Static and Dynamic Methods of Polyhedral Compilation for an Efficient Execution in Multicore Environments

Benoit Pradelle

To cite this version:

Benoit Pradelle. Static and Dynamic Methods of Polyhedral Compilation for an Efficient Execution in Multicore Environments. Hardware Architecture [cs.AR]. Université de Strasbourg, 2011. English. NNT : . tel-00733856

HAL Id: tel-00733856

https://theses.hal.science/tel-00733856

Submitted on 19 Sep 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE STRASBOURG

N° d'ordre :

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

---

UdS – INSA – ENGEES

THÈSE

présentée pour obtenir le grade de

Docteur de l'Université de Strasbourg

Discipline : Informatique

par

Benoît Pradelle

Static and Dynamic Methods of Polyhedral Compilation for an Efficient Execution in Multicore Environments

Méthodes Statiques et Dynamiques de Compilation Polyédrique pour l'Exécution en Environnement Multi-Cœurs

Soutenue publiquement le 20 décembre 2011

Membres du jury:

Directeur de thèse :	Philippe Clauss, Professeur, UdS, Strasbourg
Rapporteur :	Albert Cohen, Directeur de Recherche, INRIA - ENS, Paris
Rapporteur :	Sanjay V. Rajopadhye, Associate Professor, CSU, Fort Collins, USA
Examinateur :	William Jalby, Professeur, UVSQ - Exascale, Versailles
Examinateur :	Jean Christophe Beyler, Senior Software Engineer, Intel, Versailles

---

Laboratoire des Sciences de l'Image, de l'Informatique, et de la Télédétection **UMR 7005**

«Le véritable voyage de découverte ne consiste pas à chercher de nouveaux paysages, mais à avoir de nouveaux yeux.» — Marcel Proust

Contents

Résumé en Français **15**

1 Introduction **39**

2 Basic Concepts and Related Work **43**

2.1 Static Methods 44
2.1.1 Polyhedral Parallelization 44
2.1.2 Binary Code Parallelization and Rewriting 45
2.2 Static-Dynamic Collaborative Methods 46
2.2.1 Iterative Compilation 46
2.2.2 Hybrid Code Selection 47
2.3 Dynamic Methods 48
2.3.1 Dynamic Code Selection 48
2.3.2 The Inspector/Executor Model 49
2.3.3 Thread-Level Speculation 50
2.3.4 Binary Code Parsing 51
2.3.5 Inline Code Generation and Transformation 52
2.4 The Polyhedral Model 53
2.4.1 Mathematical Background and Notations 53
2.4.2 Scope and SCoP 55
2.4.3 Statement and Iteration Vector 56
2.4.4 Iteration Domain 56
2.4.5 Access Functions 57
2.4.6 Schedule 58
2.4.7 Data Dependence 60
2.4.8 Transformation 62
2.4.9 Code Generation 65
2.4.10 Extensions to the Polyhedral Model 65
2.4.11 Tools 66
2.5 Conclusion 67

3 Binary Code Parallelization **69**

3.1 Introduction 69
3.2 Decompiling x86-64 Executables 71
3.2.1 Basic Analysis 71
3.2.2 Memory Access Description 71

3.2.3 Induction Variable Resolution 72
3.2.4 Tracking Stack Slots 73
3.2.5 Branch Conditions and Block Constraints 76
3.3 Polyhedral Parallelization 78
3.3.1 Memory Accesses 78
3.3.2 Operations 80
3.3.3 Scalar References 82
3.3.4 Parallelization 84
3.3.5 Reverting the Outlining 84
3.3.6 Live-in Registers 85
3.3.7 Live-out Registers 86
3.3.8 Implementation 86
3.4 Evaluation 88
3.4.1 Loop Coverage 88
3.4.2 Binary-to-binary vs. Source-to-source 89
3.4.3 Binary-to-binary vs. Hand-Parallelization 92
3.5 Extension to Complex Codes 94
3.5.1 Extension to Parametric Codes 94
3.5.2 Extension to Polynomial Codes 96
3.5.3 In-Place Parallelization 99
3.6 Related Work 103
3.6.1 Decompilation and Address Expressions 103
3.6.2 Parallelization and Transformation Strategy 104
3.7 Conclusion and Perspectives 104

4 Code Version Selection **107**
4.1 Introduction 107
4.2 Selection Framework Overview 108
4.3 Generating Different Code Versions 109
4.4 Profiling the Code Versions 110
4.4.1 Strategy 1 112
4.4.2 Strategy 2 116
4.4.3 Parametric Ranking Table 122
4.5 Runtime Selection 122
4.5.1 Iteration Count Measurement 124
4.5.2 Load Balance 125
4.5.3 Predicting the Execution Time 126
4.5.4 Discussion 126
4.6 Experiments 127
4.6.1 Dynamic Scheduling of Regular Codes 130
4.6.2 Execution Context Characteristics 132
4.6.3 Execution Time Gains 134
4.6.4 Accuracy 134
4.7 Conclusion and Perspectives 137

5 Speculative Parallelization 139
5.1 Introduction . . . 139
5.2 Overview . . . 140
5.2.1 Speculations . . . 140
5.2.2 General overview . . . 142
5.2.3 Evaluation Environment . . . 143
5.2.4 Chunking . . . 144
5.3 Online Profiling . . . 146
5.3.1 Inspector Profiling . . . 146
5.3.2 Profiling on a Sample . . . 147
5.3.3 Chosen Solution . . . 147
5.4 Dependence Construction . . . 148
5.5 Scheduling . . . 148
5.5.1 PLUTO . . . 148
5.5.2 Offline Profiling and Scheduling . . . 149
5.5.3 Generic Schedules . . . 150
5.5.4 Dependence Testing . . . 151
5.5.5 Chosen Solution . . . 152
5.6 Code Generation . . . 153
5.6.1 Runtime Compilation . . . 153
5.6.2 Hybrid Code Generation . . . 155
5.6.3 Static Code Generation . . . 156
5.6.4 Chosen Solution . . . 157
5.7 Speculation Verification . . . 157
5.7.1 Parallel Speculation Verification . . . 157
5.7.2 Verification Implementation . . . 159
5.7.3 Test Implementation . . . 160
5.7.4 Chosen Solution . . . 161
5.8 Commit and Rollback . . . 161
5.8.1 Transactions and Chunks . . . 162
5.8.2 `fork`-based Transactional System . . . 163
5.8.3 `memcpy`-based Transactional System . . . 165
5.8.4 Interrupting the Threads . . . 166
5.8.5 Rollback Strategies . . . 166
5.8.6 Chosen Solution . . . 167
5.9 Putting it all Together . . . 168
5.9.1 Overview . . . 168
5.9.2 Evaluation . . . 168
5.10 Conclusion and Perspectives . . . 169

6 Conclusion and Perspectives 171
6.1 Contributions . . . 171
6.2 Future work . . . 173

Bibliography 178

List of Figures

1 Aperçu du système de parallélisation. (1) Les nids de boucles analysables sont convertis en programmes C contenant seulement les accès mémoires et le code de contrôle. (2) Cette représentation intermédiaire est envoyée à un paralléliseur source-à-source. (3) Le programme parallèle résultant est complété avec les calculs sur les données puis recompilé avec un compilateur C classique. 16
2 Implémentation du mécanisme de parallélisation de programmes binaires. 19
3 Parallélisation avec CETUS ou PLUTO depuis 1) le code source original 2) le code source extrait du programme binaire. 20
4 Comparaison de notre système à une parallélisation manuelle et à l'état de l'art. 21
5 Schéma général du sélecteur de versions. 23
6 Temps d'exécution (en secondes) réel et prédit de toutes les versions du programme `2mm` dans tous les contextes testés, trié par le temps d'exécution réel. Sur Core i7 avec la stratégie 1. 28
7 Temps d'exécution (en secondes) réel et prédit de toutes les versions du programme `2mm` dans tous les contextes testés, trié par le temps d'exécution réel. Sur Core i7 avec la stratégie 1. 28
8 Les différentes étapes de notre paralléliseur spéculatif. 29
9 Boucle d'exemple (à gauche) et sa parallélisation spéculative (à droite). 32
10 Stratégie de ré-exécution spéculative. 33

2.1 Sample loop nest (left) and the iteration domain of S (right). 56
2.2 Sample loop nest with two statements. 59
2.3 A matrix-vector product. 62

3.1 Overview of the parallelization system. (1) Analyzable portions of the sequential binary program are raised to simple C loop nests only containing the correct memory accesses, and (2) sent to a back-end source-to-source parallelizer. (3) The resulting parallel loop nest is filled with actual computations, and compiled using a standard compiler. 70
3.2 Symbolic address expressions in a loop nest of depth two in binary code in SSA form: (a) after recursive register substitution, and (b) after induction variable resolution. Indentation is used to highlight the loop levels. "@" denotes address expressions. 74
3.3 Sample CFG (left), its corresponding reversed CFG (center), and post-dominator tree (right). 77

3.4 Sample code parallelization using PLUTO. 79
3.5 Matrix multiply as it is extracted from the binary code. 81
3.6 Matrix multiply after simplifying the memory accesses. 81
3.7 Matrix multiply after forward substitution. 82
3.8 Matrix multiply after scalar to array conversion. 83
3.9 Matrix multiply after transformation by PLUTO and semantics restoration. 85
3.10 General scheme of the implementation. 87
3.11 Redirecting the execution flow to the parallel loop nests (left to right). 87
3.12 Parallelization back-ends applied to 1) the original source code, and to 2) the binary code (kernel only). 91
3.13 Speedup comparison for three parallelization strategies. (initialization + kernel). 93
3.14 Original code. 95
3.15 Code as seen by the dependence analyzer. 95
3.16 Corresponding runtime tests. 95
3.17 Execution times (and speedups) for `swim` on the `train` dataset. 96
3.18 Execution times (and speedups) for `mgrid` on the `train` dataset. 98
3.19 Overwriting loop bounds with a new loop counter (left to right). 100

4.1 Framework overview. 109
4.2 Performance of a simple parallel version relatively to the data size. . . . 110
4.3 Performance of several versions relatively to the data size. 111
4.4 Performance of the parallelized kernels relatively to the number of threads used. 112
4.5 Code pattern. The profiling code is generated from this code pattern after some syntactic replacements. 114
4.6 Sample code. 115
4.7 Sample profiling code for the loop nest presented in Figure 4.6. 116
4.8 Sample parameterized loop nest. Increasing `M` does not increase the number of iterations. 117
4.9 Sample iteration domain. A square of 5×5 iterations can fit in it, so we can guarantee that each loop level executes at least 5 consecutive iterations. 117
4.10 The intersection of $\mathcal{P}$ and $\mathcal{P}'$ defines all the points such that B can fit in $\mathcal{P}$, if A is in $\mathcal{P}$. 118
4.11 Intersection of the translated polyhedron copies in 2D space. The intersection is not empty, the polyhedron can enclose the square. 119
4.12 Intersection of the translated polyhedron copies in 2D space. The intersection is empty: the square cannot be contained in the polyhedron. . . 119
4.13 Profiling code pattern for strategy 2. 121
4.14 Code sample to profile. 121
4.15 Profiling code generated from the code in Figure 4.14. 123
4.16 A loop nest (left) and its corresponding prediction nest (right). 125
4.17 Sample loop nest and the corresponding iteration domain where the iterations are grouped by thread. 125

4.18 Speedup of OpenMP dynamic over static on Core i7. 131
4.19 Speedup of Cilk over OpenMP static on Core i7. 131
4.20 Speedup of OpenMP dynamic over static on Opteron. 131
4.21 Speedup of Cilk over OpenMP static on Opteron. 131
4.22 Speedup of OpenMP dynamic over static on Phenom II. 131
4.23 Speedup of Cilk over OpenMP static on Phenom II. 131
4.24 Execution times for `adi` on Opteron (5 threads). Version 2 is the best one for the fifth dataset. 132
4.25 Execution times for `adi` on Core i7 (5 threads). Version 2 is the worst one for the fifth dataset. 132
4.26 Execution times for `gemver` on Core i7 (1 thread). Version 1 is the best one for the first dataset. 133
4.27 Execution times for `gemver` on Core i7 (8 threads). Version 1 is the worst one for the first dataset. 133
4.28 Execution times for `mgrid` on Phenom (4 threads). Version 4 is the best one for the last dataset, but is inefficient for the third dataset. 133
4.29 Execution time of every version of `2mm` in every execution context, sorted by actual execution time on Core i7 with strategy 1. 136
4.30 Execution time of every version of `2mm` in every execution context, sorted by actual execution time on Core i7 with strategy 2. 136
4.31 Execution time of every version of `jacobi-2d` in every execution context, sorted by actual execution time on Opteron with strategy 1. 136
4.32 Execution time of every version of `jacobi-2d` in every execution context, sorted by actual execution time on Opteron with strategy 2. 136
4.33 Execution time of every version of `gemver` in every execution context, sorted by actual execution time on Phenom II with strategy 1. 137
4.34 Execution time of every version of `gemver` in every execution context, sorted by actual execution time on Phenom II with strategy 2. 137

5.1 Main phases of the speculative system. 142
5.2 Sample loop nest (left) and its chunked counterpart (right). 144
5.3 Speedup of each code over its sequential version, relatively to the chunk size. 145
5.4 Compared compilation and execution times using GCC and LLC at the `O1` and `O3` optimization levels. 154
5.5 Sample code (left) and its speculative parallel counterpart (right). . . . 160
5.6 Sample loop (top), its transformed counterpart (bottom), and the corresponding memory accesses performed (right). 162
5.7 Execution of chunk i+1 cancelled (left) or commited (right). 163
5.8 Speedup over the sequential version when using a `fork`-based transactional system. 164
5.9 Speedup over the sequential version when using a `memcpy`-based transactional system. 165
5.10 Runtime strategy. 167

Remerciements

Je profite de ces quelques lignes de liberté pour remercier les personnes qui ont influencé ces travaux. Tout d'abord, merci à Philippe pour m'avoir soutenu et aidé durant toutes ces années. Encadrer une thèse n'est pas un vain mot pour toi et j'ai beaucoup appris à tes côtés. J'emporterai avec moi l'idée que l'innovation a besoin d'une touche de folie pour se révéler.

Merci à Alain et Vincent avec qui j'ai eu le très grand plaisir de travailler. Votre expérience et votre culture m'ont été d'une très grande aide au travers de vos conseils avisés. La plupart des travaux présentés dans cette thèse n'auraient pas été possibles sans vous. Merci à Alexandre, Alexandra, et Olivier qui m'ont supporté dans notre bureau. Merci à Stéphane, Julien, Romaric, Éric, Matthieu, Jean-François, et Étienne avec qui il est toujours très agréable de partager la journée. D'une façon générale, un grand merci à toute l'équipe ICPS pour l'ambiance formidable que chaque membre s'attache à faire régner. Merci également aux anciens de l'ICPS, en particulier à Jean-Christophe qui m'a ouvert la porte de l'équipe, et à Benoît «Homie» qui m'a initié aux secrets du modèle polyédrique. New York m'accompagne désormais.

Je remercie également tous mes amis strasbourgeois. Avec vous, j'ai toujours eu (y compris pendant la thèse) un endroit où trouver de la bonne humeur, de la chaleur, et des bons moments. Merci plus particulièrement à Emil pour toutes nos discussions passionnées et passionnantes qui m'aident à prendre du recul.

Je remercie ma famille qui m'a toujours soutenu et qui m'a laissé libre de trouver ma voie. Le calme et l'apaisement que je trouve à vos côtés me sont d'une grande aide.

Enfin, merci à toi, Annick, pour ton soutien sans faille et ta patience infinie durant toute cette thèse. Du haut de ton innocence, tu colores notre monde et, sans toi, tout cela n'aurait pas la même saveur.

Résumé en Français

Introduction

Jusqu'au début des années 2000, le développement logiciel était dans une situation très favorable. En effet, les fabricants de matériel augmentaient rapidement la fréquence des processeurs à chaque nouvelle génération. Si un programme était trop lent, il suffisait simplement d'attendre la prochaine génération de processeurs pour qu'il s'exécute plus rapidement. Dans un tel contexte, le rôle d'un compilateur se borne principalement à transcrire un programme écrit dans un langage de programmation haut-niveau vers une suite d'instructions pour le processeur. Les optimisations appliquées sur le programme ne sont généralement pas vitales.

Cependant, certaines limites physiques ont été atteintes et il n'est plus possible d'augmenter encore la fréquence des processeurs. Les fabricants ont alors décidé d'augmenter le nombre de cœurs de calcul par puce afin de continuer à augmenter la puissance théorique des processeurs. Ce changement d'approche impacte fortement le développement logiciel. Pour profiter des améliorations du matériel, les logiciels doivent désormais être ré-écrits dans une version parallèle.

Cependant, les développeurs sont en général très peu sensibilisés à la programmation parallèle et préfèrent souvent ne pas la pratiquer. Étant donné que le rôle d'un compilateur est d'adapter un programme pour qu'il puisse s'exécuter efficacement sur l'ordinateur ciblé, il est raisonnable de considérer que c'est en réalité au compilateur de réaliser cette parallélisation si le programmeur ne le fait pas.

C'est ce qui a motivé toutes les approches à la parallélisation automatique proposées jusque là. L'inconvénient de ces approches automatiques est qu'elles ne sont pas vraiment prêtes pour une utilisation à grande échelle. D'immenses progrès ont été réalisés dans le domaine mais les techniques existantes ne sont pas encore assez robustes. L'urgence provoquée par l'arrivée massive de matériel multi-cœur nous pousse à développer davantage ces techniques. Nous proposons ainsi d'étendre la parallélisation automatique dans trois directions principales.

Nous proposons tout d'abord un système permettant de paralléliser les programmes qui sont exprimés sous leur forme a priori finale : le code binaire. Ensuite, nous proposons un mécanisme à l'exécution qui permet de sélectionner une implémentation efficace d'un programme parallèle parmi plusieurs afin de bénéficier au mieux des spécificités du contexte dans lequel ce programme s'exécute. Enfin nous proposons une solution de parallélisation spéculative utilisant le modèle polyédrique afin de transformer et paralléliser efficacement les programmes même lorsque leur structure est complexe et empêche la parallélisation à la compilation.

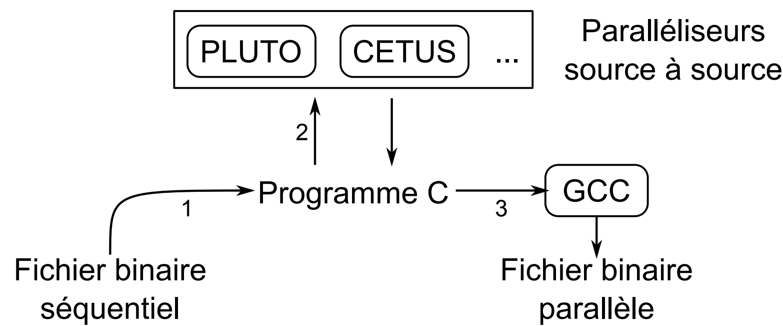

Figure 1: Aperçu du système de parallélisation. (1) Les nids de boucles analysables sont convertis en programmes C contenant seulement les accès mémoires et le code de contrôle. (2) Cette représentation intermédiaire est envoyée à un paralléliseur source-à-source. (3) Le programme parallèle résultant est complété avec les calculs sur les données puis recompilé avec un compilateur C classique.

Ces trois approches forment une réponse cohérente au problème de la parallélisation automatique des programmes. Elles utilisent tous les moments de la vie d'un programme depuis sa compilation jusqu'à son exécution afin d'exploiter autant que possible les opportunités de parallélisation. Enfin ces trois approches utilisent le modèle polyédrique dans des contextes nouveaux, permettant d'étendre son utilisation vers le code binaire, les contextes d'exécutions changeant, et les programmes dont la structure est difficilement analysable à la compilation.

Parallélisation de Code Binaire

Introduction

Alors que les outils de parallélisation automatique exploitent généralement le code source des programmes, il peut être intéressant de pouvoir paralléliser des programmes compilés. Cette approche permet la parallélisation d'applications dont le code source est inaccessible au moment de la compilation. Par exemple des programmes anciens dont le code source est perdu, des applications commerciales dont le code source est tenu secret, ou des librairies utilisées par un programme, peuvent ainsi être parallélisés. Ainsi, les programmes peuvent être parallélisés au moment de leur déploiement sur différentes machines d'exécution. De cette manière, les développeurs peuvent continuer à développer et compiler normalement leurs programmes séquentiels qui seront parallélisés lors de leur installation, par le système d'exploitation par exemple.

Nous proposons un système de parallélisation de code binaire à trois étapes. La première étape consiste à décompiler le programme binaire pour en extraire une représentation intermédiaire de ses nids de boucles affines. Cette représentation intermédiaire doit être suffisamment précise pour permettre leur parallélisation. Lorsque cette parallélisation à été réalisée, le programme est alors recompilé en un programme binaire parallèle. Un aperçu de notre système est présenté en Figure 1. Comme illustré dans la figure, notre système utilise un paralléliseur source-à-source pour réaliser la paral-

lélisation des nids de boucles. Cette capacité lui permet par exemple d'appliquer des transformations polyédriques sur les programmes binaires.

Décompilation et Analyse

La première tâche réalisée par notre système de parallélisation de code binaire est la décompilation. L'objectif de cette étape est d'analyser le programme binaire afin d'extraire une représentation intermédiaire suffisamment précise pour permettre la parallélisation des nids de boucles par un outil source-à-source. La parallélisation automatique est possible lorsque les structures de contrôle du programme ainsi que les accès mémoires sont exprimés dans une représentation facilement exploitable. Ce sont donc les deux informations que l'analyse extrait du programme compilé.

Analyse de base

Pour analyser le programme, nous utilisons principalement des techniques classiques d'analyse de programmes, combinés avec les résultats récents de Ketterlin et Clauss [67]. Le programme binaire est décompilé vers une suite d'instructions assembleur. Depuis cette représentation, les limites des blocs de base sont déterminées, et un graphe de flot de contrôle est calculé par routine. L'arbre des dominateurs est calculé et les boucles sont déterminées. Si une routine contient des boucles irréductibles, elle est ignorée. Le programme est ensuite mis sous forme SSA (Static Single Assignment). Les registres processeurs sont alors des variables et la mémoire est une variable unique M.

Substitution récursive

Dans le code binaire pour les architectures x86-64, les adresses accédées sont sous la forme $Base + s \times Index + o$, où $Base$ et $Index$ sont des registres et s et o sont des petites constantes. Cette représentation ne permet par directement une analyse de dépendances précise. Afin de simplifier ces adresses, les registres $Base$ et $Index$ sont récursivement remplacés par leurs définitions. Lors de ce remplacement, des ϕ-fonctions peuvent être rencontrées. Une résolution de variables d'induction permet alors de construire des expressions dépendant d'indices virtuels de boucles. Lorsqu'une zone mémoire est rencontrée lors de la substitution récursive, une analyse de dépendance simple est réalisée en distinguant les accès à la pile des autres accès. Cela permet de poursuivre le remplacement lorsqu'une valeur temporaire est placée dans la pile de la fonction.

Résultat

À la fin de l'analyse, les adresses accédées sont exprimées, lorsque c'est possible, sous forme de fonctions linéaires dépendant d'indices virtuels de boucle ainsi que d'autres registres définis hors de la boucle. Des conditions d'exécutions des blocs de base sont également déterminées afin d'en déduire les conditions de sortie des boucles. Toutes ces informations rassemblées permettent de construire un programme C simpliste dans

lequel le flot de contrôle ainsi que les accès à la mémoire sont équivalents à ceux présents dans le code binaire original.

Parallélisation Polyédrique

Le résultat de l'analyse génère un programme C simple qui permet la parallélisation du programme. Cependant, en l'état, ce programme simple pose plusieurs problèmes aux paralléliseurs source-à-source existant. Avant d'être parallélisé, le code source reconstruit depuis le programme binaire est donc simplifié.

De la mémoire aux tableaux

Les accès mémoires, même après la substitution récursive, sont sous une forme complexe dans le fichier binaire : les accès tableaux du code source original du programme sont linéarisés et les adresses de base des tableaux statiques sont de très grandes valeurs. C'est pourquoi, lorsque les bornes des boucles sont non paramétriques, nous proposons de reconstituer les dimensions des tableaux. Pour ce faire, la mémoire est séparée en zones disjointes et les dimensions des tableaux sont reconstruites par un algorithme simple évaluant toutes les dimensions possibles. Ces simplifications permettent une analyse de dépendance plus précise.

Les scalaires

Les écritures dans des scalaires provoquent de nombreuses dépendances qui ne sont généralement pas traitées dans les paralléliseurs source-à-source existant, en particulier les outils polyédriques. Des techniques classiques telles que la résolution de variables d'induction ainsi que l'analyse de «Forward Substitution» permettent d'éliminer certaines références à des scalaires. En plus de ces techniques, nous remplaçons, lorsque c'est possible, les références à des scalaires par des références à des tableaux existants qui génèrent généralement moins de dépendances. Cette technique peut être vue comme une expansion des scalaires suivi d'un renommage du tableau résultant de l'expansion en un autre tableau déjà présent dans le code source.

Parallélisation

Après simplification, le code à paralléliser contient des boucles et des tests ainsi que des accès mémoires. Les opérations réalisées sur les données, inutiles pour l'analyse de dépendances, sont remplacées par un opérateur générique tel que «+». Ce code simplifié est facilement analysable par les outils existants et peut donc être parallélisé. Dans notre implémentation, deux paralléliseurs automatiques source-à-source sont utilisés : CETUS [6], un paralléliseur relativement simple, et PLUTO [23], un paralléliseur polyédrique. Ces deux outils produisent en sortie un code C parallélisé avec OpenMP [95].

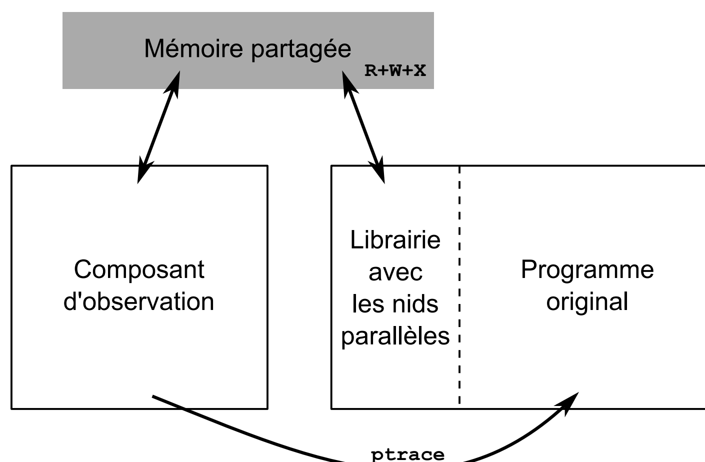

Figure 2: Implémentation du mécanisme de parallélisation de programmes binaires.

Restauration de la sémantique

Après la parallélisation du programme, les opérations sur les données sont ré-introduites dans le code source du programme. Ces opérations sont réalisées par des instructions assembleurs insérées dans le programme C parallélisé. Les registres SSA deviennent cependant des variables C classiques.

Le résultat est donc un ensemble de nids de boucles parallèles exprimés en langage C, et faisant référence à des variables dont le contenu est exploité par des opérations en assembleur. Ces nids sont finalement compilés par un compilateur classique pour produire la version parallèle du programme binaire initial.

Implémentation

La Figure 2 présente un schéma général du mécanisme permettant l'exécution des versions parallèles des nids de boucles. Les boucles parallélisées sont compilées dans une librairie dynamique, chargée dans l'espace mémoire du programme séquentiel. Au démarrage de l'application, un composant d'observation place des points d'arrêt au début de chaque boucle parallélisée dans le programme séquentiel. Lorsque ces points d'arrêts sont rencontrés, le composant d'observation reprend le contrôle et redirige l'exécution du programme vers la version parallèle du nid de boucle. Cette opération s'appuie sur une zone de mémoire partagée dans laquelle la librairie aura déclaré la position en mémoire des nids parallèles.

Évaluation

Sensibilité à l'entrée

Les optimisations appliquées lors de la compilation du programme séquentiel original ont un impact sur le nombre de boucles que notre système peut paralléliser. Afin de mieux évaluer cet impact, nous avons compilé les programmes de la suite de test PolyBench [100] avec ICC 12.1, GCC 4.5, et LLVM 2.8 en utilisant les niveaux

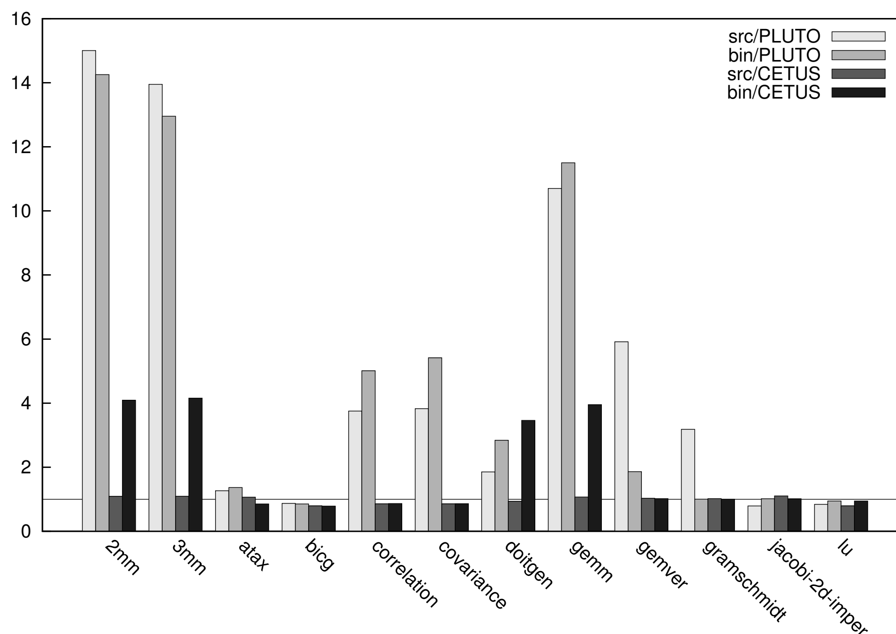

Figure 3: Parallélisation avec CETUS ou PLUTO depuis 1) le code source original 2) le code source extrait du programme binaire.

d'optimisation O2 et O3. Nous avons ensuite analysé les différents codes séquentiels résultats avec notre système. Toutes les boucles sont trouvées mais, avec un niveau d'optimisation avancé, une part non négligeable de ces boucles ne peut pas être traitée : 77 % peuvent être traitées au niveau O2, et 72 % au niveau O3. La plupart des problèmes proviennent des appels de fonctions qui compliquent l'analyse (même si celle-ci est intra-procédurale). Les transformations complexes de code sont également une source d'abandon de parallélisation.

L'analyse inter-procédurale et une étape d'inversion des transformations de codes, comme le déroulage de boucles, pourraient augmenter fortement la proportion des boucles parallélisables par notre système.

Code binaire ou code source ?

En Figure 3, nous présentons des accélérations obtenues sur une machine quadri-cœurs Intel Xeon W3520. CETUS et PLUTO sont comparés dans la figure lorsque la parallélisation est réalisée depuis le code source original du programme ou depuis le code source extrait du programme binaire par notre système.

On peut s'apercevoir que notre système permet en général une parallélisation du programme binaire à peu près aussi efficace que celle réalisée à partir du code source original du programme. L'analyse du programme est donc suffisamment précise pour permettre une parallélisation efficace et le système dynamique qui redirige l'exécution a un coût très raisonnable. On peut également noter que CETUS fonctionne mieux depuis le binaire car il bénéficie dans ce cas des simplifications réalisée lors de la compilation initiale du programme séquentiel.

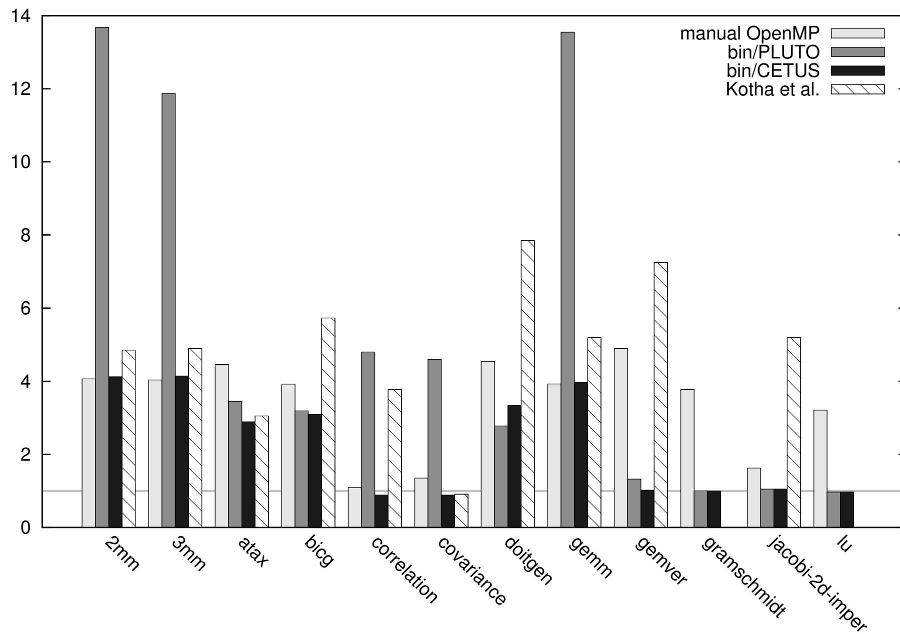

Figure 4: Comparaison de notre système à une parallélisation manuelle et à l'état de l'art.

Autres méthodes

Un graphe similaire est présenté en Figure 4, mais les deux paralléliseurs utilisés par notre système sont alors comparés à une parallélisation manuelle du code source original, ainsi qu'au système présenté par Kotha et al. dans [74]. Notre système, lorsqu'il s'appuie sur le paralléliseur PLUTO, est le seul capable de réaliser des transformations polyédriques. Cet avantage est décisif dans certains programmes tels que `2mm` ou `covariance`. Cependant, dans d'autres programmes, les fonctions d'accès ou les transformations appliquées lors de la compilation du programme séquentiel ne nous permettent pas d'analyser ou de paralléliser aussi efficacement les nids de boucles. Notre système est également très dépendant de l'efficacité des paralléliseurs utilisés et ne parvient pas à atteindre de bonnes performances lorsque les paralléliseurs ne le permettent pas. C'est le cas par exemple avec `lu` ou `jacobi-2d-imper`.

Extensions

Codes paramétriques

Après l'analyse du programme, le code extrait est simplifié pour faciliter la parallélisation. La plupart des techniques de simplification sont exactes lorsque ni les accès mémoires, ni les bornes des boucles ne contiennent de paramètres. Dans le cas contraire, notre système émet des hypothèses qui lui permettent de simplifier les programmes contenant des paramètres. Si les bornes des boucles sont paramétriques, Kotha et al. ont proposé dans [74] de diviser la mémoire en tableaux disjoints en considérant l'adresse de base des accès comme indice pour distinguer les tableaux. Nous proposons

une technique similaire lorsque l'adresse de base des tableaux est un registre, ce qui arrive fréquemment lorsque la mémoire est allouée dynamiquement dans le programme original.

Lors de l'exécution du programme, des tests sont évalués pour s'assurer que les références pour lesquelles on a supposé qu'elles accèdent à des tableaux différents, le font effectivement. Ces tests paramétriques peuvent être générés automatiquement à l'aide d'outils tels que ISL [138]. Cette technique permet de paralléliser par exemple le programme `swim` de la suite SpecOMP 2001 [5]. Sur un processeur quadri-cœurs récent, notre système atteint une accélération de 1,2 $\times$ en comparaison à une accélération référence de 1,5 $\times$.

Références polynomiales

Jusque là, nous nous sommes principalement intéressés aux références mémoire linéaires. Il est pourtant fréquent de rencontrer des références non linéaires dans les programmes compilés, par exemple si des tableaux dynamiques sont utilisés. Afin de paralléliser ces programmes, nous proposons d'utiliser une technique basée sur l'expansion de Bernstein [31] afin de s'assurer à l'exécution que la boucle externe du nid ne porte pas de dépendance et peut donc être parallélisée.

Avec cette possibilité, notre système peut paralléliser des programmes tels que `mgrid` de la suite SpecOMP 2001. Sur le même processeur que précédemment, nous atteignons une accélération de 1,8 $\times$ en comparaison à l'accélération référence de 3,1 $\times$. Comme pour l'exemple précédent, la référence consiste en une parallélisation manuelle dans laquelle un expert a indiqué que la parallélisation est possible, tandis que notre système est entièrement automatique.

Parallélisation sur place

Jusque là, chaque nid de boucle parallélisé est rajouté au code total du programme. Si un nid de boucle est parallélisé sans être transformé comme c'est le cas en présence d'accès non linéaires, il est possible de ne pas augmenter sensiblement la taille du programme en le parallélisant.

Pour cela, le composant d'observation qui redirige l'exécution vers les versions parallèles des nids de boucles va modifier le code binaire séquentiel lorsqu'un point d'arrêt est rencontré. Les fonctions présentes dans la librairie dynamique gèrent alors les threads qui exécutent simultanément le code séquentiel initial, réalisant la parallélisation. Les modifications réalisées par le composant d'observation, garantissent que l'exécution est correcte.

Conclusion

Nous avons présenté un système de parallélisation de code binaire. Ce système est l'approche statique de parallélisation, il est donc la première brique de notre approche générale. Notre système, même s'il est limité aux nids de boucles analysables et parallélisables statiquement, présente deux nouveautés principales : il est statique contrairement à la majorité des systèmes de parallélisation de code binaire proposés jusque là

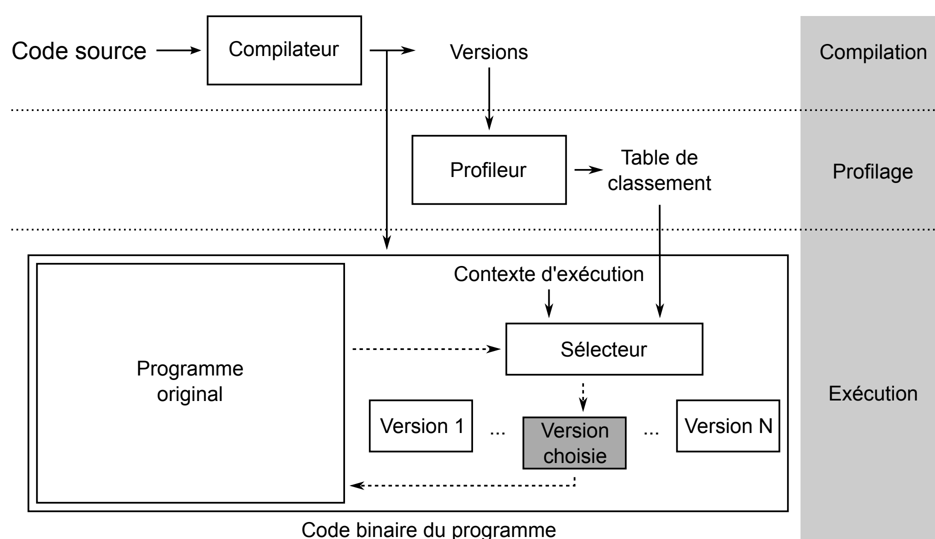

Figure 5: Schéma général du sélecteur de versions.

et il est modulaire. Différents paralléliseurs source-à-source peuvent être utilisés, y compris des paralléliseurs polyédriques qui améliorent sensiblement la performance des programmes par rapport à l'existant.

Cette approche permet également d'étendre les applications du modèle polyédrique vers les programmes binaires, pour lesquels le format du code est a priori plus complexe.

Les détails sur les techniques présentées et leur implémentation sont disponibles dans le Chapitre 3.

Sélection de Versions de Programmes

Introduction

Le deuxième système que nous proposons est un système hybride : il réalise certaines opérations au moment de la compilation du programme mais bénéficie également des informations disponibles au moment de son exécution. En exploitant ces différentes phases, notre système permet de sélectionner différentes versions de nids de boucles polyédriques en fonction du contexte d'exécution courant. Ce contexte d'exécution impacte la performance des programmes et consiste principalement en deux facteurs sur les architectures multi-cœurs modernes : l'équilibre de charge et la taille des données utilisées. Notre système se concentre donc sur ces deux facteurs.

Comme présenté en Figure 5, les différentes versions des nids de boucles polyédriques sont générées lors de la compilation, chacune résultant de l'application d'optimisations différentes sur le nid de boucle original. Ces versions sont observées dans une phase de profilage, typiquement exécutée lors de l'installation du programme. À l'exécution, le résultat de ce profilage est combiné avec le contexte d'exécution observable lors de l'exécution du programme pour sélectionner une version supposée la plus performante.

Génération des Versions

Lors de nos tests, nous avons généré les versions en appliquant manuellement des transformations polyédriques différentes sur les programmes exemples. Les transformations diffèrent par l'ordonnancement des itérations, le nombre de niveaux de tuilage, et la taille des tuiles utilisées. Les paralléliseurs polyédriques automatiques peuvent facilement générer plusieurs versions d'un nid polyédrique en utilisant des heuristiques par exemple. La nature itérative de LetSee [104] fait de ce paralléliseur un très bon candidat pour générer automatiquement différentes versions d'un nid de boucles.

Profilage

Après avoir généré les versions, elles sont profilées dans une phase unique dédiée, typiquement exécutée lors de l'installation du programme. Il s'agit alors de mesurer l'impact de l'équilibre de charge et de la taille des données sur les performances de chaque version. Pour ne pas avoir à mesurer cette performance pour toutes les tailles de données possibles, nous approximons que l'ordre relatif des versions pour des grandes données est représentatif de l'ordre relatif des versions pour toutes les tailles de données. La précision de la sélection est ainsi amoindrie, mais en pratique, le système reste très efficace avec cette approximation en particulier pour les grandes données, où le temps de calcul est plus important.

Lors du profilage, il faut alors mesurer le temps nécessaire en moyenne pour exécuter une itération du nid pour les grandes données et pour les différents équilibres de charges possibles. Nous proposons deux stratégies différentes pour réaliser cette mesure.

Profilage 1

La première stratégie de profilage a pour objectif de mesurer la performance des versions dans des domaines d'itérations aussi similaires que possible. Pour contrôler l'équilibre de charge et la taille des données, les bornes du nid de boucles sont modifiées. Après avoir appliqué la transformation polyédrique, ces bornes de boucles sont supprimées et remplacées par des contraintes simples. Ces contraintes assurent que les boucles commencent à 0, que la boucle parallèle exécute `par_sz` itérations, et que les autres boucles exécutent `N_min` itérations. Ces deux nouveaux paramètres sont modifiés au cours du profilage pour mesurer des temps d'exécution pour des grandes données et pour contrôler le nombre d'itérations parallèles (et donc l'équilibre de charge).

Cette méthode est simple et fonctionne pour n'importe quel nid de boucle. Son inconvénient principal est qu'elle détruit les bornes des boucles et donc que certains effets de bords tels que le coût du contrôle sont mal évalués lors du profilage.

Profilage 2

La deuxième méthode de profilage préserve le nid de boucle dans son état initial. Le nombre de threads actifs est contrôlé par l'utilisation de la méthode `omp_set_num_threads` d'OpenMP. La taille des données est contrôlée par les paramètres présents dans les bornes des boucles du nid.

Afin de déterminer comment augmenter la taille du domaine d'itération à l'aide des paramètres existants, nous privilégions une approche polyédrique. Un nouveau paramètre `N_min` est considéré. Il représente la taille d'un hypercube qui doit pouvoir rentrer dans le polyèdre du domaine d'itération pour garantir qu'un certain nombre d'itérations seront exécutées. Le lien entre `N_min` et les paramètres du nid de boucle est obtenu grâce aux outils polyédriques tels que PIP [45]. Le résultat pour chaque paramètre est une expression qui définit la valeur du paramètre en fonction de `N_min` pour garantir qu'un certain nombre d'itérations seront exécutées par le nid de boucles.

Utilisation du code de profilage

Quelle que soit la méthode de profilage choisie, chaque version disponible est exécutée pour une grande taille de données, et une mesure de temps d'exécution est réalisée pour chaque nombre de threads actifs. Le résultat de cette étape est stocké dans une table de classement qui sera utile pour prédire la performance de chacune des versions.

Sélection des Versions

Une fois le profilage réalisé, la sélection des versions peut avoir lieu avant chaque exécution du nid de boucles. Pour ce faire, le temps d'exécution de chacune des versions dans le contexte d'exécution courant est prédit. Cette prédiction est réalisée grâce à un nid de boucle dédié, le *nid de prédiction*, qui mesure l'équilibrage de charge courant.

