C. Kim, "The ibm blue gene/q compute chip," *IEEE Micro*, vol. 32, pp. 48–60, Mar. 2012.

[60] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Rock: A high-performance sparc cmt processor," *IEEE Micro*, vol. 29, pp. 6–16, Mar. 2009.

[61] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation," *SIGARCH Comput. Archit. News*, vol. 28, pp. 1–12, May 2000.

[62] J. Renau, K. Strauss, L. Ceze, W. Liu, S. Sarangi, J. Tuck, and J. Torrellas, "Thread-level speculation on a cmp can be energy efficient," in *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, (New York, NY, USA), pp. 219–228, ACM, 2005.

[63] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry, "The stampede approach to thread-level speculation," *ACM Trans. Comput. Syst.*, vol. 23, pp. 253–300, Aug. 2005.

[64] L. Rauchwerger, N. M. Amato, and D. A. Padua, "A scalable method for run-time loop parallelization," *IJPP*, vol. 26, pp. 26–6, 1995.

[65] J. a. Barreto, A. Dragojevic, P. Ferreira, R. Filipe, and R. Guerraoui, "Unifying thread-level speculation and transactional memory," in *Proceedings of the 13th International Middleware Conference*, Middleware '12, (New York, NY, USA), pp. 187–207, Springer-Verlag New York, Inc., 2012.

[66] L. Rauchwerger and D. 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, (New York, NY, USA), pp. 218–232, ACM, 1995.

[67] F. Dang, H. Yu, and L. Rauchwerger, "The R-LRPD test: speculative parallelization of partially parallel loops," in *Parallel and Distributed Processing Symposium, IPDPS 2002*, April 2002.

[68] A. Jimborean, *Adapting the polytope model for dynamic and speculative parallelization*. PhD thesis, Université de Strasbourg, Sept. 2012.

[69] A. Jimborean, L. Mastrangelo, V. Loechner, and P. Clauss, "VMAD: An Advanced Dynamic Program Analysis and Instrumentation Framework," in *Proceedings of the 21st International Conference on Compiler Construction*, CC'12, (Berlin, Heidelberg), pp. 220–239, Springer-Verlag, 2012.

[70] S. Aldea, D. Llanos, and A. González-Escribano, "Support for Thread-Level Speculation into OpenMP," in *OpenMP in a Heterogeneous World* (B. Chapman, F. Massaioli, M. Müller, and M. Rorro, eds.), vol. 7312 of *Lecture Notes in Computer Science*, pp. 275–278, Springer Berlin Heidelberg, 2012.

[71] A. Jimborean, P. Clauss, J. M. Martinez, and A. Sukumaran-Rajam, "Online Dynamic Dependence Analysis for Speculative Polyhedral Parallelization," in *Euro-Par 2013*, vol. 8097 of *LNCS*, (Aachen, Germany), pp. 191–202, Springer, Aug 2013.

[72] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas, "POSH: a TLS compiler that exploits program structure," in *PPoPP '06*, ACM, 2006.

[73] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar, "Speculative thread decomposition through empirical optimization," in *PPoPP '07*, ACM, 2007.

[74] E. Raman, N. Vachharajani, R. Rangan, and D. I. August, "Spice: speculative parallel iteration chunk execution," in *CGO '08*, ACM, 2008.

[75] M. Ravishankar, J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan, "Code generation for parallel execution of a class of irregular loops on distributed memory systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), IEEE Computer Society Press, 2012.

[76] E. D. Berger and B. G. Zorn, "DieHard: probabilistic memory safety for unsafe languages," in *in PLDI '06*, pp. 158–168, ACM Press, 2006.

[77] G. Novark and E. D. Berger, "DieHarder: Securing the heap," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, (New York, NY, USA), pp. 573–584, ACM, 2010.

[78] M. Griebl and J. Collard, "Generation of synchronous code for automatic parallelization of while loops," in *Euro-Par '95 Parallel Processing, First International Euro-Par Conference, Stockholm, Sweden, August 29-31, 1995, Proceedings*, pp. 315–326, 1995.

[79] J.-F. Collard, "Automatic parallelization of while-loops using speculative execution," *Int. J. Parallel Program.*, vol. 23, pp. 191–219, Apr. 1995.

[80] S. J. Geuns, M. J. Bekooij, T. Bijlsma, and H. Corporaal, "Parallelization of while loops in nested loop programs for shared-memory multiprocessor systems," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2011*, pp. 1–6, IEEE Computer Society, March 2011.

[81] C. Lengauer and M. Griebl, "On the parallelization of loop nests containing while loops," Tech. Rep. MIP-9414, Universität Passau (DE), 1994.

[82] J.-F. Collard, D. Barthou, and P. Feautrier, "Fuzzy Array Dataflow Analysis," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pp. 92–101, ACM, 1995.

[83] A. Venkat, M. Shantharam, M. Hall, and M. M. Strout, "Non-affine extensions to polyhedral code generation," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, (New York, NY, USA), pp. 185:185–185:194, ACM, 2014.

[84] "Dlog: A libary to dump and visualize data." http://github.com/intelaravind/DLOG.

# Index

UNIVERSITÉ DE STRASBOURG Aravind SUKUMARAN RAJAM msii ÉCOLE DOCTORALE

# Beyond the Realm of the Polyhedral Model: Combining Speculative Program Parallelization with Polyhedral Compilation


## Summary

In this thesis, we present our contributions to APOLLO (Automatic speculative POLyhedral Loop Optimizer), which is an automated compiler combining Thread Level Speculation (TLS) and the polyhedral model to optimize codes on the fly. By doing partial instrumentation at runtime, and subjecting it to interpolation, Apollo is able to construct a speculative polyhedral model dynamically. The speculative model is then passed to Pluto -a static polyhedral scheduler-. Apollo then selects one of the statically generated code optimization skeletons and instantiates it. The runtime continuously monitors the code for any dependence violation in a decentralized manner. Another important contribution of this thesis is our extension of the polyhedral model to codes exhibiting a non linear behavior. Thanks to the dynamic and speculative context offered by Apollo, non-linear behaviors are either modeled using linear regression hyperplanes forming tubes, or using ranges of reached values. Our approach enables the application of polyhedral transformations to non-linear codes thanks to an hybrid centralized-decentralized speculation verification system.

## Résumé

Dans cette thèse, nous présentons nos contributions à Apollo (Automatic speculative POLyhedral Loop Optimizer), qui est un compilateur automatique combinant la parallélisation spéculative et le modèle polyédrique, afin d'optimiser les codes à la volée. En effectuant une instrumentation partielle au cours de l'exécution, et en la soumettant à une interpolation, Apollo est capable de construire un modèle polyédrique spéculatif dynamiquement. Ce modèle spéculatif est ensuite transmis à Pluto, qui est un ordonnanceur polyédrique statique. Apollo sélectionne ensuite un des squelettes d'optimisation de code générés statiquement, et l'instancie. La partie dynamique d'Apollo surveille continuellement l'exécution du code afin de détecter de manière décentralisée toute violation de dépendance. Une autre contribution importante de cette thèse est notre extension du modèle polyédrique aux codes exhibant un comportement non-linéaire. Grâce au contexte dynamique et spéculatif d'Apollo, les comportements non-linéaires sont soit modélisés par des hyperplans de régression linéaire formant des tubes, soit par des intervalles de valeurs atteintes. Notre approche permet l'application de transformations polyédriques à des codes non-linéaires grâce à un système de vérification de la spéculation hybride, combinant vérifications centralisées et décentralisées.