Nombre d'itérations

Le nid de prédiction est en charge de mesurer le nombre de threads actifs pour chaque itération du nid de boucles. C'est de cette mesure que l'équilibre de charge sera déduit plus tard.

Le nid de prédiction est construit à partir du nid de boucle exécuté. Le contenu de la boucle parallèle est remplacé par un polynôme d'Ehrhart qui définit le nombre d'itérations exécutées dans l'itération courante de la boucle parallèle. Ce nombre d'itérations est assigné à un compteur propre au thread qui le calcule. À la fin de l'exécution du nid de prédiction, il est donc possible de consulter le nombre d'itérations exécutées par chaque thread.

Temps d'exécution

En faisant l'hypothèse que le temps d'exécution par itération est constant, on peut facilement déduire la mesure inverse du nombre d'itérations exécutées par chaque thread, c'est à dire le nombre de threads actifs lors de l'exécution de chacune des itérations. Pour cela, il suffit de trier les mesures obtenues par le nid de prédiction en ordre décroissant et de considérer les différences entre deux entrées successives. Par exemple, si on mesure qu'un thread a exécuté trois itérations et que l'autre thread en a exécuté 5, on peut déduire que les deux threads ont été actifs pendant l'exécution de trois itérations alors qu'un seul thread a été actif pendant l'exécution de deux itérations.

On appelle NC le nombre de cœurs de calcul disponibles, Nit_t^v le nombre d'itérations exécutées par t threads (le résultat du nid de prédiction de la version v), et ET_t^v le temps d'exécution d'une itération en utilisant la version v et t threads (le résultat du profilage). Le temps d'exécution prédit pour la version v est alors défini par :

$$\sum_{t=0}^{NC}(t \times Nit_t^v \times ET_t^v)$$

En utilisant cette formule simple, on peut prédire un temps d'exécution pour chaque version juste avant d'exécuter le calcul. Il suffit alors d'effectuer le calcul en utilisant la version pour laquelle le temps d'exécution prédit est le plus petit.

Évaluation

L'évaluation de notre système a été réalisée sur trois machines, décrites dans le Chapitre 4, en utilisant différentes versions de programmes, dont les détails sont également disponibles dans ce chapitre. Les versions sont évaluées dans différents contextes qui se distinguent par la taille des données utilisées et par le nombre de cœurs de calculs disponibles sur chacun des ordinateurs considérés.

Influence du contexte

En évaluant les versions des programmes pour plusieurs tailles de données, on s'aperçoit que le contexte d'exécution influence les programmes de différentes manières. En premier lieu, et de façon évidente, on peut remarquer que la machine qui exécute le programme influence la performance de ce programme. Ce n'est alors pas toujours la même version du programme qui est la plus efficace selon l'ordinateur considéré. Ensuite, la taille des données a aussi un impact sur la performance du programme. Selon la taille des données, ce n'est pas toujours la même version qui est la plus rapide. Enfin, le nombre de cœurs de calculs disponible influence également la performance relative des versions.

Différents exemples chiffrés sont présentés dans la section expérimentale du Chapitre 4.

Temps d'exécution

Afin d'évaluer la performance de notre système de sélection, nous avons mesuré l'accélération des programmes qu'il réalise dans différents contextes d'exécution par rapport à un système de sélection de versions parfait. Les résultats sont présentés dans la Table 1. Un résultat de 100% indique une performance maximale en considérant les versions disponibles. La performance de chacune des stratégies de profilage est présentée avec celle de la meilleure version statique. Celle-ci est la plus rapide dans tous les contextes d'exécution testés, elle a été déterminée après avoir testé toutes les versions. Notre système atteint des performances comparables à cette meilleure version statique, et arrive même à la dépasser dans certains cas. Le surcoût du nid de prédiction est considéré dans ces mesures. On peut déduire de ces mesures que notre système de sélection est efficace et permet de bénéficier d'accélérations spécifiques à certains contextes d'exécutions.

Processeur	Programme	Stratégie 1	Stratégie 2	Meilleure version statique
Corei7	2mm	100.0 %	98.9 %	100.0 %
	adi	99.6 %	98.7 %	97.5 %
	covariance	96.6 %	95.8 %	99.7 %
	gemm	93.4 %	84.7 %	93.5 %
	gemver	80.6 %	98.3 %	91.6 %
	jacobi-1d	99.5 %	99.5 %	99.9 %
	jacobi-2d	90.2 %	94.8 %	99.6 %
	lu	91.2 %	-	98.3 %
	matmul	98.5 %	97.9 %	98.5 %
	matmul-init	100.0 %	97.6 %	100.0 %
	mgrid	97.0 %	99.9 %	97.0 %
	seidel	99.5 %	99.8 %	99.6 %
Opteron	2mm	100.0 %	100.0 %	100.0 %
	adi	99.1 %	99.6 %	97.3 %
	covariance	99.8 %	99.8 %	99.8 %
	gemm	97.8 %	96.5 %	96.7 %
	gemver	99.7 %	99.4 %	99.8 %
	jacobi-1d	99.6 %	99.6 %	100.0 %
	jacobi-2d	100.0 %	98.5 %	100.0 %
	lu	100.0 %	-	100.0 %
	matmul	100.0 %	96.9 %	100.0 %
	matmul-init	100.0 %	100.0 %	100.0 %
	mgrid	96.2 %	99.0 %	98.5 %
	seidel	98.9 %	99.5 %	98.3 %
Phenom	2mm	100.0 %	100.0 %	100.0 %
	adi	98.7 %	99.5 %	97.5 %
	covariance	99.9 %	99.9 %	99.9 %
	gemm	99.2 %	97.2 %	96.9 %
	gemver	99.7 %	99.1 %	99.8 %
	jacobi-1d	99.7 %	99.7 %	100.0 %
	jacobi-2d	99.4 %	98.7 %	100.0 %
	lu	100.0 %	-	100.0 %
	matmul	100.0 %	100.0 %	100.0 %
	matmul-init	100.0 %	100.0 %	100.0 %
	mgrid	95.9 %	99.7 %	98.1 %
	seidel	99.0 %	98.9 %	99.0 %

Table 1: Accélération des deux stratégies de profilage et de la meilleure version statique par rapport à un mécanisme parfait de sélection de versions.

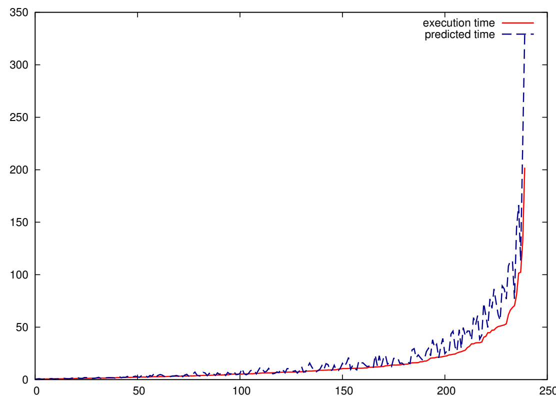

Figure 6: Temps d'exécution (en secondes) réel et prédit de toutes les versions du programme `2mm` dans tous les contextes testés, trié par le temps d'exécution réel. Sur Core i7 avec la stratégie 1.

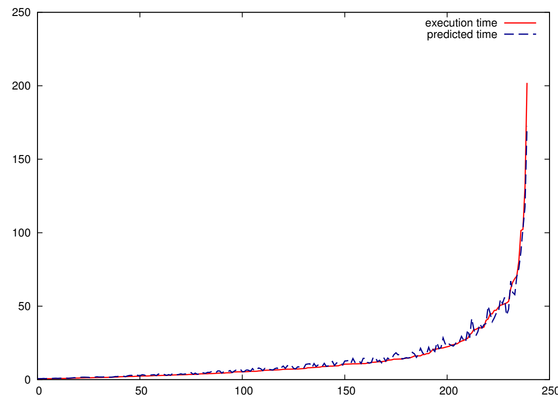

Figure 7: Temps d'exécution (en secondes) réel et prédit de toutes les versions du programme `2mm` dans tous les contextes testés, trié par le temps d'exécution réel. Sur Core i7 avec la stratégie 1.

Précision

Même si la performance atteinte en utilisant les deux stratégies de profilage est comparable, nous avons évalué la précision de chacune de ces stratégies. Pour ce faire nous présentons en Figure 6 et 7 le temps d'exécution réel et prédit de chacune des stratégies pour toutes les versions dans tous les contextes d'exécution testés pour un programme sur un processeur. Le trait plein indique le temps d'exécution réel, le trait pointillé indique le temps d'exécution prédit. Les versions dans les différents contextes d'exécutions sont présentées horizontalement. Le temps d'exécution est représenté sur la direction verticale. Les mesures sont triées par temps d'exécution réel croissant. On peut s'apercevoir que les deux courbes sont proches : le temps prédit est très proche du temps d'exécution réel. La stratégie 2 semble cependant plus précise. D'autres mesures similaires sont présentées dans le Chapitre 4.

Conclusion

Nous avons présenté un système de sélection de version hybride qui constitue la deuxième brique de notre approche générale. Bien que limité aux boucles affines, il exploite à la fois la période de compilation et celle de l'exécution du programme pour effectuer une sélection de versions parallèles efficace et précise dès la première exécution du nid. Il permet ainsi de bénéficier du contexte d'exécution courant pour accélérer les programmes par rapport à une version unique.

Comme il exploite le modèle polyédrique, notre système peut être vu comme une extension de ce modèle vers l'amélioration de la performance des programmes en profitant des particularités des contextes d'exécution.

Le Chapitre 4 présente les détails de la méthode, les implémentations réalisées, et d'autres mesures pour argumenter de la validité de notre approche.

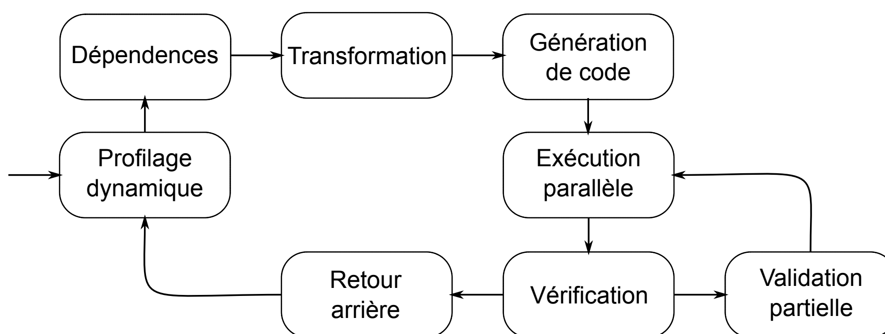

Figure 8: Les différentes étapes de notre paralléliseur spéculatif.

Parallélisation Spéculative

Introduction

Le troisième mécanisme que nous proposons est dynamique, il exploite principalement la période d'exécution des programmes pour les paralléliser. Il consiste en un système de parallélisation spéculative capable d'appliquer des transformations polyédriques sur les nids de boucle pour les paralléliser efficacement même s'ils ne sont pas analysable statiquement.

Les systèmes spéculatifs sont intéressants car ils permettent la parallélisation de codes complexes qui contiennent par exemple des accès pointeurs impossibles à analyser statiquement. Ils permettent également le traitement de programmes qui ne sont parallèles que dans certaines phases de leur exécution. L'apport principal de notre méthode est la possibilité d'appliquer des transformations polyédriques de façon spéculative.

La Figure 8 présente les différentes étapes d'un tel système. Tout d'abord, le programme est observé sur une courte période, plusieurs éléments tels que les accès mémoires, la valeur de certains scalaires, et les bornes de certaines boucles sont instrumentées. Si ces éléments peuvent être décrits par une fonction linéaire, alors le système suppose que cette caractérisation sera valable pour toute l'exécution du programme. Partant de cette hypothèse, les dépendances du programmes sont construites puis une transformation polyédrique est choisie. Le code correspondant est alors généré puis exécuté. Pendant l'exécution parallèle, les hypothèses émises par le système sont vérifiées et, si elles s'avèrent incorrectes, l'exécution est partiellement annulée avant de recommencer un cycle de parallélisation.

Profilage en-ligne et Dépendances

Lorsqu'un nid de boucle intéressant est exécuté, la première tâche réalisée par le système est d'observer le comportement de ce nid. Cette observation a pour principal objectif de déterminer si les accès mémoires peuvent être décrits par des fonctions linéaires, condition préalable à l'application du modèle polyédrique. Certains scalaires sont fréquemment la source de nombreuses dépendances dans les nids de boucles. Dans certains cas ils peuvent être traités statiquement, par privatisation par exemple. Les codes complexes que notre système considère contiennent fréquemment des scalaires qui

ne peuvent pas être traités statiquement mais dont la valeur est une fonction linéaire des indices des boucles englobantes. Ces valeurs sont donc également observées afin de supprimer les dépendances associées aux scalaires et permettre la parallélisation. Pour permettre l'implémentation d'un mécanisme transactionnel, le nombre d'itérations des boucles `while` est également observé et exprimé sous forme de fonction linéaire lorsque c'est possible.

Profilage

Les adresses accédées sont donc instrumentées ainsi que la valeur des scalaires qui provoquent des dépendances qui ne peuvent pas être éliminées statiquement. De la même façon, le nombre d'itérations des boucles `while` est également mesuré pour former une fonction linéaire lorsque c'est possible. Cette instrumentation est exécutée pendant trois itérations de chaque niveau de boucle, le temps de mesurer un nombre suffisant de valeurs pour construire les fonctions linéaires. Cependant, dans certains cas, il est nécessaire d'observer les boucles pendant une plus longue période. C'est le cas par exemple si un accès mémoire est présent dans un test qui n'est pas pris à chaque itération. Si des données d'instrumentation sont manquantes après trois itérations, le profilage continue jusqu'à au plus dix itérations.

Construction des dépendances

Lorsque tous les éléments profilés sont identifiés comme étant linéaires, le système fait l'hypothèse que ce comportement est représentatif du comportement général de l'application. C'est l'aspect spéculatif du système. Sous cette hypothèse, qui sera vérifiée plus tard, les dépendances du nid de boucle sont construites. Ces dépendances sont représentées sous forme de polyèdres de dépendance qui sont une représentation compacte mais exacte des dépendances dans le cadre polyédrique. Ces dépendances sont utilisées plus tard pour choisir une transformation pour le nid de boucle.

Ordonnancement

Les dépendances construites sont spéculatives. Si les éléments observés comme étant linéaires pendant le profilage en ligne le sont effectivement, alors ces dépendances sont celles qui ont lieu lors de l'exécution du nid. Il faut alors choisir une transformation polyédrique qui respecte ces dépendances. Cette transformation sera correcte et ne nécessitera pas de retour arrière si les dépendances spéculatives sont correctes durant toute l'exécution du programme.

Profilage hors-ligne

Le choix d'une transformation polyédrique en fonction de dépendances est extrêmement complexe et les techniques existantes ont un coût qui ne peut pas être facilement amorti dans un système dynamique. Notre système effectue donc un profilage hors-ligne (avant l'exécution du programme) afin d'étudier le comportement des éléments qui seront profilés plus tard lors de l'exécution du programme par la phase de profilage en-ligne.

Si, lors du profilage hors-ligne, il s'avère que ces éléments sont majoritairement linéaires, alors les dépendances observées sont reconstruites et un ensemble de transformations possibles est identifié. Ces transformations sont alors embarquées dans le programme séquentiel et pourront être sélectionnées si les dépendances spéculatives le permettent.

Ordonnancements génériques

En plus des transformations choisies lors de la phase de profilage hors-ligne, il est intéressant de considérer d'autres transformations qui peuvent être également embarquées avec le programme pour être sélectionnées si jamais aucune autre transformation ne peut l'être.

Dans ce cas, il faut générer un ensemble de transformations qui ont une chance raisonnable d'être compatibles avec les dépendances spéculatives et qui permettent une parallélisation relativement efficace. Une des nombreuses possibilités pour atteindre cet objectif est de considérer des transformations simples telles que des inversions de boucles pour ramener successivement toutes les boucles vers l'extérieur combinées avec du tuilage pour améliorer la localité des données. Un parcourt diagonal peut aussi être intéressant pour certains codes. Ces différentes transformations ne sont pas nécessairement valides ou efficaces. Elles représentent juste des transformations souvent suffisantes pour paralléliser les programmes et le tuilage limite les effets d'une mauvaise localité des données.

Validité d'un ordonnancement

Les ordonnancements construits avant l'exécution du programme sont donc embarqués avec le programme. Après la construction des dépendances spéculatives, il faut déterminer lesquels sont valides en fonction des dépendances. Pour cela, des tests de vacuité sont classiquement appliqués aux polyèdres de dépendance construits. Cette méthode est coûteuse et peut être approximée par des méthodes plus simples mais moins robustes tels que les vecteurs de distance de dépendance [143].

Une fois un ordonnancement valide trouvé, un niveau de boucle parallèle doit être détecté, s'il en existe au moins un. Pour ce faire, une approche classique consiste là encore en un certain nombre de tests de vacuité sur des polyèdres de dépendance. Des tests heuristiques existent également tels que le test GCD, ou le I test. Notre système utilise ces heuristiques combinées à des tests exacts pour déterminer un ordonnancement valide et un niveau de boucles parallèle dans cet ordonnancement.

Génération de Code

Étant donné que les transformations sont choisies avant l'exécution du programme, le code correspondant à chacun de ces ordonnancements peut être généré hors-ligne également. Les techniques de génération de code existantes sont utilisées sans modification majeure. Les hypothèses de spéculations qui seront faites puis vérifiées garantissent que ce code parallèle est correct, ou effectueront un retour-arrière dans le cas contraire.

```
while (p != NULL) {
  ...
  p = p->next;
}
```

```
while (!end) {
  forall (i = CHUNK_START; i < CHUNK_END; i++) {
    p = base1 + i * scale1;
    if (p == NULL) { end = 1; break; }
    ...
    assert &(p->next) == base2 + i * scale2;
    p = p->next;
    assert p == base1 + (i + 1) * scale1;
  }
  CHUNK_START = CHUNK_END;
  CHUNK_END = CHUNK_END + CHUNK_SIZE;
}
```

Figure 9: Boucle d'exemple (à gauche) et sa parallélisation spéculative (à droite).

Vérification de la Spéculation

Une fois le code parallèle disponible, il est exécuté. Mais il faut encore vérifier que les hypothèses émises depuis la fin du profilage en-ligne sont effectivement valides. Pour cela, notre système surveille tous les éléments prédits pour s'assurer que leur comportement réel est bien représenté par la fonction linéaire qui a été construite lors du profilage en-ligne.

La vérification consiste simplement en une comparaison entre la valeur réelle et celle prédite pour chaque élément spéculé. Cette vérification est réalisée concomitamment à l'exécution parallèle du nid. Nous prouvons dans le Chapitre 5 que cette vérification est correcte même après avoir transformé le nid de boucles.

La Figure 9 présente un nid d'exemple simple et sa version parallèle dans laquelle le code de vérification est ajouté. On peut en effet apercevoir différentes assertions qui échouent si la valeur du scalaire spéculé `p` ne respecte pas la prédiction. Ce même scalaire est initialisé en début de boucle afin de pouvoir le privatiser et supprimer ainsi la dépendance qu'il induit entre chaque itération de la boucle initiale. Les coefficients des fonctions linéaires `base1`, `scale1`, `base2`, et `scale2` sont ceux déterminés lors du profilage en-ligne et définissent les fonctions linéaires spéculatives. On peut également voir que l'adresse de l'accès pointeur est aussi prédite et vérifiée. Enfin, on remarque que l'exécution a lieu par tranches d'itérations consécutives. Cette découpe du domaine d'itération en tranches permet d'appliquer efficacement le retour arrière si une erreur de prédiction est détectée.

Système Transactionnel

En cas d'erreur de prédiction, la validité des dépendances n'est plus garantie et donc la sémantique du programme parallèle n'est plus forcément identique à celle du programme séquentiel. Il faut donc annuler le dernier groupe d'itérations exécuté pour recommencer l'exécution à l'aide d'un ordonnancement sûr : l'ordre séquentiel.

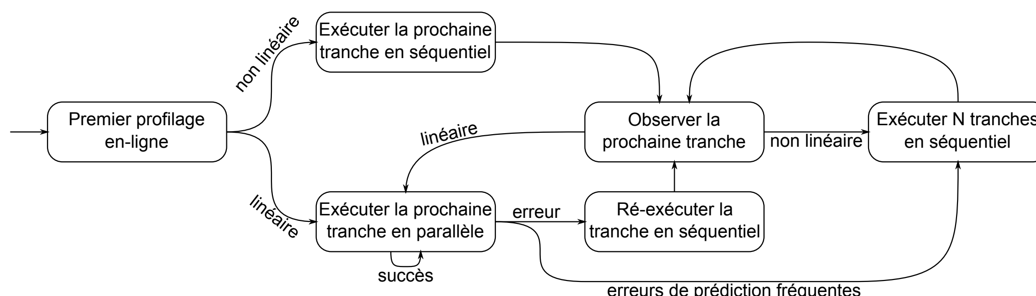

Figure 10: Stratégie de ré-exécution spéculative.

Arrêt des threads

La première chose à faire en cas d'erreur de spéculation est d'arrêter les threads. Pour ce faire, un bit est associé à chacun d'entre eux. Tous ces bits sont activés lorsqu'un thread détecte une erreur de prédiction, provocant l'arrêt de tous les threads. Cela implique que tous les threads observent régulièrement l'état du bit d'arrêt qui leur est associé, et donc un surcoût lors de l'exécution parallèle.

Annulation des itérations

Une fois les threads arrêtés, il faut annuler les calculs réalisés dans la tranche du domaine d'itération courante. Avant d'exécuter la tranche courante, un copie mémoire simple aura sauvegardé toutes les cases mémoires écrites dans la tranche courante. Cette copie est donc restaurée pour annuler les calculs éventuellement réalisés avant la détection d'une erreur de prédiction.

Les outils polyédriques classiques tels que PIP permettent de calculer facilement une sur-approximation des éléments mémoire écrits par chacun des accès dans la tranche d'itérations courante. Si aucune erreur n'est détectée, la copie effectuée pour la tranche courante est simplement écrasée par la suivante.

Stratégie de ré-exécution

Le calcul annulé doit être ré-exécuté afin de poursuivre l'exécution du programme. La prédiction n'est pas correcte, il n'est donc plus possible d'utiliser directement une version parallèle du nid. La dernière tranche d'itérations est donc ré-exécutée en utilisant l'ordonnancement séquentiel original du programme. Nous démontrons dans le Chapitre 5 que cette ré-exécution partielle est suffisante pour garantir que la sémantique du programme est bien identique à celle du programme original en cas d'erreur de prédiction.

Une fois la tranche d'itérations ré-exécutée, une stratégie dynamique est entreprise afin de reprendre efficacement l'exécution. Cette stratégie est présentée en Figure 10. L'exécution parallèle continue normalement tant qu'il n'y a pas d'erreur de prédiction. Les erreurs de prédiction rares provoquent une ré-exécution de la tranche courante en séquentiel, mais les erreurs fréquentes provoquent l'exécution de plusieurs tranches séquentielles. Cette distinction est basée sur l'observation qu'un programme peut avoir

Programme	Référence Parallèle	Spéculation OK	Séquentiel	Spéculation ÉCHEC
`ind`	280 *ms* (×5.3)	600 *ms* (×2.5)	1,495 *ms*	620 *ms*
`llist`	495 *ms* (×6.3)	985 *ms* (×3.1)	3,100 *ms*	1,050 *ms*
`rmm`	330 *ms* (×31.3)	890 *ms* (×11.6)	10,330 *ms*	890 *ms*
`switch`	620 *ms* (×6.4)	1,400 *ms* (×2.8)	3,960 *ms*	1,500 *ms*

Table 2: Évaluation du système spéculatif.

différentes phases dont certaines ne sont pas parallélisables, et qu'il peut également rencontrer un comportement temporairement non linéaire.

Évaluation

Le système que nous avons décrit a été évalué avec des programmes simples qui ne peuvent être parallélisés que par un système spéculatif. Les transformations polyédriques réalisées par notre système permettent non seulement de paralléliser ces programmes mais également d'améliorer la localité des données lors de l'exécution.

La Table 2 présente les temps d'exécution et accélérations de quatre programmes de tests. Les résultats de la version parallélisée manuellement (Référence Parallèle) accompagne le résultat de notre système lorsqu'aucune erreur de spéculation n'a lieu (Spéculation OK). La version séquentielle est également présentée (Séquentiel) ainsi que le surcoût du système lorsque toutes les tranches sont annulées mais pas ré-exécutées (Spéculation ÉCHEC). Le pire coût de notre système correspond environ au temps séquentiel plus un sixième de la dernière entrée en considérant que cinq tranches sont exécutées si des erreurs fréquentes sont détectées. Ce surcoût est provoqué par la restauration de la mémoire écrasée plus l'exécution parallèle qui a été annulée. Dans le meilleur cas, notre système peut atteindre une accélération de près de la moitié de ce qui peut être obtenu manuellement après avoir ré-écrit le programme pour s'assurer qu'il peut être parallélisé.

Conclusion

Ce système est la troisième brique de notre approche générale. Il exploite principalement la phase d'exécution du programme pour pouvoir le paralléliser. Il utilise également le modèle polyédrique et étend donc en quelque sorte ce modèle pour des programmes qui ne sont pas analysables statiquement. Son implémentation complète permettra l'application du modèle polyédrique aux programmes qui ne sont pas analysables statiquement.

Le Chapitre 5 présente différentes alternatives envisageables pour implémenter chaque partie du système. Les choix présentés sont évalués et comparés à ces alternatives. Les preuves de validité du mécanisme de vérification et de retour arrière sont également détaillées.

Conclusions et Travaux Futurs

Contributions

Nous avons présenté trois systèmes différents qui profitent des différents moments de la vie d'un programme pour le paralléliser.

Le premier système présenté permet la parallélisation statique de code binaire. Il exploite uniquement une étape de compilation pour paralléliser des programmes dont le format est complexe. Les principaux atouts de ce système sont les suivants :

- Il s'agit d'un système statique. Contrairement à la majorité des mécanismes de parallélisation de code binaire existants, il ne nécessite pas de support pour la parallélisation spéculative, y compris matériel.
- Il est modulaire. N'importe quel paralléliseur source-à-source existant ou futur peut être utilisé pour paralléliser les programmes, à condition que l'information extraite soit suffisante pour qu'il fonctionne. La plupart des systèmes existants ne permettent qu'une seule technique de parallélisation.
- Il est compatible avec les transformations polyédriques. Ces transformations permettent d'améliorer sensiblement les performances des programmes parallélisés. Elles étaient jusque là impossibles à appliquer sur du code binaire.
- Les extensions que nous proposons permettent notamment de paralléliser des programmes pour lesquels les fonctions d'accès aux données ne sont pas linéaires. Il est également possible de paralléliser les programmes sans augmenter sensiblement la taille du programme lorsque le nid de boucles n'est pas transformé.

Le deuxième système présenté est un mécanisme hybride de sélection de versions de nids de boucles affines. Il exploite une phase de profilage en plus du programme en lui même pour pouvoir le paralléliser. Les principales contributions de ce système sont les suivantes :

- Il permet la sélection de versions parallèles. La plupart des systèmes existants ne peuvent fonctionner qu'avec des programmes séquentiels. Les programmes parallèles induisent des défis particuliers qui sont relevés par notre système.
- La version sélectionnée l'est dès la première exécution. Beaucoup des systèmes proposés jusque là nécessitent de nombreuses ré-exécutions des parties du programme traitées. Notre système peut donc fonctionner avec des nids de boucle rarement exécutés.
- Notre système est entièrement automatique. Contrairement aux autres mécanismes de sélection, il ne nécessite pas de décrire le programme, ou de le ré-écrire dans un langage spécifique, ni de l'alimenter avec des données d'entraînement qui doivent souvent être sélectionnées manuellement.
- Le modèle polyédrique contraint la forme des boucles qui peuvent être traitées. Cette limite est la garantie de sa précision. Les codes traités sont connus et

clairement définis. Il est donc beaucoup moins facile de construire un programme qui sera mal prédit que pour les autres systèmes qui peuvent être piégés par de nombreux programmes aux performances très irrégulières.

La troisième partie présente un mécanisme de parallélisation spéculative, d'une nature dynamique. Le cœur de la parallélisation est réalisé lors de l'exécution du programme. Les principales contributions du système spéculatif sont les suivantes :

- La parallélisation spéculative peut être accompagnée de transformations polyédriques. Ces transformations sont jusque là ignorées dans les travaux existants. Notre système est donc le premier à pouvoir profiter des bénéfices du modèle polyédrique avec des programmes qui ne sont pas analysables statiquement.
- La parallélisation polyédrique est souvent considérée comme étant applicable seulement statiquement. Nous présentons ici une approche dynamique, aboutissant à utiliser le modèle polyédrique sur des programmes jusque là hors de sa portée.
- Le système spéculatif est efficace sans nécessiter de support pour la spéculation. Aucun matériel spécifique n'est imposé pour effectuer la parallélisation.

Les trois parties, mises en commun, forment une approche complète et cohérente qui répond au problème de la parallélisation des programmes. Nous ne résolvons bien entendu pas l'ensemble du problème mais proposons des avancées significatives. Les différents systèmes permettent en effet d'améliorer les performances des programmes afin de mieux exploiter les ressources parallèles disponibles. Chacune des étapes de la vie d'un programme sont exploitées dans ce but.

Perspectives

Le système de parallélisation statique peut être amélioré dans plusieurs directions différentes. Sa sensibilité aux optimisations appliquées aux programmes binaires séquentiels peut être amoindrie en annulant certaines de ces optimisations avant de paralléliser les programmes. Par exemple le déroulage de boucles peut être annulé pour améliorer la robustesse de l'analyse. Certaines parties du flot de contrôle qui restent complexes à traiter peuvent être cachées dans une boîte noire et approximées pour réaliser la parallélisation. Par exemple des tests sur les données peuvent être considérés comme toujours vrais afin de sur-approximer les dépendances. Enfin, si le programme binaire est entièrement analysable, il peut être intéressant de le traduire complètement vers une autre représentation binaire afin de permettre la parallélisation de code binaire x86 vers GPU ou FGPA par exemple.

Le système de sélection de code peut également être amélioré. La limite majeure de notre approche est la limitation aux nids de boucles affines. Cette contrainte pourrait être légèrement relâchée mais la précision du système risque alors d'être amoindrie. Réaliser une sélection de versions dans un contexte hétérogène est une autre extension intéressante. Ainsi, il pourrait être bénéfique de considérer des versions pour CPU et d'autres pour GPU et de déterminer automatiquement quelle version sera la plus rapide avant de l'exécuter.

La parallélisation spéculative n'a pas été complètement implémentée. Le premier objectif est de terminer cette implémentation et d'ajuster éventuellement la stratégie proposée. Partant de ce système, de nombreuses perspectives sont ouvertes. Une parallélisation polyédrique entièrement dynamique permettrait d'améliorer des programmes qui restent aujourd'hui encore hors du champ d'application du modèle polyédrique. Cette évolution vers plus de dynamisme peut se faire progressivement à partir de l'approche que nous proposons.

L'apparition rapide des systèmes multi-cœurs s'est accompagnée d'une attente très forte pour des systèmes logiciels plus efficaces. Les évolutions du matériel que nous entre-apercevons aujourd'hui permettent de supposer que cette pression ne fera qu'augmenter au cours des prochaines années. Les problèmes de cohérence de cache, d'économie d'énergie, et l'hétérogénéité croissante des ordinateurs sont des exemples des problèmes que le logiciel, et donc les compilateurs, devront résoudre.

Chapter 1

Introduction

Until the early 2000's, the software developers were in a very favorable situation, described by Sutter in [129] as a "Free Lunch" situation. This period is characterized by the rapid increase in processor frequencies. Those frequencies were increasing nearly as fast as the number of transistors in chips. In this context, a slow program just had to wait for the next processor generation to be executed faster. It was then sufficient to write a sequential program and to compile it once to automatically benefit from the improvements of the hardware. The compiler role was then mostly to perform a simple translation of the program source code to a low-level representation. Some optimizations performed by the compiler were already complex but mostly interest a small community of users having specific needs. Those optimizations were not really vital for users.

As stated by Sutter, "the free lunch is over". It is over since the early 2000's because the hardware designers have met physical limits which prevent them to increase further the processors frequency. In order to still enhance their products, they have come to the solution of increasing the number of cores instead. This decision has two major consequences for the software developers. First, the free lunch is over: a sequential program will not anymore automatically benefit from the progresses achieved by the processors. To exploit parallelism, the programs have to be rewritten or, at least re-compiled, which has severe consequences for code legacy. Second, there is a tremendous pressure on parallelism extraction. Languages and compilers transformations for parallelism have been designed at a time when the only parallel computers where rare machines designed for scientific computations. Then, as there were only a few concerned users, the software compilation techniques have somehow accumulated a delay compared to the impressive progress of hardware. Nowadays, nearly every computing machine sold has some parallel computing resources, and the softwares have to exploit them.

The programmers have at their disposal several languages and programmings models which allow them to write parallel programs. However, they are often not trained for parallelism and prefer the sequential model unless being forced to go into parallel programming. An interesting survey can be performed by any computer user, who can notice that, except for some domain-specific compute-intensive applications such as multimedia tools, most of the programs installed on a general-purpose computer are

sequential ones. One could argue that it is the compiler responsibility to make the best usage of the available hardware resources. Thus, automatic parallelization of sequential programs has to be targeted, with as less as possible help from the program developers. In fact, several automatic parallelization tools have been proposed with this objective in mind. Important progresses have been made in this field in the past decades but those technologies are not yet fully operational for a large scale deployment and still fail to parallelize programs for many different reasons.

Among the different existing techniques, the polyhedral model and associated tools have emerged as part of the solution for automatic parallelization. The polyhedral model is a mathematical framework which allows a precise representation of some parts of the programs. In this representation, some loop transformations can be performed to enhance the data locality and to uncover parallelism. Several polyhedral parallelizers are already able to exploit this model to automatically parallelize the "easily" analyzable parts of the programs.

Aiming at re-opening the "free lunch", we also target a setting where the software developers can continue to develop sequential applications, while a compiler automatically parallelizes them. With this goal in mind, we propose to extend the existing techniques and tools in three directions. First, the existing programs have to be parallelizable, even if they are legacy programs for which the source code is lost. For that purpose, we present a system able to parallelize sequential binary programs. Second, the parallelism extraction has to take into account the current external environment to reach the maximal performance. We present the possible gains and propose an efficient system to exploit the maximal amount of parallelism in different execution contexts. Third, the parallelization can be complex or impossible to perform using only static tools. Thus, we propose a speculative parallelization system to handle programs which are hard to analyze.

The tremendous pressure on parallelism extraction pushes us to exploit any situation. For that reason, the three proposed systems runs at different stages.

The binary parallelization is performed statically, before the program execution. It is able to precisely analyze a binary program in order to extract an intermediate representation. This intermediate representation is precise enough to automatically parallelize the program using the existing source-to-source parallelizers.

The second system is able to select the best implementation of a loop nest considering the current context. It exploits both static and dynamic information to perform an efficient and accurate code selection. An offline profiling step extracts information about the different parallel implementations of the loop nests. This information allows a runtime system to predict the execution time of each version in the current execution context. The version predicted as being the fastest is then executed.

Finally, the speculative system is mostly dynamic and exploits as much as possible the information available when the program is running. Existing polyhedral parallelizers are used at compile time to parallelize the program based on a profiling run. If the runtime behavior of the loop nest seems to allow the use of this parallel version, it is speculatively run, back-tracking the execution if the program behavior is finally not the one expected.

Those three systems form a fully fledged approach which addresses a broad range

of issues related to automatic parallelization of programs.

The polyhedral model representations and tools are exploited by the proposed systems to reach their goals. In some sense, they all extend the polyhedral model scope to unexpected horizons. The binary parallelization shows that the polyhedral representation and techniques are useful on low-level representations, i.e. even when the code format is complex. The code selection mechanism extends the use of the polyhedral model to complex execution contexts. Finally, the speculative parallelization system allows the polyhedral model to be used on programs which have a complex inherent code structure.

All those three systems, which form the basic bricks of our general approach are presented in the next chapters. First, we describe the base notions of the polyhedral model and present the existing techniques related to our work. In Chapter 3, we present the static binary parallelizer. Chapter 4 describes the code selection mechanism, while Chapter 5 details our last proposal: the speculative parallelizer based on the polyhedral model.

Chapter 2

Basic Concepts and Related Work

There are two important epochs in the life of a program: the compile time and the execution time. We consider those epochs to classify the existing code transformation techniques in three families.

The *static methods* are applied offline, before executing the program. They can be complex and time consuming as the program is compiled only once by its developers. However, the information available at this time is only the program itself, usually as a source code. This program description is often insufficient to determine the exact behavior of the programs in every possible execution context. Thus, the transformations belonging to this family have usually only little information on the impact of a specific runtime factor on the program performance. For that reason, they generally target an average execution context and try to not over-specialize the program for a specific runtime context, such as a specific workload for instance.

To get a more precise program characterization, some *collaborative static and dynamic methods*, or *hybrid methods,* have been proposed. The first category of hybrid systems combine a compile time approach with a runtime extension. As much as possible, the transformation is applied at compile time, and a dynamic system performs the final steps of this transformation when a more precise program description can be obtained. The second category of hybrid systems evaluate the programs during one or several profiled executions in order to collect some statistical information. This better characterization of the programs is used to perform more precise code transformations at compile time. In some sense, they also combine a static and a dynamic approach.

The dynamic methods transform the programs while they execute, allowing the transformations to be specific to the current execution context. This enables a very effective transformation as more information related to the program execution is available. However those methods have to maintain a very low overhead as they are run simultaneously to the program. Notice that the dynamic systems often require at least a small static step to be effective. For instance, a Java program has to be compiled into bytecode before running it. When the main transformation is performed at runtime, we still consider that the system is dynamic.

The recent wide spreading of multicore processors heavily increases the pressure on parallelism extraction. The current parallelization techniques have to be extended in several ways in order to exploit any possible gain. Thus, we propose a new par-

allelization mechanism in each of those families. We propose to statically transform and parallelize binary programs using the polyhedral model. We also show that a fast and accurate code selection mechanism can be implemented using an hybrid static-dynamic approach. Finally, we propose a speculative parallelization system able to perform advanced code transformations at runtime on complex programs. Despite a part of this system is executed at compile time, its core functionalities are run while the program is executed, defining a mostly-dynamic system. We present in this chapter an overview of the existing systems related to our proposals and compare them. We introduce in Section 2.1 some static systems which can be related to our binary parallelizer. In Section 2.2, some important collaborative static and dynamic mechanisms are presented and compared to our proposals. Finally, we discuss in Section 2.3 the relation of some dynamic mechanisms with our systems. Although the presented techniques do not cover all the existing program transformations, they should give to the reader a general idea of the main contributions related to our proposals.

The mechanisms that we propose are exploiting the polyhedral model to perform precise program analysis and advanced loop nest transformations. We present in Section 2.4 the common notations and concepts used in this model, and we introduce the main techniques and tools exploiting this representation.

2.1 Static Methods

The static transformations are applied on the program before executing it. They are performed in a unique compilation phase, usually by the application developers. Thanks to this property, those transformations can be complex and can consume a large amount of resources. We present in Chapter 3 our static system able to perform polyhedral transformations on loop nests at the binary level. To better evaluate the contributions of this work, we briefly discuss the links of this parallelization system with other existing static mechanisms.

2.1.1 Polyhedral Parallelization

The main contribution of our binary parallelizer is its ability to perform polyhedral transformations and parallelization at the binary level. In the past, the polyhedral model has been successfully experienced on different representations. Obviously the source level is preferred for its expressiveness, and many polyhedral parallelizers are source-to-source compilers. In CHiLL [27], the user can specify a list of transformations to apply on a loop nest including common polyhedral transformations and other advanced loop transformations. PLUTO [23, 99] uses heuristics to automatically determine a transformation sequence to apply on the loop nest. The LetSee compiler [105, 104] is an exception as it is indeed an hybrid system, using iterative compilation to determine the best optimization sequence. Although those parallelizers are research tools, some production-level polyhedral compilers have been designed such as RStream [119]. More details on those tools and the operations they perform are provided in Section 2.4.

Recently, efforts have been made to port the polyhedral model in production compiler as with GIMPLE in GCC [135], and Polly in LLVM [55]. This implies to apply polyhedral transformation on the compiler intermediate representation, which is significantly lower-level than the source code. The system that we propose goes one step deeper. The representation we use, the binary code, is less expressive than any other experimented representation. However, we show that the automatic transformations and parallelization of loop nests are still possible, while still using the existing source-to-source compilers.

No representation appears to be ideal, each one has its own pros and cons. At high level, more information is embedded and the analysis is easier, while low-level representations allow optimizations specifically targeting a given architecture. Moreover, those low-level representations also makes it possible to parse any program independently from its origin: any programming language is handled, and libraries or proprietary codes can also be parallelized.

2.1.2 Binary Code Parallelization and Rewriting

Despite they are not able to perform polyhedral loop transformations, several binary code parallelizers have been proposed. To our knowledge, Kotha and colleagues have proposed the only fully static binary parallelizer [74]. Since this work is very close to ours, we present a topic-by-topic comparison with their approach after having detailed our system in Chapter 3. The other binary code parallelizers are dynamic systems which are presented in the appropriate section.

Kotha et al. use a static binary rewriter to inject the parallelized loop nests back into the original program. Several static binary rewriters have been proposed in the past, and, as they are an important component of binary code parallelization, we first discuss them here.

One of the common optimization targeted by those static binary rewriter is code reorganization based on a previous profiling run. Spike [35] propose this optimization on WinNT/Alpha systems, `alto` [91] has a similar approach on a different operating system, and PLTO [126] solves similar issues on IA32 processors. ATOM [127] is a generic framework where the user can define its own analysis tool. The tools can be called before or after any procedure, basic block, or instruction. At runtime, ATOM exposes to the tools some low-level information such as current register values, allowing them to observe the program execution. Etch [120] is targeting Win32/x86 platforms but provides similar facilities. Pebil [75] is also a generic framework but focuses on reducing the overhead of the instrumentation tools by preparing the program before instrumenting it. Diablo [136] is also a generic framework but it is retargetable and can use several platforms.

Other tools such as HPCVIEW [85] or MAQAO [42] perform a precise analysis of binary programs with the final goal of providing optimization hints to the programmers. MAQAO also allows the users to insert some dynamic instrumentation in programs in order to build a more precise program characterization.

Two points have to be distinguished here: the analysis capacities of the tools and their binary rewriting functionalities. The binary program analysis is often naive and

most of the rewriters do not need to extract more information than basic block boundaries or function boundaries. A notable exception is presented with the program analyzers such as MAQAO. Those tools usually extract precise information related to the program such as loop structures. However, they usually do not perform a complex static memory analysis such as the one we perform to parallelize loop nests in the binary parallelizer presented in Chapter 3.

Relatively to the binary rewriting capacities, we can observe a broad variety of systems, often dedicated to a specific operating system and processor architecture. The other systems which offer genericity are also often adding some extra-overhead compared to specialized systems which can more easily perform platform-specific optimizations. We have implemented our own binary rewriting tool to parallelize binary applications. It is compatible with our experimental configuration and it is specialized to our usage, limiting its overhead. However, it is dynamic as the implementation of such dynamic rewriting tools is easier on Linux platforms. We present other dynamic rewriting tools in the dynamic methods section, and compare them with our approach.

2.2 Static-Dynamic Collaborative Methods

Static methods only exploit the program listing which may be an insufficient source of information. In some cases, a transformation can benefit from additional information which is difficult or impossible to obtain at compile time. Thus, the static method can be combined with a dynamic system to form a hybrid mechanism.

We present in this section two main categories of hybrid mechanisms related to our proposals. First, the iterative compilers can be considered as a special form of code selection systems as they are able to determine which optimization performs best among several. Second, we present the most relevant code selection mechanisms that use an hybrid approach. As the purely dynamic approach is often privileged by such systems, dynamic selection systems are evaluated in the next section.

2.2.1 Iterative Compilation

The first type of code selection mechanisms we present target code selection at compile time. In this setting, the compiler evaluates several distinct optimization sequences before choosing one for the final program. The compilation is said *iterative* as the program is compiled and evaluated several times before deciding which optimization sequence has to be applied.

In [21], Bodin et al. present a case study where an efficient optimization sequence is searched using a simple algorithm. They show that a few evaluations are required to obtain an efficient program. Kisuki et al. present in [72] an iterative compiler able to outperform a static compiler with a few evaluations. The approach is generalized in [38] where a generic iterative compiler is presented. It is able to minimize an objective function specified by the user and is then not restricted to the execution time. Some machine-learning techniques are used to traverse the optimization space. This is also the case in [134], where Triantafyllis et al. combine machine-learning techniques with execution simulations to accelerate the evaluations. They also propose to use heuristics

to direct the search. Similarly, in [1], the authors propose to use machine learning to build a model directing the search. The model predicts the optimizations that have a high chance of being profitable for a given class of programs. A recent production-scale implementation of an iterative compiler has been proposed with Milepost GCC [52]. It exploits a database, shared among the users, to collect data linking some program features to programs performance.

Fursin et al. have proposed in [51] a system taking advantage of the program phases to accelerate the evaluation of optimization sequences. Their system predicts phases in the execution to evaluate several versions in a single program run. When all the versions have been evaluated, the best version is also used for the rest of the execution. This system is singular since it is a dynamic code selection system used for offline code selection.

The polyhedral model provides a large set of loop transformations but no analytical performance model exists to determine their profitability. Thus, the polyhedral model is a natural target for iterative compilers. The first attempt in that direction was proposed by Nisbet with the GAPS framework [93] which uses genetic algorithms to select an efficient transformation using the UTF framework [66]. This framework is also used by Long and Fursin in [80] where a more comprehensive system is proposed. In [105, 104], Pouchet et al. describe an iterative strategy to select efficient polyhedral loop nest transformations in the polyhedral model. They propose to restrict the search space to the legal schedules only in order to limit its size. They have proposed later to combine iterative compilation with a model-driven approach in [106] in order to accelerate the search in that space. Park et al. have proposed in [96] to use a machine learning approach with hardware counters to perform a fast transformation selection.

Iterative compilation is a code selection mechanism but is intended to be used offline. No dynamic system is associated with iterative compilers. The different optimization sequences are generally evaluated on full runs, at compile time. For that reason, the results obtained are closely related to the execution context occurring during the compilation. They are not generally suitable to any circumstance — for instance, to distribute an efficient multi-platform executable, or to fine-tune memory and processor loads. On the contrary, our code selection framework is able to exploit the specificities of the current execution model to select at runtime an efficient version among several ones.

2.2.2 Hybrid Code Selection

More closely to the code selection mechanism that we present in Chapter 4, we evaluate some hybrid code selection systems. Our system is based on preliminary profiling runs, it is then hybrid as the other code selection mechanisms presented below.

Numerical libraries, tuned at installation time, are commonly used for scientific computing. For instance, the ATLAS project [141] is a linear algebra library where empirical timings are used at installation time in order to choose the best computation methods for a given architecture “in a matter of hours”. Such libraries suffer from the same flaws as iterative compilers: finding an efficient version is time consuming and only one version is finally generated.

The STAPL adaptive selection framework [130] runs a profiling execution at install time to extract architectural-dependent information. This information is used at runtime to select the best version, combined with previous runs and training runs performance measurements through machine learning. This system requires many training runs before being able to take relevant decisions.

More recently Tian et al. [133] propose an input-centric program behavior analysis for general programs, which is a statistical approach where program inputs have to be characterized differently depending on the target application — input size, data distribution, etc. —, and the program behavior is represented by relations between some programs parameters, as for instance loop trip-counts. This modeling can be exploited to achieve some dynamic version selection. Such relevant statistical relations seem difficult to be determined for any kind of programs. Moreover, the system requires several distinct valid inputs emphasizing different characteristics of the program in order to build the different versions. In general, it may be extremely complex to determine and provide such inputs to the system. Overall, this work does not consider parallel programs.

Those systems require several training runs before being efficient, and the parallelism is often out of the scope of the proposed systems. We show in Chapter 4 that our hybrid code selection system is able to select an efficient version since the very first execution of the loop nest and can handle the specific challenges of parallelism. Moreover, our system is fully automatic and does not require any human intervention to propose the versions, their description, or some training data.

More code selection mechanisms are presented in the next section and are compared to our system.

2.3 Dynamic Methods

While some approaches are performed at compile time, it is useful in some cases to apply the code transformations at runtime. Indeed, when the program is running, it can be more precisely analyzed and its execution context is known. Then, some aggressive transformations, specific to the current execution context, can be applied.

Among the existing dynamic systems, some have a direct link with our proposals. We present in this section some relevant mechanisms and compare them to ours.

2.3.1 Dynamic Code Selection

We have already presented the hybrid code selection mechanisms, however, most of the existing code selection frameworks are dynamic.

For instance, in the ADAPT system [140], a specific language allows the user to describe optimizations and heuristics for applying these optimizations dynamically. However, the resulting optimizer is run on a free processor or on a remote machine across the network. Such an approach seems only suitable for programs with long execution times and a lot of available hardware resources, particularly when tuning parallel programs.

PetaBricks [2] provides a language and a compiler, where having multiple implementations of algorithms is the natural way of programming. The associated runtime system uses a choice dependency graph to select one or another algorithm and implementation at different steps of the whole computation. Such an approach is suitable for programs where it is obviously possible to switch from one algorithm to another while still making progress in the whole computation, such as in divide and conquer algorithms. Further, the changing behavior of the executed algorithms must not induce overheads due to a compulsory cache flush for instance.

Mars and Hundt's static/dynamic SBO framework [83] consists in generating at compile time several versions of a function that are related to different dynamic scenarios. These scenarios are identified at runtime thanks to the micro-processor event registers. Execution is dynamically rerouted to the code relevant to the current identified scenario. However, execution is not rerouted during a function call, but for the next calls. Further, it seems difficult to use this approach with parallel programs since it is actually quite challenging with multicore processors to deduce accurate global multithreaded program behaviors from registers disseminated on the cores.

The dynamic code selection systems suffer from the same flaws as the hybrid ones. They often require several executions before being efficient, they do not target parallel codes, and they are not fully automatic. Our code selection mechanism, presented in Chapter 4, addresses all those issues by providing a fast automatic code selection mechanism for parallel codes.

2.3.2 The Inspector/Executor Model

In Chapter 5, we present a speculative parallelization scheme using the polyhedral model. This system is part of a broader category, namely the *dynamic parallelizers*, which is based either on the inspector/executor model, or the speculative methods.

In the inspector/executor model, sequential programs are divided in two components. The first one, called the inspector, is in charge of extracting the program dependences, usually between loop iterations. Then, an executor runs the tasks as soon as all their dependences have been satisfied. This model has been first proposed by Zhu and Yew in [147], and has been later extended in many directions. Chen et al. describe the CYT model [28] which allows one to reuse the result of the inspector across loop executions, and to partially overlap loop iterations. Many other extensions have been proposed in this generic frame. For instance, Padua, Rauchwerger et al. primarily focus in applying privatization and reduction to expose more parallelism [115, 114, 113]. Saltz et al. have extensively studied different runtime scheduling techniques that can be used in that model [124, 123, 88, 102]. Leung and Zahorjan later extend in [77] their parallel inspector into a more efficient one. Another interesting parallelization scheme for the inspector loops is proposed by Philippsen et al. in [86], where the inspection phase is performed by two highly optimized parallel loops.

Other inspector/executor systems, able to perform advanced program transformations, have also been proposed. For instance Ding and Kennedy propose in [41] to apply locality grouping and dynamic data packing on irregular codes to improve their performance. Mitchell et al. also propose in [89] to use the inspector/executor model

combined with some other data optimizations to enhance the performance of irregular applications. Strout et al. later unify this technique in a general framework in [128]. Despite they apply runtime transformations using the inspector/executor model, those techniques are restricted to indirect array references. Moreover, the proposed transformations are related to data reordering, and ignore program transformations such as those provided by the polyhedral model.

In general, this model is efficient if the addresses computation is clearly separated from the actual computation, allowing the creation of an efficient inspector. Moreover, to capture the dependences with no restriction, some control bits are commonly associated to every array element during the inspector phase. This often restricts those methods to array accesses, and can lead to major memory overheads. Moreover, pointer references can strongly disturb the automatic inspector creation, limiting the applicability of this method.

2.3.3 Thread-Level Speculation

Other methods performing dynamic parallelization are based on Thread-Level Speculation (TLS). In these methods, some assumptions are made on the general behavior of the application, allowing one to speculatively parallelize the program. During the parallel execution, the hypothesis are confronted to the real application behavior, and if an assumption appears to be wrong, the parallel execution is partially back-tracked and re-executed in a different way, usually using the original sequential program.

POSH [79] is a compilation framework for transforming the program code into a TLS-compatible version, by using profile information to improve speculation choices. A similar approach is presented in [63]. The Mitosis compiler [109] generates speculative threads as well as pre-computation slices (p-slices) dedicated to compute in advance the values required for initiating the threads. The LRPD test [116] speculatively parallelizes forall loops that access arrays and performs runtime detection of memory dependences. Such technique is applicable only when the array bounds are known at compile time. Tian et al. [132] focus on the efficient exploitation of pipeline parallelism using a data-speculation runtime system which creates on-demand copies of statically, as well as dynamically allocated data.

SPICE [111] is a technique using selective value prediction to convert loops into data parallel form. A similar approach is proposed in [131]. In [30], a speculative parallelization in chunks of the outermost loop is proposed, using a sliding window for reducing the impact of load imbalance. However this last proposal is limited to array-only applications.

Softspec [24] is a technique whose concepts represent preliminary ideas of our approach presented in Chapter 5. Linear memory accesses and scalar values sequences are detected, but only for innermost loops. Hence one-variable interpolating functions are built and used for simple dependence analysis via the GCD test. Thus, only the innermost loop can be parallelized.

Finally, Zhong et al. present in [146] several code transformation techniques to uncover the hidden parallelism and improve TLS efficiency, as speculative loop fission, infrequent dependence isolation, or speculative prematerialization. Our system goes

further, as it is able to perform more advanced loop transformations, and does not require any specific hardware.

The existing speculative systems are often able to handle any kind of dependency. It allows them to consider a broad variety of codes, but it can also lead to important memory and performance overheads, which are often deported to some dedicated hardware mechanisms. As in Softspec, we restrict our speculative parallelizer to loops where the addresses accessed define a linear function of the loop indices. Using this representation, we can apply advanced transformations on the loops, significantly improving their efficiency, while limiting the runtime overhead induced by the speculation. Our system could be combined with an existing dynamic system to handle non-linearity as well as linear codes.

2.3.4 Binary Code Parsing

Thread level speculation is currently a major category of systems that has to extract high-level information from low-level, executable code. Then, it is interesting to compare the analysis we perform for the binary code parallelization that we propose in Chapter 3 to those systems.

A typical example of TLS mechanism is the POSH system [79], where the code is statically analyzed to extract tasks and where the runtime environment is responsible for verifying the absence of conflicts. In the words of the authors [79, Introduction]: “[TLS] compilers are unique in that they do not need to fully prove the absence of dependences across concurrent tasks — the hardware will ultimately guarantee it”. More closely related to binary code parallelization, DeVuyst and colleagues write [40, Section 3]: “Without the high-level code, we cannot guarantee that parallel iterations of [a] loop wont attempt to modify the same data in memory”. These two short quotations are sufficient to highlight what makes our static binary parallelizer original regarding binary code parsing.

TLS systems do not perform static dependence analysis, but rather rely on some transactional memory mechanisms to ensure the correctness of the parallel execution [60]. On the other hand, static binary parallelizers produce a parallel program off-line, which means that they need some precise and definitive characterization of the data dependences right out the binary code. Therefore, most TLS systems for binary code perform control-flow analysis, extracting routines and loops [94, 40, 60]. Some go as far as putting the program into SSA form to perform a simple form of data-flow analysis [144]. However, as far as we know, no system is able to build symbolic expressions for memory accesses. This is easily understandable: static binary parallelization systems need this description to perform static dependence analysis, whereas TLS systems leave that part to the runtime environment. On the other hand, binary parallelization systems restrict themselves to loop nests, whereas TLS systems target a much wider range of program structures, including, unfortunately, loop nest that could be handled statically.

2.3.5 Inline Code Generation and Transformation

Going further in the dynamic approach leads to fully dynamic code generation and transformation mechanisms. The most common occurrences of those dynamic systems are just-in-time compilers in virtual machines [4]. Java [53] or Common Language Runtime [84] are well-known examples of this approach.

Some other tools perform dynamic code instrumentation, allowing the user to insert at runtime extra instructions to observe a program. One of the first dynamic instrumentation tool proposed is DynInst [62] developped in the Paradyn project [87]. This tool is able to add trampolines in the code in order to redirect the execution toward instrumentation code. A similar approach is developed in Vulcan [44]. The main difference with DynInst is the high-level form used to expose the program structure to the user. Pin [82] is another famous example of dynamic code instrumentation tool. The user writes C++ PinTools using the Pin API to implement the desired instrumentation. PIN inserts calls to the PinTool directly in the executed binary code. Several optimizations, such as code caches, are used to limit the overhead of this instrumentation strategy. Another approach is chosen by Valgrind [92], where the final performance is not the main concern. This tool targets heavy instrumentation as every basic block is decompiled to a high-level representation. The instrumentation is inserted in this high-level representation, which is later recompiled.

Some systems go further and propose to modify the code of the program while it is running. Dynamo [8] and DynamoRIO [25] interpret the code to determine hot regions of the program while it is running. Once those regions are found, they are dynamically transformed, including with conventional optimizations such as loop unrolling. Kistler and Franz present another system in [71] which can continuously profile a running program and optimize it using data reordering and trace optimizations. Several versions of the program are generated during its execution to perform those transformations. ADORE [81] uses a second thread and hardware counters to observe the phases in the program execution in order to apply simple optimizations such as prefetching when the program behavior is stable. Beyler has proposed later a fully software solution to instrument memory loads and model their behavior in order to dynamically insert prefetching instructions when it is profitable [15, 16]. Another kind of dynamic code optimization is performed by dynamic translators where the program is converted to another instruction set while it is running. For example, DAISY [43] translates a program to VLIW primitives and parallelizes them at runtime.

One case of dynamic optimization is particularly interesting to us. It is proposed by Yardımcı and Franz in [145], and represents the first attempt at binary parallelization. Their system is mostly dynamic and identifies control flow that behaves like loops at runtime, including recursive function calls. It is able to dynamically parallelize or vectorize those code slices using naive techniques to test the dependences and to perform the parallelization. As their system is highly dynamic, it can handle complex code structures, however, it cannot rely on complex decision algorithms. Therefore, it can only parallelize loops where no data dependence occurs, which strongly limits its scope.

The static binary parallelization presented in Chapter 3 outperforms this dynamic parallelizer as it can apply complex loop transformations on the code. Similarly for

the speculative parallelizer presented in Chapter 5. Despite those techniques can only handle loop nests, they can perform exact dependence analysis and parallelize loops in the presence of dependences, going one step further.

The code instrumentation and manipulation tools can be compared to our static binary code parallelizer implementation. Indeed, in Chapter 3, we use a simple dynamic binary optimization process to re-inject the parallelized loop nests in the original binary program. Most of the existing dynamic code instrumentation tools are huge machineries with large APIs and infinite instrumentation possibilities. The price to pay for this genericity is usually large runtime overheads. Moreover, the tools which are not generic framework are limited to some specific architectures and exploit architecture-specific methods. Our binary re-injection process is naive and not well optimized. However, it works on our architecture and provokes a negligible overhead which can often not be guaranteed by more complex frameworks.

2.4 The Polyhedral Model

In the next chapters, we propose three systems able to enhance the program performance using each of the three different approaches. All of them make an intensive use of the *polyhedral model* and its associated tools. This model aims at representing exactly the execution of the program at compile time. In order to provide useful and consistent representations, it largely exploits linear algebra, linear programming, and convex optimization theory. The premises of this model are proposed by Karp, Miller, and Winograd with a method to automatically map computations of uniform recurrence equations [64]. It is later extended to techniques related to systolic arrays, before being connected to imperative programs [45, 48].

Our work is not intended to extend this model. We rather show that this model and the associated techniques can be successfully exploited in difficult contexts, whereas it is sometimes considered as being limited to a very restricted class of programs. Despite we are users of this model, we present the main internal characteristics of the polyhedral model, in order to help the reader to better understand the possibilities it offers. We do not provide an extensive review of the polyhedral model here, but we present the main terms, representations, and tools associated to the model and which are used later in this dissertation. For a more comprehensive understanding of the model, the reader is referred to the extensive work of Feautrier [45, 46, 48, 49, 50].

2.4.1 Mathematical Background and Notations

The notions used in the polyhedral model are based on simple mathematical notions that we introduce here before going into the details of the polyhedral model itself.

A vector is noted $\vec{v}$, its dimensionality is noted $|\vec{v}|$, and the ith element of $\vec{v}$ is $\vec{v}[i]$. To clarify the reading, we make no distinction between a row vector and a column vector. This property can be determined according to the context. Two vectors can be concatenated using $|$, $\vec{v_1}|\vec{v_2}$ is the vector whose elements are made of those of $\vec{v_1}$, followed by those of $\vec{v_2}$. A matrix made of real elements with n rows and m columns is noted $A \in \mathbb{R}^{n \times m}$. The element at row i and column j is noted $A[i, j]$, while the

i^{th} row is $A[i,*]$ and the jth column is $A[*,j]$. A matrix-vector product is noted $A\vec{v}$. Depending on the context, 0 can denote the integer value or a vector of an arbitrary size containing only zeros.

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

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

We also use the terms *linear function* to name those functions.

Definition. (*Affine hyperplane*) An *affine hyperplane* is a $n-1$-dimensional subspace splitting a n-dimensional space in two distinct parts. Considering a vector $\vec{a} \in \mathbb{K}^n$ and a scalar $b \in \mathbb{K}$, an affine hyperplane is defined by the vectors $\vec{x} \in \mathbb{K}^n$ such that:

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

Definition. (*Affine half-space*) An affine hyperplane splits a space into two *half-spaces.* Considering a vector $\vec{a} \in \mathbb{K}^n$ and a scalar $b \in \mathbb{K}$, an affine half-space of $\mathbb{K}^n$ is defined by all the vectors $\vec{x} \in \mathbb{K}^n$ such that:

$$\vec{a} \cdot \vec{x} \geq b$$

Definition. (*Convex polyhedron*) A *convex polyhedron* $\mathcal{P} \subseteq \mathbb{K}^n$ is a set defined by the intersection of a finite number of affine half-spaces of $\mathbb{K}^n$. A polyhedron can also be considered as a set of m affine constraints, represented in a matrix $A \in \mathbb{K}^{m\times n}$ and a vector $\vec{b} \in \mathbb{K}^m$:

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

Definition. (*Parametric polyhedron*) The set $\mathcal{P}(\vec{p}) \subseteq \mathbb{K}^n$ is a polyhedron parameterized by the vector of symbolic values $\vec{p}$. It can be represented by a matrix $A \in \mathbb{K}^{m\times n}$, and a matrix of parameter coefficients $B \in \mathbb{K}^{m\times p}$ if we consider p parameters, i.e. $|\vec{p}| = p$, and a vector $\vec{b} \in \mathbb{K}^m$ such that:

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

Definition. (*Polytope*) A polyhedron is a *polytope* if it can be enclosed in a hypercube, i.e. if it is bounded.

For more details, the reader is referred to Schrijver's book [125]. In the rest of this dissertation, we are mostly interested in the points with integer coordinates contained in polyhedra. Thus, unless if specified differently, we consider that $\mathbb{K} = \mathbb{Z}$. Notice that this representation is not the most precise one and that other representations can sometimes be considered in the polyhedral model such as in the $\mathcal{Z}$-polyhedral model [57].

2.4.2 Scope and SCoP

Traditional compiler representations usually focus on static relations between the instructions. For instance the abstract syntax trees, the control flow graphs, or the Static Single Assignment analysis ignore the time when an instruction is executed. If this instruction is executed several times, they do not provide sufficient information to distinguish between those different executions. Then, they are insufficient to represent and manipulate the different executions of the instructions.

The polyhedral representation aims at solving this issue. It is able to represent, among other things, the different executions of an instruction and their associated execution order. Using those representations, it becomes possible to reason about the dynamic execution of the program at compile time. This compile-time representation of the execution can be exact only for the parts of the programs whose control flow is statically known. Those program regions are called *Static Control Part*s (SCoP) [34].

Definition. (*Static Control Part*) A Static Control Part (SCoP) is a program region defined as the longer sequence of consecutive instructions such that the only allowed control structures are *affine loops* and conditional instructions, where the loop bounds and conditions are defined as boolean combination of affine functions of the surrounding loop indices and global parameters.

A parameter in the polyhedral model is a symbolic value whose value is unknown at compile-time, which is defined outside of the SCoP, and then remains constant during its execution. The so-defined SCoPs are sequences of instructions and loop nests which are not necessarily perfectly nested, i.e. some instructions are not at the maximal depth in the nest. The following code, for instance, defines a valid SCoP:

```
out = 0;
for (iter = 0; iter < tsteps; iter++) {
  for (i = 0; i <= length - 1; i++)
    for (j = 0; j <= length - 1; j++)
      c[i][j] = 0;
  for (i = 0; i <= length - 2; i++) {
    for (j = i + 1; j <= length - 1; j++) {
      sum_c[i][j][i] = 0;
      for (k = i + 1; k <= j-1; k++)
        sum_c[i][j][k] = sum_c[i][j][k - 1] + c[i][k] + c[k][j];
      c[i][j] = sum_c[i][j][j-1] + W[i][j];
    }
  }
  out += c[0][length - 1];
}
```

The constraints put on the SCoP format cover a large set of scientific computation kernels, despite full programs can generally not been represented.

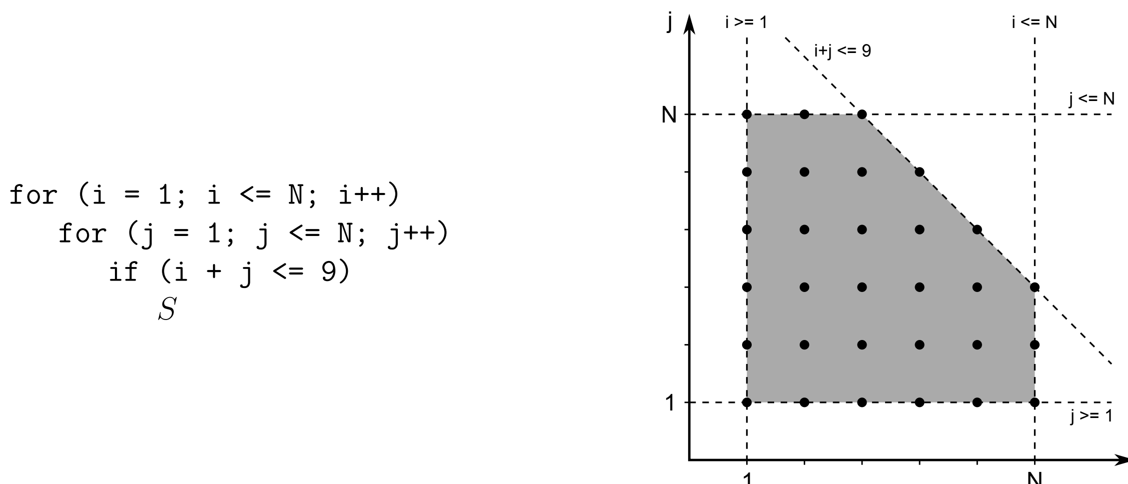

Figure 2.1: Sample loop nest (left) and the iteration domain of S (right).

2.4.3 Statement and Iteration Vector

The polyhedral model does not pay attention to the operations performed during the program execution. A common usage is to consider that those operations are performed in *statements* which represent an instruction or a sequence of consecutive instructions in a basic block. The statements are considered as atomic black boxes and the only information which is associated to them is the memory accesses they perform.

The statements represent a static feature of the program which is similar to the notion of instruction in a source code. It is not very helpful to represent the dynamic execution of the program. Thus, we introduce the *iteration vectors* to represent the different time steps in the overall SCoP dynamic execution.

Definition. (*iteration vector*) An *iteration vector* $\vec{i}$ of a given statement S consists in the value of the indices of the loops surrounding S, ordered by increasing loop depth.

A couple made of a statement S and an iteration vector $\vec{i}$ defines an *instance* $\left\langle S, \vec{i} \right\rangle$ of S. It corresponds to a specific execution of a given statement. This notion of instance is at the base of the polyhedral representation as it allows us to represent different executions of a single statement.

2.4.4 Iteration Domain

Consider a statement S enclosed in d loops. An instance $\left\langle S, \vec{i} \right\rangle$ of this statement can be represented in $\mathbb{Z}^d$, using $\vec{i}$ as coordinates. For instance, consider the loop nest presented in Figure 2.1. We can represent the first execution of S as the point $(1, 1)$ in $\mathbb{Z}^2$, called the *iteration space* of S. One can notice that not all the points in the iteration space are actually executed because of the loop bounds and tests surrounding the statement. The iteration space of S, restricted to the actually executed iterations, is called the *iteration domain* of S. The iteration domain of S in our example is represented graphically in Figure 2.1.

A more precise definition of the iteration domain is given by the following definition.

Definition. (*iteration domain*) If a statement S is enclosed in d loops, the set of iterations vectors $\vec{i}$ describing the different executions of S defines the iteration domain of S which can be represented as a parametric polyhedron:

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

The notion of iteration domain is extremely useful as it characterizes exactly all the executed instances of a given statement. This characterization is at the base of the rest of the model. The execution of the program can be seen as scanning the integer points in an iteration domain. The execution order is then the order used to perform this scanning. This strong connection between the iteration domain and the loop nest leads to consider that a loop level in the source code is similar to a dimension of the iteration domain.

As the constraints defining the iteration domain are affine inequalities, we can express this iteration domain as a set of affine constraints. Using the matrix representation defined earlier, we can represent the iteration domain of the loop shown in Figure 2.1 as being:

$$\mathcal{D}^S = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} \begin{pmatrix} N \end{pmatrix} + \begin{pmatrix} -1 \\ 0 \\ -1 \\ 0 \\ 9 \end{pmatrix} \geq 0$$

A more compact representation is commonly considered, appending the column of the two matrices with the vector, resulting in the following representation:

$$\mathcal{D}^S = \left[\begin{array}{rr|r|r} 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & -1 & 1 & 0 \\ -1 & -1 & 0 & 9 \end{array}\right] \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} \geq 0$$

2.4.5 Access Functions

To guarantee that an exact program analysis can be performed later, we require that the memory accesses are defined by array accesses where the subscript functions are affine. The scalars are treated as arrays where only the first element is accessed. In that frame, we can represent the addresses reached by memory references as couples $\langle \mathrm{A}, f \rangle$, where A is the name of the array and f is a multi-dimensional affine function mapping the loop indices to an array element.

To precisely describe the behavior of a statement S, we distinguish the values which are written in memory and those only read when S is executed. We note $\mathcal{W}^S$ the set of memory writes performed by S, and $\mathcal{R}^S$ the set of read accesses in S.

Consider the following instruction, where `i` and `j` are the indices of the two surrounding loops:

```
A[i][j+1] += B[i+j][j] - c;
```

This statement contains the following memory references:

$\mathcal{W}^S = \{\langle \mathtt{A}, f_{\mathtt{A}} \rangle\}$

$\mathcal{R}^S = \left\{\langle \mathtt{A}, f_{\mathtt{A}} \rangle, \langle \mathtt{B}, f_{\mathtt{B}} \rangle, \left\langle \mathtt{c}, f_{\mathtt{c}}(\vec{i}) = 0 \right\rangle\right\}$

The array subscript functions are defined as:

$$f_{\mathtt{A}}(\vec{i}) = \begin{pmatrix} \mathtt{i} \\ \mathtt{j}+1 \end{pmatrix}$$

$$f_{\mathtt{B}}(\vec{i}) = \begin{pmatrix} \mathtt{i}+\mathtt{j} \\ \mathtt{j} \end{pmatrix}$$

The affine subscript functions can be represented using the matrix representation. If a single parameter N is used in the SCoP, the affine functions have the following form:

$$f_{\mathtt{A}}(\vec{i}) = \left[\begin{array}{cc|c|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{array}\right] \begin{pmatrix} \mathtt{i} \\ \mathtt{j} \\ N \\ 1 \end{pmatrix}$$

$$f_{\mathtt{B}}(\vec{i}) = \left[\begin{array}{cc|c|c} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{array}\right] \begin{pmatrix} \mathtt{i} \\ \mathtt{j} \\ N \\ 1 \end{pmatrix}$$

This representation is commonly used as it is very compact and can be easily implemented on computers using integer matrices.

2.4.6 Schedule

We have defined several representations allowing us to describe the execution of a SCoP. However, we still lack of a fundamental feature as we are not yet able to describe the relative execution order of the statement instances. For that purpose, we assign a logical execution date to every statement instance. The logical dates define an order which can be used to sort the instances by execution order. To assign an execution date to every instance, we use *scheduling functions*.

Definition. (*Scheduling function*) The *scheduling function* of a statement S, or *schedule* of S, is a function that maps every instance of S to a logical timestamp, defining when this instance can be executed relatively to the other instances executed in the SCoP.

$$\forall \vec{i} \in \mathcal{D}^S,\ \theta^S(\vec{i}) = t$$

```
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++) {
      S1
      S2
   }
```

Figure 2.2: Sample loop nest with two statements.

Timestamps

The timestamps t can be scalar values [48], however Feautrier has shown that unidimensional timestamps cannot represent the schedules in every class of programs. He has proposed in [49] to use instead multidimensional timestamps, which are able to represent the schedules in every possible SCoP.

The multidimensional timestamps are vectors which can be seen as logical clocks. The most important value is the first one, while the last vector element is the less important. Intuitively, this is equivalent to vectors containing days, hours, minutes, seconds, ... Two multidimensional timestamps are sorted according to the lexicographic order, noted $\ll$, and defined as:

$$(a_1, \ldots, a_n) \ll (b_1, \ldots, b_n) \Leftrightarrow \exists i : 1 \leq i \leq n, \forall m : 1 \leq m < i, \, a_m = b_m \wedge a_i < b_i$$

The timestamps, once lexicographically sorted, describe the original execution order of their associated statements instances.

One could think that the iteration vectors can be directly used as multidimensional timestamps to characterize the original sequential execution order. However, it cannot express the textual order information which also accounts to determine the execution order. For instance, if we consider the loop nest in Figure 2.2, we can see that during the whole execution of the nest, both statements have the same iteration vectors despite they are not executed exactly at the same time: S_1 is executed before S_2.

A simple solution to that problem is to interleave the iteration vector entries with constant values to represent the relative textual order at every common loop level. In our example, the following timestamps can be used to represent the sequential execution order:

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

The two first constant values are identical, representing the fact that both statements are enclosed in the same loops. The last constant value indicates that S_2 is executed after S_1 at every iteration of the innermost loop.

Interestingly, the timestamps can still be sorted by lexicographical order to represent the execution order of their associated instances. This solution has been proposed by Feautrier in [49], and can be applied automatically by traversing the AST to determine the relative textual position of the statements.

Scheduling Matrices

We impose that the schedules are affine functions in order to perform code generation later [12]. Thus, we can use a matrix to represent a schedule, leading to the following definition:

$$\forall \vec{i} \in \mathcal{D}^S,\ \theta^S(\vec{i}) = \Theta^S \vec{i} = \vec{t}$$

Where Θ^S is the *scheduling matrix* of S, and is such that $\Theta^S \in \mathbb{Z}^{d^t \times (d+p+1)}$ with $d^t = \left|\vec{t}\right|$.

A canonical form of the scheduling matrix proposed by Girbal et al. in [11, 34]. This format encodes the scheduling matrix as:

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

In this format, the scheduling matrix is actually made of three sub-matrices: A^S, Γ^S, and β^S. The first one, A^S, is related to the loop indices and is then able to represent any combination of loop interchange, reversal, and skewing. The second component, Γ^S, is applied on parameters and can represent shifting transformations. The last sub-matrix, β^S, is in charge of encoding the relative textual position of the statements, which allows one to perform loop fission, loop fusion, and code motion. The two matrices A^S and Γ^S are the *dynamic components* of Θ^S as they can reorder the instances of S, while β^S is the *static component* of Θ^S, in charge of representing the relative order of the statements in the source code.

We can use this format to describe the original execution order of our example from Figure 2.2. The resulting matrices actually generate the same timestamps as presented earlier.

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

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

The scheduling matrix can be exploited to encode different execution orders than the original sequential execution order. The modifications applied on matrices have a direct impact on the represented execution order. Thus, as shown later, we can perform code transformations directly in this representation.

2.4.7 Data Dependence

Before performing code transformations, we have to define a tool able to represent the correctness of a program with regard to the original sequential semantics. The *data dependences* have been created for that purpose.

Definition. (*data dependence*) Two statement instances $\left\langle S_a, \vec{i_a} \right\rangle$ and $\left\langle S_b, \vec{i_b} \right\rangle$ are data dependent if they access to the same memory location, and if one of those accesses is a write. By extension, the statements S_a and S_b are said data dependent if one of the following case occurs:

$$\mathcal{W}^{S_a} \cap \mathcal{R}^{S_b} \neq \emptyset$$

$$\mathcal{R}^{S_a} \cap \mathcal{W}^{S_b} \neq \emptyset$$

$$\mathcal{W}^{S_a} \cap \mathcal{W}^{S_b} \neq \emptyset$$

If $\left\langle S_a, \vec{i_a} \right\rangle$ is executed before $\left\langle S_b, \vec{i_b} \right\rangle$, the statement S_a is called the *source* of the dependence, and S_b is the *destination*. Different kinds of data dependences are distinguished, depending on the order of the memory accesses performed. If we consider two instances a and b accessing the same location with a being executed before b, then, the following dependences can occur:

- A *true dependence* ($a\delta b$) if a writes a memory element later read by b. Those dependences are also called Read-After-Write (RAW) dependences.
- An *anti dependence* ($a\delta^{-1}b$) if a reads a memory element overwritten by b afterwards. They are also called Write-After-Read (WAR) dependences.
- An *output dependence* ($a\delta^{o}b$) if a writes to a location overwritten by b afterwards. They are also called Write-After-Write (WAW) dependences.

A WAR or WAW dependence appears when a memory location is reused. Such dependence can be removed by separating the memory locations used by the two instances. Techniques such as scalar expansion or array expansion can be used for that purpose [47, 26]. On the other hand, the true dependences capture the program semantics and necessarily have to be considered when applying code transformations on the program.

Data dependence is of major importance because it is strongly related to the program semantics. If two instances are data dependent, then they cannot be executed in any order as it may impact the program semantics. For example, if an instance a is writing to a memory location read later by another instance b, it is incorrect to execute a after b, which would read an incorrect value in that case. However, nothing prohibits the reordering of two instances which are not data dependent. The schedules representing execution orders which preserve the correct ordering of the dependent instances, are said *legal*, or *valid*. Notice that the correct ordering of dependent instances is defined by the original sequential program.

We use a compact representation of the dependences proposed by Feautrier in [46]. In this representation, considering two dependent statements S_a and S_b, we build a *dependence polyhedron* $\mathcal{P}_e$ which is a subset of the Cartesian product of the iteration domains of S_a and S_b. To be correctly defined, a dependence polyhedron contains the following constraints:

- The instances of the two statements have to exist, i.e. their iteration vectors have to be in the iteration domain of the statement: $\vec{i_a} \in \mathcal{D}^{S_a}$ and $\vec{i_b} \in \mathcal{D}^{S_b}$.

```
  for (i = 0; i < N; i++) {
S1:  b[i] = 0;
     for (j = 0; j < N; j++)
S2:     b[i] = b[i] + A[i][j] * x[j];
  }
```

Figure 2.3: A matrix-vector product.

- The two accesses have to access the same memory location. This is the case when the two conflicting memory references use the same array and access to the same element: $f^{S_a}(\vec{i_a}) = f^{S_b}(\vec{i_b})$.
- The source has to be executed before the destination in the original execution order: $\theta^{S_a}(\vec{i_a}) \ll \theta^{S_b}(\vec{i_b})$.

A dependence between a source statement S_a and S_b, the destination, is then expressed using the following notation $\delta(S_a, S_b, \mathcal{P}_e)$.

Consider the matrix-vector product presented in Figure 2.3. There is a true dependence provoked by the write `b[i]` in S_1 which is read later in S_2. The corresponding dependence polyhedron $\mathcal{P}_e$, defining all the conflicting instances $\langle S_1, (i_1)\rangle$ and $\langle S_2, (i_2, j_2)\rangle$ can be expressed by the following constraints:

$$\mathcal{P}_e = \left\{ ((i_1), (i_2, j_2)) \;\middle|\; \begin{array}{c} 0 \leq i_1, i_2, j_2 < N \\ i_1 = i_2 \\ i_1 \leq i_2 \end{array} \right\} \begin{array}{l} \text{Iteration domains} \\ \text{Same element accessed} \\ \text{Source before destination} \end{array}$$

This polyhedron represents all the pairs of iterations vectors (i_1) and (i_2, j_2) such that $\langle S_1, (i_1)\rangle$ is dependent with $\langle S_2, (i_2, j_2)\rangle$.

This representation is very convenient as all the dependent instances are represented in a polyhedron which can be expressed as matrices describing a set of affine constraints.

A legality property can be expressed using this notation: considering a given dependence $\delta(S_a, S_b, \mathcal{P}_e)$, the schedule expressed by θ^{S_a} and θ^{S_b} is legal iff

$$\forall(\vec{i_a}, \vec{i_b}) \in \mathcal{P}_e,\ \theta^{S_a}(\vec{i_a}) \ll \theta^{S_b}(\vec{i_b})$$

2.4.8 Transformation

The data dependences constrain the execution orders which produce the same result as the sequential execution. Despite those constraints, the instances can often be executed in a different order while preserving the program semantics. Such new execution order can be defined by new scheduling matrices.

Definition. (*program transformation*) A *program transformation* is a transformation of the statement instances execution order. It is often performed in order to enhance the locality of the program, or to exhibit a parallel loop level for instance. The result of a program transformation is expressed by scheduling matrices Θ^S associated to the statements S.

Obviously, we are interested in legal schedules, where the dependent statement instances are executed in the same order as in the original program. Two different approaches have been proposed to constrain the transformations by the dependences.

One could first envisage to build a schedule and test if it is legal with regard to the dependences afterwards. Different methods can be used to determine if the schedule is legal. The simplest one consists in ensuring that the legality constraint is respected in every dependence polyhedron.

Consider a dependence $\delta\,(S_a, S_b, \mathcal{P}_e)$, then, to test if the schedules θ^{S_a} and θ^{S_b} lead to a semantically correct execution, we augment $\mathcal{P}_e$ with the following constraint:

$$\forall(\vec{i_a}, \vec{i_b}) \in \mathcal{P}_e, \theta^{S_b}(\vec{i_b}) \ll \theta^{S_a}(\vec{i_a})$$

which is the inverse of the legality constraint. The emptiness of the polyhedron is then computed. If this augmented dependence polyhedron is empty, i.e. if it contains no point with integer coordinates, then, the scheduling functions are correct with regard to this dependence. The test performed actually ensures that there is no couple of dependent instances such that the destination of the dependence is executed before the source, considering the execution order described by θ^{S_a} and θ^{S_b}. The emptiness check can be performed by a Fourier-Motzkin elimination restricted to the integer solutions [108]. Notice that, if the schedules are not legal, Vasilache et al. have proposed in [137] a solution to correct them by loop-shifting whenever possible.

The second approach to constrain the transformations by the dependences, is to express the space of the valid transformations and to consider only the program transformations in this space. This characterization is based on the affine form of the Farkas Lemma [125]:

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

$$f(\vec{x}) = \lambda_0 + \vec{\lambda}(A\vec{x} + \vec{b}), \textit{ with } \lambda_0 \geq 0 \textit{ and } \vec{\lambda} \geq 0$$

Consider a dependence $\delta\,(S_a, S_b, \mathcal{P}_e)$. Let Θ'^{S_a} and Θ'^{S_b} be two scheduling matrices whose entries are unknown. Those matrices represent all the possible schedules for S_a and S_b. We want to constrain their entries to only represent the legal schedules with regard to the dependence. In order for the dependence to be considered, the source has to be executed before the destination. This can be expressed as

$$\forall(\vec{i_a}, \vec{i_b}) \in \mathcal{P}_e,\ \Theta'^{S_b}(\vec{i_b}) - \Theta'^{S_a}(\vec{i_a}) \gg 0$$

We can represent this difference in the form presented in the Farkas lemma, leading to the following equality, where $A^{\mathcal{P}_e}$ and $\vec{b}^{\mathcal{P}_e}$ define the constraints defining $\mathcal{P}_e$:

$$\forall \vec{i_a}, \vec{i_b},\ \Theta'^{S_b}(\vec{i_b}) - \Theta'^{S_a}(\vec{i_a}) = \lambda_0 + \vec{\lambda}(A^{\mathcal{P}_e}(\vec{i_a}|\vec{i_b}) + \vec{b}^{\mathcal{P}_e}), \text{ with } \lambda_0 \geq 0 \text{ and } \vec{\lambda} \geq 0$$

On the left-hand side, all the entries of the iteration vectors $\vec{i_a}$ and $\vec{i_b}$ have unknown coefficients defined by the generic scheduling matrices. On the right-hand side, those

vector entries coefficients depend on $\vec{\lambda}$, and on the coefficients of the dependence polyhedron matrix $A^{\mathcal{P}_e}$, which are known. Those iteration vectors coefficients of both sides are equated, resulting in a new system involving the Farkas multipliers λ_0 and $\vec{\lambda}$ and the generic scheduling matrices entries. This system is augmented with the positivity constraints on the Farkas multipliers, exploiting the Farkas lemma. Using the Fourier-Motzkin projection algorithm [125], one can eliminate the Farkas multiplier from the system and build a set of constraints only involving the scheduling matrix coefficients. Those constraints represent the space of valid transformations only. More details on this method and examples can be found in the literature, as for instance in [103]. This second approach is the most common one and is used in many existing scheduling methods [48, 49, 78, 22, 103].

Once a transformation is chosen for the SCoP, with one of those two presented methods, one can determine if a dimension in the new execution order is parallel. We introduce the notion of *loop-carried* dependence for that purpose. A dependence is *carried* by a loop if the same memory element is accessed at two distinct iterations of that loop. A loop that carries no dependence is parallel.

In order to determine if it exist a loop at level l which carries the dependence $\delta(S_a, S_b, \mathcal{P}_e)$, we add the following constraint to every dependence polyhedron in the SCoP:

$$\forall(\vec{i_a}, \vec{i_b}) \in \mathcal{P}_e,\ \Theta^{S_b}[l,*] \cdot \vec{i_b} < \Theta^{S_a}[l,*] \cdot \vec{i_a}$$

If this augmented dependence polyhedron is not empty, then, some instances are such that the destination is executed at a different iteration than the source, meaning that the loop at this level carries the dependence. If several loops carry the dependence, it is considered that only the outermost loop carries the dependence.

For instance, consider the example presented in Figure 2.3. In this example, we focus on the dependence provoked by the write to `b[i]` performed in S_1 and the corresponding read in S_2. The corresponding dependence polyhedron has been presented before. We perform a loop fission on the original program to move the initialization S_1 before the computation nest, resulting in an initialization loop with only S_1 followed by a different computation nest with S_2 only. This transformation is legal and can be expressed as the following scheduling matrices:

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

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

In order to determine if the dependence prohibits the parallelization at the outermost level, we test for the emptiness of the augmented dependence polyhedron:

$$\mathcal{P}_e = \left\{ (i_1, i_2, j_2) \;\middle|\; \begin{array}{c} 0 \le i_1, i_2, j_2 < N \\ i_1 = i_2 \\ i_1 \le i_2 \\ 1 \times i_1 > 1 \times i_2 + 0 \times j_2 \end{array} \right\} \begin{array}{l} \text{Iteration domains} \\ \text{Same element accessed} \\ \text{Source before destination} \\ \text{Parallelization test for level 0} \end{array}$$

We can see that the extra constraint contradicts the others. Thus, the augmented dependence polyhedron is empty, meaning that the dependence is not carried by the loops at this level. If similar results are obtained with all the dependences, the loops at this level can be parallelized.

2.4.9 Code Generation

Once a transformation is chosen, an interesting feature in the polyhedral model is the ability to generate a program which can execute the SCoPs according to the new execution order defined by the transformed schedules. This generated code actually scans the points in the iteration domains of the statements using the new execution order.

Bastoul et al. have designed CLooG [12] which is currently one of the most efficient code generator in the polyhedral model. It significantly outperforms previous techniques such as Kelly's et al. in the Omega framework [65]. CLooG is an extension of the algorithm proposed by Quilleré et al. in [110] which was the first algorithm able to directly eliminate redundant control in the generated code, as opposed to previous methods where a naive code was generated and simplified afterwards.

This code generator can produce very efficient codes to scan polyhedra. However, they deal with complex problems where a lot of cases has to be evaluated separately, possibly leading to large compilation times for some unfavorable inputs. For more details about CLooG, the reader is referred to Bastoul's thesis [13].

In our different proposals, we use CLooG, directly or through polyhedral compilers, in order to generate transformed programs once a transformation has been chosen.

2.4.10 Extensions to the Polyhedral Model

Starting from the base representation and techniques presented before, several extensions have been developed. As the polyhedral model accepts only a very restricted set of loops; extending its scope is a natural evolution. Several researchers have proposed to extend the polyhedral model representation to handle more complex code structures such as `while` loops or data dependent conditions. Collard has proposed a speculative model to transform and parallelize loop nests with a `while` loop in [36]. In his proposal, some iterations of a `while` loop can be optimistically executed, and then reverted using a back-tracking mechanism, when it turns out that those iterations should not have been executed. On the other hand, Griebl et al. have proposed a conservative scheme where only relevant iterations are executed. The two main problems are then to know if a particular iteration has to be executed, and to determine when the execution has ended (see for example [76]). More recently, Benabderrahmane et al. have proposed in [14] an extension to the polyhedral model representation where the statements can be predicated by data-dependent conditions. They show that, with an adequate code generation algorithm, implemented in irCLooG, most of the expressiveness of the polyhedral model can be used on loop nests with `while` loops and data-dependent conditions.

Despite those proposals allow the parallelization of complex code structures, they

generally rely on common linear dependence analysis, or approximate non-linear dependence analysis such as Fuzzy Array Dataflow Analysis [37] or the Range Test [18]. To parallelize complex binary codes in Chapter 3, we use a more precise approach based on the symbolic Bernstein expansion strategy proposed by Clauss and Tchoupaeva in [31].

Grösslinger et al. present in [56] another extension targeting non-linear iteration domains and show how it can be used to handle parametric tiling for instance. Despite they adapt the polyhedral model to handle some more complex programs, they are specifically targeting non-linear domain constraints. Moreover, they perform complex operations on polynomials which limits the scalability of their approach.

Recently, Baghdadi et al. have presented in [7] a preliminary study targeting a synergistic use of the polyhedral model both offline and online. The speculative parallelizer presented in Chapter 5 addresses similar issues although our approach is more dynamic-oriented.

Tiling is an important optimization which can have a major impact on the program performance. The polyhedral model can represent with no difficulty a tiled iteration domain when the tile sizes are fixed, although efficient code generation might be an issue [54]. However, the model cannot naturally handle parametric tiling, where the tile size is a compile-time parameter instantiated at runtime. Several systems have been proposed to address the different issues of parametric tiling in the polyhedral model. Renganarayanan et al. have proposed TLOG [117] which can generate sequential parametrically tiled code. They have later extended it with HiTLoG [69] to handle multiple levels of tiling. Those systems are limited to perfectly nested loops but the authors show in [68] how to handle imperfectly nested loops. Another approach to handle imperfectly nested loops is proposed by Hartono et al. with PrimeTile [59]. The parallelization of those loop nests can be performed by PTile [10] or with the fully dynamic system DynTile [58]. Those different tools are often based on the polyhedral representation and polyhedral code generation tools which serve as a base for the overall tiling process. In some sense, the parametric tiling is a selection system which can be compared to our proposal in Chapter 4. Although parametric tiling systems can represent an arbitrary number of versions corresponding to the different tile sizes, our selection mechanism is not restricted to tiling and can handle the full set of transformations defined the polyhedral model.

2.4.11 Tools

In the polyhedral model representation, many tools are available and allow a comfortable but powerful manipulation of the different representations. We use several of those techniques in next chapters. In order to give to the reader a quick overview of the polyhedral model opportunities, we present here several tools used to implement the techniques described later.

Several geometric operations are useful to directly manipulate polyhedra. For instance, computing the union or the intersection of two polyhedra is a commonly performed operation. The convex hull of two polyhedra can be helpful to build simple approximation. Some affine transformations can also be applied on polyhedra.

Finding the lexicographic minimum or maximum in a polyhedron is a very useful

operation. It can be used for instance to determine what is the iteration vector of the first valid iteration in a domain, to determine a random point in a polyhedron, or to check for emptiness. The emptiness operation can also be performed using the Fourier-Motzkin elimination which solves a system of affine inequalities.

Another important operation is the computation of the number of integer points in a polyhedron. The resulting count is expressed as a parametric Ehrhart polynomial which is a piecewise affine polynomial depending on the polyhedron parameters [32]. This counting can serve several purposes such as estimating the memory use of a SCoP.

Concrete implementations have been proposed to perform all these operations. The PolyLib [101] and the Parma Polyhedral Library [107] provide facilities to manipulate polyhedra. PIP [45] allows to determine the lexicographic minimum in a parametric polyhedron. The PolyLib and ISL/barvinok [139, 138] are able to build the Ehrhart polynomials counting the integer points in a polyhedron. Currently, ISL [138] is one of the most advanced tools and implements many important operations on integer sets and relations.

The polyhedral compilers are probably the most evolved tools exploiting the polyhedral model. Those compilers use internally the polyhedral representation to perform different operations from memory analysis to automatic transformation and parallelization of loop nests. Some production-scale compilers implement the polyhedral model such as IBM XL/C, the GRAPHITE extension of GCC [135], Polly [55] in LLVM, or the source-to-source parallelizer R-Stream [119] from Reservoir Labs. We extensively use PLUTO [23, 99] in our different proposals which is a research-level polyhedral compiler able to perform automatic transformations and parallelization of SCoPs, while optimizing communications and locality. Another relevant tool is the PoCC framework and more precisely LetSee [105, 104] which uses iterative compilation to determine an efficient transformation to apply on programs. This tool can be used to automatically generate efficient versions for our code selection mechanism.

2.5 Conclusion

We have presented the most relevant techniques related to our work. This should give to the reader a better understanding of the relation between our proposals and the existing techniques. It also presents the main issues with those techniques which are solved in this thesis.

We have also introduced the important notions related to the polyhedral model to delimit the possibilities in this model. We do not extend the model in this dissertation. Instead, we use the existing representations, techniques, and tools to perform advanced binary code parallelization, efficient dynamic code selection, and speculative polyhedral transformations. We show that the polyhedral model can be successfully used in difficult contexts to enhance the performance of programs which are not initially considered as its natural targets.

Chapter 3

Binary Code Parallelization

3.1 Introduction

There is a persistent hiatus between the software vendors, having to distribute generic programs, and the users, running them on a variety of hardware platforms, with varying amount of available parallel hardware resources. The next decade may well see an increasing variety of parallel hardware, as it has already started to appear in the embedded system market. In the same time, one can expect more and more architecture-specific automatic parallelization techniques, e.g., GPU or FPGA oriented scheduling algorithms. Therefore, the widening gap between software production and its execution becomes a critical issue in the adoption of parallelism as a mean to efficiently build, deploy, and use computing infrastructures.

In this chapter, we present an approach that could be named *parallelization as a service*, whose aim is to solve that issue. In this setting, the operating system or runtime environment provides tools that take sequential binary programs and transforms them into parallel executable code. The transformation is performed statically, e.g., at installation time, and may use any source-to-source parallelization back-end. As it is performed on binary programs, closed-source third party applications, legacy programs, and the libraries used by the program can be parallelized, independently from their original source language.

The basic strategy of a static binary program parallelizer can be summed up in three phases: raising, parallelizing, and lowering. The *raising phase* has to bring the binary code into an intermediate representation that is precise enough to conduct any analysis required for parallelization. The *parallelizing phase* uses this result to generate a set of transformations to apply to the code. Finally, the *lowering phase* is in charge of creating the resulting binary program, ready to run on the target hardware. The system we propose works applying this scheme on x86-64 binary programs. It targets loop-intensive kernels and applies different parallelization strategies depending on the code complexity. This includes advanced polyhedral transformations on relevant loop nests. To our knowledge, such advanced parallelization of binary codes, if already considered, has never been implemented before.

We present in Figure 3.1 an overview of our parallelization system in which the three main steps can be clearly identified. The raising phase transforms the analyzable

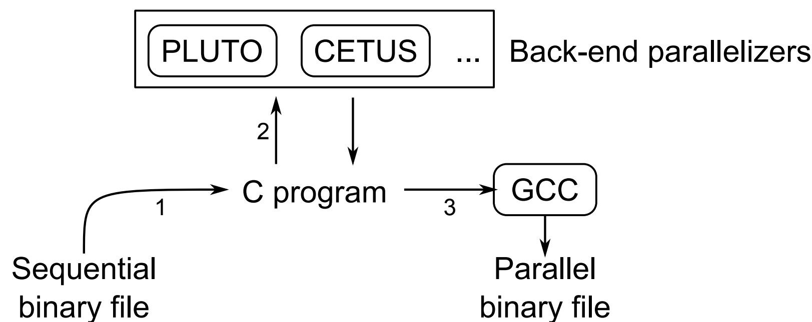

Figure 3.1: Overview of the parallelization system. (1) Analyzable portions of the sequential binary program are raised to simple C loop nests only containing the correct memory accesses, and (2) sent to a back-end source-to-source parallelizer. (3) The resulting parallel loop nest is filled with actual computations, and compiled using a standard compiler.

loop nests in the sequential executable file to restricted C programs which contain only analyzable loop nests and memory references. Those programs are sent to a back-end source-to-source parallelizer such as PLUTO [23], a polyhedral parallelizer, or CETUS 1.3 [6], a non transforming parallelizer. The original computations on the memory values are re-injected in the resulting parallelized loop nest, which is then compiled using a classical C compiler.

Our static approach contrasts with the existing binary code parallelizers where a dynamic and speculative approach is often chosen by the authors [40, 60, 94]. Despite the static approach limits the complexity of the programs that can be treated, it provides two main advantages over the existing dynamic methods. First, our static system has no runtime overhead and does not require any speculation framework nor any hardware mechanism to support speculation. Second, we can perform advanced loop transformations which often greatly improve the performance of the parallel codes. Our approach is in fact complementary to the dynamic approaches: our system should be used to parallelize loop nests that can be handled statically, while the more complex codes should be parallelized using the existing dynamic binary parallelizers. This static system is also the first brick of our general approach and statically applies polyhedral transformations on the programs when their format is complex.

The raising phase is detailed in Section 3.2. We explain how the binary code is analyzed, and what kind of intermediate representation is extracted. In Section 3.3, we detail some adaptations performed on the intermediate representation in order to meet the usual requirements of our source-to-source parallelizers, and we also describe the lowering phase. We evaluate the efficiency of our approach in Section 3.4 by comparing with equivalent parallelization from the original source code. We also compare our system to the current state of the art of binary parallelization. We extend our technique in Section 3.5 to target more complex codes. Static binary code parallelization has not seen much work in the last years. Before concluding the chapter, we give a detailed comparison of our approach and methodology with the one used in the single paper we have found with similar ambitions in Section 3.6.

3.2 Decompiling x86-64 Executables

Compiled applications are packed in executable files, using the ELF format on Unix for example, which basically contain all the program code and data, and describe the program memory layout. The executable code can be extracted from such files and raised to a more usable representation in an operation called *disassembling*. This operation can be extremely complex for example in the case of obfuscated code or self-modifying codes. We consider those cases as out of scope in this chapter. The representation resulting from disassembling a binary program is a list of assembly instructions. This instruction list cannot be directly parallelized as it is: since we target loop nests, they have to be found back, and the memory accesses have also to be identified in order to perform the dependence analysis prior to the loop nest parallelization. The analysis techniques presented in this section are either basic code analysis methods or recent results of Ketterlin and Clauss [67].

3.2.1 Basic Analysis

The first step is to handle one routine at a time, extracting basic block boundaries and a complete control-flow graph (CFG). A dominator tree is computed from the CFG, and natural loops are recognized and organized into a hierarchy. Routines where the CFG cannot be reconstructed even after applying some compiler-specific heuristics, or routines that contain irreducible loops, are discarded and will not be transformed. Similar passes appear in virtually every optimizing compiler, and excellent descriptions of their details are available in [3, 90].

The next step consists in a data-flow analysis, where each instruction is analyzed to extract the sets of variables it uses and defines. When dealing with binary code, all machine registers, including flags, are considered, and a single monolithic variable called M is used to represent memory. Each definition of M, i.e. each memory store, implicitly uses the currently visible definition of M, i.e., M is weakly updated. The Static Single Assignment (SSA) form of the program is then computed, using essentially the original algorithm [39]. The result is a set of use-def links, as well as new ϕ-functions placed on appropriate basic blocks entry point. The SSA form is the basic intermediate representation on which the rest of the process applies.

3.2.2 Memory Access Description

An accurate description of the memory accesses is required to perform an exact data dependence analysis. The most important knowledge to extract from the assembly code is obviously the addresses targeted by the memory accesses. On x86-64 architectures, addresses appearing in the code are of the form $Base + s \times Index + o$, where $Base$ and $Index$ are register names and s and o are small immediate values, all terms are optional. This expression effectively represents an address, but it is usually equivalent to a function of virtual loop indices and global parameters (symbolic values invariant for the loop execution). Expressing those memory accesses in terms of virtual loop indices and parameters actually simplifies the dependence analysis performed later.

To build back those expressions from the addresses, one can recursively follow SSA use-def links from the uses of *Base* and *Index*. The resulting symbolic expression uses definitions that dominate the memory access whenever possible. This recursive substitution process is applied to all memory operands in all instructions, effectively traversing all instructions that participate in the address computation, while ignoring others.

Given a representation model for the addresses, e.g., the integer linear model, this recursive substitution stops in four distinct situations:

1. when reaching the routine entry point, i.e., when using an input parameter;
2. when reaching an instruction that, after parsing, would lead to an expression that is outside the representation model, e.g., an instruction CVTSD2SI that converts a floating point value into an integer;
3. when reaching one of the ϕ-functions introduced by the SSA form;
4. when reaching a definition that uses memory.

The first two situations are strict limitations: the first is due to the fact that we use an intra-procedural analysis, and the second is due to the expressiveness of our description model. Note that this limitation applies only to instructions that are actually involved in address computations. The last two limitations, however, can be overcome. The ϕ-functions, when resolved by *scalar evolution*, can be replaced by their equivalent expression involving virtual loop indices. This is described below. The following subsections also describe techniques to solve cases where address computations involve memory cells.

At the end of this resolution, every versioned register involved in an address computation is associated with an expression whenever possible. It is then legal to replace every use of these registers version by their equivalent expressions.

3.2.3 Induction Variable Resolution

This section explains how the third limitation to address expression reconstruction, namely ϕ-functions, can be overcome. A ϕ-function at the head of a loop typically expresses the fact that the register (or memory cell) "enters" the loop with an initial value, which is then modified at each iteration. Because we target regular programs manipulating arrays, ϕ-functions used in address computations are, more often than not, linear induction variables. Consider the following typical simple example, where superscripts indicate defined version, subscripts are used versions, and where the line starting with # is a comment giving the definition of the ϕ-function:

```
    mov r9^4, 0x20
L:
    # r9^5 = φ (r9_4, r9_6)
    ...
    add r9^6_5, 0x8
    jmp L
```

A closed form for r9^5 is easy to derive: starting with value 0x20 and incremented by 0x8 at each iteration, r9^5 has value 0x20+0x8×i at the ith iteration.

Therefore, to solve induction variables, we introduce a new, unique virtual counter for each loop, and consider all ϕ-functions that are used in address computations. For any ϕ-function $r = \phi(r_1, r_2)$, where r_2 is the value defined inside the body of the loop:

- if, after complete expression substitution, r_2 is of the form $r + \alpha$ with α being loop-invariant, the definition of r becomes $r_1 + i \times \alpha$;
- if r_2 is of the form $\beta + i \times \gamma$, with both β and γ being loop-invariant, this expression becomes the definition of r if r_1 and β are identical.

In both cases, i is the normalized virtual loop counter, whose initial value is zero and which is virtually incremented by one at each new iteration. Once the definition of a ϕ-function is known, every occurrence of the ϕ-function inside the loop body can be replaced by its definition, thereby replacing loop-varying register occurrences with occurrences of the virtual loop counter and of loop-invariant expressions.

Figure 3.2 shows an extract from the swim_m benchmark of the SPEC OMP-2001 benchmark suite [5]. This figure shows how memory access descriptions are formed, and how induction variables are solved. After register substitution and induction variable resolution, all memory accesses, preceded by @, are expressed in terms of normalized loop counters, along with register definitions that cannot be further substituted. However, in some cases, these definitions can use a value stored in some memory cell, which is often a stack slot. The next section explains how to handle such cases.

3.2.4 Tracking Stack Slots

All but the simplest programs use memory cells to store intermediate results, which are later used in address computations. For the class of programs we target, these memory cells are almost always stack slots. A typical example is given in the following code fragment, where comments give the address expressions obtained so far, rsp$_1$ is the value of the stack pointer upon routine entry, and i is some loop counter:

```
1. mov [rsp+0x20], rax    # address = rsp1 - 0x38
...
2. mov [rcx+8*r9], xmm1   # address = rdi1 + 30416 × i
...
3. mov r12, [rsp+0x20]    # address = rsp1 - 0x38
...
4. movsdq xmm0, [r12]
```

In this fragment, register r12 is used to address memory at line 4. The definition of r12 at line 3 uses memory: recursive substitution is not possible anymore because of the use of a memory operand ([rsp+0x20]). Even though it seems obvious on this fragment that the value assigned to r12 is the value of rax used the instruction at line 1, the substitution process must be able to infer that the instruction at line 2 does not interfere with values stored at rsp+0x20: in x86-64 code, no implicit rule can

Instructions	Definitions	(a)	(b)
mov r15, 0x1	$\texttt{r15}^2 = \text{0x1}$		
[...]	$\texttt{r15}^3 = \phi(\texttt{r15}_2, \texttt{r15}_4)$		1+I
mov rsi, 0x1	$\texttt{rsi}^3 = \text{0x1}$		
[...]	$\texttt{rsi}^4 = \phi(\texttt{rsi}_3, \texttt{rsi}_5)$		1+J
lea r11, [rsi+0x1]	$\texttt{r11}^4 = \texttt{rsi}_4 + \text{0x1}$		2+J
[...]			
movsdq xmm0, [rax+r9*8]		@0x...b0+8×$\texttt{rsi}_4$	@0x...88+8×J
		+30416×$\texttt{r15}_3$	+30416×I
addsdq xmm0, [rax+rbx*8]		@0x...a8+8×$\texttt{rsi}_4$	@0x...80+8×J
		+30416×$\texttt{r15}_3$	+30416×I
mulsd xmm0, xmm4			
mulsdq xmm0, [rax+rdx*8]		@0x..70+8×$\texttt{rsi}_4$	@0x...48+8×J
		+30416×$\texttt{r15}_3$	+30416×I
movsdq [rax+rdx*8+0x...], xmm0		@0x...90+8×$\texttt{rsi}_4$	@0x...68+8×J
		+30416×$\texttt{r15}_3$	+30416×I
[...]			
mov rsi, r11	$\texttt{rsi}^5 = \texttt{r11}_4$	$\texttt{rsi}_4$+0x1	2+J
[...]			
add r15, 0x1	$\texttt{r15}^4 = \texttt{r15}_3\text{+0x1}$		2+I

Figure 3.2: Symbolic address expressions in a loop nest of depth two in binary code in SSA form: (a) after recursive register substitution, and (b) after induction variable resolution. Indentation is used to highlight the loop levels. “@” denotes address expressions.

ensure this property from the code fragment shown above, thus it has to be proved by a further analysis.

To have a tractable and fast memory cell tracker, we have chosen to distinguish between two separate memory regions: first the current stack frame, which is expected to hold intermediate address computations that we would like to track; and second the rest of the address space, which is expected to hold program data, and where we have small hope of being able to follow any data-flow. Our analysis proceeds in two steps:

1. find out which register points to which region, at all program points;

2. use this information to find the "last write" to any memory cell used at some point in an address computations.

The first problem can be solved by associating two bits to each register definition, which indicate whether the register may contain an address inside the corresponding region. At routine entry, the `rsp` register is known to contain an address in the current stack frame only, and all other registers are supposed to point to the rest of the address space. From this initial situation, this simple "points-to" information is propagated along all control-flow edges until a fixed point is reached. For each instruction and ϕ-function, each bit of the defined registers is set by applying the logical OR on the corresponding bits of all used registers. Memory has two pairs of bits, one for the stack frame, the other for the rest of the address space. In practice, this simple analysis is remarkably robust and surprisingly accurate.

Typically, in the example given above, the result of the simple points-to analysis just described would be that:

- both occurrences of $\texttt{rsp}_1\texttt{-0x38}$ represent addresses located in the current stack frame, because so does $\texttt{rsp}_1$;

- address $\texttt{rdi}_1\texttt{+30416}\times\texttt{i}$ represents an address *not* pointing to the current stack frame.

This is enough to assume independence between the use of memory at line `3` and the definition of memory at line `2`.

Once we know where every register points to, the source of the value of a memory cell can often be found, especially if that cell is in the current stack frame. Starting with a use like `[rsp+0x20]` in the instruction at line `3`, all what is required is to be able to follow the chain of the memory stores, from nearest to farthest, while checking whether we have found the specific cell we are looking for. We use a simple decision procedure: if both addresses point to distinct regions, they cannot interfere; else if the difference of their address expressions is not constant, we cannot decide and take a conservative "may" decision; otherwise, if the difference is zero we have found the last write, and if the difference is non zero we can continue the search at the previous memory store.

This simple dependence test is clearly tailored to locate stack slots, whose addresses are usually of the form $\texttt{rsp+}\alpha$, and would give weak results on other memory cells. That's exactly what it was designed for, and gives excellent results on the programs we tested. In our example, addresses at lines `1` and `3` both resolve to the same expression,

namely $\texttt{rsp}_1$-0x38. Therefore, we can insert a synthetic use-def link between the memory operand at line 3 and the memory operand at line 1. Also, parsing the instruction lets one conclude that the address accessed at line 4 is actually equal to the value of rax at line 1, and the recursive substitution can proceed by following the use-def link starting there.

However, there is one point we have hidden so far, namely the case where the search for the source reaches a ϕ-function on M, the variable representing memory as a whole. In that case, a new ϕ-function is created for the specific memory cell whose source is searched for, if such a ϕ does not already exist at that program point. The parameters of this new ϕ-function are initialized with the result obtained by recursively calling the search procedure on the parameters of the ϕ-function on M. Those parameters are then memory accesses possibly aliasing with the searched source. This is actually equivalent to defining "virtual registers" corresponding to specific memory cells. These virtual registers are used like any other, real register, and, in particular, may be subjected to induction variable resolution as explained above. In all situations, memory operands used in memory computations for which one is able to find the source behave exactly like registers, and exact points-to links can be used similarly to the SSA use-def information.

3.2.5 Branch Conditions and Block Constraints

The last aspect of the binary code that needs to be captured are the conditions that govern the control-flow inside loop bodies, from which we derive the loop bounds. To extract these characteristics, we introduce two notions:

- a *branch condition* is a simple logical comparison between zero and an expression, with one of the operators $<$, $\leq$, $>$, $\geq$, $=$ and $\neq$; branch conditions are typically parsed from conditional jump instructions;
- a *block constraint* is a logical combination of branch conditions, of arbitrary complexity, typically represented in disjunctive normal form.

Branch conditions are directly extracted from the binary. Consider the following fragment of code appearing inside a loop with normalized counter i, where the comment shows the result of induction variable resolution on a stack slot:

```
L:
  # M[rsp+0x8] = 0x0 + 0x8*i
  ...
  cmp r15_2, [rsp+0x8]
  jg L
```

From this code fragment, the branch condition is equivalent to $\texttt{0x8}\times\texttt{i} < \texttt{r15}_2$.

Block constraints are constructed with the help of the control-dependences, which are given by the dominator tree of the reverse control-flow graph, i.e. the post-dominator tree. A node w in the CFG is control-dependent on a node u if a condition evaluated by u determines if w is executed. The nodes w control-dependent on u are such that it exists a CFG edge $u \to v$ and w post-dominates v, but w does not strictly

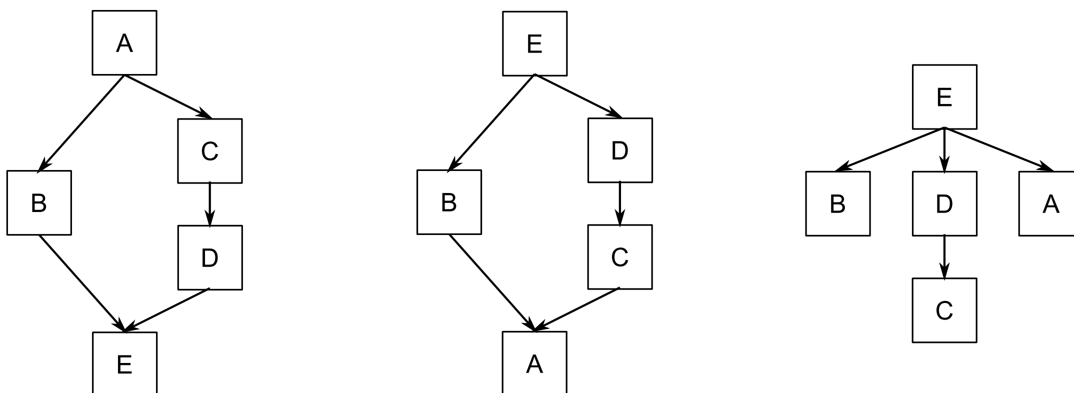

Figure 3.3: Sample CFG (left), its corresponding reversed CFG (center), and post-dominator tree (right).

post-dominates u [90, ch. 9]. The first condition expresses that, if v is executed, it is guaranteed that w will be executed too. The second condition tells us that the end of the CFG can be reached from u without executing w. It means that u ends by a conditional jump and all the blocks w are executed only when the condition of going from u to v is true, provided the control has reached u.

If we note C_u the constraints applying to block u, and $\langle u \to v \rangle$ the condition of going from u to v, then the basic blocks w control-dependent on u are constrained by:

$$C_u \wedge \langle u \to v \rangle$$

This is illustrated in Figure 3.3 with, for example, the edge from A to C in the CFG. Nodes C and D post-dominate C and do not strictly post-dominate A, they are then control-dependent on A. Intuitively, we see on the CFG that a conditional branch is taken when going from A to C and the basic blocks C and D are executed only when this branch is taken. The condition under which those two blocks are executed is then the condition to reach A and the condition of going from A to C. The node E post-dominates C, but also post-dominates A, thus it is not control dependent on A. This is logical as the condition of executing E does not depend on the condition of going from A to C for instance.

To compute constraints on all blocks of a loop with head h, we first set C_h to true, and C_b to false for all $b \neq h$. Then we consider the loop body, i.e., the sub-graph of the CFG restricted to the blocks of the loop with back-edges removed: this is an acyclic graph, which therefore defines a topological order on the blocks. The blocks are then traversed in that order. For each block ending with a conditional jump, we apply the constraint propagation just described. After all blocks have been processed, constraints will have been propagated to all blocks of the loop. The constraints propagated back to the head of the loop will define the condition under which a new iteration is started: if this condition is formed from expressions containing only the loop counter and loop-invariant quantities, then it defines the loop trip-count.

For many common cases, the procedure outlined above produces an expression for the loop bound. Edges branching out of the loop are often placed in such a way that the whole loop can be translated into a for-loop, i.e., the conditions on the loop

counter are the same for all blocks defining the loop body. Conditional control-flow inside loops is anyway preserved, potentially leading to generate tests inside the loop body. The whole decompilation procedure has proved to be extremely effective for the programs we targeted. The next section shows several examples of C code that is directly derived from the SSA intermediate representation, and that can serve as input to the parallelization tools of the following phase.

3.3 Polyhedral Parallelization

At the end of the raising phase, a C source code, equivalent to the assembly representation of the loop nests, is built. For-loops are generated using the virtual loop indices and the discovered loop bounds. The memory accesses are expressed as functions of those loop indices and of some parameters. In this section, we specifically focus on affine loop nests. Other cases are discussed in Section 3.5.

We consider two different back-end source-to-source parallelizers to perform the loop nest parallelization: CETUS 1.3 and PLUTO. CETUS [6] is a classical parallelizer which uses approximate dependence analysis to determine if a loop in the nest can be parallelized. It is not able to perform data locality optimizations. PLUTO [23, 99] is a polyhedral parallelizer which can perform advanced loop nest transformations to enhance data locality and exhibit a parallel loop level. Both parallelizers consider a C program as input, affine access functions and loop bounds, but are very sensitive to several characteristics of the input program. In order to allow them to work at their full potential, we adapt the C code which is extracted from the binary program in order to better suite their needs. This section describes all the adaptations performed.

3.3.1 Memory Accesses

Array Splitting

The memory accesses extracted by the analysis steps all refer to a unique one dimensional array `M` whose base address is zero. As the programs' data segments are usually far from address zero, huge values can appear in the memory access functions. This often makes the dependence analyzers fail. We alleviate this issue in the following way. When the loop bounds are numeric constants, we can statically determine the range of addresses reached by every memory reference. Thus, we can easily split the unique memory array `M` into several different arrays defined over non-overlapping chunks of memory.

The algorithm to split these accesses is fairly simple: for each memory reference, its minimal and maximal addresses are computed using polyhedral tools such as PIP, defining an address space. If this address space overlaps an other one, both are merged together in a single array. Otherwise the address space defines a new array. Once every memory reference has been processed, a set of arrays with their associated base addresses is obtained, and a unique name is assigned to each array. Those arrays can then be used for the dependence analysis, hiding the large memory offsets. For example a reference to `M[0x600e850 + 8*i]` can become the reference `A[8*i]`, considering

```
for (i = 0; i < 10; i++)                    forall (j = 0; j < 10; j++)
   for (j = 0; j < 10; j++)     ——→            for (i = 0; i < 10; i++)
      A[i][j] = A[i+1][j];                        A[i][j] = A[i+1][j];


                                           for (x = 0; x < 64; x++)
for (i = 0; i < 10; i++)                      forall (y = max(⌈(5*x)/6⌉, x-9);
   for (j = 0; j < 10; j++)      ——→           y <= min(⌊(5*x+9)/6⌋, x);
      A[10*i+j] = A[10*i+j+10];                y++)
                                                 A[5*x-4*y]=A[5*x-4*y+10];
```

Figure 3.4: Sample code parallelization using PLUTO.

that the array `A` starts at address `0x600e850`. The parametric cases are addressed specifically in the following sections.

Array Dimensions Rebuilding

We show on Figure 3.4 two equivalent codes (on the left) being parallelized by PLUTO (on the right). On the top, a classical multi-dimensional access is performed leading to a simple loop interchange, while, on the bottom, the access is linearized and PLUTO selects a more complex transformation. We clearly see that the schedule chosen by PLUTO is influenced by the format of the access functions. In the C language, it is assumed that all the array subscript expressions are positive and lower than the dimension size. This leads to a slightly stricter dependence analysis of multi-dimensional accesses and then to better transformations. Unfortunately, after compilation, all the array accesses are linearized; we then rebuild the array dimensions whenever possible.

Consider that the linearized access function has the following form:

$$base + C + \sum_{l=0}^{NL}(\vec{c}[l] \times \vec{i}[l])$$

Where the full expression is a constant C plus a sum of loop indices $\vec{i}[l]$ times a coefficient $\vec{c}[l]$ for each loop level l, while $base$ is the array base address.

A brute-force algorithm presented in Algorithm 3.1 is then run to rebuild the array dimensions. In the algorithm, `dom_max` is the function returning the maximum values of the given expression in the iteration domain. Similarly, `dom_min` returns the minimum expression value in the iteration domain. All the possible dimension sizes are tested in decreasing order. For each tested dimension size, two subscripts are generated to define a two-dimensional array access equivalent to the original one-dimensional access. For instance, A[8*i+j] can be expressed as A[i][j] if the last array dimension size is eight. A dimension size is considered as a possible result if it leads to valid subscripts, i.e. if the right subscript always take positive values lower than the tested dimension size and if the left subscript always take positive values. The algorithm fails if no such solution exists. To rebuild an arbitrary number of dimensions, this algorithm can be repeated on the left subscript as long as it does not fail. In any case, care must be

Algorithm 3.1 Array dimensions reconstruction.

```
max_c ← max(abs(c⃗[l]), 0 ≤ l < NL)
for each dimension size L from max_c to 2
    fn_left ← ⌊C/L⌋
    fn_right ← C % L
    left_min ←⌊C/L⌋
    right_max ← C % L
    for each loop level l from 0 to NL
        coeff_left ← ⌊c⃗[l]/L⌋
        coeff_right ← c⃗[l] % L
        fn_left = fn_left + coeff_left * i⃗[l]
        fn_right = fn_right + coeff_right * i⃗[l]
        left_min ← left_min + dom_min(i⃗[l] * coeff_left)
        right_max ← right_max + dom_max(i⃗[l] * coeff_right)
    if right_max >= 0 and right_max < L and left_min >= 0
        return [fn_left][fn_right] (L)
failure
```

taken to ensure that all references to the same array have the same dimensions (number and sizes). The result of the algorithm is then made of dimension sizes and subscripts realizing a multi-dimensional access equivalent to the linearized access.

3.3.2 Operations

The C code reconstructed from the binary program does not contain the operations on data. Indeed, the processor instructions used in the binary code have often no equivalent representation in pure C code. This is not a problem anyway as, most of the time, the parallelizers only exploit the data and control flows information. To create a parsable C program, we *outline* the operations on data: every operation is considered to be equivalent to a neutral operation, noted $\odot$. This operation can be implemented by any operator in the actual C code sent to the parallelizer, for example "+". Each instruction in the binary code is then represented as an equality where the registers are C variables, and the memory accesses are array references. The written (defined) operand is on the left-hand side of the equality, while the read (used) ones are on the right-hand side, combined by $\odot$.

We show in Figure 3.5, the C code extracted from a matrix multiply binary program. Figure 3.6 shows the same code after splitting `M` into three disjoint arrays and rebuilding the array dimensions. Notice that all the operations on data are outlined in the neutral operator $\odot$.

```
for (t1 = 0; -511 + t1 <= 0; t1++)
   for (t2 = 0; -511 + t2 <= 0; t2++) {
      M[10494144 + 4096*t1 + 8*t2] = 0;
      xmm1 = 0;
      for (t3 = 0; -511 + t3 <= 0; t3++) {
         xmm0 = M[6299808 + 4096*t1 + 8*t3];
         xmm0 = xmm0 ⊙ M[8396960 + 8*t2 + 4096*t3];
         xmm1 = xmm1 ⊙ xmm0;
      }
      M[10494144+4096*t1+8*t2] = xmm1;
   }
```

Figure 3.5: Matrix multiply as it is extracted from the binary code.

```
for (t1 = 0; -511 + t1 <= 0; t1++)
   for (t2 = 0; -511 + t2 <= 0; t2++) {
      A2[t1][8*t2] = 0;
      xmm1 = 0;
      for (t3 = 0; -511 + t3 <= 0; t3++) {
         xmm0 = A1[t1][8*t3];
         xmm0 = xmm0 ⊙ A3[t3][8*t2];
         xmm1 = xmm1 ⊙ xmm0;
      }
      A2[t1][8*t2] = xmm1;
   }
```

Figure 3.6: Matrix multiply after simplifying the memory accesses.

```
for (t1 = 0; -511 + t1 <= 0; t1++)
   for (t2 = 0; -511 + t2 <= 0; t2++) {
      A2[t1][8*t2] = 0;
      xmm1 = 0;
      for (t3 = 0; -511 + t3 <= 0; t3++)
         xmm1 = xmm1 ⊙  (A1[t1][8*t3]⊙ A3[t3][8*t2]);
      A2[t1][8*t2] = xmm1;
   }
```

Figure 3.7: Matrix multiply after forward substitution.

3.3.3 Scalar References

Every reference to a register in the program results in a scalar reference in the C code. These scalar references usually cause many data dependences and could often better be handled by privatization. Although privatization is a simple task in non-transforming parallelizers, it is much more complex in the case of polyhedral compilers, since statements may be moved around by the scheduling transformations. At the time we performed our experiments, support for privatization appeared to be broken in two of the most advanced parallelizing compilers: PLUTO and PoCC [106]. Thus, we have to remove as many privatizable scalar references as possible before parallelizing the C code.

The induction variable resolution performed during the analysis is often able to detect that a given SSA version of a register is equivalent to an expression depending on the loop indices and, sometimes, on parameters. This is especially frequent for registers that contain addresses. It allows us to replace those scalar references by their equivalent affine function. As all the uses of those register versions are replaced, we also ignore the instructions that define them.

Classical forward substitution is also applied, helping to remove many scalar dependencies. Figure 3.7 shows the matrix multiply code after forward substitution. Notice that references to scalar `xmm0` have been removed, but the `xmm1` variable is still present, causing data dependences at each loop level.

The last applied scalar simplification tries to replace those remaining scalar references by array references, considering that they usually generate less data dependencies. We consider scalars whose first reference in the loop nest is a write, and which are not defined as a ϕ-function: they correspond to the privatizable scalars. For each of such scalars, we look for a memory slot unused during the scalar life-span and necessarily written after every reference to the scalar. If this memory slot does not intersect with any other slot accessed since the scalar definition, then it can be used in place of the scalar while preserving the code semantics. The detailed algorithm is given in Algorithm 3.2 and can be seen as a *in-place scalar expansion*, i.e. a scalar expansion with no extra memory requirement.

In the presented algorithm, the terms “before”, “after”, “first”, and “last” actually refer to the execution order in a single loop iteration. When considering a single basic block, the execution order is actually equivalent to the textual order of the instructions

Algorithm 3.2 Scalar to array replacement algorithm.

```
for each scalar s whose first access is a write
   fsref ← first reference to s
   lsref ← last reference to s
   for each memory write mw executed after lsref
      dependence ← false
      for each memory access ma executed after fsref and before mw
         if mw intersects with ma
            dependence ← true
      if not dependence
         replace s by the element written by mw
         continue with the next s
```

```
for (t1 = 0; -511 + t1 <= 0; t1++)
   for (t2 = 0; -511 + t2 <= 0; t2++) {
      A2[t1][8*t2] = 0;
      for (t3 = 0; -511 + t3 <= 0; t3++)
         A2[t1][8*t2] = A2[t1][8*t2]
            ⊙  (A1[t1][8*t3] ⊙  A3[t3][8*t2]);
   }
```

Figure 3.8: Matrix multiply after scalar to array conversion.

in the source code. For multiple basic blocks, this textual order can still be used under two conditions:

1. The first reference to the scalar s must dominate every other reference to s.

2. The memory write mw must post-dominate every reference to the scalar s.

The first condition ensures that the scalar is privatizable, i.e. that, for each loop iteration, the scalar is always written before being used. The second condition ensures that the memory slot used to replace the scalar reference is always written after all the scalar references: whatever is the value of the scalar during its life-span, the memory slot is overwritten later.

Applying this algorithm on our example results in the code shown in Figure 3.8. Notice that no scalar reference remains: the refactored code causes less dependences and is easier to handle by source-to-source parallelizers. If some scalar references remain, they could be handled using classical scalar expansion, but usually at the price of an heavily increased memory consumption.

3.3.4 Parallelization

The main specificity of codes extracted from binary program is their inherent complexity. The assembly instructions often refer to registers, causing many scalar references. The memory accesses have been linearized and rebuilding array accesses is not always a straightforward task. The control flow can also be more complex for some machine-specific reasons: for instance GCC generates data-dependent tests on floating point values to specifically handle `NaN` on x86-64. Finally the compiler optimizations such as loop unrolling, vectorization or basic block merging can transform a simple loop nest in a very complex code structure which is hard to raise to a C loop nest. Those limitations are especially met on highly optimized codes and could probably be overcome by using powerful de-optimization methods.

In our "parallelization as a service" model, we consider that binary executable should be distributed in their intermediate-optimized form, which is already the case in practice for portability reasons. However, even at intermediate optimization levels, some difficulties can remain and the code simplifications we propose are not sufficient to reach a representation simple enough for a common source-to-source parallelizer.

When a simple code representation can be reached, the sequential C program is sent to a back-end parallelizer. This modular approach is very interesting as it means that potentially, any source-to-source parallelizer can be used. We show with PLUTO that even transforming parallelizers can be used, unleashing the power of such advanced tools to binary programs. Currently there is a unique limit put on the back-end parallelizers: they are expected not to split a statement in several others, in order to simplify the code generation performed later. Apart from that, they can perform any code transformation, including duplicating and moving statements in the parallel code.

Once the loop nest has been parallelized, some operations are performed to restore the code semantics that was hidden to the parallelizer, and to link the parallel code to the original binary program. This is the topic of the next subsections.

3.3.5 Reverting the Outlining

The state of the program at that point is a parallel C code where every operation is hidden in $\odot$. Before compiling the code back to a x86-64 binary, we need to revert this outlining and to set back the correct semantics in the parallel code.

One could simply consider replacing each C instruction in the parallelized loop nests by the corresponding assembly code extracted from the original binary program. This would be wrong since the induction variable resolution leads to replace some registers with equivalent affine functions. Thus, the dependence analysis is performed on the code without those register references, and we cannot guarantee a correct semantics if we directly re-inject them. Instead, the registers are represented as equivalent thread-local C variables, and their definitions are evaluated in pure C. The actual operations on these registers are expressed as inlined assembly code. When SIMD instructions are present in the original binary code, they are re-injected using SIMD compiler intrinsics. Indeed, for each SIMD instruction, there is often an equivalent compiler intrinsic function which provides more information to the compiler than the equivalent inline assembly code, generally considered as a black box until code generation.

```
#pragma omp parallel for private(t1,t2,t3,t4)
for (t1=0; t1<=15; t1++)
  for (t2=0; t2<=15; t2++)
    for (t3=32*t1; t3<=32*t1+31; t3++)
      for (t4=32*t2; t4<=32*t2+31; t4++)
        *((double*)(10494144+4096*t3+8*t4)) = 0;

#pragma omp parallel for private(t1,t2,t3,t4,t5,t6,xmm01,xmm02)
for (t1=0; t1<=15; t1++)
  for (t2=0; t2<=15; t2++)
    for (t3=0; t3<=15; t3++)
      for (t4=32*t1; t4<=32*t1+31; t4++)
        for (t5=32*t2; t5<=32*t2+31; t5++)
          for (t6=32*t3; t6<=32*t3+31; t6++) {
            xmm01 = _mm_loadsd((double *)(6299808 + 4096*t4 + 8*t6));
            m128d tmp1 = _mm_loadsd((double*)(8396960+8*t5+4096*t6));
            xmm02 = _mm_mulsd(xmm01, tmp1);
            m128d tmp2 = _mm_loadsd((double*)(10494144+4096*t4+8*t5));
            tmp2 = _mm_addsd(tmp2, xmm02);
            _mm_storesd((double *)(10494144+4096*t4+8*t5), tmp2);
          }
```

Figure 3.9: Matrix multiply after transformation by PLUTO and semantics restoration.

For example, if $\texttt{rax}_3$ has been determined as being equal to $1024 + 8 \times \texttt{i}$, then the instruction `mov rbx`^{7}`, rax`$_3$, is replaced by the following code:

```
long int rbx7;
long int rax5 = 1024 + 8 * i;
asm ("mov %0, %1", rbx7, rax5);
```

We show in Figure 3.9 the matrix multiply code after transformation, parallelization, and code semantics restoration. One can see that the initialization of the result matrix has been split out of the main loop nest by PLUTO. Some C variables, representing hardware registers, are used in addition to some SIMD intrinsic functions. The parallelization is achieved through simple OpenMP directives.

3.3.6 Live-in Registers

In the parallel code, when the code semantics has been restored, the memory addresses are evaluated as C expressions, and each register version becomes a C variable which has to be initialized. Distinct situations lead to generate different code to initialize those variables:

- the register versions for which an equivalent expression is known are simply initialized with this expression;

- the registers removed by forward substitution or in-place scalar expansion are re-introduced as thread-private C variables, and left uninitialized as their first reference is necessarily a write;
- some registers, used in expressions, are initialized before executing the loop nest. If their definition is alive at the loop nest entry, then the value of the corresponding hardware register at the nest entry is used to initialize the variable. Otherwise, the value of the register version is tentatively rebuilt as a linear combination of the other alive register values. If this is not possible, the value of the corresponding hardware register is read during the life-span of the register versions, using an instrumentation code. The different instrumentation codes are merged, as much as possible, into common profiling points where several register values are read, limiting the overhead of this instrumentation

3.3.7 Live-out Registers

An equivalent problem also occurs for registers updated during the loop nest execution and used after it has completed. The loop nest is now parallel and may have been rescheduled. It is then complex to determine in the general case what should be the register value at the end of the loop nest execution. Complex mechanisms could be designed where the threads dynamically check which one of them computes this value, giving it the role to store the register value at the proper iteration. However, it is extremely rare to observe that a register used in the actual computation is read after the loop nest execution.

On the other hand, registers devoted to the control flow are commonly used after the loop nest execution, but their value is known. For instance the register used to store the loop counter can sometimes be used for the next loop nest. When those registers are handled by scalar evolution, the correct value of those registers at loop exit can be easily computed using common polyhedral tools and the corresponding affine function.

3.3.8 Implementation

After parallelization, the parallel loop nests could have been re-injected in the application using an existing binary rewriter such as PLTO [126], or DIABLO [136]. The parallel versions of the loops would then be part of the binary executable and permanent links would be set from the sequential code to the parallelized loop nests.

We actually use another technique which allows a more dynamic behavior: no permanent modification of the original application is performed, and the execution can be easily rerouted, depending on some runtime criteria, to the sequential or to the parallel version. This is especially important for the runtime tests presented in Section 3.5 as the execution flow may be redirected back to the sequential loop nest if they fail. Moreover, the binary executable file is left untouched, allowing one to execute the application in sequential or parallel by running, or not, the dynamic process. This can really ease the debugging process.

In our system, two processes coexist: the main application and an automatically generated runtime component (RC) which is in charge of managing the execution of

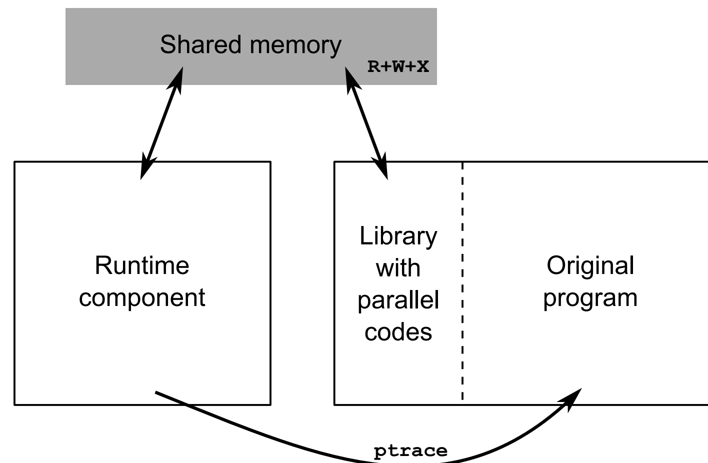

Figure 3.10: General scheme of the implementation.

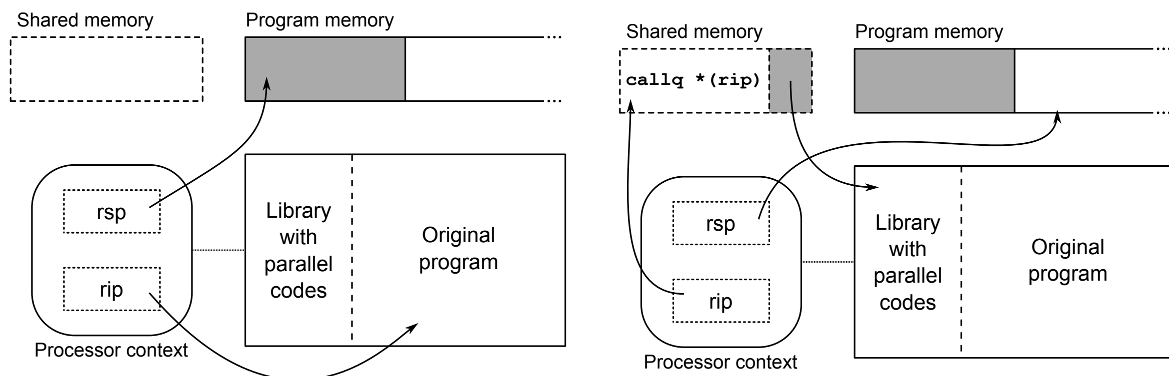

Figure 3.11: Redirecting the execution flow to the parallel loop nests (left to right).

the application. This is illustrated in Figure 3.10. The main application is extended by a dynamic library which mainly contains the parallelized loop nests, compiled in different functions by a standard C compiler at the end of the parallelization process. This dynamic library is loaded in the memory space of the original application at startup using the `LD_PRELOAD` environment variable. A shared memory space is created with all required permissions (including code execution) to allow communication between the RC and the main shared library. Most of those communications are related to executable code positions in both components. The RC has a full control of the application thanks to the `ptrace` system call, which allows it to read and write in the full memory space of the application, including the processor registers.

Before calling its entry point, the original application stalls. The runtime component takes advantage of it to overwrite the head of the loops which have been parallelized with a breakpoint. The program is then run, and the RC stalls.

When a breakpoint is met, the RC redirects the execution flow from the sequential loop nest to the parallel version of the loop nest. To do so, it writes in the shared memory space the instruction `call *(rip)`. This instruction performs a regular function call whose destination address is stored just after the instruction itself. The program counter of the original application hardware context is then set to this shared memory

area: the next instruction that the application executes is this function call. The stack pointer register is set to a free memory space in order for the parallel loop function to execute in a safe environment. Once this is done, the application is resumed from that point. This is illustrated in Figure 3.11 where the original situation (just before the breakpoint is met) is presented on the left, while the state of the program after the RC intervention is presented on the right. On x86-64, the program counter is called `rip` and the stack pointer register is `rsp`.

The nest parallelization is achieved using OpenMP which automatically handle thread management, including thread creation, synchronization, and destruction. Thus, no extra control is required for the thread management. Once a parallel loop nest has finished its execution, the RC is awaken and performs the opposite operation, resuming the execution back to the end of the original sequential loop nest. The overhead of those two code redirections is mainly two `ptrace` system calls per loop nest execution, which is negligible compared to the benefits of the parallelization.

3.4 Evaluation

We have evaluated our implementation on the PolyBench benchmark suite [100]. This suite is made of kernels commonly used in scientific and multimedia codes. The goal of this section is to provide answers to three basic questions one may want to ask about static parallelization of binary code:

- How many loops can be extracted by the decompilation phase? Does the use of binary code entail any loss in coverage? What is the effect of compiler optimizations on coverage?
- Does the use of binary code entails any loss in performance compared to the use of source code? Does this depend on the power of the parallelization back-end?
- How does the performance speedup obtained by automatic binary code parallelization compares to that of hand-made, directive-based parallelization? To other published systems?

The next three sections focus on each of these questions in turn, using an Intel Xeon W3520 processor with four cores and two threads per core, and running Linux as the testing platform.

The reader should be warned of an easily missed characteristic of the Polybench programs: they include a main computation *kernel*, along with an initialization loop nest. This initialization loop is trivially parallelizable and can account for a major part of the execution time. Even though every program is clearly built around its kernel loop nest, some researchers have also included the initialization loop in evaluation runs. In the following experiments, we mention explicitly which part of the programs we use.

3.4.1 Loop Coverage

To evaluate the quality of the decompilation process, we have used three different compilers, each one using two distinct optimization levels. The compilers involved are

Clang 2.8 from the LLVM suite, GCC 4.5, and the Intel ICC compiler version 12.1. Each benchmark program was compiled at the `-O2` and `-O3` optimization levels with each of these compilers. The resulting binary was the submitted to the static binary code decompilation described in Section 3.2.

In all cases, all loops have been correctly *located* in the binary code. However, not all loops are *usable*.

We define the coverage of our system as the number of loops that can be safely transmitted to the parallelization back-end. We have chosen to not make use of "background knowledge" about library function calls. For instance, a single call to `sqrt` in the main loop of the `correlation` program makes the outer loop non parallelizable. This has to be contrasted with source-to-source tools that require explicit directives (e.g. PLUTO), where the programmer is supposed to leave only "harmless" function calls, which are then ignored. We have left the usage of external knowledge on function calls for future research.

Table 3.1 shows counts and percentages for every combination of compiler and optimization level. For each such combination, the table shows first the number of loops that can safely be transmitted to the parallelization back-end, then the total number of loops located in the binary code, and finally the corresponding percentage, with an average at the bottom of the column.

There are two main causes for abandoning a loop. The first one is the presence of function calls. All benchmarks include a final loop nest in charge of printing the computed data structure. This accounts for the vast majority of loops ignored in Clang and GCC binaries. The remaining failures are caused by the difficulty of tracking register values across calls without making simplifying assumptions. Besides `printf`, some PolyBench programs include calls to `sqrt`, and some compiler's handling of this call lead to data-dependent control-flow, which the decompilation phase cannot handle. For instance, GCC use a native processor instruction to perform this operation, except if the value is `NaN`, in which case a library call is performed. The second major cause for abandoning loops is the use of sophisticated program transformations. As it appears from the numbers in Table 3.1, ICC makes heavy use of loop transformations, at both `-O2` and `-O3` levels. In the worst case (`2mm` at `-O3`), less than half of the loops remain analyzable. We suspect this is due to loop tiling, which ICC seems to apply automatically. The problem here is that tiling produces non-strictly linear address expressions, using `max` compute bounds on "side-tiles". Even though we could detect such cases, we think a more general loop restructuring phase is needed to handle highly optimized codes, and have left this phase for future research.

3.4.2 Binary-to-binary vs. Source-to-source

The goal of our second set of experiments is to evaluate whether parallelization applied to binary code is fundamentally inferior to automatic parallelization of the source code. The core of our binary parallelizer uses a source-to-source back-end parallelizer acting on a skeleton program extracted from the binary. Then, we compare the code generated by the back-end parallelizers when their input is 1) the original source code, or 2) the skeleton code extracted from the binary. The experiments in this case are applied to

Program	Clang		GCC		ICC	
	-O2	-O3	-O2	-O3	-O2	-O3
`2mm`	6/8 75%	6/8 75%	6/8 75%	6/8 75%	18/21 86%	10/23 43%
`3mm`	9/11 82%	9/11 82%	9/11 82%	9/11 82%	26/29 90%	15/33 45%
`atax`	6/7 86%	6/7 86%	5/7 71%	5/7 71%	5/7 71%	5/7 71%
`bicg`	5/6 83%	5/6 83%	5/6 83%	5/6 83%	4/6 67%	4/6 67%
`correlation`	10/13 77%	10/13 77%	8/13 62%	8/14 57%	21/27 78%	19/25 76%
`covariance`	11/13 85%	11/13 85%	11/13 85%	11/16 69%	16/22 73%	16/23 70%
`doitgen`	10/13 77%	10/13 77%	10/13 77%	10/13 77%	9/14 64%	9/15 60%
`gemm`	9/11 82%	9/11 82%	9/11 82%	9/11 82%	11/14 79%	4/12 33%
`gemver`	9/10 90%	9/10 90%	9/10 90%	9/10 90%	15/17 88%	11/13 85%
`gramschmidt`	8/10 80%	8/10 80%	7/10 70%	7/10 70%	8/11 73%	8/11 73%
`jacobi-2d`	7/9 78%	7/9 78%	7/9 78%	7/9 78%	5/10 50%	5/10 50%
`lu`	6/8 75%	6/8 75%	4/7 57%	4/7 57%	9/14 64%	9/14 64%
average	81%	81%	76%	74%	74%	61%

Table 3.1: Coverage of our analysis step. The notation X/Y indicates that X loops are successfully raised to a C program which can be treated by a back-end parallelizer among the Y loops present in the program.

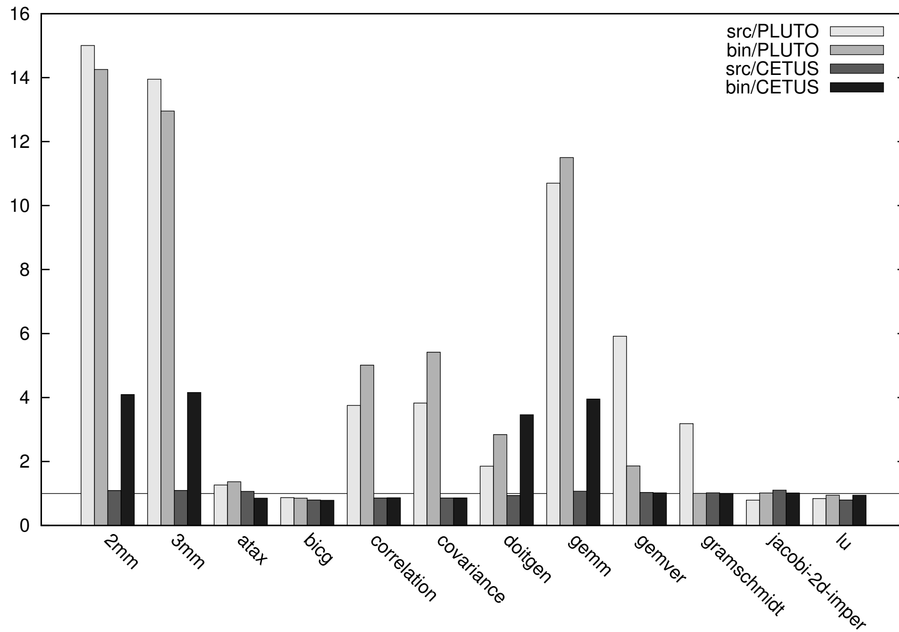

Figure 3.12: Parallelization back-ends applied to 1) the original source code, and to 2) the binary code (kernel only).

the kernel part only.

We have selected two source-to-source parallelizers, CETUS 1.3 and PLUTO, to act as back-end components. Each of them has been used to:

1. generate a parallel version of the source code, which is then compiled with `gcc -O3`;

2. generate a parallel version of the loops extracted from the binary code, which has been compiled with `gcc -O2`.

The resulting speedups are shown on Figure 3.12, with the base execution time being the sequential execution of the program. The goal here is not to compare both parallelizers, since they use significantly different strategies, but rather to compare the speedups obtained on both original source code and source extracted from the binary.

The behavior of both back-end parallelizers is clearly different. In the case of PLUTO, the performance varies slightly between both versions in most of the cases. However, these variations can hide large differences in the nature of transformations applied. For instance, binary-to-binary parallelization slightly outperforms source-to-source for `correlation`, even though fewer loops have been parallelized in the binary case because of the presence of a call to `sqrt`. Conversely, the presence of a function call in `gramschmidt` completely annihilates performance in the binary case, whereas on source code, PLUTO simply assumes that the call may not interfere with the rest of the loop nest. The only case where the difference is significant is `gemver`, where the techniques applied on the binary code to remove the scalar references are insufficient, preventing PLUTO from applying a parallelization as efficient as with the source code.

The case of CETUS is also interesting, in that it doesn't find anything profitable to do on source programs. This is due to the timing functions present in the benchmark programs which make the alias analysis of CETUS 1.3 fail. In some cases this has been corrected when using binary code directly, because the analyzable loop nests are extracted from the rest of the program by our system (in `2mm`, `3mm`, `doitgen`, and `gemm`). In other cases, the loop nests were simply too complex or required parallelizing transformations, which were out of reach of CETUS.

3.4.3 Binary-to-binary vs. Hand-Parallelization

The goal of our last experiment is to compare our solution first to a skilled programmer using OpenMP directives, and second to the best results achieved by a published automatic binary parallelizer (as far as we know). The four contenders are:

- a unnamed programmer using OpenMP directives, placing parallelism directives in the best possible way on the original source code without transformation;
- our binary parallelization system with PLUTO as a back-end (which performs code transformations);
- our binary parallelization system with CETUS 1.3 as a back-end (no transformation);
- the results published by Kotha and colleagues [74] (no transformation).

Numbers in the last category have been taken directly from the paper [74, Section VIII, Table II] when available, using experiments on an architecture similar to the one we have used. No attempt has been made to reproduce their system (see their "Acknowledgments" section). The various speedups are shown on Figure 3.13.

An important aspect of this experiment is that parallelization is applied to the whole program, and not only on the kernel loop nests. For those experiments, all loops in sight have been parallelized when possible, including initialization loops (see the discussion of the PolyBench programs at the beginning of this section). The reader may wish to compare the strong impact of including initialization in the resulting numbers by comparing speedups show on Figure 3.12 with those of Figure 3.13 for similar setting, e.g., on `atax` or `bicg`. We would like to state here that we do not consider whole-program benchmarking to be a significant way to evaluate parallelization systems on the PolyBench suite; we show these results for the sake of comparison with other systems.

The reader should also remember that, among the four parallelizers used, only the PLUTO back-end is able to apply code transformations, giving it a significant advantage on some programs. We have kept the polyhedral back-end in this set of results to illustrate the modularity of our system.

These results can be roughly divided into four categories:

1. `2mm`, `3mm`, and `gemm`, where all non-transforming systems obtain similar results, and polyhedral transformations lead to a spectacular gain; `atax` could be included in this category, except that polyhedral techniques cannot compensate for the fact that the only parallel loop is buried inside a non-parallel outer loop;

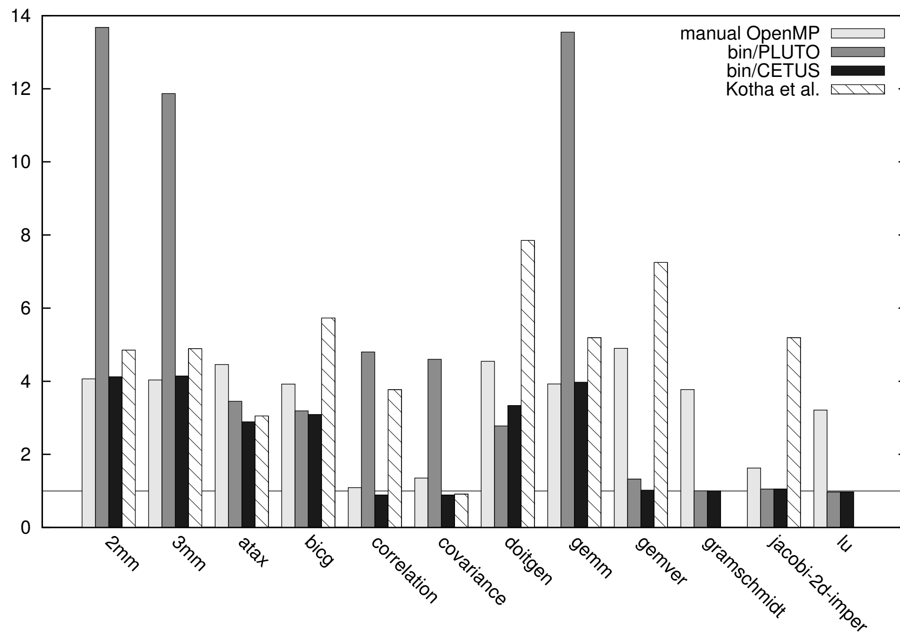

Figure 3.13: Speedup comparison for three parallelization strategies. (initialization + kernel).

2. `covariance` and `correlation`, where polyhedral techniques also dominate clearly, although at a smaller scale: the locality optimization allowed by the polyhedral model has a strong impact, compensating in both cases for the fact that some loops were not parallelized;

3. `gemver`, `gramschmidt`, and `lu`, where the presence of function calls and the complex expansion and privatization requirements make most automatic system fail. The system described by Kotha and colleagues seems to perform very well on `gemver`, but results on the last two programs have not been published;

4. `bicg`, `doitgen`, and `jacobi`, where our automatic system is clearly suboptimal compared to manual, OpenMP parallelization (for reasons similar to the previous category), and where both of these are far below the system by Kotha and colleagues. We have to admit that we have no explanation for this last fact: despite our best efforts (including hand tweaking), we have not been able to even approach the results published in [74], especially on `jacobi`, which is a stencil-like kernel.

Overall, our conclusion on this diverse set of experiments is first that our automatic binary parallelization competes with equivalent systems in most cases, and second that the ability to use polyhedral techniques can make a significant difference in some cases.

3.5 Extension to Complex Codes

Our system is performing well on the kernels provided in the PolyBench benchmark suite. However, those kernels are not representative of all existing programs. Strong restrictions prevent the application of polyhedral transformations on more complex codes. We present in this section some extensions which widen the scope of the handled binary codes. As the codes we target here are more complex, we do not transform the loop nests, but target immediately parallelizable loop nests.

3.5.1 Extension to Parametric Codes

The programs tested in the previous section are all made of non-parametric loop bounds and access functions. However, in many situations, a developer chooses to use parametric loop bounds or dynamic arrays, leading to registers appearing in the code extracted from the binary code. There is no strong restriction preventing our system to handle these cases, as the polyhedral model and its associated tools are already able to solve parametrized problems.

In the case of parametric loop bounds, Kotha et al. [74] propose to group memory accesses according to their base addresses: if the base addresses of two references differ by less than a determined constant threshold, they are considered as being part of the same *reference group* (RG). Memory references in the same group are then considered as referring to the same array. For example, consider to memory references, accessing the addresses $8 \times i + 4000$ and $8 \times i + 4005$. We can consider that both references are actually using the same array `A`, whose base is at address 4000. Then, the first access becomes `A[`$8 \times i$`]` and the second access becomes `A[`$8 \times i + 5$`]`.

Some runtime tests are generated to ensure that the references in different groups are effectively disjoints. The runtime tests proposed by Kotha et al. are based on simple patterns filled by the loop bounds [74, Section III]. This pattern represents the minimal and maximal addresses reached by a memory reference. The address ranges of all the groups are then compared to determine if they are actually disjoints.

In the polyhedral model, some tools such as ISL [138] allows one to determine the minimum and maximum of a linear expression in a given iteration domain. Computing those parametrized minimum and maximum for every memory reference in each group, and checking for range intersection between the groups at runtime, leads to more robust runtime checks, as we can consider non-rectangular iteration domains, and guarded memory accesses. Those runtime tests are generated at compile time and run before the loop nest execution. The general form of those tests is presented in Algorithm 3.3. If one of such test fails, the sequential version of the loop nest is executed as the assumptions made for the dependence analysis cannot be proven.

There are generally a few different reference groups that are generated by this procedure. Thus the time spent performing the runtime check is negligible. In the Algorithm 3.3, the minimal and maximal addresses of reference groups are used. They are simply computed by finding at runtime the minimal or maximal addresses over all the references in the group.

This solution is far from being exact. For example, it can fail if large offsets are used in access functions, or if constants are used in the largest subscript of a multi-

Algorithm 3.3 Generated runtime check to ensure reference groups separation.

For each reference group g_1

a_1^{min} ←minimal address accessed in g_1
a_1^{max} ←maximal address accessed in g_1
For each reference group $g_2 \neq g_1$

a_2^{min} ←minimal address accessed in g_2
a_2^{max} ←maximal address accessed in g_2
if $[a_1^{min}; a_1^{max}] \cap [a_2^{min}; a_2^{max}] \neq \emptyset$
failure (sequential execution required)

```
for (t1 = 0; t1 <= rsi; t1++)
   for (t2 = 0; t2 <= rdi; t2++) {
      M[rax + 8*t1] = ...
      ...
      M[rbx + 8*t1 + 8*t2] = ...
   }
```

Figure 3.14: Original code.

```
for (t1 = 0; t1 <= rsi; t1++)
   for (t2 = 0; t2 <= rdi; t2++) {
      A0[8*t1] = ...
      ...
      A1[8*t1 + 8*t2] = ...
   }
```

Figure 3.15: Code as seen by the dependence analyzer.

```
if (rax + 8*rsi >= rbx && rax <= rbx)
   return FAIL;
if (rbx + 8*rsi + 8*rdi >= rax && rbx <= rax)
   return FAIL;
```

Figure 3.16: Corresponding runtime tests.

dimensional array reference. It can also be over-approximative if some arrays are smaller than the threshold used to distinguish the groups. However, this technique allows parallelization in many simple cases and is correct thanks to the runtime check.

If the program uses dynamic array allocation, registers can also appear as base address in the memory references. In that case, we group the memory references using the registers used as base addresses. The rest of the process remains identical: the same tests are performed and they are generated exactly in the same way as when the base address is a numerical constant.

We present in Figure 3.14 a sample loop nest that could have been extracted from a binary program. The loop bounds are parametric and the nest body is made of two memory references whose base addresses are also parameters. On Figure 3.15, one can see the effects of grouping memory accesses: the two references have different base addresses and are then put in two different groups. Each group and its corresponding base address are associated to a different array name. Those array names are used

Sequential	Source parallelization	Binary parallelization
47.06 s	30.76 s (1.5)	40.79 s (1.2)

Figure 3.17: Execution times (and speedups) for `swim` on the `train` dataset.

for the dependence analysis to assume that the two references never reach the same memory location. Once the parallelization is performed, some runtime tests are added before the parallel code. They are presented in Figure 3.16. They ensure that the two references actually cannot intersect considering the value of the registers used. In case of possible memory conflict, the sequential code is executed.

This technique allows us to handle codes like `swim` from the SPEC OMP-2001 benchmark suite [5]. As our scalar removal process cannot handle all possible cases, polyhedral transformation of this code is not yet possible. However, our system is still able to parallelize it using non-transforming parallelization techniques. One can see on Figure 3.17 that our system is able to perform a reasonable speedup compared to the speedup obtained from the reference source code parallelization. The executions are performed on our Intel Xeon W3520 with four cores and two threads per core. It is important to remember here that our system is automatic and that it does not need the source code of the program, whereas the reference speedup is reached after human intervention to provide parallelization directives to the compiler in that source code.

3.5.2 Extension to Polynomial Codes

Another common issue in binary codes is provoked by array linearization. The compilation pass that transforms multi-dimensional array accesses into flat memory access functions often yields non-linear expressions. For instance, the following code fragment shows an array declaration and a simple use of this array in a loop nest:

```
void fun(int n, int m) {
    int A[n][m];
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            A[i][j] = ...
    ...
}
```

If the array sizes `n` and `m` are not known at compile time, the compiler transforms the array access into the expression $Base + 4 * j + 4 * i * m$ where $Base$ represents the array base address and can be either a register or a constant value. This expression is not linear and thus cannot be analyzed using the polyhedral model. Unfortunately such non-linear expressions are frequent in common binary codes, especially in Fortran programs where dynamic arrays are commonly used.

In order to parallelize such programs, we propose to use an approximate and conservative dependence analysis method handling parametrized polynomial memory references. We denote by $r_k(\vec{i_k}, \vec{p_k})$ a memory reference where $\vec{i_k}$ is the list of variables,

and $\vec{p}_k$ the list of parameters used. Variables $\vec{i}_k$ are the indices of the loops enclosing the memory reference. Those loop indices are constrained by the loop bounds: their possible values define a convex polyhedron $\mathcal{D}_k$, the iteration domain of the memory reference r_k. We note $\vec{i}_k[d]$ the iterator of the loop at depth d in the hierarchy of the loops enclosing the memory reference. Parameters $\vec{p}_k$ are usually array-size parameters whose values are stored in processor registers.

Consider two memory references $r_1(\vec{i}_1, \vec{p}_1)$ and $r_2(\vec{i}_2, \vec{p}_2)$ inside a loop nest. At least one of those accesses is a write, and both $r_1(\vec{i}_1, \vec{p}_1)$ and $r_2(\vec{i}_2, \vec{p}_2)$ are non-linear parametrized expressions, more precisely multivariate parametrized polynomials. If we are able to ensure that those memory accesses do not provoke any dependency between iterations of the outermost loop, then this loop can be parallelized. A sufficient condition is the statement:

$$\begin{array}{l} \text{for any } \vec{i}_1 \in \mathcal{D}_1 \text{ and } \vec{i}_2 \in \mathcal{D}_2 \text{ such that } \vec{i}_1[0] \neq \vec{i}_2[0], \\ r_1(\vec{i}_1, \vec{p}_1) < r_2(\vec{i}_2, \vec{p}_2) \text{ or } r_1(\vec{i}_1, \vec{p}_1) > r_2(\vec{i}_2, \vec{p}_2) \end{array} \tag{3.1}$$

This condition ensures that there is no intersection between the values reached through $r_1(\vec{i}_1, \vec{p}_1)$ and the values reached through $r_2(\vec{i}_2, \vec{p}_2)$ across distinct iterations of the outermost loop, i.e. $r_1(\vec{i}_1, \vec{p}_1) \neq r_2(\vec{i}_2, \vec{p}_2)$ for any $\vec{i}_1[0] \neq \vec{i}_2[0]$, and thus, that those memory accesses do not induce any dependence carried by the outermost loop. Ensuring (3.1) for every couple of memory accesses where at least one is a write allows us to parallelize the outermost loop. Notice that r_1 and r_2 can be the same memory write to handle self-dependences. Very little information or even nothing is known about the parameters values, usually only positivity of some parameters. Hence (3.1) can only be satisfied subject to some conditions on the values of the parameters.

To find some sufficient conditions on the parameter values ensuring that (3.1) is satisfied, we use a method based on symbolic Bernstein expansion of polynomials defined over parametrized convex polyhedra, described in [31, 33] and implemented in the ISL library [138]. This method allows us to compute the maximum value that can be reached by a polynomial. Hence when the maximum value of $r_1(\vec{i}_1, \vec{p}_1) - r_2(\vec{i}_2, \vec{p}_2)$ (respectively $r_2(\vec{i}_2, \vec{p}_2) - r_1(\vec{i}_1, \vec{p}_1)$) is strictly negative, we obviously prove that $r_1(\vec{i}_1, \vec{p}_1) < r_2(\vec{i}_2, \vec{p}_2)$ (respectively $r_1(\vec{i}_1, \vec{p}_1) > r_2(\vec{i}_2, \vec{p}_2)$).

Consider the following example built from one of the handled binary codes:

$$\begin{cases} r_1(i_1, j_1, k_1, M, N, P) = 8 + P + 8M + 8N + 8i_1N + 8j_1M + 8k_1 \\ r_2(i_2, j_2, k_2, M, N, Q) = 8 + Q + 8M + 8N + 8i_2N + 8j_2M + 8k_2 \\ 0 \leq i_1 \leq L-1,\ 0 \leq j_1 \leq L-1,\ 0 \leq k_1 \leq L-1 \\ 0 \leq i_2 \leq L-1,\ 0 \leq j_2 \leq L-1,\ 0 \leq k_2 \leq L-1 \\ i_1 \neq i_2 \\ L > 2,\ M > 0,\ N > 0,\ P > 0,\ Q > 0 \end{cases}$$

Sequential	Source parallelization	Binary parallelization
89.29 s	28.98 s (3.1)	48.46 s (1.8)

Figure 3.18: Execution times (and speedups) for `mgrid` on the `train` dataset.

Using the ISL library, we automatically compute the sufficient condition statements:

$$\begin{cases} \textbf{if } (-8+8L)M+(-8+8L)N+8L+P-Q-8<0 \\ \quad \textbf{then } r_1(i_1,j_1,k_1,M,N,P) < r_2(i_1,j_1,k_1,M,N,Q) \textbf{ when } i_1 > i_2 \\ \textbf{if } (-8+8L)M-8N+8L-P+Q-8<0 \\ \quad \textbf{then } r_1(i_1,j_1,k_1,M,N,P) > r_2(i_1,j_1,k_1,M,N,Q) \textbf{ when } i_1 > i_2 \\ \\ \textbf{if } (-8+8L)M-8N+8L+P-Q-8<0 \\ \quad \textbf{then } r_1(i_1,j_1,k_1,M,N,P) < r_2(i_1,j_1,k_1,M,N,Q) \textbf{ when } i_1 < i_2 \\ \textbf{if } (-8+8L)M+(-8+8L)N+8L-P+Q-8<0 \\ \quad \textbf{then } r_1(i_1,j_1,k_1,M,N,P) > r_2(i_1,j_1,k_1,M,N,Q) \textbf{ when } i_1 < i_2 \end{cases}$$

The conditions computed with ISL can be merged in a single test considering that the two references must not intersect when $i_1 < i_2$ and when $i_1 > i_2$. In our example, the generated test is then:

$$\begin{array}{ll} \textbf{if} & (\ (-8+8L)M+(-8+8L)N+8L+P-Q-8<0 \\ & \textbf{or} \\ & (-8+8L)M-8N+8L-P+Q-8<0\ ) \\ \quad \textbf{and} & \\ & (\ (-8+8L)M-8N+8L+P-Q-8<0 \\ & \textbf{or} \\ & (-8+8L)M+(-8+8L)N+8L-P+Q-8<0\ ) \\ \textbf{then} & \text{there is no dependency between } r_1 \text{ and } r_2 \text{ carried at level 0} \end{array}$$

Similar tests are generated for every couple of memory accesses where at least one is a write. Those tests are evaluated at runtime and, if one of them fails, the parallelization of the outermost loop cannot be proven correct and the sequential version of the loop nest has to be executed. Notice that we have focused on the outermost loop parallelization only. Similar tests could be performed for other levels with several disadvantages, including the necessity of having several parallel versions (one per parallel level), a large increase in the number of tests that have to be performed to decide if a loop is parallelizable, and possible slowdowns when parallelizing the inner loop levels. Thus, we only consider the parallelization of the outermost loop.

We have implemented this parallelization strategy as a different parallelization back-end alongside the other back-ends. This strategy enables our system to parallelize codes with polynomial memory references such as the `mgrid` code from the SPEC OMP benchmark suite. Figure 3.18 shows the execution times and speedups reached by our system compared to the reference parallelization from the source code. The execution times have been measured on an Intel Xeon W3520 processor with four cores and

two threads per core. Once again, it is a good result, considering that our automatic system does not need the source code and that few common parallelization techniques can be applied in this case, since the access functions are not linear. It is also a good example of the modularity of our approach, where various dependence analyzers and/or parallelizers can be "plugged in" to handle various classes of problems. The overhead induced by the tests performed before the loop nest execution is relatively low: there are many tests (one per couple of memory accesses) but the execution of each one accounts only for a few processor cycles which is negligible in regard to the benefits offered by the loop nest parallelization.

3.5.3 In-Place Parallelization

During the analysis phase, interesting loop nests are raised to C codes, amenable to parallelization by existing source-to-source parallelizers. After parallelization, the loop nest semantics is restored and the parallel code is compiled as a new function, distinct from the sequential code. However, many issues can be encountered using this strategy.

First, some performance issues can arise if the compiler has precisely optimized the original sequential loop nest and is not able to perform as well on the parallelized code. Second, creating a new version of the loop nest increases the code size. This can be critical on some platforms such as embedded systems. Third, it can also be difficult to raise some very specific assembly constructs to functional C code. For example, GCC can generate some tests on floating point values to ensure that they are different than `NaN` using the conditional flags of the processor. To translate this sequence as an inline assembly code extract in a C program, a complex and inefficient sequence of floating point operations, tests, and register savings and transfers have to be used in the parallel code. The problem is exacerbated when the compiler does not support the full set of instructions as inline assembly.

For those reasons, we have also considered an alternative parallelization method. The key point of that method is to parallelize the binary code *in-place*, i.e. without modifying nor moving the loop body. As the parallel binary code is identical to the sequential one, no code transformation is allowed anymore. This is then particularly suited to the extension on polynomial codes we have presented before. In this setting, the threads are not anymore implicitly created using OpenMP but managed internally using standard Linux threading facilities.

This parallelization method solves all the problems aforementioned: the resulting parallelization uses the same loop body as the sequential loop nest and then benefits from the same compiler optimizations, the loop nest does not need anymore to be raised to C code and recompiled, and there is only one version of the loop nest for both sequential and parallel execution.

A specific implementation is required for this technique, it is based on the same two-processes approach as presented in Subsection 3.3.8.

Loop Bounds Overwrite

When the polyhedral parallelization strategy is used, the full parallel loop nest becomes a function in the shared library loaded by the RC. When using in-place parallelization,

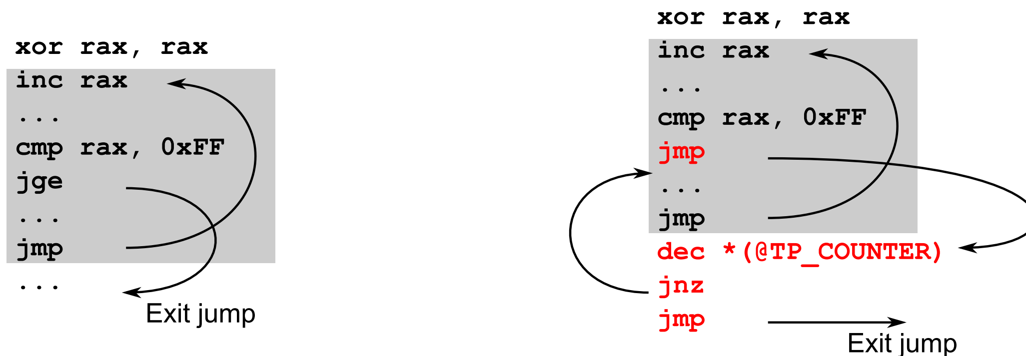

Figure 3.19: Overwriting loop bounds with a new loop counter (left to right).

the original sequential loop nest has to be executed in parallel without modifying or moving it. A function in the library still contains some initialization and termination code, but the original computation remains in the original program.

In the library, an initialization code of each loop nest is principally managing a thread pool, created at the application start. When the parallel loop nest execution starts, this initialization sequence also computes the first and last iterations of the parallel loop that each thread has to execute. Once this is determined, every thread is unlocked and executes its specific parallel loop iterations.

To control the iterations executed by each thread, a slight modification is applied on the loop nest. The loop exit condition is usually performed using a conditional jump evaluating the exit condition, i.e. the loop upper bound. We overwrite this conditional jump with a unconditional jump which points to a "safe" code area, made of consecutive instructions in the function body which are not part of the loop instructions, and which can then be safely overwritten during the loop nest execution. We temporary overwrite those instructions with the desired exit condition.

The exact code sequence computing the new exit condition is:

```
dec *(@TP_COUNTER)
jnz @LOOP_CONT
jmp *(@TERMINATION)
```

This short sequence decrements a thread-private loop counter, if this counter is not equal to zero, the execution continues, otherwise a termination code is called. The full code fragment can be stored in a 23 bytes binary sequence and does not modify any register except from the flag register.

The main challenge is to find twenty three consecutive bytes in the function body that are not used by the loop nest, and that are reachable from the exit jump. This can be difficult as the overwritten exit jump can be a short jump, using a single byte as offset. Those small offsets can prevent us to reach an address outside of the loop nest body. Moreover such jumps cannot be overwritten with long jumps that are encoded in longer binary sequences on the x86-64 architecture. Even if they exist, such cases do not appear often in practice.

The loop bounds overwrite is illustrated in Figure 3.19 where the original loop code (on the left) is transformed to use the new loop bounds (on the right). After

transformation, the new exit sequence is executed at every loop iteration, and the execution stops only when the new loop bound is reached. The process is reverted at the end of the nest execution: when the exit jump in the code fragment is executed, the original loop bound is restored.

Notice that, to start the execution at the correct parallel iteration, it is sufficient to correctly initialize registers and memory locations whose values depend on the parallel loop counter. This means that, if the loop nest is not fully analyzable, this parallelization is not possible.

Fast Thread-Private Data

The parallel loop has a new loop counter, used by the new exit sequence. This new loop index has to be thread-private as the loop is parallel and each thread executes a distinct set of iterations. Ideally it also has to be directly reachable for the code to be efficient: no register should be written, and there should be no function or system call to read or write the loop index value. Subject to those constraints, it is not possible to use any of the classical threading functionalities, such as the `pthread` local storage support, or the thread unique identifier.

We have used a specificity of our operating system implementation to find a directly addressable thread-private memory location. All the recent operating systems with multi-thread support have to maintain the threads' states. This state includes, for example, the current scheduling options, the synchronization data, and some private-data facilities. On Windows and Linux for x86 and x86-64 platforms, this is achieved through a segment register. One of those specific processor registers is pointing to a thread-specific memory area, filled with the information related to the thread. On Linux, the `FS` register is used for that purpose. The address `FS:0` is then the address of the information structure. Interestingly, on recent implementations, the structure contain unused fields and some padding bytes.

On our system, as for most of the current x86-64 Linux systems, 64 bytes are then available starting at address `FS:0x280`. This memory space is writable for the thread, and thus, we store the loop counter at `FS:0x280` which is a directly addressable memory location, causing no specific overhead when accessing it. No function call is performed and no register requires to be set to access this memory location.

Address in Position Independent Code

When the parallel loop execution ends, the RC has to be awaken in order to cleanup the code and safely resume the sequential program execution. When polyhedral parallelization is performed, OpenMP handles the thread synchronization and the execution becomes sequential at the end of the parallel loop. But when in-place parallelization is used, the thread synchronization has to be ensured and the execution has to be resumed in sequential.

A naive solution would be to use a breakpoint at the end of the loop nest execution. Each thread would then execute it to signal to the RC that it has finished its task. Despite this solution is simple, this would interrupt all the threads every time a thread finishes, provoking a huge overhead. Instead, at the end of the loop nest execution,

Algorithm 3.4 Finding the instruction-location canary.

```
uint8_t *addr;
for (addr = (uint8_t*) @FUNCTION; cnt < 7; addr++) {
   if (*addr == 0xCC) {
      cnt++;
   } else {
      cnt = 0;
   }
}
while (*(addr++) == 0xCC);
return addr;
```

we redirect all the threads back to the shared library, right after the instruction that provoked the execution of the parallel loop in the shared library. At that point, all the threads are then synchronized and only one of them calls the breakpoint to let the RC resume the execution.

To do so, the address of the instruction calling the parallel loop has to be determined in the library. The library is necessarily compiled as position-independent code, meaning that it is impossible to determine it at compile time, and no specific language functionality exists to determine at runtime the location of a specific instruction.

However, this can be achieved using a canary system: a specific instruction sequence is inserted just after the call to the parallel loop nest. This sequence has not to be generated by the compiler if not specifically asked for. It is searched for during the initialization, starting at the beginning of the function code. Once found, we know the address where the execution has to be redirected to, when the parallel loop nest ends. We have chosen the canary to be seven occurrence of the specific `int 3` instruction (encoded as the byte `0xCC`). The `int 3` instruction is very specific and the compiler has no reason to generate it without having been requested to do so. The seven occurrences of this instruction avoids any confusion with program data such as memory addresses encoded using this specific byte.

The algorithm used to find this canary is presented in Algorithm 3.4 as a C program fragment. We can see that seven occurrences of the specific byte `0xCC` are searched for. Once found, the address is incremented until reaching a different byte, as the first `0xCC` bytes met can be the last bytes of some instructions preceding the canary sequence.

The address found can then be used to redirect the threads at the end of the parallel execution. The thread synchronization code directly follows the canary sequence.

Putting it all Together

At startup, the library is injected in the original program memory space. The breakpoints are inserted at the beginning of each loop which has been parallelized.

When a breakpoint is met, the application stalls and the RC overwrites the loop bounds with the code sequence presented before. The RC then redirects the application to a function in the shared library. In that function, the parallel loop iterations are

divided among the threads, and each thread initializes its specific loop counter, and all the data depending on this loop counter. The address of the termination code is found in the function body and set in the data used by the loop exit code sequence. Every thread then jumps to the original loop to execute it in parallel. At the end of the loop nest execution, every thread jumps back to the termination code in the library function. The threads are then synchronized and a master thread awakes the runtime component. The RC cleans the code and redirects the program after the sequential loop nest in the program.

When the execution branches from the library code to the original application code and back to the library code, all the registers are saved and restored in a procedure similar to what is done by `ptrace`. It is also sometimes required that the threads use private stacks, for instance when some loop indices are in the stack. When it is possible to statically determine the stack area used in the loop nest, then the stack is privatized, otherwise, the parallelization has to be aborted.

The cost of this system remains low and is mainly induced by the `ptrace` system calls. One occurs at the beginning of the loop nest execution and one at the end. Each call costs about a thousand processor cycles on recent hardware and Linux versions. The full parallelization process has a negligible overhead compared to the gains usually provided by the loop nest parallelization.

3.6 Related Work

In Chapter 2, we have presented several binary code parallelizers which are very often dynamic systems. Recently, Kotha and colleagues have proposed a static binary parallelizer [74]. To our knowledge, this is the only static binary parallelizer proposed so far. Since this work is very close to ours, we present in this section a topic-by-topic comparison with their approach.

3.6.1 Decompilation and Address Expressions

The first major difference is how both systems extract address expressions. The system proposed by Kotha et al. extracts address expressions by pattern matching binary instructions with three classes of instructions:

- initial assignments of the form $i \leftarrow \alpha$, where α is a loop invariant quantity;
- increments of the form $i \leftarrow i + d$ where d depends on the dimensions of the assumed underlying array;
- loop counter upper bounds of the form $i < u$, where u is the upper bound of the loop counter.

On the other hand, our system captures the data-flow between registers and stack slots, and uses symbolic analysis to reconstruct address expressions built around normalized loop counters. Although the naive approach seems sufficient on PolyBench programs, we have found it inefficient on more complex loops, like the ones our decompilation

pass extracts from SPEC benchmarks (see Section 3.5). There are two major reasons why we have switched to a more advanced analysis of binary code. The first is that most programs put enough pressure on the register allocator to force spilling on registers containing loop counters, if any. In fact, this is even true on PolyBench programs, when they are compiled for a 32 bits x86 architecture. The second is that dependence analysis needs loop trip-counts, which are seldom expressed as single compare instructions. The approach we have developed in Subsection 3.2.5 is, in our experience, absolutely crucial to obtain reasonable constraints on blocks and avoid an overly conservative dependence analysis.

3.6.2 Parallelization and Transformation Strategy

Kotha et al. use several simple dependence tests to decide whether a given loop is parallel or not. The dependence vectors are extracted with ad-hoc pattern-matching techniques similar to the ones used in extracting addresses. A trivial analysis of the dependence vector components selects parallel loops. This strategy is known to be fundamentally inferior to polyhedral techniques, where each dependence is represented as a polyhedron, an abstraction strictly more expressive than dependence vectors. When using PLUTO as a back-end parallelizer, we are able to benefit from this precise dependence analysis. Another aspect is that decompilation and dependence analysis appear strongly entangled in [74]. In contrast, we have demonstrated that the parallelization phase can be “off-loaded” to an external component, and used two distinct such components (PLUTO and CETUS).

The authors also claim to be able to extend their system with program transformations, on a way “orthogonal to [their paper]” [74, Section 5]. However, we consider this claim to be overoptimistic. First because they do not substantiate their claim by any experiment, and second because they do not seem to realize that applying affine transformations to a program requires full knowledge of the control flow inside loop bodies, e.g., conditionals that place additional constraints on some blocks of the body (see our Subsection 3.2.5). By going from simple branch conditions to block constraints, we have proved our system to be able to apply any combination of loop transformations to binary programs. Our experiments have shown an actual implementation of a transforming parallelizer.

3.7 Conclusion and Perspectives

This chapter presents a system able to efficiently parallelize binary programs. Our system is able to parallelize the program even if some advanced transformations are required.

The contributions of this work are manifold. First, we present a parallelization system able to use existing any existing source-to-source compiler to parallelize the binary program. Second, we show that the static parallelization of complex binary codes is possible. We present techniques to handle codes with parameters in loop bounds or memory references, and non-linear memory references. Third, we present a parallelization technique that limits the code size expansion and ensures that the

optimizations applied on the sequential code can be exploited for non-transforming parallelization. Fourth, the implementation of the system and its evaluation show that the chosen solutions are viable and lead to good performance, comparable to equivalent systems exploiting the source code, whereas the ability to use polyhedral compilers provide a clear benefit over existing techniques.

This system is the first brick of our general approach: it is a static system which can automatically extract parallelism and enhance the performance of sequential binary programs, whose format is complex. It demonstrates that an efficient parallelization can be performed when the source code of the program is not available, without requiring any specific hardware to support a speculative approach. This static parallelizer can nicely complement any other dynamic system in order to efficiently parallelize complex programs where some parts can be treated statically.

From those results, we could envisage in the future to extend the same reasoning to more complex optimizations. For instance, when the code is fully analyzable, this work can be combined with the dynamic selection system presented in the next chapter to perform this code selection at the binary level. A full decompilation could also be achieved, allowing us to recompile the loop nests for different architectures. This would result in a binary translation system able to generate optimized parallel code for GPUs or FPGAs from sequential binary x86 programs. The static analysis could be enhanced to revert more complex optimizations such as loop unrolling or tiling. This could allow our system to parallelize more complex binary programs. Such advanced parallelizer, combined with a dynamic binary parallelizer, would lead to a very robust and efficient solution to parallelize binary programs.

Chapter 4

Code Version Selection

4.1 Introduction

During the long history of compilation, many optimization passes have been designed and recent compilers are exploiting most of them. For instance, recent versions of GCC exhibit more than 150 different options to control the optimizations applied on a program. Considering this large number of optimization passes, one of the main critical tasks for a compiler is then to find the ordered set of optimizations leading to the best output, considering a given program and architecture. Due to the large number of available optimization passes, to their interactions, and to the complexity of the target architectures, the selection of a good optimization sequence is particularly difficult in the general case [21, 72, 38, 134, 51, 1, 105, 52]. For instance, if we focus on a loop, unrolling it can provide large performance improvement but, due to the increased number of scalars, the register allocation pass may produce a less efficient result, degrading the efficiency of the generated code.

Even worse, in many cases, the performance of the compiled application depends on runtime parameters such as the input data size, or the number of available processor cores. For instance, we observed on a matrix multiplication code using simple control and data structures, that depending on the input data size, distinct parallel versions provide the best performance, while running all versions on a single computer. Further, we observed that the version offering the best performance is also not the same when run on different machines, even with the same input data. Those parameters, impacting the execution time of a program, define an *execution context*.

The static compilers cannot generate one unique code version always reaching the maximal performance in every context. To reach this maximal performance, it is required to consider several functionally equivalent versions of the application. Each version can be optimized in a different way, targeting a distinct execution context. At runtime, the best version according to the current execution context has to be executed. The performance that can be reached with such systems can then outperform any statically generated version.

Codes with deterministic and regular behavior are of particular interest for selection systems as their performance can be precisely characterized. This is a fundamental property when selecting code versions as the version performance must be predicted

at some point in order to determine which version to execute in the current context. Another interesting property for selection systems is the version diversity. Having very different versions resulting from the application of a wide variety of optimizations can help the selection system to adapt to very different execution contexts. As presented in Chapter 2, the polyhedral model cumulates those two aspects: the loop nests in this model have a statically analyzable behavior with regular memory accesses, and a large number of different optimizations can be applied on them. Thus, the polyhedral model is a good frame in which performing dynamic code selection.

In the previous chapter, we have presented a parallelization system able to statically parallelize binary code. We present here a hybrid static-dynamic code selection mechanism which can enhance the performance of a polyhedral loop nest when different execution contexts are met. For this system, we consider that an execution context is mostly defined by the current architecture, by the input data size and shape, and by the number of available processor cores. When various execution contexts are met by a program, we show that this hybrid system nicely complement a static polyhedral parallelizer such as the one we have presented earlier.

We present an overview of our selection system in Section 4.2. It is able to predict execution times for polyhedral loop nests in order to select the most efficient one for a given execution context. The different code versions can be automatically generated using existing tools as presented in Section 4.3. Those versions are then profiled on the targeted computer in order to extract a performance characterization allowing us to predict their execution times. This profiling step is presented in Section 4.4. At runtime, before each loop nest execution, the performance of every version is predicted using the strategy detailed in Section 4.5. Our system has been implemented; its accuracy and performance are evaluated on a set of polyhedral loop kernels in Section 4.6.

4.2 Selection Framework Overview

Our framework aims at selecting the best code version, i.e. the fastest one, before any loop nest execution. To achieve this goal, we predict each code version execution time in the current execution context and run the version predicted to be the fastest. Predicting an accurate execution time in all the possible execution contexts is a very difficult problem that we do not try to solve here. But rather, our framework computes an approximation of the execution times which is accurate enough to provide a realistic ranking of the different code versions.

To predict those execution times, our framework first measures, during an offline profiling step, the execution time of each code version in various execution contexts expected to be as representative as possible of all the actual contexts. Those execution times are gathered in a parametric ranking table. A runtime prediction step adapts those results according to the current context in order to predict an execution time for each code version. This runtime prediction is preceding each execution of a loop nest. It is embedded in the application binary and is run through a simple function call replacing all the target loop nest executions.

Figure 4.1 shows an overview of the framework. The different phases are represented vertically from the generation of the different code versions at compile time to the

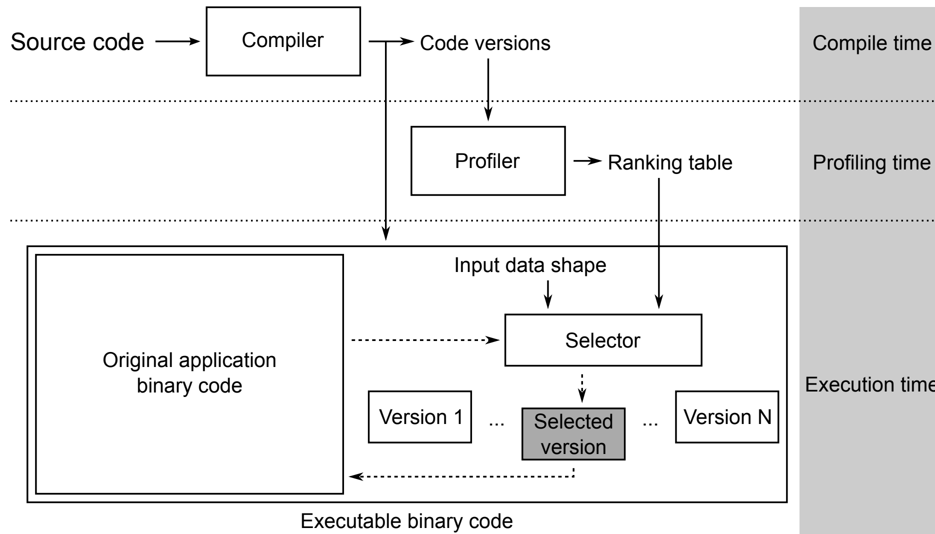

Figure 4.1: Framework overview.

execution of the loop nest. All those steps are detailed below.

4.3 Generating Different Code Versions

The different versions of the loop nest differ in the optimizations that were applied to it. Among all the existing optimizations, the polyhedral transformations are of particular interest. Indeed, the polyhedral framework allows a large variety of high-level transformations able to improve parallelism and data locality. Their impact on performance is usually greater than common non-polyhedral optimization. We chose to compare different versions where the main differences between the code versions are the parallel schedules and the tile sizes. Both have a strong impact on performance: degree of parallelism, load balancing, communication volume, cache locality, and vectorization capability depend on them. Moreover, both the tiling levels and sizes [59, 118], and the parallel schedules [105, 104] are difficult to determine at compile time, making them of particular interest for a dynamic system.

For our benchmarks, we generated different versions by hand. In order to automatically build several versions, current automatic parallelizers such as PLUTO [23] or LetSee [105, 104] could be used to determine several probably efficient optimization sequences. The iterative nature of LetSee makes it particularly amenable to this task. Generating a tiled and a untiled version is also generally a good idea as the untiled version is often more efficient for small problem sizes. Auto-parallelizing tools could also consider different parallel schedules when they all lead to equivalent memory performance. Another interesting approach would be to generate code versions supposed less efficient due to some hardware hypothesis. For example a code version expected to be inefficient due to its expected cache misuse could be the best one on a hardware accelerator without a cache. The incorporation of such versions would give to our system the opportunity of selecting and running efficient versions on very different hardwares.

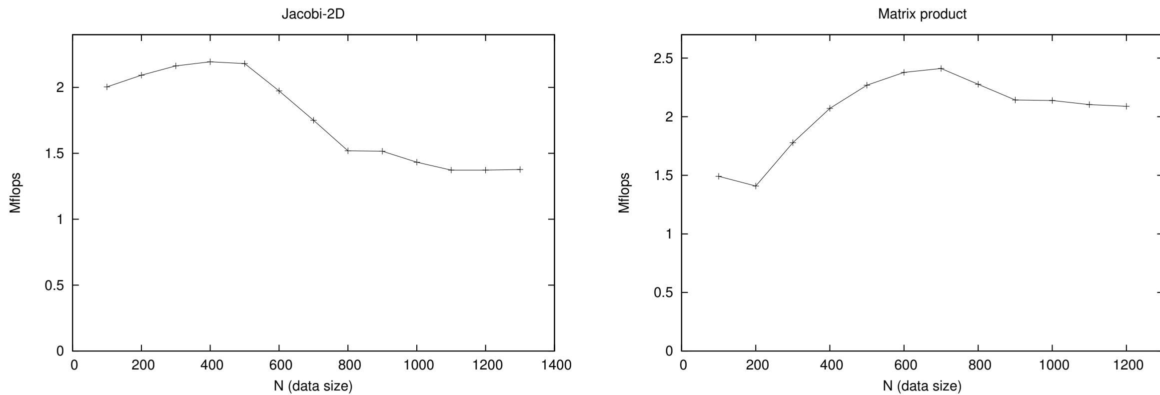

Figure 4.2: Performance of a simple parallel version relatively to the data size.

4.4 Profiling the Code Versions

The final goal of the system is to select the fastest version in the current execution context among all the provided ones. For this purpose, the system has to predict the execution time of each version or at least to rank every version against the others. Existing runtime selection systems often use previous runs of the program to "predict" the performance of each version in the future: each version is executed and evaluated in the encountered contexts. If a similar context is encountered again, this information is used to select the correct version. This obviously imposes many executions of the versioned code fragment. We want to predict execution times before every execution of the loop nest in order to be more reactive, and then more efficient.

Naively, one could measure the execution time of each version in one specific execution context and use the results to rank the versions in every execution context. This would be correct if all the contexts lead to the same execution times. This is not the case. When measuring the execution time of an application, one has first to evaluate the impact of some characteristics of the context, the *static* factors, whose impact is constant during the whole execution of a code fragment in every possible context. They are taken into account when measuring the execution time of any run. For example, the micro-architecture characteristics are static factors: instruction cycle, pipeline, branch prediction, out-of-order execution, floating point operation latency, amount of processor memory cache, etc. However other factors, the *dynamic* ones, can change with the execution context and also have to be measured.

In Figure 4.2, one can see the performance of two polyhedral kernels (Jacobi-2D and a matrix product) with different increasing data sizes on a recent computer. Those kernels have been parallelized by hand. We clearly see that the performance of both kernels evolves with the data size. This evolution is due to several complex factors related to the memory hierarchy of the computer on which the kernels are run. For instance the cache usage plays a large role on this performance evolution.

The impact of the data size on the execution time is difficult to characterize due to the complexity of the underlying memory architecture. Moreover, each code version can exhibit a specific behavior and, for a given code version, some large performance variations can appear even with nearly identical data sizes. It means that, to precisely

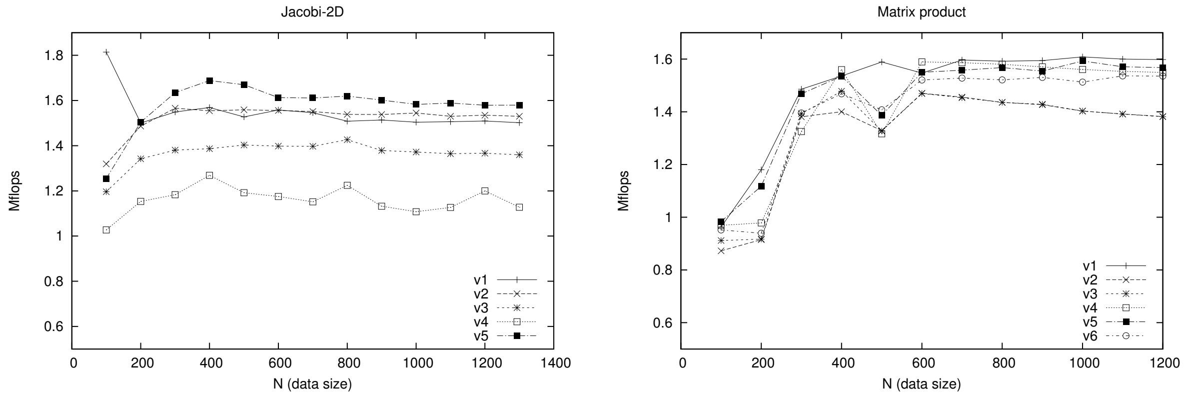

Figure 4.3: Performance of several versions relatively to the data size.

characterize the performance of a code version, a fine sampling of the execution time for many different sizes is required. However, such sampling would lead to large profiling results and to very long profiling steps.

Nevertheless, one common point can generally be observed: at very large data sizes, the processor caches are less effective and most of the performance perturbations due to the caches are not occurring anymore. Moreover, code selection systems are more useful for long computations, generally occurring when processing large data. Hence, our framework measures the execution time of each code version when the data cannot fit in the processor cache memory. The rank of each version is exact only for that specific data size. However, we assume that this ranking is a good approximation over all the possible sizes. In general, the different versions are indeed following performance patterns that are roughly similar, except for some specific values and at small data sizes. This is illustrated in Figure 4.3, where for both example kernels, we compare the performance of different versions for several increasing data sizes. The versions differ in the number of tiling levels, the tile sizes and the schedules applied. One can see that there are fluctuations, especially at small data sizes. But the relative order of the versions for the biggest data size remains a good approximation of the general order over all the possible sizes.

We have chosen not to depend on any architecture-specific mean to detect when the data overfills the cache memory. Instead, the profiling step performs a few measurements while increasing the data size. When two successive measurements lead to similar performance, we assume that this data size is bigger than size of the first cache levels. This is a best-effort method that is never guaranteed to detect cache overfills, but which works well in practice.

The second factor that can impact the code performance is the number of threads used to perform the computation. For example, on Figure 4.4, we can see the performance of a naive parallelization of both sample kernels when using different number of threads. We obviously see that the number of used threads strongly impacts the performance of the application.

There are two main reasons for variable numbers of available threads during the execution of an application. First the operating system can decide to allocate a given number of threads to the application according to the amount of available hardware

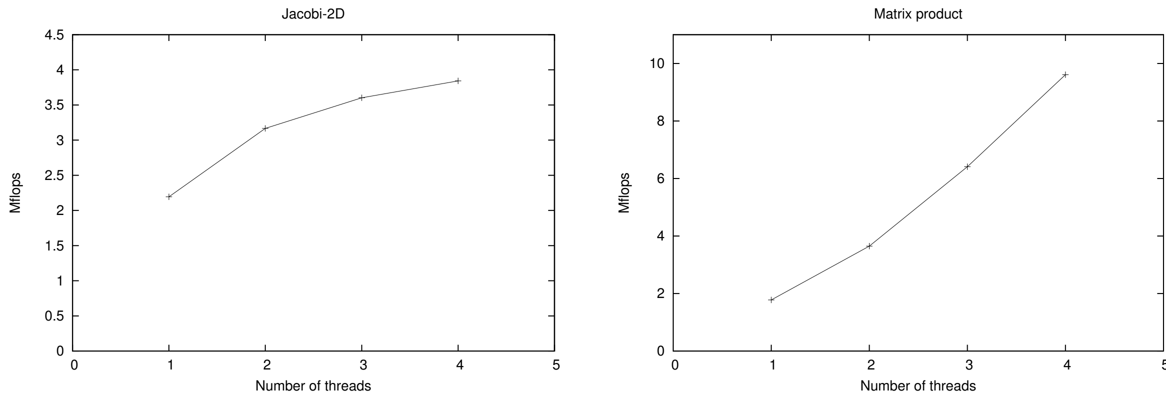

Figure 4.4: Performance of the parallelized kernels relatively to the number of threads used.

resources. This amount can change depending on other applications running concurrently and on possible hardware failures. Second, the data size directly impacts the load balance of the loop nest: if the trip count of the parallelized loop depends on the data size, the performance may greatly vary with nearly identical datasets. For example on a four-core processor, having four or five iterations to be executed by the parallel loop leads to very different efficiency.

To precisely handle this factor, the profiling is repeated in successive measurements for different number of active threads ranging from one to the number of logical processor cores. The profiling result is therefore made of execution times for each possible number of active threads. A runtime system is then in charge of determining how many threads will be active during each iteration of the nest, in order to deduce an execution time prediction. For the runtime system to be able to deduce execution times at the iteration granularity, the result of the profiling step is actually made of average execution times per iteration instead of absolute execution times.

In the following subsections, we propose two different profiling strategies. Those strategies comply with the same contract: for each code version, they build an associated profiling code that can be substantially different from the executed code, but that is dedicated to measure relevant execution times per iteration on large domains and for every possible number of active threads.

4.4.1 Strategy 1

In this subsection, we present a first strategy which aims at profiling the different code versions in execution contexts which are as comparable as possible. We accept here to perform major changes in the profiled code structure to ensure that all the versions are evaluated with iteration domains that are as similar as possible. A few challenges have to be addressed here: first, in order to perform the measurements while increasing the data size (to overfill the first cache levels), the iteration domain size has to be increased. Second, the number of used threads has to be controlled, and third, to ensure that the domains are comparable, the strategy has to ensure that only full tiles are executed when considering tiled loop nests.

Strategy Goals and Concepts

The domain size has to be increased at each step of the profiling process and the number of active threads has to be controlled. A simple way to do so is to remove all the iteration domain boundary constraints, and then to bound the parallel loop with a new parameter, `par_sz`, and the other dimensions with another new parameter, `N_min`. The latter allows us to control the overall domain size, while `par_sz` controls the number of active threads by defining the number of parallel iterations. It is equivalent as considering that the loops range from zero to either `N_min` or `par_sz`.

To be efficient, the codes parallelized with polyhedral transformations are usually tiled [142, 112]. Significant performance variations can be observed between codes where tiles are fully executed and where only incomplete tile executions occur. The number of incomplete tiles depends both on the schedule used in each code version and on the iteration domain shape. Thus, a given code version can contain more incomplete tiles than another version for a given domain shape. Since incomplete tiles usually appear at the domain boundaries, there are generally less incomplete tiles than full tiles, and thus we choose to build the profiling domain exclusively with full tiles in this strategy. In order to ensure that full tiles are executed, only the first tiling level is constrained by the new parameters, while others are left unconstrained. However, if the domain is not tiled, all the dimensions are constrained.

The resulting profiling code domain is then made of two constraints per loop for the first tiling level: one expressing the lower bound of the loop, and the second one for the upper bound which is either `N_min` or `par_sz`. Those constraints are applied on the transformed loop nest.

Time measurement instructions are added and execution times per iteration are measured. In order to increase the domain size, the value of parameter `N_min` is doubled at each step until reaching stable measurements, i.e. having less than 5% of variation between two successive measurements. The value of `par_sz` is increased from one to the number of logical processor cores to simulate all the possible threads availability.

Building the Profiling Domain

The algorithm used to build the profiling loop nest for a given code version is given in Algorithm 4.1. It generates a *profiling nest* from the polyhedral definition of the considered loop nest version. The full profiling code, i.e. the profiling nest and the code to control the new parameters, is generated by a syntactic replacement from the code pattern shown in Figure 4.5. The profiling loop nest is inserted in the pattern, and its iteration count, `PROFILING_NEST_SIZE`, defined by an Ehrhart polynomial, is used to compute execution times per iteration. This Ehrhart polynomial is automatically generated at compile time using the Barvinok library [139]. Notice the token `MEMORY_ALLOCATION` which is replaced by some array allocation code. Those instructions allocate all the arrays such that they occupy nearly all the available memory. This array allocation allows the system to profile the code for large domain sizes with a single memory allocation.

The iteration domain bounds have been replaced in the profiling nest to introduce

Algorithm 4.1 Profiling code generation with the first strategy.

Apply the transformation
Remove all the boundary constraints
Create two parameters: N_min and par_sz
Constrain the parallel iterator p by $1 \leq p \leq par_sz$
$tiles1 \leftarrow$ loop iterators from the first tiling level
for i in $tiles1$
  if $i \neq p$
    Constrain i by $0 \leq i < N_min$
Generate the code

```
MEMORY_ALLOCATION
do {
    old_result = copy_array(result);
    N_min = N_min * 2;
    for (par_sz = 1; par_sz <= NB_CORES; par_sz++) {
        start = time();
        PROFILING_NEST
        end = time();
        result[par_sz] = (end - start) / PROFILING_NEST_SIZE;
    }
} while (difference(result, old_result) > 0.05
        && enough_memory(N_min*2));
MEMORY_FREE
```

Figure 4.5: Code pattern. The profiling code is generated from this code pattern after some syntactic replacements.

```
for (iT = 0; iT <= M / 64; iT++)
   forall (jT = 0; jT <= N / 64; jT++)
      for (i= 64 * iT; i <= min(64 * iT + 63, M); i++)
         for (j = 64 * jT; j <= min(64 * jT + 63, N); j++)
            S[j] += A[i][j];
```

Figure 4.6: Sample code.

the two parameters `par_sz` and `N_min`. Hence, there is no more guarantee that the array subscript functions are always positive. For example consider a loop iterating the index `i` starting from 1 and an array access where the element at position `i - 1` is referenced. After replacing the domain constraints, the loop can start at 0 and the first accessed element is then at position -1. To overcome this issue, the array base is actually positioned in the middle of an allocated memory space. It does not guarantee that no incorrect access happens but in practice, this is sufficient as the negative offsets used in subscript functions are often small constants. More complex analysis could be performed for cases where a loop index is used in a subscript function with a negative coefficient. In particular, the relation between the first and last element accessed and the loop bounds can be determined using the lexicographic minimum and maximum of the subscript function. Such relation can be used to determine a value range that the parameters `par_sz` and `N_min` can have for the access to fit in memory.

Figures 4.6 and 4.7 illustrate the profiling code generation. In Figure 4.6, we present a code summing the columns of array `A` in array `S`. The loop nest is tiled and parallelized. Figure 4.7 shows the corresponding profiling code generated from the pattern presented in Figure 4.5. The domain boundaries are eliminated and the domain size is controlled by both new parameters `par_sz` and `N_min`. While `N_min` is doubled until reaching a stable measurement, `par_sz` is incremented from 1 to the number of available processor cores. The domain size increase is stopped if the computation requires too much memory. A simple implementation of this test is performed using the operating system memory protection mechanisms. In practice, during our experiments, stability has always been reached before overfilling the memory. The result is made of the last measured execution times per iteration, for each considered number of parallel loop trip counts.

For each value of both parameters, the execution time is measured using the regular operating system timing function, although only the execution time for the last value of `N_min` is saved. The expression used to compute the number of iterations `(N_min * par_sz * 64 * 64)` is the Ehrhart polynomial representing the iteration domain size. In the case of multiple statements in the nest, the domain size is the sum of each statement domain size. This leads to consider the average execution time per iteration over all the statements as the profiling results.

In this first strategy, the code versions are evaluated in execution contexts that are as comparable as possible, considering that this similarity leads to better results. The main drawback of this strategy is the strong code modifications performed to build the profiling code. The loop bounds and the associated control code can be very different from the loop bounds used in the original loop nest. This can lead the compiler to

```
do {
   old_result = copy_array(result);
   N_min = N_min * 2;
   for (par_sz = 1; par_sz <= NB_CORES; par_sz++) {
      start = time();

      for (iT = 0; iT < N_min; iT++)
        forall (jT = 0; jT < par_sz; jT++)
          for (i= 64 * iT; i <= 64 * iT + 63; i++)
            for (j = 64 * jT; j <= 64 * jT + 63; j++)
              S[j] += A[i][j];

      end = time();
      result[par_sz] = (end - start) / (N_min * par_sz * 64 * 64);
   }
} while (difference(result, old_result) > 0.05
        && enough_memory(N_min*2));
```

Figure 4.7: Sample profiling code for the loop nest presented in Figure 4.6.

apply different optimizations on the profiled code and on the actually executed loop nest. Then, irrelevant profiled execution times can be measured.

4.4.2 Strategy 2

We present a second strategy whose goal is to maintain the source code as much as possible in its original form, in order to profile a code as similar as possible to the actually executed code. During the profiling, the number of active threads and the iteration domain size are evolving. In this strategy, the number of active threads is controlled using the OpenMP [95] `omp_set_num_threads` function, allowing us to control this factor while preserving the original loop nest bounds. To control the iteration domain size, we directly change the value of the parameters that are used in the loop bounds, avoiding any code modification. If no such parameter exists, then the data size is fixed and a single profiling for this specific data size is sufficient.

One could think that we could simply assign increasing values to the parameters to increase the iteration domain size. It is actually not always possible as the loop bounds can be defined as complex functions of those parameters. For instance, Figure 4.8 shows a simple loop nest where increasing simultaneously the value of both parameters does not increases the iteration domain size.

The Enclosed Hypercube Problem

We want to find parameter values that ensure a minimal iteration domain size. If we can guarantee that each iteration domain dimension, i.e. each loop level, has at least a given number of iterations, then we can ensure that the iteration domain has a corresponding minimal size. Geometrically, an efficient way to do so is to ensure that

```
for (i = 0; i < N; i++) {
   for (j = 0; j < i - M ; j++) {
      ...
   }
}
```

Figure 4.8: Sample parameterized loop nest. Increasing `M` does not increase the number of iterations.

```
for (i = 0; i <= 10; i++)
   for (j = 1; j <= i + 1; j++)
      ...
```

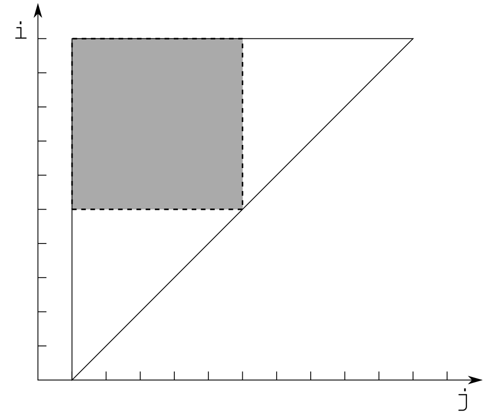

Figure 4.9: Sample iteration domain. A square of 5×5 iterations can fit in it, so we can guarantee that each loop level executes at least 5 consecutive iterations.

the iteration domain polyhedron contains an axis-aligned hypercube, representing the desired minimal domain size.

For instance, we can see in Figure 4.9 a two-dimensional example loop nest and its corresponding polyhedral representation. We can ensure that the iteration domain has at least 5 consecutive iterations for each loop level as we can fit a square of dimension 5×5 inside the polyhedron. The square represents the desired minimal size of the iteration domain: if it can fit in the polyhedron, the domain has the required size.

Our goal is to increase this minimal domain size step by step during the profiling process; this can be done by increasing the size of the hypercube that has to fit in the iteration domain. The hypercube size is then controlled using a parameter N_{min}. Thus, the general problem is to know what are the iteration domain parameter values that guarantee that an axis-aligned hypercube of size N_{min} fits in the iteration domain. Since the hypercube size is a parameter, the values of the domain parameters are then expressed as functions of N_{min}.

In a d-dimensional space, consider a vector $\overrightarrow{AB}$ with $A, B \in \mathbb{Z}^d$ and a convex $\mathbb{Z}$-polyhedron $\mathcal{P}$. We want first to know if $\overrightarrow{AB}$ can fit in $\mathcal{P}$, i.e. if there exists two points A and B inside $\mathcal{P}$, whose relative position is defined by $\overrightarrow{AB}$.

For a given point A in $\mathcal{P}$, B is obviously the point whose coordinates are those of A translated along $\overrightarrow{AB}$. Thus, as all the possible points A are defined by $\mathcal{P}$, all the possible points B are defined as $\mathcal{P}'$, a copy of $\mathcal{P}$ translated along $\overrightarrow{AB}$. For $\overrightarrow{AB}$ to fit in $\mathcal{P}$, B has also to be enclosed in $\mathcal{P}$, thus all the possible B such that $\overrightarrow{AB}$ is enclosed

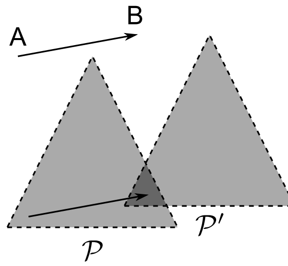

Figure 4.10: The intersection of $\mathcal{P}$ and $\mathcal{P}'$ defines all the points such that B can fit in $\mathcal{P}$, if A is in $\mathcal{P}$.

Algorithm 4.2 Algorithm to determine if a hypercube is included in a polyhedron.

$\mathcal{P} \leftarrow$ `Iteration domain polyhedron`
$\mathcal{P}_{inter} \leftarrow \mathcal{P}$
$H \leftarrow$ `Hypercube to test against` $\mathcal{P}$
$v_1 \leftarrow$ `A vertex of` H `considered as its origin`
$V_H \leftarrow$ `Vectors defined by` v_1 `and any other vertex of` H
`for` $\overrightarrow{vec}$ `in` V_H

$\mathcal{P}_i \leftarrow$ `copy of` $\mathcal{P}$
`translate` $\mathcal{P}_i$ `along` $\overrightarrow{vec}$
$\mathcal{P}_{inter} \leftarrow \mathcal{P}_{inter} \cap \mathcal{P}_i$

`return` $\mathcal{P}_{inter} \neq \emptyset$

in $\mathcal{P}$ are defined by the intersection of $\mathcal{P}$ and $\mathcal{P}'$. An illustration of this situation is presented in Figure 4.10.

Let us now consider a hypercube H with NV vertices, v_i being the ith vertex of H. A specific vertex v_1 with minimal coordinates is called the origin of H. We define the set of vectors $V_H = \{\overrightarrow{v_i v_1}, 2 \leq i \leq NV\}$ made of all the vectors generated from any vertex except the origin of H and pointing to the origin vertex v_1. By definition, if all the vectors in V_H can simultaneously fit in $\mathcal{P}$, all the vertices of H can fit in $\mathcal{P}$ and then, H can fit in $\mathcal{P}$. We have seen before how to ensure that a vector is enclosed in a hypercube. The same principle can be used for all the vectors in V_H: a copy of $\mathcal{P}$ is created and translated for each vector in V_H. The intersection of all the translated copies with $\mathcal{P}$ defines all the points where the origin of H can be placed. The full algorithm is presented in algorithm 4.2 and illustrated in Figures 4.11 and 4.12.

Controlling the Parameters

We have not yet solved the whole problem: now we can determine if a hypercube can fit in an iteration domain but we still have to determine the parameter values such that the hypercube can fit in it.

We call N_{min} the size of the hypercube: every loop level has to execute at least N_{min} iterations. In the algorithm 4.2, the intersection $\mathcal{P}_{inter}$ defines all the points in $\mathcal{P}$ starting from which every loop level executes at least N_{min} iterations. The points

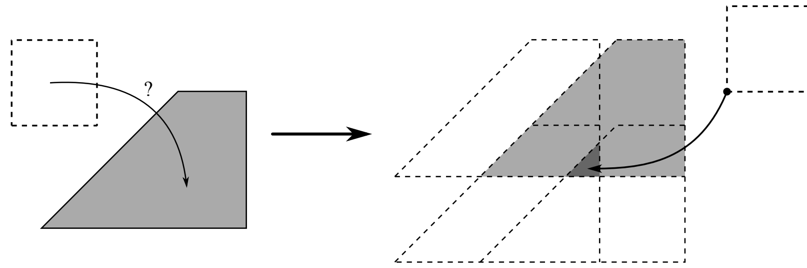

Figure 4.11: Intersection of the translated polyhedron copies in 2D space. The intersection is not empty, the polyhedron can enclose the square.

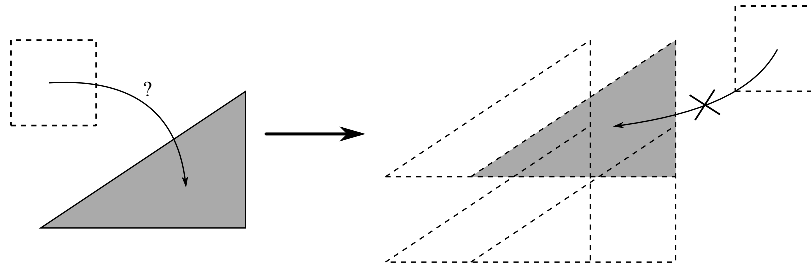

Figure 4.12: Intersection of the translated polyhedron copies in 2D space. The intersection is empty: the square cannot be contained in the polyhedron.

in $\mathcal{P}_{inter}$ should have one coordinate per loop level but we actually consider that the iteration domain parameters are regular dimensions in the same way as any loop level. It means that the iteration domain space has actually as many dimensions as the depth of the nest plus the number of parameters[1]. The parameter dimensions coordinates of integer points in $\mathcal{P}_{inter}$ provide a value for the parameters for which the hypercube fits in $\mathcal{P}$. Thus, if we pick a point in $\mathcal{P}_{inter}$, its coordinates in the parameter dimensions provide us parameter values which ensure that each loop level executes at least N_{min} iterations. As the hypercube size is parameterized, we actually obtain a relation between each loop nest parameter and N_{min}. The size of the iteration domain can then be controlled through this N_{min} parameter.

We present in Algorithm 4.3 the complete algorithm to compute the parameter values. This algorithm successively considers all the statements. The hypercube problem is solved for a hypercube of size N_{min} for every statement domain: for each statement s, we compute the polyhedron $\mathcal{P}^s_{inter}$ which defines the possible origins of the hypercube. If the resulting intersection is empty, no relation can be found, the strategy fails, and the first strategy must be used instead. Otherwise, we pick one point in $\mathcal{P}_{result}$ and consider its parameter dimensions coordinates. All the parameterized translations and intersections can directly be computed on the polyhedra defining the iteration domains. To pick a point in $\mathcal{P}_{result}$, we compute the lexicographic minimum of the set using the PIP [45] library. In the case of tiled loops, the operations are performed over the intra-tile dimensions to control the number of iterations in the domain, and not the number of tiles. An example is provided below.

[1] Such parametric polyhedron representation is called the combined representation.

Algorithm 4.3 Computation of the parameters values.

for each statement s

  $\mathcal{D}^s \leftarrow$ iteration domain of s
  $\mathcal{P}^s_{inter} \leftarrow$ hypercube_pb(N_{min}, $\mathcal{D}^s$)

$\mathcal{P}_{result} = \bigcap \mathcal{P}^s_{inter}$
if $\mathcal{P}_{result} = \emptyset$

  failure

$lex_min \leftarrow$ lexicographic_minimum($\mathcal{P}_{result}$)
return coordinates of lex_min at parameter dimensions

Profiling Code Generation

Once the parameters values are known, the profiling code generation is straightforward. The considered loop nest is syntactically inserted in a generic profiling code pattern given in Figure 4.13.

The profiling code pattern used by this strategy is very similar to the code generated with the first strategy (subsection 4.4.1). The iteration domain parameters are expressed as a function of the hypercube size, `N_min`, which indirectly controls the domain size. The arrays referenced in the computation nests often have sizes defined as functions of the parameters, thus we reallocate the arrays when the parameters change. As in the previous strategy, the profiling is repeated for every possible number of threads, from one to the number of cores, and until reaching less than 5% of variation between two consecutive measurements. In this strategy the maximum number of active threads is controlled through a call to the OpenMP `omp_set_num_threads` function.

Example

Figure 4.14 shows a code sample to be profiled with our system. The first step in the profiling strategy is to determine the value of the parameters relatively to the parameter `N_min`. The initial iteration domain is defined by four inequalities:

$$\begin{cases} i & \geq & 0 \\ -i + N & \geq & 0 \\ j & \geq & 0 \\ -3 \times j + 2 \times M & \geq & 0 \end{cases}$$

We apply a translation of $-N_{min}$ on every combination of dimensions and intersect the results. N_{min} is the size of the hypercube and is therefore positive, we then add this constraint to the system. The resulting domain is the following:

```
do {
    old_result = copy_array(result);
    N_min = N_min * 2;
    PARAMETER_INITIALIZATION
    MEMORY_ALLOCATION
    COMPUTE_LOOP_NEST_SIZE
    for (par_sz=1; par_sz<=NB_CORES; par_sz++) {
        omp_set_num_threads(par_sz);
        start = time();
        LOOP_NEST
        result[par_sz] = (time() - start);
        result[par_sz] = result[par_sz] / LOOP_NEST_SIZE;
    }
    MEMORY_FREE
} while (difference(result, old_result) > 0.05
        && enough_memory(N_min*2));
```

Figure 4.13: Profiling code pattern for strategy 2.

```
int a[N+1][2*M+1];
...
forall (i = 0; i <= N; i++) {
   for (j = 0; 3 * j <= 2 * M; j++) {
      a[i][j] *= 5;
   }
}
```

Figure 4.14: Code sample to profile.

# of cores	version 1	version 2	version 3
1	30 ms	28 ms	32 ms
2	10 ms	14 ms	15 ms
3	7 ms	9 ms	8 ms
4	5 ms	8 ms	6 ms

Table 4.1: Sample parametric ranking table built on a 4-cores processor for 3 code versions. The table content is made of execution times per iteration.

$$\begin{cases} N_{min} & \geq & 0 \\ i & \geq & 0 \\ -i - N_{min} + N & \geq & 0 \\ j & \geq & 0 \\ -3 \times j - 3 \times N_{min} + 2 \times M & \geq & 0 \end{cases}$$

These inequations define the intersection of the translated domains. We now search for an integer point in this polyhedron, by computing its lexicographic minimum. The PIP library computes the lexicographic minimum as the point:

$$\begin{cases} i & = & 0 \\ j & = & 0 \\ N & = & N_{min} \\ M & = & 2 \times N_{min} - (N_{min}/2) \end{cases}$$

The two last coordinates of this point define the relation between the program parameters and the hypercube size, N_{min}. The profiling code generated from those parameter definitions is shown in Figure 4.15. We can see on the figure that the pattern shown in Figure 4.13 is simply filled. The original computation nest is inserted, and the parameter initialization is used to control the domain size depending on the value of `N_min`. Notice the presence of an Ehrhart polynomial to compute the number of executed iterations (`dom_size`).

4.4.3 Parametric Ranking Table

The ranking table, containing the execution times per iteration measured during the profiling step, is two dimensional. One dimension is made of the different versions while the second dimension represents the number of used processor cores. For a given version, and a given number of cores available to the application, the ranking table gives the average execution time per iteration of this version. A simple example of a ranking table is presented in Table 4.1.

4.5 Runtime Selection

The profiling phase characterizes the performance of the code versions depending on the number of threads used. In order to predict an execution time for each version, the number of threads which can be used in the current execution context has to be

```
do {
    old_result = copy_array(result);
    N_min = N_min * 2;
    N = N_min;
    M = 2 * N_min - N_min / 2;
    a = array_allocation(N + 1, 2 * M + 1);
    dom_size = ((2./3.*M + ((M%3==0) ? 1:
                (M%3==1) ? 1./3. : 2./3.)) * (N+1));
    for (par_sz=1; par_sz<=NB_CORES; par_sz++) {
        omp_set_num_threads(par_sz);
        start = time();
        forall (i = 0; i <= N; i++) {
            for (j = 0; 3 * j <= 2 * M; j++) {
              a[i][j] *= 5;
            }
        }
        result[par_sz] = (time() - start);
        result[par_sz] = result[par_sz] / dom_size;
    }
    array_free(a);
} while (difference(result, old_result) > 0.05
        && enough_memory(N_min * 2));
```

Figure 4.15: Profiling code generated from the code in Figure 4.14.

Simple counters	No false sharing	Ehrhart polynomial
3,665 *ms*	1,222 *ms*	20 *ms*

Table 4.2: Time to measure a number of executed iterations.

evaluated at runtime. A runtime component is then run to measure it before each loop nest execution.

4.5.1 Iteration Count Measurement

The runtime component executes a very simplified copy of each version loop nest called *prediction nest*. Its goal is to count how many threads are running simultaneously when each loop iteration is executed. A simple solution to build such nest is to replace every statement by the incrementation of thread-specific counters, the result can then easily be deduced from them. The major performance issue resulting from this solution is due to false-sharing between processor cores. Then, one must ensure that the counters are not allocated contiguously in memory. Another performance issue is due to the full execution of the loop nest while incrementing the counter. A possible solution is to replace the inner sequential loops by the instantiation of the Ehrhart polynomial representing the number of iterations executed by each thread. Such polynomial can be built by tools such as Barvinok [139], considering that the indices of outer loops until the parallel one are parameters. The polynomial can then be instantiated at each iteration of the parallel loop with the current value of the parameters and outer loop indices. Its value is then the number of iterations executed by the loops in the parallel loop.

We have compared those approaches on a simple loop nest made of a single statement enclosed by two tiled loops and where the outermost loop is parallel. The nest executes ten billion iterations in total and four cores are available on the computer. Table 4.2 presents the execution time needed for each method to compute the number of iterations executed by each thread: first using a simple counter per thread, second when avoiding false sharing and third when instantiating an Ehrhart polynomial. We clearly see that the last solution outperforms naive counting. Indeed, it requires only a few tests and computations to instantiate the polynomial, and the time needed for each instantiation does not depend on the number of iterations executed inside the parallel loop.

This is the solution we have chosen for the prediction nest. At compile time, we syntactically replace the content of the parallel loop by the computation of the Ehrhart polynomial counting the number of iterations executed in loops enclosed by the parallel loop. The value defined by the polynomial is added to a counter, specific to the thread which executes the current parallel iteration. The operation is performed per statement: the counter contains the sum of the iteration counts for all the statements. At the end of the prediction nest execution, each counter then stores the number of executed iterations per thread.

Figure 4.16 presents a sample code and its associated prediction nest. Observe that the innermost loop and the statement are replaced by a counting code. This counting

```
for (i = 0; i < M; i++)
   forall (j = 0; j < N; j++)
      for (k = i; k < j; k++)
         A[j][k] = B[i] * 2;
```

```
for (i = 0; i < M; i++)
   forall (j = 0; j < N; j++)
      cnt[thread_id] += j - i;
```

Figure 4.16: A loop nest (left) and its corresponding prediction nest (right).

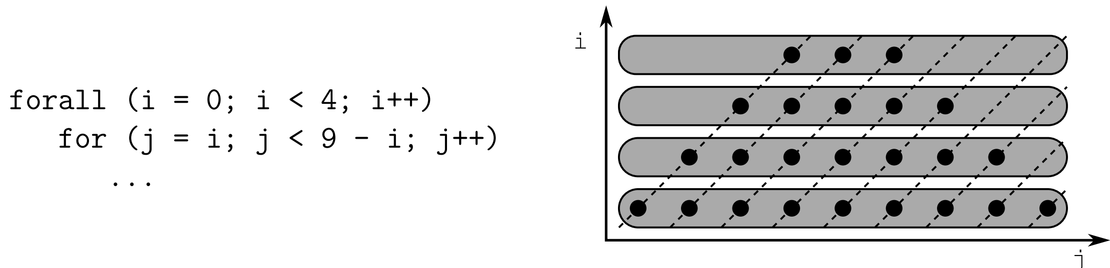

Figure 4.17: Sample loop nest and the corresponding iteration domain where the iterations are grouped by thread.

code increments the thread-specific counter with the trip count of the removed loop. The Ehrhart polynomial $j - i$, generated at compile time, computes this number of iterations. Despite this polynomial is very simple in this example, the method is robust and can handle arbitrarily complex affine loop nests.

4.5.2 Load Balance

We assume that all the loop nest iterations have identical execution times. Under this hypothesis, one can compute the number of iterations executed by each thread count. First, the array of counters `cnt` is sorted in descending order. Then, for each position $1 \leq i <$ number of threads, we can state that `cnt[i-1] - cnt[i]` iterations have been executed in parallel by `i` threads.

In Figure 4.17, we present a sample two-dimensional iteration domain and its corresponding code. The dimension `i` is parallel, four threads are available. The iterations are grouped by thread in the figure, where each gray capsule is a group. The dashed lines cross iterations which are executed simultaneously on different threads. No synchronization is performed in the loop nest but we have assumed that the iterations are all executed in the same amount of time, which leads to this regular execution pattern. Notice that the threads do not necessarily have their first iteration when `j` = 0. One can see that the first thread, on the bottom, executes 9 iterations, the second 7 iterations, the third 5, and the fourth thread executes 3 iterations. The result of the prediction nest is then `cnt = {9, 7, 5, 3}`. Using the technique described before, we can deduce that 2 iterations are executed by one thread $(9 - 7)$, 2 by two threads simultaneously $(7 - 5)$, 2 by three threads $(5 - 3)$, and 3 iterations are executed by all the threads, each one executing 3 iterations. The same result can be observed on Figure 4.17 considering the number of dashed lines for each possible number of active threads. For instance, one can see three dashed lines crossing an iteration for all threads.

The polyhedral loop nests usually have a very regular behavior. The different executions of a statement often lead to similar execution times. Thus, assuming that all the iterations can be executed in the same amount of time is a realistic hypothesis.

4.5.3 Predicting the Execution Time

The execution time can be directly deduced from the parametric ranking table and the prediction nest results. We call NC the number of available processor cores, Nit_t^v the number of iterations executed by t threads (provided by the prediction nest of version v), and ET_t^v the execution time of one iteration of the nest in version v when t threads can be used (provided by the profiling). We can then compute a predicted execution time for the version v as being:

$$\sum_{t=0}^{NC}(t \times Nit_t^v \times ET_t^v)$$

For instance, consider the previous example where 3 iterations are executed simultaneously by all the threads, and where one, two, and three threads execute simultaneously 2 iterations while the other threads are idle. Consider also the profiled execution times of the first profiled version in Table 4.1. The execution time is then computed as $4 \times 3 \times 5$ *ms* for the part of the iteration domain executed by four threads, $3 \times 2 \times 7$ *ms* for the iterations executed by three threads, $2 \times 2 \times 10$ *ms* for the iterations executed by two threads plus $1 \times 2 \times 30$ *ms* for the sequential iterations. This sum predicts an execution time of 202 milliseconds. A predicted execution time is computed in the same way for every version of the loop nest with the current execution context, before running the one that has been predicted as being the fastest one.

Note that if any limit on the number of usable processor cores exists on the system, for example to handle hardware failures, we assume that it remains constant from the execution of the runtime component to the execution of the loop nest. The number of threads available to execute the prediction nest is then constrained by the same limit as when executing the actual loop nest.

4.5.4 Discussion

Links with the Polyhedral Model

Our system uses the polyhedral model at every step. The different versions can be generated automatically while applying very different optimizations thanks to the polyhedral model and its associated tools. In both profiling strategies, the profiling nest can be automatically generated as the loop nest polyhedral representation is available. The datasets used for the profiling can also be automatically generated as they mostly correspond to different parameter values. The profiling and prediction phases use the polyhedral model to efficiently count the number of iterations that have been or will be executed.

One could see the limits imposed on the code structure as a drawback. We claim that it is an advantage for a code selection framework. When considering the full set of possible code fragments, one has to consider for example loops with exit conditions

Model	Frequency	Processor cores	Threads per core
Intel Core i7 920	2.6 Ghz	4	2
AMD Opteron 2431	2.4 Ghz	6	1
AMD Phenom II 965	3.4 Ghz	4	1

Table 4.3: Experimental configuration.

depending on the data. In that case, subtle changes in the dataset can have a large unpredictable impact on the loop execution time. The polyhedral model offers a strict frame where the loop nests are guaranteed to perform regular memory accesses in iteration domains that are statically defined. This ensures that the execution time predictions made by our system are accurate in many different execution contexts even if only a few profiling runs have been launched. Thus, on the contrary of the other existing systems, the execution contexts which can make our selection system fail are much rarer, leading to a more robust selection.

Binary Size Increase

Each loop nest version is generated in a separate function in the binary file. Those versions are increasing the total size of the binary file. This size increases heavily depending on the number of registered versions, the target architecture and the loop nest size. During our experiments, we measured a binary size increase of about a few kilo-bytes for each loop nest version. This size growth has to be related to program sizes and can be considered as negligible in modern general-purpose computers. Moreover, if this factor is considered as highly important for a given architecture, e.g. embedded systems, one can choose to limit the number of considered versions to a small but very efficient subset.

Dynamic Thread Mapping

To predict an execution time, it is assumed that the mapping of the parallel iterations to the threads is the same in the prediction nest and in the corresponding loop nest version. It then excludes dynamic thread mapping, often implemented as work-stealing methods, from the scope of our framework. Such systems would dynamically assign each parallel iteration to a thread at runtime. Dynamic mapping systems have been proven to be effective to enhance load balancing [20] but we show in the experimental results section that such system tends to be counter-performing on very regular codes such as the affine loop nests that our system targets.

4.6 Experiments

We run our experiments on three computers with different multicore processors described in Table 4.3. All of them run a Linux 2.6.35 system. The programs are compiled using the `O3` optimization level of GCC 4.4.

The benchmark programs are 12 common polyhedral loop nests. The code `2mm` is made of two matrix multiply $D = A \times B \times C$, `adi` is the ADI kernel provided for example with PLUTO, `covariance` is a covariance matrix computation, `gemm` and `gemver` are taken from BLAS [17], `jacobi-1d` and `jacobi-2d` are the 1D and 2D versions of the Jacobi kernel, `lu` is a LU decomposition kernel, `matmul` is a simple matrix multiply, `matmul-init` is a matrix multiply combined with the initialization of the result matrix, `mgrid` is a kernel extracted from the `mgrid` code in SPECOMP [5] and `seidel` is a Gauss-Seidel kernel as provided with PLUTO. Those kernels are typically put in libraries and called many times by applications cumulating the benefits of our system.

For each benchmark program, many versions are generated. One of those version is automatically generated by the PLUTO parallelizing compiler. Other versions have been designed by an expert, changing the number of tiling levels and the tile sizes, and performing polyhedral loop transformations. As explained in Section 4.3, those versions could have been generated by automatic tools. Table 4.4 details how each considered version has been built. In this table, each code version is associated with its tile size and its parallel schedule. When codes are made of many statements, the tile size is actually the tile size of the most time consuming statement. A letter next to a tile size denotes different tiling of less representative statements. From each version, the profiling and prediction codes are generated by a set of fully automatic scripts in our implementation. The result of the prediction codes determines the framework choice in each execution context.

Table 4.4: Considered code versions.

Name	Tile sizes	Sched.
`2mm`	$1 \times 1 \times 1$	A
	$16 \times 16 \times 16$	B
	$32 \times 32 \times 32$	B
	$1 \times 32 \times 32$	C
	$1 \times 16 \times 16$	C
`adi`	$32 \times 32 \times 32$	A
	$16 \times 16 \times 16$	A
	$16 \times 16 \times 16$	B
	$32 \times 32 \times 32$	C

Continued on next page...

Name	Tile sizes	Sched.
covariance	32 × 32 × 32	A
	32 × 32 × 32	B
	32 × 32 × 32 (a)	C
	32 × 32 × 32 (b)	C
	32 × 32 × 32 (c)	C
	32 × 32 × 32	D
gemm	32 × 32 × 32	A
	1 × 1 × 1	A
	64 × 64 × 64	A
	32 × 32 × 32	B
	16 × 2 × 16 × 16 × 128 × 16	C
	32 × 32 × 32	B
	52 × 52 × 52	D
gemver	20 × 16 × 400 × 16	A
	64 × 64	B
	16 × 16	B
	16 × 16	A
	64 × 64	A
jacobi-1D	1 × 1	A
	256 × 256	B
	256 × 256	C
	256 × 256	D
jacobi-2D	32×32×32	A
	32×32×32	B
	32×32×32	C
	16×16×16	C
	32×32×32	D
lu	16 × 2 × 16 × 16 × 100 × 16	A
	32 × 32 × 32	B
	32 × 32 × 16	B
matmul	16 × 2 × 16 × 8 × 128 × 8	A
	16 × 16 × 16	A
	32 × 32 × 32	A
	64 × 64 × 64	A
	128 × 128 × 64	A
matmul-init	8 × 2 × 8 × 8 × 128 × 8	A
	16 × 16 × 16	A
	32 × 32 × 32	A
	16 × 16 × 16	B

Continued on next page...

Name	Tile sizes	Sched.
`mgrid`	$32 \times 32 \times 32$	A
	$16 \times 16 \times 16$	A
	$32 \times 32 \times 32$	B
	$32 \times 32 \times 128$	B
`seidel`	$10 \times 15 \times 15$	A
	$16 \times 16 \times 16$	A
	$16 \times 16 \times 16$	B
	$16 \times 16 \times 16$	C
	$8 \times 8 \times 8$	A

The evaluated execution contexts are made of six different input data with increasing random sizes and shapes. For example, if the input data is a matrix, some datasets can be made of square and rectangular matrices of different sizes. The chosen data usually lead to execution times ranging from half a second to a couple of minutes. For each data size, all the different number of available cores is considered.

4.6.1 Dynamic Scheduling of Regular Codes

To measure the load balance in the current execution context, the prediction nest is run. The way this measurement is done imposes that the mapping of the loop iterations to the threads is reproducible from the load balance measurement to the actual execution. This prevents us from using our system with a dynamic scheduler.

The dynamic schedulers are usually efficient for irregular applications, with various task sizes and complex communication schemes [19]. They are however much less efficient on regular applications such as the loop nests we target, meaning that the static mapping of threads on processor is often the best solution for those applications. To illustrate this, we have measured the speedup of two dynamic schedulers over the default static blocked scheduling provided by OpenMP.

We show in Figures 4.18 to 4.23 the speedup of the OpenMP dynamic strategy (on the left) and of Cilk version 8503 [19] (on the right) over the OpenMP static blocked scheduling strategy. The tests have been run on the three test machines. Each presented value is the average speedup of the best version of one program, over six different data sizes.

We clearly notice on the figures that both dynamic schedulers fail to achieve good performance in most of the tested cases: the dynamic strategies rarely outperform the static one. Those runtime systems usually count on the tasks irregularity to compensate their overhead. The programs we consider are regular and seem to rarely exhibit enough irregularity to let those dynamic schedulers reach good performance. It means that, not using a dynamic scheduler with our system is not a strong limitation.

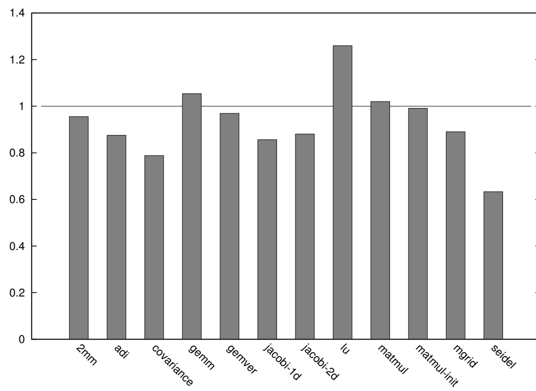

Figure 4.18: Speedup of OpenMP dynamic over static on Core i7.

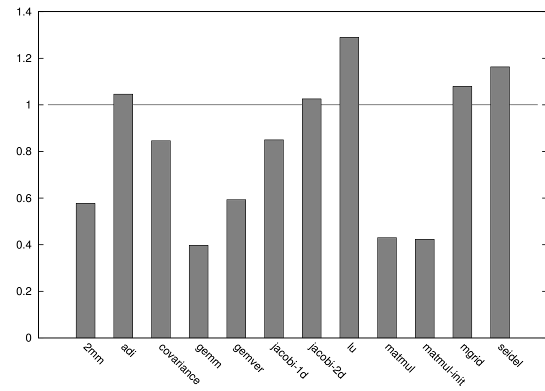

Figure 4.19: Speedup of Cilk over OpenMP static on Core i7.

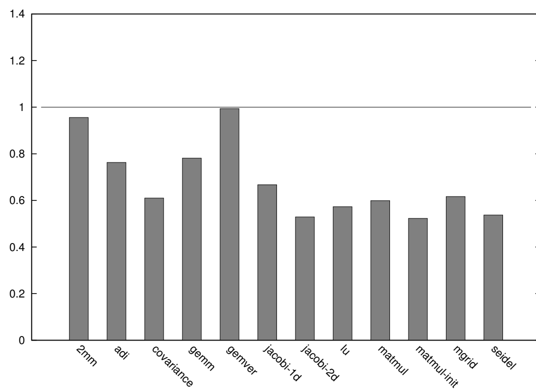

Figure 4.20: Speedup of OpenMP dynamic over static on Opteron.

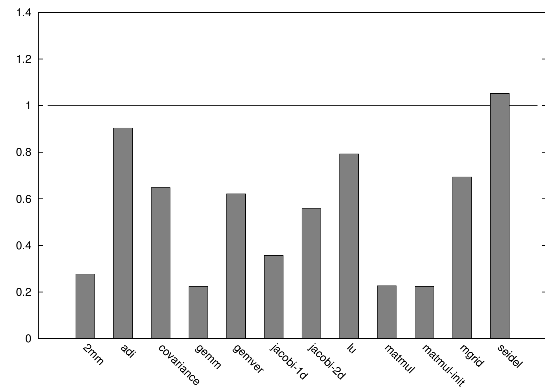

Figure 4.21: Speedup of Cilk over OpenMP static on Opteron.

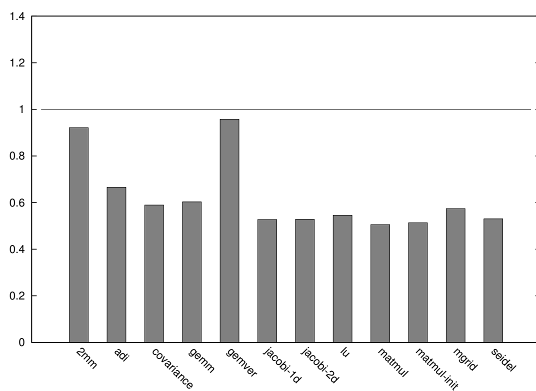

Figure 4.22: Speedup of OpenMP dynamic over static on Phenom II.

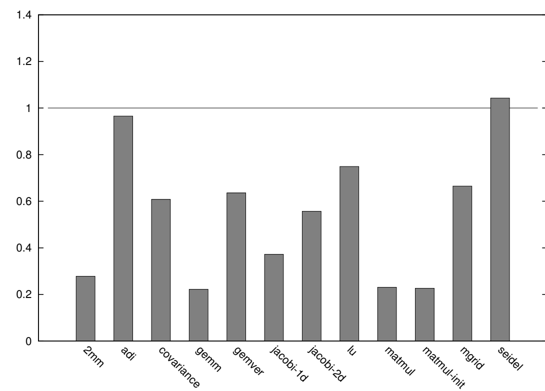

Figure 4.23: Speedup of Cilk over OpenMP static on Phenom II.

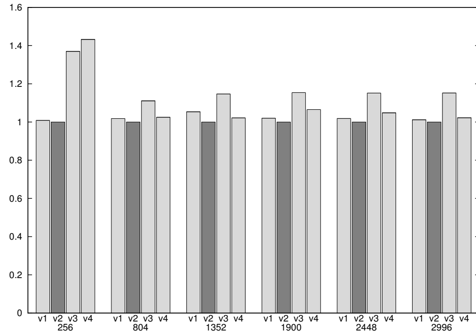

Figure 4.24: Execution times for `adi` on Opteron (5 threads). Version 2 is the best one for the fifth dataset.

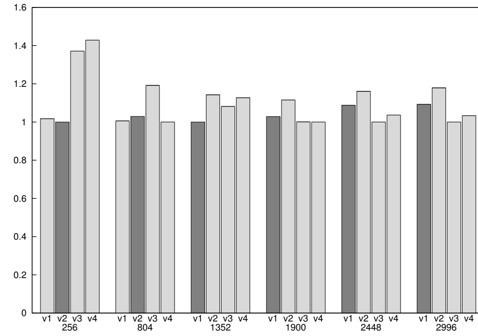

Figure 4.25: Execution times for `adi` on Core i7 (5 threads). Version 2 is the worst one for the fifth dataset.

4.6.2 Execution Context Characteristics

In this subsection, we illustrate different cases using histograms. The histograms in Figures 4.24 to 4.28 represent execution times of every version of a specific program on a given architecture using a fixed number of threads. The execution times are normalized over the one of the best version in each context: the best version has always an execution time of 1, others are above as they necessarily perform worse. The different data sizes and the different versions are distributed on the horizontal axis. The darker bar is the version that our system selects using the strategy 2 to profile the versions.

During our experiments, we observed many performance variations. First, significant performance variations can be observed for a given code version among different computers for a fixed problem size and number of cores. For example with `adi`, when considering a problem size of 2448 (fifth dataset on figures) and 5 available processor cores, version 2 is the best one with the Opteron processor (Fig. 4.24) but the worst one with the Core i7 processor (Fig. 4.25).

We also observed performance variations when the number of available cores on a given computer varies for a fixed problem size. For example, with the Core i7 processor, with `gemver` and for a problem size of 10000 (first dataset on figures), the first version is the best one when only one processor core is available (Fig. 4.26) but the worst one when all the cores can be used (Fig. 4.27).

Finally, when the problem size changes, we can also observe performance variations for a fixed computer and a fixed number of available cores (Fig. 4.28). For example, with all the processor cores of the Phenom processor, the fourth version of `mgrid` is inefficient for a problem size of 320 (third dataset on the figure) but is the best one for a problem size of 500 (last dataset on figure).

This illustrates how the execution context, made of the input data, the architecture, and the number of available processors, impacts the execution time of a program. Thus, a dynamic system able to quickly detect the best version for the current context leads to speedups. We show in the next subsections that our system is able to quickly and accurately detect the best version in various contexts.

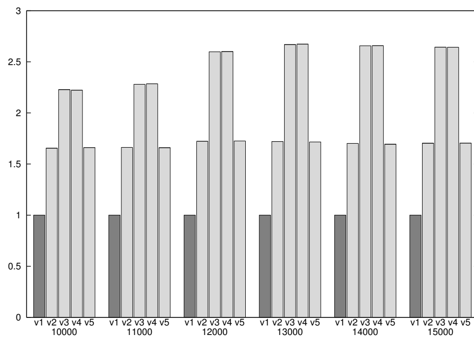

Figure 4.26: Execution times for `gemver` on Core i7 (1 thread). Version 1 is the best one for the first dataset.

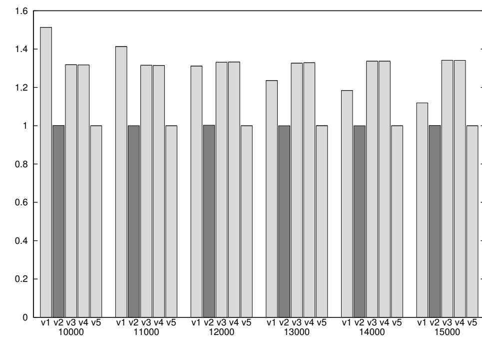

Figure 4.27: Execution times for `gemver` on Core i7 (8 threads). Version 1 is the worst one for the first dataset.

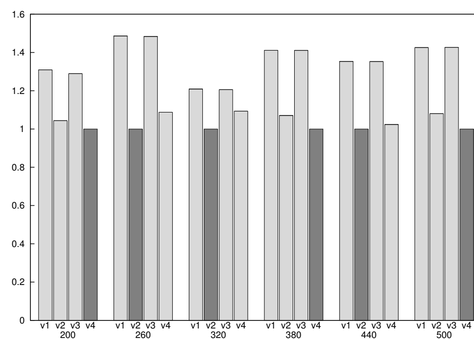

Figure 4.28: Execution times for `mgrid` on Phenom (4 threads). Version 4 is the best one for the last dataset, but is inefficient for the third dataset.

4.6.3 Execution Time Gains

Our runtime system is evaluated on each of the three computers, for each code with six different problem sizes. Each measurement is repeated with a number of threads varying from one to the number of logical cores, simulating different resource availability. The statistics presented in Table 4.5 then enclose more than one thousand runs, each measurement being the median value out of five executions (although we did not notice much difference between different runs in identical contexts).

The theoretic best runtime mechanism would select at no cost the best code version among all considered versions for any given execution context. All measurements presented in Table 4.5 are speedup compared to this theoretic best runtime mechanism, so they are necessarily below or equal to 100%. Having 100% in a table cell means that the maximal performance has been reached using the considered code versions in all the execution contexts.

For each processor and program, we present three measurements. The first two columns correspond to the two profiling strategies presented in this chapter. The last one, the *best static version* is one unique version of the program that performs best in average in all the tested execution contexts. If one considers an oracle static compiler, it is the version that this perfect offline compiler would generate. Note that no system is currently able to generate this code version, it has been chosen here after having tested all the possible versions in every execution context.

A data is missing with `lu` for the strategy 2 of our system because, due to its code characteristics, the strategy 2 is not able to build a correct profiling code. Also note that the presented numbers takes the runtime system overhead into account.

One can see that both strategies provide very good results. The runtime system is able to exploit almost all of the available performance in all the tested cases. There is no clear winner between the two strategies. One can also see that our system is able to outperform in some cases the best static version. It means that even an oracle static compiler would not be able to exploit as much performance as our system is able to in those cases thanks to its dynamic nature. In every case, our system is able to compete with this best static version.

4.6.4 Accuracy

We present in Figures 4.29-4.34 a graphical evaluation of our system accuracy in the tested execution contexts. The horizontal axis represents the code versions of a program in each execution context: each value on that axis represents one code version evaluated on a specific dataset, using a specific number of threads. For each of those entries, we present the actual execution time (full line) and the predicted execution time (dashed line). Those values are sorted by the actual execution time. On the left, the predictions are made using the strategy 1, on the right using the strategy 2. A good prediction leads to a dashed line on the graphs that closely follow the variation of the full line.

We can see on the figures that the predicted execution times are very precise and closely follow the actual execution times. There is no large misprediction: no efficient version is predicted as being inefficient and no inefficient version is predicted as being efficient.

Processor	Program	Strategy 1 performance	Strategy 2 performance	Best static performance
Corei7	`2mm`	100.0 %	98.9 %	100.0 %
	`adi`	99.6 %	98.7 %	97.5 %
	`covariance`	96.6 %	95.8 %	99.7 %
	`gemm`	93.4 %	84.7 %	93.5 %
	`gemver`	80.6 %	98.3 %	91.6 %
	`jacobi-1d`	99.5 %	99.5 %	99.9 %
	`jacobi-2d`	90.2 %	94.8 %	99.6 %
	`lu`	91.2 %	-	98.3 %
	`matmul`	98.5 %	97.9 %	98.5 %
	`matmul-init`	100.0 %	97.6 %	100.0 %
	`mgrid`	97.0 %	99.9 %	97.0 %
	`seidel`	99.5 %	99.8 %	99.6 %
Opteron	`2mm`	100.0 %	100.0 %	100.0 %
	`adi`	99.1 %	99.6 %	97.3 %
	`covariance`	99.8 %	99.8 %	99.8 %
	`gemm`	97.8 %	96.5 %	96.7 %
	`gemver`	99.7 %	99.4 %	99.8 %
	`jacobi-1d`	99.6 %	99.6 %	100.0 %
	`jacobi-2d`	100.0 %	98.5 %	100.0 %
	`lu`	100.0 %	-	100.0 %
	`matmul`	100.0 %	96.9 %	100.0 %
	`matmul-init`	100.0 %	100.0 %	100.0 %
	`mgrid`	96.2 %	99.0 %	98.5 %
	`seidel`	98.9 %	99.5 %	98.3 %
Phenom	`2mm`	100.0 %	100.0 %	100.0 %
	`adi`	98.7 %	99.5 %	97.5 %
	`covariance`	99.9 %	99.9 %	99.9 %
	`gemm`	99.2 %	97.2 %	96.9 %
	`gemver`	99.7 %	99.1 %	99.8 %
	`jacobi-1d`	99.7 %	99.7 %	100.0 %
	`jacobi-2d`	99.4 %	98.7 %	100.0 %
	`lu`	100.0 %	-	100.0 %
	`matmul`	100.0 %	100.0 %	100.0 %
	`matmul-init`	100.0 %	100.0 %	100.0 %
	`mgrid`	95.9 %	99.7 %	98.1 %
	`seidel`	99.0 %	98.9 %	99.0 %

Table 4.5: Speedup of our two strategies and the best static version compared to the theoretic best runtime mechanism.

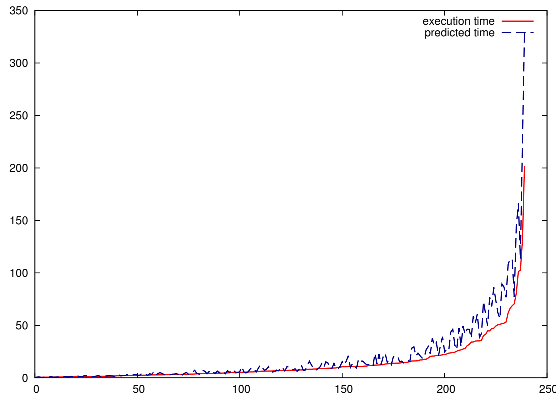

Figure 4.29: Execution time of every version of `2mm` in every execution context, sorted by actual execution time on Core i7 with strategy 1.

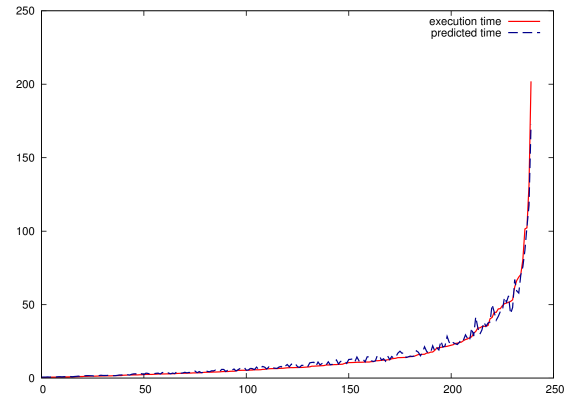

Figure 4.30: Execution time of every version of `2mm` in every execution context, sorted by actual execution time on Core i7 with strategy 2.

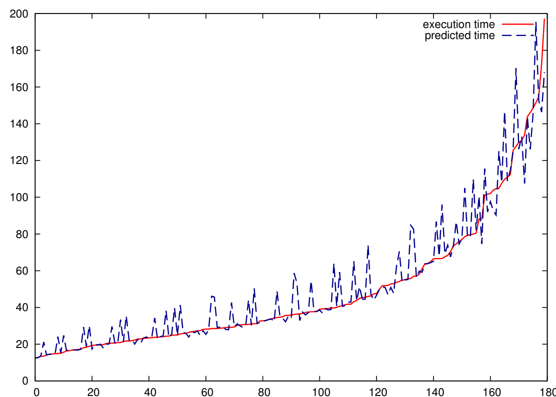

Figure 4.31: Execution time of every version of `jacobi-2d` in every execution context, sorted by actual execution time on Opteron with strategy 1.

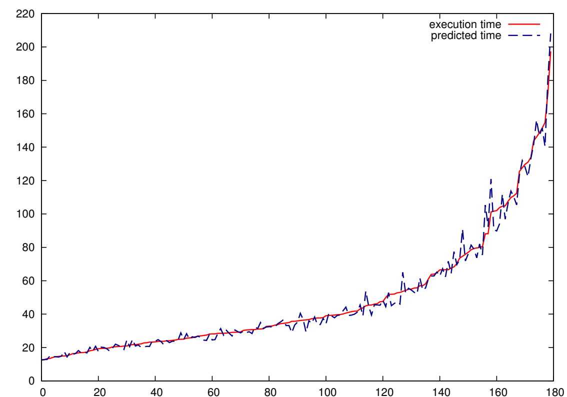

Figure 4.32: Execution time of every version of `jacobi-2d` in every execution context, sorted by actual execution time on Opteron with strategy 2.

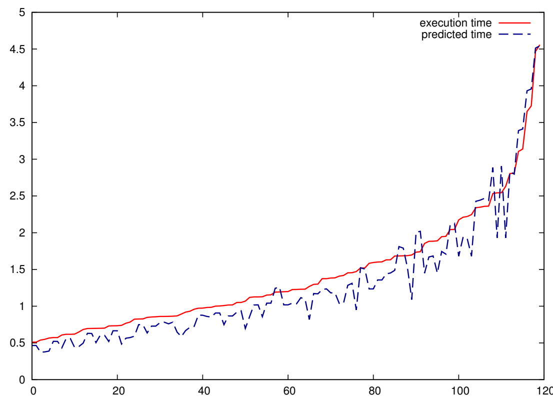

Figure 4.33: Execution time of every version of `gemver` in every execution context, sorted by actual execution time on Phenom II with strategy 1.

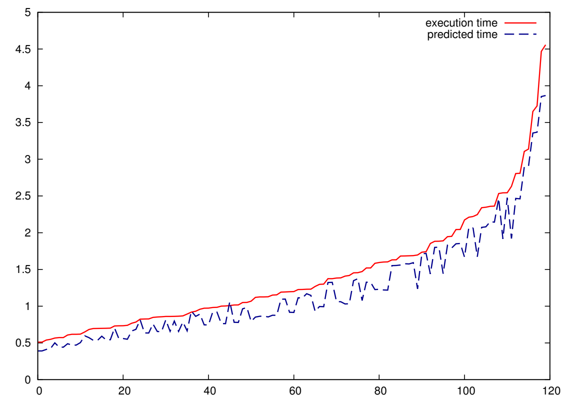

Figure 4.34: Execution time of every version of `gemver` in every execution context, sorted by actual execution time on Phenom II with strategy 2.

One can also notice that the second strategy seems to be more precise in the presented cases as the predicted execution times are closer to the actual ones compared to the first strategy. It means that it is preferable to use the strategy 2 whenever possible as it leads to more precise predictions. With this strategy, the system is able to select a good version even when only small benefits can be expected. This behavior is generally observable on all the tested codes.

4.7 Conclusion and Perspectives

We have presented in this chapter a new framework able to select at runtime the fastest version of a polyhedral loop nest. The contributions of this work are multiple. First we identify two main factors that impact performance of a precisely defined set of codes structures on modern multicore architectures. Second, we propose two fully automatic techniques able to generate profiling codes and to profile those code structures according to the identified factors. Third, we show how to automatically generate a very efficient runtime component able to measure the identified performance factors. We also show that, when put all together, those components constitute a very efficient code selection system able to automatically select an efficient version of a parallel affine loop nest at nearly no cost and since the very first execution.

The polyhedral model puts strong constraints on the codes which can be handled by our system. A natural extension would then be to consider more complex code. Some recent extensions to the polyhedral model [14] could be considered. The main risk is a loss of precision as the loops performance would not be as stable. It could also be interesting to adapt our system to use dynamic optimizations such as parametric tiling: the versions would then be parallel parametrically tiled loop nests with different schedules. Going further in the dynamic aspect, a fully dynamic system could be built where no profiling is required but actual executions would fill the parametric ranking table.

Chapter 5

Speculative Parallelization

5.1 Introduction

We present in this chapter the last part of our general approach, which consists in a dynamic parallelization system. It complements the two other systems presented earlier, and addresses another challenge for the polyhedral model, namely handling more complex codes.

The polyhedral model is originally targeting statically analyzable loop nests with affine loop bounds and memory references. However, many programs do not enter in this category and contain some complex constructs which can often not be analyzed offline. Different extensions to the polyhedral model have been proposed to specifically overcome some of those obstacles such as non-linearity [31, 56], or data-dependent control flow [14]. However, despite techniques such as points-to analysis [61, 29] can help to statically analyze pointer references, several memory references are unanalyzable at compile-time. To handle those codes, dynamic parallelization has to be used.

Dynamic parallelization consists in parallelizing the program while it executes. More information is actually available at runtime, allowing one to parallelize a program which cannot be handled statically. For instance, in some programs, the memory accesses performed allow the parallelization only during some phases of the program execution. They require a dynamic system to detect and exploit those parallel phases. We call those programs *partially parallel* programs.

The dynamic parallelization systems can be divided in two families: those based on the inspector/executor model, and the speculative systems. The formers extract the information required to validate and perform the parallelization at runtime. On the other hand, the speculative systems emit hypothesis on the program behavior in order to parallelize it. Those assumptions are verified while the program is run in parallel, requiring a back-tracking system to rollback a part of the execution in case of misprediction. Speculative systems can handle a broader class of programs as they can parallelize programs for which it would be too costly, or even impossible to perform a sufficient analysis before the parallelization. In this chapter, we focus on speculative systems, and propose a speculative mechanism performing polyhedral transformations on loop nests to parallelize them. Such system allows us to apply polyhedral transformations on programs which are currently out of the scope of the model, extending the

benefits of the model to a wider class of programs. With such a system, the polyhedral model can handle partially parallel programs, applying polyhedral transformations only in some phases of the loop nests execution. It can also handle programs which are not statically analyzable, and can parallelize loop nests with complex code structures and memory references.

We present in Section 5.2 an overview of our parallelization system. We also introduce some base concepts which are required for the rest of the chapter. In the sections 5.3 to 5.8, we detail every phase of the parallelization and evaluate several possible implementations. In Section 5.9 we decide on a general system and evaluate it before concluding.

5.2 Overview

5.2.1 Speculations

The following code extract is a typical example of the loop nests targeted by our speculative parallelization system:

```
while (p != NULL) {
   ...
   if (error) {
      ...
   }
   p = p->next;
}
```

In that code, a linked list is traversed and a computation is performed on every element. The loop bound is unknown at compile time, the linked list pointer `p` is updated at each iteration provoking a dependence between every successive iterations. What has to be particularly noticed is that if the list is allocated consecutively in memory, the addresses reached by memory references can be described as an affine function. The error test in the sample code is hopefully never taken in a normal execution.

The first adaptation performed by our system is to create a loop counter for each while loop. This counter is starting at zero and incremented at every loop iteration. It allows every loop to have a counter, independently from their control structure.

To exploit polyhedral techniques on such complex loop nests, several kinds of speculations are targeted by our system:

1. Address speculation: whenever possible, the parallelization system must be able to infer that the memory references are linear in order to use polyhedral techniques. In our example, we cannot statically prove that property for the memory references accessing the linked list elements. Thus, our system checks if this can be observed on a few loop nest iterations, assumes that this property is true for the whole execution, and verifies the perpetuation of this property when the program is run in parallel.

2. Value speculation: the pointer `p` prevents any parallelization of this loop nest as it generates dependences between every couple of successive iterations. For each iteration, the first access to this scalar is a read, thus, it cannot be privatized. However, if one is able to characterize the successive values assigned to `p`, it could be initialized at the beginning of every iteration and privatized, suppressing the problematic dependences. Thus, our system speculates that the value of `p` can be defined as an affine function of the surrounding loop indices and global parameters. This function is then used to initialize it at the beginning of every iteration.

3. Control speculation: if some dependences occur in the error test, they can usually be ignored. Ignoring the dependences provoked by such infrequent memory accesses is the first objective of control speculation. Our system will then observe the program for a few iterations and detect infrequent memory references. They will be speculated to be never executed during the parallel execution, causing a failure if the speculation appears to be incorrect. The second objective of control speculation is to speculate that the trip count of while loops are also linear functions, allowing the system to use some of the techniques associated to the polyhedral model even in presence of while loops.

Address speculation is performed for every memory reference which cannot be analyzed statically. It allows us to have a linear characterization of those memory accesses, enabling the use of polyhedral techniques afterwards.

Value speculation is performed only for scalars which cannot be privatized, which are written in the loop nest, and whose successive values cannot be statically defined through scalar evolution analysis. It is also limited to the scalars which participate in address computations, as their is only little hope to observe linear behaviors for the scalars used for the computations. The speculation allows us to initialize the concerned scalars at the beginning of every iteration and to privatize them. This leads to remove the dependences they provoke, and then to parallelize more loop nests.

Control speculation for while loops is obviously not attempted on the outermost while loop. Indeed, it requires several full loop nest executions before being able to characterize the trip count of such loops. For the other while loops, the linear characterization of their trip counts allows us to exploit some polyhedral techniques on the nest such as precisely determining the memory area reached by the accesses during the nest execution. The while loops with data-dependent exit conditions are a challenge in the polyhedral model and have to be specifically handled. A static method able to apply loop transformations and parallelization in presence of such loops has been proposed in [14]. We exploit this technique to allow such complex loop structures in the parallelized program. We could have exploited the speculation on the while loop trip counts to consider any loop as an affine for loop. The main issue here is the speculation verification which has to ensure that the speculated trip count is correct, even if the loop level may have been transformed. This is left for future research.

When put all together, the different speculations allow us to use the polyhedral model and the associated techniques in presence of arbitrary complex control structures and memory references, as long as the speculated values can be defined by affine

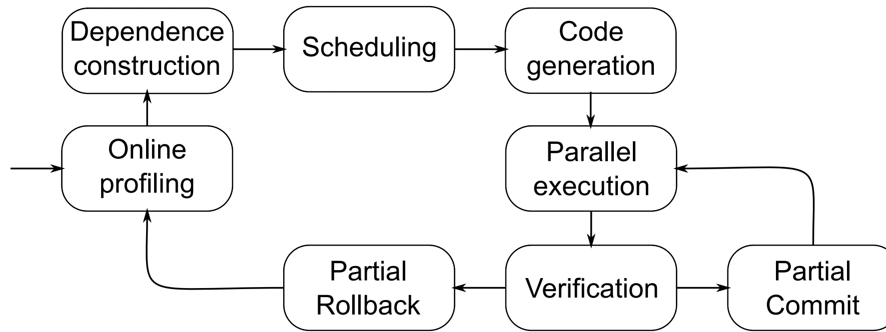

Figure 5.1: Main phases of the speculative system.

functions.

5.2.2 General overview

We present in Figure 5.1 a scheme describing a system able to perform the speculations aforementioned. The first step is to observe the application behavior during a short online profiling step. In this phase, the speculated memory accesses, loop bounds, and scalar values are instrumented to check if they can be characterized by a linear function. This online profiling only last for a few loop iterations. If a linear characterization of the loop nest can be built at this step, it is assumed to be valid for the rest of the execution and used as a predictor.

In the next step, a dependence test is performed. From this dependence test, a polyhedral transformation may be applied on the loop nest and the resulting parallel code is generated. The resulting loop nest is then executed and, in the mean time, a verification is performed to ensure that the speculation is actually correct. If the actual program behavior does not follow the predicted one, a rollback is performed to cancel the incorrect parallel execution part. Those iterations are then re-executed using another schedule, and a new profiling is performed to check if the behavior of the loop nest has changed. Then, a new schedule may be used to parallelize the next iterations of the loop nest. All those steps are performed at runtime, and form a dynamic parallelization system.

It is interesting to point out that there are no more invalid parallel schedules with such a speculative parallelizer. In case of mispredictions, a rollback mechanism will anyway cancel and replay the invalid iterations using another valid execution order (usually the sequential one), leading to a valid execution in any case. However, the number of rollbacks and their associated cost must be minimized in order to limit the runtime overhead and maximize the effective parallelism. For instance, consider a loop nest where the outermost loop carries a dependence and which is then not parallel. One could try to speculatively parallelize it, leading to a rollback at every iteration. On the other hand, if a loop interchange can bring a loop which does not carry any dependence at the outermost level, the speculative parallelization of this loop does not anymore provoke any rollback. The polyhedral model enables all sorts of transformations, leading to different number of rollbacks. As our system speculates on the dependences and transforms the nests accordingly, its goal is to minimize the

program	sequential time	# iterations	# outermost iterations	nest depth
`ind`	1,495 *ms*	25,000,000	5,000	2
`llist`	3,100 *ms*	50,000,000	50,000,000	1
`rmm`	10,330 *ms*	1,000,000,000	1,000	3
`switch`	3,960 *ms*	64,000,000	8,000	2

Table 5.1: Main characteristics of the reference programs.

number of rollbacks when the nest is not naturally parallelizable, compared to a naive speculative parallelization of the original loop nest.

For clarity reasons, we still evoke in this chapter "valid" parallel schedules. This validity notion is the classical non-speculative one, where the schedule is valid regarding the speculated dependences, although any schedule is "valid" in the context of such a speculative system.

5.2.3 Evaluation Environment

Different design options can be considered to build our speculative system. We describe the most relevant ones in the following sections, their pros and cons are evaluated separately on four example codes:

- `ind` which uses array indirections to compute the average of neighbors elements. Such stencil computation is typical in image processing where a mask determines where to apply the operation.
- `llist` is a linked list traversal where a computation is performed on every list element.
- A matrix multiply `rmm`, where the arrays have been dynamically allocated, making any static code analysis impossible for most of the existing tools.
- `switch` applies two different operations on the elements of an array according to the value of linked list elements. Two different parallel schedules have to be used to parallelize this application, as each operation references different stencil of neighbors in the array.

Even a perfect static code analyzer could not determine the memory behavior of most of those codes as it depends on dynamic characteristics such as the memory location of linked list elements: if those elements are allocated successively, there is a high probability to be located in consecutive memory locations, however, this cannot be guaranteed at compile time. Thus, the presented programs are good targets for speculative parallelization systems. Moreover, they all have a distinct behavior and emphasis different functionalities that an advanced speculative system should provide.

The different steps of the speculative system and their associated alternative mechanisms are evaluated on a Linux 2.6.38 system, using a Intel Xeon W3520 processor with four cores and two threads per core. The reference sequential execution time of every benchmark program is presented in Table 5.1, accompanied by the number of

```
for (i = 0; i < N; i++) {
   ...
}
```

```
for (cnum = 0;
  cnum <= ceil(N / csize);
  cnum++)
{
   for (i = cnum * csize;
    i < N && i < (cnum + 1) * csize);
    i++)
   {
      ...
   }
}
```

Figure 5.2: Sample loop nest (left) and its chunked counterpart (right).

iterations executed in the nest, the number of outermost loop iterations, and the loop nest depth.

5.2.4 Chunking

Before going into the details of every phase, we present our chunking mechanism. Grouping the iterations of a parallelized loop nest by chunks is a common practice of speculative parallelizers. Usually the chunks are assigned to different processors leading to a parallel execution. We use a different approach which we detail here.

In the different phases of our speculative system, several operations have to be performed only for a few successive iterations of the loop nest. For instance, the online profiling has to be executed only for a few iterations, just enough to extract a speculative characterization of the program. To allow different operations on successive parts of the iteration domain, we divide it into several *chunks*.

Those chunks are groups of consecutive iterations of the original outermost loop. The chunking is easily obtained by strip-mining this outermost loop. This transformation can also be applied on while loops as we have introduced virtual loop counters. We show in Figure 5.2 a simple for loop, transformed to be executed by chunks of `csize` iterations. This chunk size can be adapted to form chunks of different sizes if needed. The resulting outermost loop, iterating over the chunks, is considered as not being a part of the loop nest. If a polyhedral transformation is applied on the loop nest later, this outermost loop is then left unchanged. It leads to transform and parallelize the iterations in the chunks, while sequentially iterating over the chunks. This is especially important for the completeness and correctness of the speculation verification, as explained in Section 5.7.

Performing this chunking on programs is not neutral for their performance, especially when a loop nest transformation is applied on the chunked program. This can lead to extra controls in the parallelized version of the nest, and then to performance drops in extreme cases. Moreover, the chunk size has a direct influence on the loop nest performance, and choosing a correct chunk size is of capital importance. Indeed, in a speculative context, larger chunks increase the cost of a misprediction as more iterations have to be re-executed, while small chunks can provoke a shortage situation

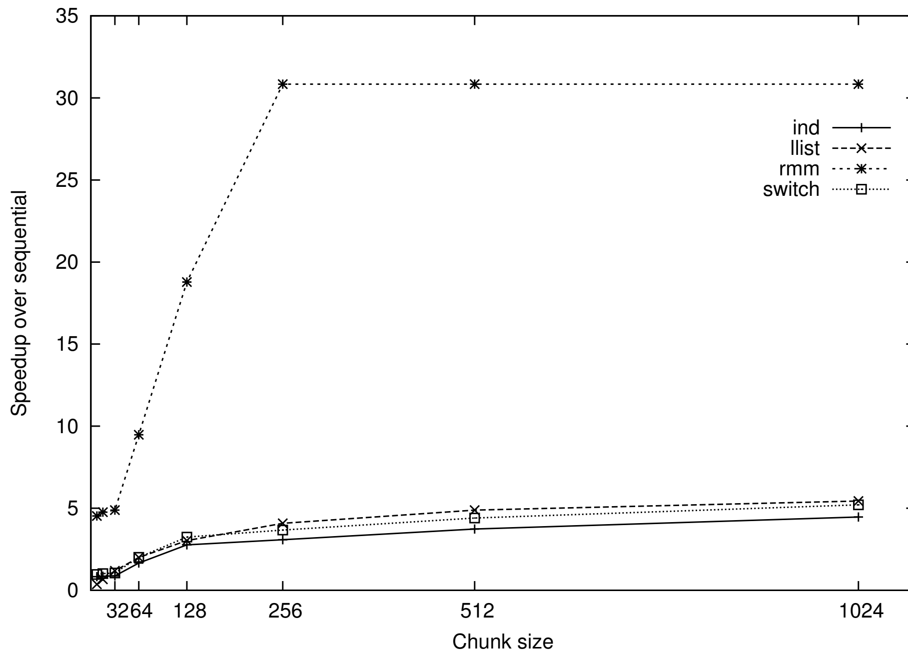

Figure 5.3: Speedup of each code over its sequential version, relatively to the chunk size.

with unoccupied processor cores. For example, consider a loop nest which is not transformed but only tiled and parallelized on the outermost tiling level. If the chunk size, i.e. the number of iterations of the outermost loop nest dimension, is lower or equal to the tile size, then only one iteration of the parallel loop is run during the execution of each chunk, leading to execute the loop nest in sequential.

To illustrate this, we present in Figure 5.3 the speedup of the four example programs parallelized and tiled with PLUTO, after having been rewritten to allow their static parallelization. The speedups are computed over the original sequential version of each loop nest. One can observe that the programs perform well only after a minimal chunk size around 256 iterations. This size corresponds to the total number of processor cores multiplied by the tile size used: good performance is reached when each processor thread can be occupied. The case of `rmm` is specific as large improvements are provided even with very small chunks thanks to the transformation applied by PLUTO which greatly enhances the data locality. However, even in that case, one can observe a clear performance increase when using large chunks.

Our system makes an heavy use of this chunking system for various purposes, including the parallel execution. During this parallel execution, the chunk size is fixed to be $NP \times TS$ where NP is the number of processor threads, and TS is the tile size used. This chunk size is the minimal chunk size ensuring that the chunks does not result in a severe load imbalance. Notice that other phases of the system can use different chunk sizes, as for instance the online profiling step.

5.3 Online Profiling

One of the first step in our speculative system is the online profiling. During that phase, the different speculated elements (memory accesses, scalar values, and while loop trip counts) are observed during a few iterations of the original loop nest. The result of this short profiling period is used to build a linear function describing every speculated element, whenever possible. Those linear functions of surrounding loop indices and global parameters are then assumed to represent the actual behavior of the program for the rest of the parallelization process. Later, a runtime verification phase ensures that this is effectively true.

Every memory reference is instrumented and affine functions describing the addresses they reach are tentatively built. This includes the references which can be statically analyzed. The goal is to obtain an exact characterization of the accessed addresses in order to perform dependence tests directly on the affine functions. For instance, if a pointer reference and an array access are both present in the loop nest, we must build a linear function at runtime for both accesses in order to determine if they can alias. The knowledge of the array access linearity is not sufficient for that purpose, the addresses reached are required. However, if an access is statically analyzable, the corresponding linear function is not speculative and does not need to be verified during the parallel execution.

The value of the speculated scalars is also observed and affine functions are tentatively built to describe their successive values. A similar operation is performed for the instrumented while loops trip counts.

This online profiling is performed before any program transformation. It benefits from the chunking transformation described earlier to execute only a few iterations of the original loop nest.

5.3.1 Inspector Profiling

As in the inspector/executor model, we can notice that some of the actual computations are sometimes not required to compute the correct values of the instrumented elements. For instance, computations on linked list elements are often decoupled from the linked list traversal. Thus, an inspector could be built only with the instructions required to perform the correct memory accesses, scalar references, and number of iterations of while loops. This instrumented inspector could then be used to gather the desired data without having to execute all the computations. Note that we do not execute the inspector for the whole loop nest execution as it is done in the inspector/executor model. We only use it to perform a faster profiling during a few loop iterations.

The obvious advantage of that method is a limited overhead compared to profiling the whole loop nest body. However, the computations that have not been executed during the profiling still have to be performed later. Moreover, it can be tricky to automatically generate such an inspector as we target some codes that can contain complex pointer references.

Nevertheless, the performance of both a simple profiling and an inspector-based approaches are presented in Table 5.2. The presented execution times correspond to a full loop nest execution. The reference code is a sequential execution of the program

program	reference	full profiling	inspector profiling
`ind`	1,495 *ms*	1,910 *ms*	1,986 *ms*
`llist`	3,100 *ms*	3,720 *ms*	3,700 *ms*
`rmm`	10,330 *ms*	20,900 *ms*	26,120 *ms*
`switch`	3,960 *ms*	5,030 *ms*	4,970 *ms*

Table 5.2: Evaluation of the two profiling methods execution times.

without any instrumentation. We can see on the table that the costs of both profiling methods are very high compared to the reference execution. It even reaches a slowdown of more than 2× with `rmm`. The inspector profiler shows no clear benefit over the full profiling method when a complete sequential execution is performed after the profiling.

Considering the small number of instrumented iterations, the gain which can be obtained with that method is negligible, especially as the inspector profiler construction may be extremely complex in some cases.

5.3.2 Profiling on a Sample

Three iterations per loop level are sufficient to build a linear function in the favorable cases. However it is convenient to instrument more iterations to handle complex situations.

Some instructions can be enclosed in tests which are not taken at least three times during the profiling step. In such case, the code has better to be profiled on a longer period, increasing the probability of taking the branch at least three times. If this is not the case despite this longer profiled run, one can assume that the test is never or rarely taken. The dependences induced by the references in that test can be ignored in later phases, but the system must verify whether the test is actually taken at runtime. This leads the system to actually perform control speculation.

The extra cost induced by this longer profiling period can avoid an even higher overhead if the parallel code is run and back-tracked later because of this non-linearity.

5.3.3 Chosen Solution

Considering the difficulty to build an inspector from some loop nests, the chosen method is to profile the whole body of the original loop nest on a few iterations. Profiling three iterations by default and continue for at most ten iterations can help the system to quickly detect non-linear accesses and to handle conditional branches. If some profiled memory references, scalar values, or while loops are enclosed in a test which is not taken at least three times during the profiling phase, then they are ignored in later steps. If that test is actually taken during the parallel execution, the system considers that a misprediction has occurred.

For all those profiled items, the result is either a failure: at least one of them is not linear and no parallelization can be performed, or a linear function is associated with each profiled value. Those linear functions are speculated to be the behavior of the program for the next iterations and the dependences are deduced from those functions.

5.4 Dependence Construction

In the usual static polyhedral compilation sequence, the dependences are first built from the program source code in order to generate a valid transformation later. Our system targets programs which are not statically analyzable, and for which the dependences cannot be determined at compile time. It uses instead the linear functions obtained during the profiling step to build the dependences. Those dependences are correct if the behavior observed during the profiling is representative of the rest of the execution. We speculate that this is true and perform the dependence analysis under this hypothesis. The following steps verify at runtime if this assumption is effectively correct.

We exploit the polyhedral model to represent the dependences as polyhedra [46]. This representation is convenient as it is a very compact way to represent dependences between affine memory references. For each pair of references, a polyhedron is built from the iteration domains of each reference in the pair, from an equality between their linear functions, and from precedence constraints between the source and the destination of the dependence. More details are provided in Chapter 2. Building those dependence polyhedra from the linear functions is a trivial task whose cost is negligible. It can even be accelerated by partially creating them offline, using the information statically available. The main difference with common offline dependence construction is that the only parameters used are those defining the bounds of the currently executed chunk, and a fake bound for the while loops [14]. Indeed, those polyhedra are created at runtime, when all the other parameter values are known. Thus, the dependence polyhedra are simpler and the corresponding issues can be solved more easily. The speculative dependence polyhedra are used at the next step to select a code transformation to apply on the loop nest.

5.5 Scheduling

At that point, the speculated dependences are known and restrict the set of valid transformations. A parallelizing transformation has to be chosen accordingly. The selected transformation ideally has to optimize the data locality while exposing parallelism. However, there exists some valid transformations leading to inefficient parallel executions. For this reason, the scheduling is generally a complex task which can only be efficiently achieved by a few advanced tools, such as the PLUTO compiler, extended with the framework presented in [14].

5.5.1 PLUTO

We have evaluated the execution time of PLUTO when parallelizing the sample programs. The results are presented in Table 5.3. In the first column, PLUTO is evaluated when no tiling is requested, while the second one contains its performance when parallelization and tiling are both requested. This execution time is measured using simplified versions of the programs where all the memory references are rewritten to be statically analyzable. We can see on the table that such an advanced transformation selection is time consuming. If we use it in our system, PLUTO may have to be run

program	PLUTO -notile	PLUTO -tile
`ind`	155 *ms*	230 *ms*
`llist`	150 *ms*	220 *ms*
`rmm`	180 *ms*	370 *ms*
`switch`	150 *ms*	180 *ms*

Table 5.3: Execution time of PLUTO on our sample programs.

multiple times in case of mispredictions, and even if code caches are used, the induced overhead would be hard to amortize.

Notice however that there is bias introduced in our measurements: some operations which are not required at runtime are actually performed here. For example, the code parsing and the dependence construction are performed by PLUTO whereas they are not required by our system. On the other hand, the considered programs are simple and contain only a few memory references, whereas the execution time of PLUTO exponentially grows as the programs becomes larger. Thus, although the cost of scheduling is slightly over-approximated in our experiments, it can still be much higher on more complex codes.

One could object that PLUTO is not intended to be used at runtime, which is true: as all the existing polyhedral compilers, it has been designed as an offline tool. A different conception could lead to better performance. For instance, we could imagine to statically generate some intermediate representation of the problem, simplified using the information available offline, and which could be easily parsed. However, the main issue would probably still be the exponential complexity of the core algorithms used to compute the new schedule. This leads us to the conclusion that lighter methods have to be used.

5.5.2 Offline Profiling and Scheduling

One possible lighter approach is to perform an offline profiling of the program which could provide a first approximation of the loop nest behavior. The addresses reached by memory accesses, value of speculated scalars, and the trip count of while loops can be profiled similarly to what is achieved by the online profiling. The non-linear loop nests can be detected and ignored. Finally, the loop nests accounting for an insignificant part of the execution can also be identified and left unparallelized.

The linear functions resulting from the offline profiling can be used to compute the loop nest dependences that occurred during that profiling run. A valid and efficient transformation can be deduced from those dependences, using PLUTO for instance (extended to correctly handle while loops). The principle is to assume that the loop nest behavior, observed during the offline profiling, is representative of what will happen at runtime. The generated transformation has then a high probability of being valid during the actual execution. Moreover, this transformation is expected to be efficient as it is computed by an advanced parallelizer. Notice here that a different memory allocation is probably performed at runtime, leading to different linear functions. Although the linear functions can differ from what has been observed during

the offline profiling, the dependences can remain identical, still allowing the schedule to be used in that case.

If different successive linear functions, characterizing different phases of the execution, can be associated to the memory references during the profiled execution, all of them can then be considered to generate several corresponding transformations. At runtime, one of these transformations can be selected according to the speculative linear functions. If the behavior of the program changes, a misprediction will occur followed by a short online profiling, which can lead to successively consider different transformations.

The offline profiling can be accelerated using sampling. The loop nest is then periodically profiled for a few iterations, reducing the overall instrumentation cost. The system can also benefit from previous executions of the program to better characterize the profiled accesses.

This offline method results in several transformations that are computed offline by advanced tools. Those transformations are efficient as they are computed by advanced parallelizers, add no runtime overhead as they are generated offline, and have a high probability of being valid during the actual loop nest execution.

5.5.3 Generic Schedules

It may be possible that the loop nest behavior observed during the offline profiling appears not to be representative of the actual loop nest behavior. In such case, the transformations statically generated may not be valid for the dependences observed at runtime. Thus, we have to provide some backup schedules which can be used in that case.

The solution we envisage is to provide other generic schedules, alongside those built during the offline profiling. Those new schedules are expected to allow the parallelization of common loop structures, but are not intended to reach the maximal performance. In that sense, they are generic schedules, providing a decent backup solution when the offline profiling result cannot be used.

The main issue with this method is that the space defining the valid transformations is often huge, and many transformations in that space are clearly inefficient [103]. Thus, one cannot randomly pick a transformation and hope for it to be efficient. However, the space can be restricted to vital loop transformations. The transformations often leading to inefficient parallel codes can be ignored, even if they are sometimes required to parallelize some other applications. We restrict the iteration space in the following way:

- The schedules can be generated considering the statically analyzable dependences of the targeted loop nest, ensuring that no obviously incorrect schedule is generated. This already restricts the transformations to a limited set when a part of the loop nest is statically analyzable.

- We ignore some advanced transformations often not required to reach decent performance. For example loop splitting, reversal, fission, and fusion can be ignored.

- On the other hand, tiling usually provides large performance improvement on programs, and is then systematically applied. A single tiling level using default tile sizes is generally sufficient to enhance the programs performance.

- We also consider a single transformation for every statement in the loop nest.

A few transformations can be chosen in this limited space. For example the identity schedule, the loop interchanges bringing any loop at the outermost level, and a diagonal schedule can be selected.

Again, nothing ensures that the parallel codes generated from those schedules are efficient. For instance, the data locality can be deteriorated by the transformation (despite the tiling limits that). The data dependences discovered at runtime can also prohibit all those transformations, even if the loop nest is actually parallelizable. However, this strategy ends up with a few transformations which are expected to have a reasonable probability of being valid with regard to the speculated dependences, and to provide decent speedup.

When a set of transformations is statically generated, the scheduling mainly consists in detecting which of the available transformations are valid according to the dependences speculated at runtime. This is done in a dependence test which verifies if each schedule is correct according to those dependences.

5.5.4 Dependence Testing

Two problems have to be solved in the dependence test: considering a schedule, statically generated, and a set of speculated dependences, we have to determine if the schedule is valid, and if at least one loop in the transformed code can be parallelized. With the generic schedules for instance, nothing ensures that a loop level is parallel even if the new execution order is valid with regard to the speculative dependences.

The validity problem can be precisely solved by several emptiness tests on the dependence polyhedra augmented by some precedence constraints (see Chapter 2). Such emptiness tests can be non-trivial depending on the polyhedra. To evaluate the cost of those emptiness tests, we have generated several thousands of random polyhedra with realistic coefficients, and used ISL to check for their emptiness. The check is performed in parallel: multiple polyhedra are evaluated simultaneously. In average, we are able to evaluate 1,950 polyhedra per seconds on our reference computer. Considering this performance, the dependence testing can quickly become a limitation on large problems, especially as the number of dependence polyhedra to test exponentially increases with the number of memory references and with the number of loops enclosing them. Alternatively, other exact dependence tests such as the Omega test [108] can also be considered, however, as many exact dependence analysis techniques, it is also known to be time-consuming [97].

The validity test can also be performed by less precise methods such as dependence direction or dependence distance vectors [143]. With those simple dependence representations, the validity test can be performed by a few comparisons for every couple of accesses. The main issue with those representations is their low precision: they can lead to consider a schedule as invalid whereas it is in fact valid.

Once the schedule validity is proven, we have to determine if a loop level can be parallelized in the resulting transformed loop nest. This can also be performed directly and precisely on dependence polyhedra, using emptiness tests (see Chapter 2 for more details). As the cost of this exact dependence testing may cause severe overheads, one could envisage to use approximate methods, which can conservatively answer that the parallelization is not possible, although it actually is. For example the GCD test [90], could be considered despite it might be too approximate. Interestingly, Kim et al. present in [70] a possible dynamic use of the GCD test, although they do not extend its precision. More precise approximate tests can be used such as the Banerjee test [9], or its extension, the I Test [73]. It may also be interesting to consider tests specifically designed for runtime parallelization. For instance, the proposals of Rus et al. [122, 121] could be adapted to our system, allowing it to perform an hybrid static and dynamic test.

We have evaluated the I Test to have a better idea of the performance of such approximate methods. We have generated several random couples of access functions with reasonable coefficients and evaluated them using the I Test implementation provided in the PLATO library [98]. We have measured that 560,000 tests can be performed per second on our reference computer. This performance illustrates how such approximate dependence tests can be interesting alternatives to exact dependence tests. However, note that such tests are heavily approximating the problem, especially as they can often only handle simple loop structures. As an example, the loop bounds have to be constants to use the I Test, which is obviously not the case for many loop nests, especially after having been transformed.

As much as possible, exact dependence analysis should be preferred for its precision. However, when the loop nest contains many memory references, heuristics could be used to limit the induced overhead, at the cost of a reduced prediction. A gradual analysis could also be considered, starting with heuristics to eliminate simple cases, and using exact analysis when the heuristics could not prove the schedule validity or the parallelism.

5.5.5 Chosen Solution

Considering the presented alternatives, we consider that the best solution is to profile the whole application once. From that profiling, linear functions can be built to precisely characterize the memory references. They are used by an offline parallelization tool, and the generated transformations, alongside other generic transformations, are considered. If several linear functions can be associated to a memory reference, different schedules are generated by the automatic parallelizer, corresponding to the different possible cases. This number can be bounded to a given threshold in order to avoid generating too many schedules.

At runtime, the schedule selection is performed by exact dependence analysis on the speculative dependences. If there are many memory references, the problem is first approximated using inexact dependence tests such as the dependence distance vectors and the I Test.

5.6 Code Generation

Once the schedule is chosen, the corresponding parallel code has to be generated. Different solutions are possible to generate this code, ranging from fully dynamic code generation to statically generated code versions. Each method has different costs and benefits which we evaluate in this section before deciding of an appropriate strategy.

5.6.1 Runtime Compilation

The number of considered parallel versions can be large, either because dynamic scheduling is used, or because a large number of schedules are embedded in the application. In that case, the size of the binary executable can grow unreasonably if the code of every considered schedule is linked in the executable. Runtime code generation is one solution that can avoid this.

Runtime code generation can be divided in two main parts: first the code generation, and second the code compilation. The former consists in converting the polyhedra representing the transformed iteration domains into loop nests. The latter part compiles these loop nests into a binary form, executable by the processor. For our example programs, the first step can be performed in a few tenth of milliseconds, using irCLooG for instance. This tool is an extension of CLooG which supports data-dependent control flow, including while loops [14]. This execution time is relatively short but the exponential nature of the algorithms used in such code generators can quickly lead to execution times in order of seconds or minutes, depending on complexity of the polyhedra.

The output of irCLooG can be any high-level representation as a C or a Fortran program. It can also be a medium-level representation such as the LLVM intermediate representation. However, directly generating an efficient sequence of low-level instructions is a complex task and it does not seem reasonable to adapt irCLooG to directly generate it. For example register allocation has to be performed and is not trivial, similarly for instruction selection, code alignment, vectorization, ... One can notice that the efficient binary code generation problem has already been addressed in existing compilers, so the best tools able to perform this task are probably those compilers. Then, we have measured their execution times on parallel versions of the example programs to evaluate a possible runtime use where irCLooG could generate high-level code which is later compiled.

We have compared three common C compilers: GCC 4.5, ICC 12.1, and LLVM 2.8 in a fair context: the only used flags are `-O,` to specify an optimization level, and `-fopenmp,` to activate OpenMP. Notice that the LLVM front-end currently does not support OpenMP, thus, we have used instead LLVM-GCC, which is a modified version of GCC made of the GCC front-end plus the optimizations passes and back-ends of LLVM. We measured two different usages of LLVM-GCC: first to generate a binary program, and second to generate LLVM bytecode, later compiled into a binary program using LLC. The latter case is particularly interesting as LLC uses LLVM bytecode as input. Thus, its overhead is limited to some last-minute optimizations and code generation. The bytecode is already optimized by LLVM and can be parsed more easily than any high-level language. Thus, LLC is a good choice for runtime compilation.

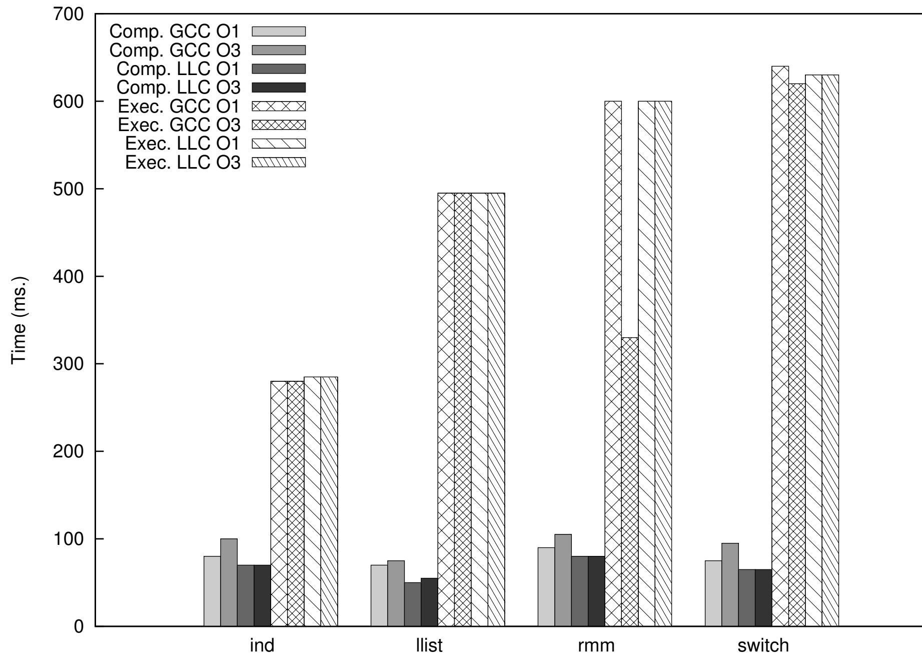

Figure 5.4: Compared compilation and execution times using GCC and LLC at the `O1` and `O3` optimization levels.

program	GCC			ICC			LLVM-GCC			LLC		
	`O1`	`O2`	`O3`	`O1`	`O2`	`O3`	`O1`	`O2`	`O3`	`O1`	`O2`	`O3`
`ind`	80	90	100	150	190	200	115	100	100	70	70	70
`llist`	75	75	75	160	170	170	90	90	90	50	55	55
`rmm`	90	100	105	160	250	250	120	110	110	80	80	80
`switch`	85	90	95	160	180	190	95	95	100	65	65	65

Table 5.4: Compilation time (in milliseconds) of the sample programs using different compilers and optimization levels.

We present the measured compilation times in Table 5.4, and the corresponding execution times in Table 5.5. In the tables, the parallel version, generated by PLUTO, of each rewritten program is compiled using every compiler at three different optimization levels. LLVM-GCC refers to a complete compilation by LLVM, i.e. from the source code to a binary program, while LLC refers to the transformation of the LLVM bytecode into executable code. A graphical representation is provided in Figure 5.4 to visually compare the execution and compilation times obtained with GCC and LLC.

With no surprise, we can see that LLC is the fastest to compile the loop nests. Interestingly, the performance of the generated codes are often comparable to the other compilers, illustrating the efficiency of LLC despite its reduced execution time. It can then be considered as the best alternative. However, the absolute compilation times remain too high: an overhead of several tenth of milliseconds just to compile the code is not tolerable considering the execution times of the programs. This is exacerbated as several different schedules can be required during the parallel execution, if the program

program	GCC			ICC			LLVM-GCC			LLC		
	O1	O2	O3	O1	O2	O3	O1	O2	O3	O1	O2	O3
`ind`	280	355	280	800	770	780	280	300	300	285	285	285
`llist`	495	495	495	800	700	650	495	495	495	495	495	495
`rmm`	600	330	330	520	450	500	600	600	600	600	600	600
`switch`	640	630	620	1,450	1,500	1,480	620	645	630	630	630	630

Table 5.5: Execution time (in milliseconds) of the sample programs compiled with different compilers and optimization levels.

behavior evolves during the execution.

This solution is clearly the most robust, but it also exhibits a large overhead. Then, other solutions, with lower execution times, have to be investigated to limit the overall runtime overhead.

5.6.2 Hybrid Code Generation

Hybrid code generation is an alternative to runtime code generation. In this strategy, a parallel code pattern is generated offline, and filled at runtime according to the chosen schedule. The code pattern is made of the loops, the tests, and the computations. However, the loop bounds and memory access functions are left unspecified in the pattern. It is a generic version of the loop nest and is meant to represent the parallel loop nest code for several different transformations. However, the patterns obviously restrict the set of transformations which can be applied. For example different loop splittings, fissions, or fusions cannot be performed in a single pattern.

Different implementations of this strategy can be considered, and we have measured the impact of three distinct approaches to implement the hybrid code generation:

- Using function calls: in that implementation, every loop bound and access function is performed by a distinct function. When the schedule is selected, each function is generated as a short binary sequence computing the correct loop bound or memory address.

- Through parametrized expressions: loop bounds and access functions are replaced by expressions depending on all the loop indices with unspecified coefficients. The coefficients are filled to compute the correct functions according to the schedule selected at runtime.

- Patching the executable code is the last studied implementation. The access functions and the loop bounds are not generated in the pattern, but some padding is inserted. This padding is then replaced at runtime by instructions computing the corresponding expression directly in the loop nest binary code.

We have evaluated those three methods on the benchmark programs. The results are presented in Table 5.6 and show the performance of the programs when the different approaches have been tried. The reference version is the statically generated parallel version with the linear functions exposed in the source code. To simulate code

program	reference	function call	parametrized expression	code patching
`ind`	280 *ms*	302 *ms*	295 *ms*	290 *ms*
`llist`	495 *ms*	515 *ms*	495 *ms*	510 *ms*
`rmm`	330 *ms*	1,955 *ms*	815 *ms*	710 *ms*
`switch`	620 *ms*	690 *ms*	680 *ms*	670 *ms*

Table 5.6: Execution times of the parallel programs for different hybrid code generation strategies.

patching, some padding is inserted in the parallel code, disturbing the code generation and increasing the code size. Considering those results, it is clear that the function call approach is not efficient. Parametrized expressions are easier to implement and reach performance comparable to the code patching, they are then the privileged approach among those we have tested.

Let us now consider the pattern instantiation performed at runtime. It exists to our knowledge no technique able to generate a loop nest scanning a polyhedron, where the resulting nest is constrained by a given pattern. However, one can use existing techniques to generate the transformed loop nest, and update the pattern according to the generated code, aborting if the pattern is not generic enough to represent the transformed loop nest. The main issue here is again the exponential complexity of the existing code generation algorithms, which can lead to major overheads in complex cases.

The last issue is the offline pattern generation. To achieve it, one can simply consider the original sequential loop nest, and overwrite every memory access and loop bound by a generic one. The complexity of the loop bound expressions can be arbitrary, ranging from a single linear expression to a complex sequence of min, max, integer divisions and linear expressions. Depending on the complexity of the pattern, different transformations can be applied on the nest.

This method limits the overhead of code generation to a few writes after having computed the correct access functions and loop bounds. However, this latter computation is not trivial and the pattern can strongly restrict the set of possible schedules if it is not generic enough. Moreover, an extra cost has to be paid during the whole parallel execution: the pattern is a generic code whose performance is worse than that of a specialized parallel code, as shown in Table 5.6. Those drawbacks push us to explore lighter approaches.

5.6.3 Static Code Generation

If we assume that the schedules are selected at compile time, as described in Section 5.5, we can statically generate the corresponding transformed loop nests. Once a transformation is chosen, the existing code generation tools such as irCLooG can statically generate the corresponding parallel code even when while loops or data-dependent conditional instructions are present. The complex memory references actually cause no difficulty for the code generation. They are transparently copied from the sequen-

tial source code to the parallel loop nest. The scheduling phase takes care of the dependences they provoke, and guarantees that the transformed loop nest with those complex memory references is correct. As for classical loop nest transformations, when the loop indices are used in a statement, the initial loop indices are expressed as a linear combination of the transformed loop indices in the parallel loop nest.

The main drawback of this code generation method is that many different parallel versions have to be embedded in the application (one per possible schedule of each loop). This is not always a severe issue, especially for general-purpose computers with weak restrictions on memory, however, it can be critical on embedded platforms for instance. This method guarantees that the code generation is feasible, which is not always the case with hybrid code generation, and induce no particular runtime performance overhead.

5.6.4 Chosen Solution

Considering the previous choices, the static code generation is the preferred solution for general-purpose computers. A threshold can be used to decide between static code generation and hybrid code generation. For example, above a given number of transformations, hybrid code generation can be considered to avoid a large binary size increase.

A fully runtime code generation seems to induce a large overhead, excepted for hot loop nests where the compilation cost might be amortized by numerous and time-consuming executions. If very fast polyhedral code generators and compilers are developed in the future, this approach should be re-considered as it does not restrict the transformations that can be applied on the nest, and does not increase the program size. Of course, it makes even more sense if a dynamic scheduling can be performed before code generation, leading to a fully dynamic polyhedral system.

5.7 Speculation Verification

Once the parallel codes have been generated, they can be executed but the predicted scalar values, memory addresses, and loop trip counts have to be checked against their speculative linear functions to ensure that the whole speculative process is correct. In Softspec [24], the addresses reached by the memory references are compared to the speculative linear functions at each parallel iteration, using a simple equality. If the addresses do not match the speculation, the semantics of the parallel code is not guaranteed, and the speculative execution is then canceled. In our case, we have to perform a similar operation, however, a transformation has been applied on the loop nest. Thus, the parallel schedule is different from the sequential one and it is not obvious to determine whether the verification can still be performed in parallel.

5.7.1 Parallel Speculation Verification

We prove in this subsection that we can actually perform the speculation verification in parallel even after a code transformation. For that purpose, one has to notice that

the only way to modify the addresses reached by a reference is to write something in a memory slot which is read later to compute the address of the reference. For instance, consider the indirect memory reference A[B[i]], the only way to change the element accessed by A is to write in B. This fact is at the base of the proof developed below.

In this section, we distinguish between a memory reference which is an instruction accessing the memory, possibly executed many times, and a memory access which is an instance of a memory reference, i.e. a specific execution of a memory reference.

We consider two different execution orders of the program: the original sequential execution order and the transformed parallel one. It is possible to verify if the speculation is correct in both execution orders. In the sequential version, the verification is necessarily exact as this version serves as a reference. In this execution order, if a memory access reaches an address which was not speculated, then, there is no reason why the verification would not notice it. This is however not guaranteed for the parallel execution order.

We impose two constraints related to those execution orders. First, the transformation used to build the parallel execution order has to be valid with regard to the speculative dependences. Second, in both execution orders, the same operations are performed on data. If a given operation is performed in one execution order, then it has to be also performed in the other execution order, possibly at a different iteration.

Let a and b be two memory accesses. We note $a \prec_{seq} b$ the case when a is performed before b in the sequential loop nest execution order. Similarly, we note $a \prec_{par} b$ the case when a is executed before b in the parallel execution order. The relation induced by the two operators is not a total order as it may be possible that two memory references are executed in any order, especially in the parallel execution order.

We consider that every memory access a is always in one of the two following states:

- $\diagup$ if a reaches an address corresponding to the speculated linear function;
- $\sim$ if the address reached by a does not correspond to the speculative linear function.

We note S^a_{seq} the state of a if the program is executed in the sequential order, and S^a_{par} the state of a during the parallel execution. Then, we have $S^a_{seq}, S^a_{par} \in \{\diagup, \sim\}$.

Consider the case when a misprediction is not detected in the transformed parallel execution order despite it would occur if the loop nest would have been executed in the original sequential order. That case is an issue as the parallel execution would not fail despite the program semantics would differ from the sequential semantics. We derive a correctness definition from this remark. A speculation verification is said correct if a misprediction is detected in parallel if one occurs when the loop nest is executed in the original sequential order. This property can be expressed by the following lemma.

Lemma. *Given a memory access a such as $S^a_{seq} = \sim$, it exists a memory access b such that $S^b_{par} = \sim$.*

Proof. Let a be a memory access such as $S^a_{seq} = \sim$. Let ar be another memory access which reads a value used to compute the address reached by a. Let x be a memory access leading to $S^a_{par} = \diagup$ (despite $S^a_{seq} = \sim$). To influence the state of a, the access x writes to the location read by ar. The accesses ar and x are then data-dependent.

We decompose our proof in two main points.

First, there are two situations where x can influence a such that $S^a_{par} = \diagup$: either $x \prec_{seq} ar$ and $ar \prec_{par} x$, or $ar \prec_{seq} x$ and $x \prec_{par} ar$.

In the first case, x overwrites the value read by ar in the sequential execution, changing the state of a from correctly predicted to mispredicted. If the access x is scheduled after ar in the parallel execution, it cannot modify the state of a when the loop nest is executed in the parallel order. This results in $S^a_{seq} = \sim$, while $S^a_{par} = \diagup$.

In the second case, x overwrites the value read by ar after its execution in the sequential order. Then, during the sequential execution, x has no impact on the state of a. However, if x is run before ar in the parallel order and changes the state of a, it leads to $S^a_{seq} = \sim$ and $S^a_{par} = \diagup$.

Second, it is important to notice that, if $S^{ar}_{par} = \diagup$ and $S^x_{par} = \diagup$, then the dependence between x and ar is known and is taken into consideration by the transformation, leading ar and x to be executed in the same order during both the sequential and the parallel execution. This contradicts the first point. Thus, either $S^{ar}_{par} = \sim$, or $S^x_{par} = \sim$.

In conclusion, three possibilities can occur: (1) It exists no such access x and $S^a_{par} = \sim$. (2) There is an access ar which participate in the address construction of a, and which is such that $S^{ar}_{par} = \sim$. (3) The memory access x, changing the state of a, is such that $S^x_{par} = \sim$.

In any case, there is an access b such that $S^b_{par} = \sim$ when $S^a_{seq} = \sim$. □

This lemma is proven true, meaning that the verification performed in parallel is correct even if the sequential execution order is different from the parallel execution order. If a misprediction occurs when executing the code in the original sequential order, then a misprediction also occurs in parallel, possibly when checking a different memory access.

5.7.2 Verification Implementation

To perform this verification during the parallel execution of the loop nest, some verification code is added to the parallel loop nest in order to ensure that the speculation is correct. For every speculated memory reference, the actually accessed addresses are tested against the speculative linear function, before performing the access. The speculative functions built by the online profiling phase on the sequential loop nest are transformed with the same transformation that has been applied to the loop nest.

The speculated scalars are initialized at the beginning of every iteration according to their speculative function. A test is added at the end of the iteration to ensure that their actual values match their predicted values for the next iteration in the sequential execution order. Determining the value of the loop indices in the next sequential iteration is not an issue, even when the transformation matrix is not invertible. Indeed, to maintain correct array references, the code generation algorithm is forced to build a relation between the loop indices in the transformed space and those in the original iteration space. This relation can be used to determine what is the value of the transformed loop indices for the next iteration in the sequential order, and then to compute the scalar value for the next sequential iteration.

```
while (p != NULL) {
   ...
   p = p->next;
}
```

```
while (!end) {
   forall (i = CHUNK_START; i < CHUNK_END; i++) {
      p = base1 + i * scale1;
      if (p == NULL) { end = 1; break; }
      ...
      assert &(p->next) == base2 + i * scale2;
      p = p->next;
assert p == base1 + (i + 1) * scale1;
   }
   CHUNK_START = CHUNK_END;
   CHUNK_END = CHUNK_END + CHUNK_SIZE;
}
```

Figure 5.5: Sample code (left) and its speculative parallel counterpart (right).

Notice first that the operations performed on the predicted scalars do not depend on the execution order of the loop iterations. Moreover, if the memory accesses are correctly predicted, the values used by those operations during the parallel execution are identical to those used in the sequential execution. Thus, the same operations are performed on the same data in both execution orders. Then, the described verification for the speculated scalar values is necessarily correct, i.e. a misprediction that would occur in the original sequential execution order is detected when executing the code in parallel.

The while loops are also verified when their trip counts are speculative. A counter incrementation is added in the sequential while loops, before transforming the nest. When the transformation is applied on the loop nest, this counter increment is considered as a regular statement which can be handled by reduction. In the transformed nest, when the while loop exit condition is met, it contains the number of while loop iterations executed. It is then compared to the speculation to ensure that this speculation is correct.

The code used for the verification is illustrated with an example in Figure 5.5. In that figure the value of the scalar `p` is speculated: in the parallel nest, it is initialized at the beginning of the iteration and its value is tested at the end of the iteration. The memory access performed at the address defined by `p->next` is also compared to the speculative linear function. The coefficients `base1`, `base2`, `scale1`, and `scale2` are determined during the online profiling which associates a speculative linear function to every predicted scalar value, memory reference, and loop trip count. In our example, the outermost loop is a while loop, which is chunked to allow its parallelization by conventional techniques. In our case, the exit condition is never true as the value of `p` is speculated. However, a misprediction will happen for this scalar at the last iteration, leading to execute the last chunk in sequential before stopping the execution.

5.7.3 Test Implementation

We have considered two implementations to check if the speculative value matches the actual one. The first one compares both speculative and actual values using a test at

program	reference	conditional instruction	boolean flag
`ind`	280 *ms*	365 *ms*	330 *ms*
`llist`	495 *ms*	505 *ms*	510 *ms*
`rmm`	330 *ms*	780 *ms*	1,305 *ms*
`switch`	620 *ms*	645 *ms*	660 *ms*

Table 5.7: Execution time of the different verification strategies.

each iteration. The second one updates a boolean flag, testing its value only once after several iterations, typically at the end of a chunk. This later strategy limits the number of tests performed and is then expected to be faster, at the price of an increased overhead in case of mispredictions. Indeed, as the misprediction is detected several iterations later, when the actual test is performed, more parallel iterations are executed before being canceled. We evaluate both strategies on our reference computer, augmenting the parallel code generated by PLUTO with the validity checks and measuring the resulting execution times. They are presented in Table 5.7, where the reference version is the original parallel code without any validity check. Surprisingly, the second strategy does not provide any clear benefit over the naive one. Most of the time, the compiler, GCC in our case, performs better at optimizing the tests than the boolean operations. This can also be observed with LLVM and ICC. Both implementations lead to overheads, especially for `rmm` whose reference version is highly optimized, but the speculation verification is mandatory and its overhead remains limited compared to the sequential execution times.

5.7.4 Chosen Solution

As we can ensure that the parallel verification is valid, we use this technique to verify if the speculation is indeed correct. Considering the different test implementations results, we use simple conditional instructions at every iteration. This approach seems to be as efficient as the one based on the boolean flag, but is able to engage the back-tracking as soon as a misprediction is detected.

5.8 Commit and Rollback

A remaining challenge is to build a transactional system able to revert the execution in case of misprediction. Although some existing hardware mechanisms are able to perform this operation, they remain very uncommon and software solutions should be privileged for the sake of portability. In this section, we show that only the currently executed chunk has to be canceled and re-executed sequentially to maintain the correct program semantics. We also present two implementations of a simple software system able to perform the required back-tracking. The first one is based on operating system processes, while the second is a more common approach exploiting the benefits from the polyhedral model.

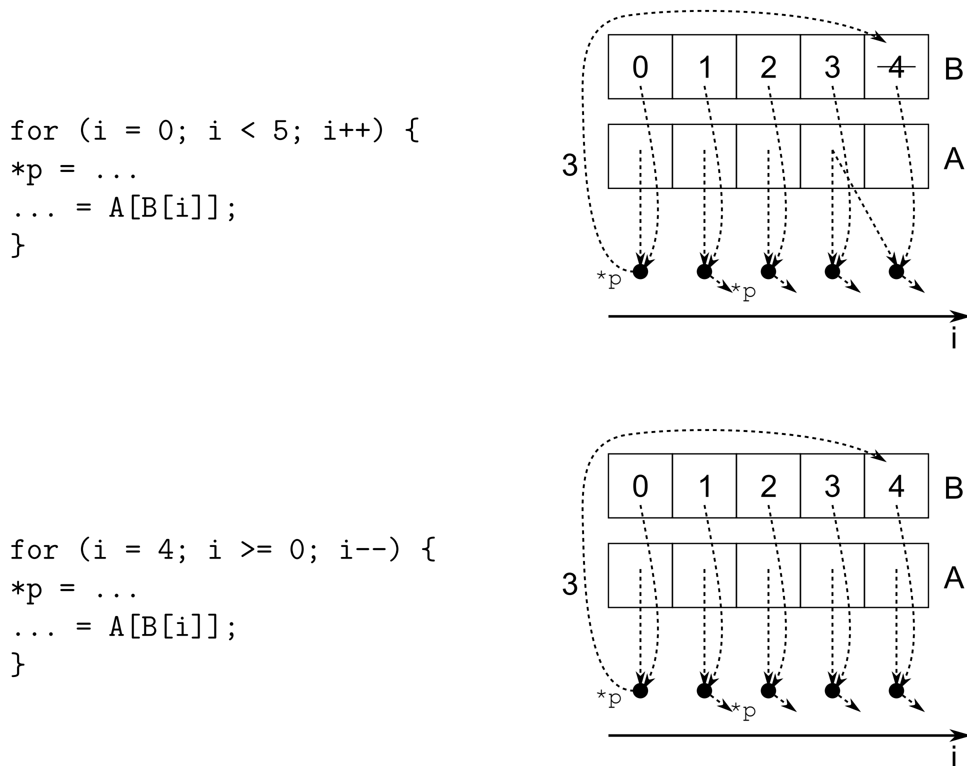

Figure 5.6: Sample loop (top), its transformed counterpart (bottom), and the corresponding memory accesses performed (right).

5.8.1 Transactions and Chunks

One can notice that the parallel validation can fail when testing an access which is not related to the first access mispredicted in the sequential order. This is illustrated in Figure 5.6, where an original loop is transformed by a loop reversal. The memory accesses performed are represented on the right in the figure. In the initial loop, on the top, the very first pointer access writes 3 in B[4], causing the last memory access to be mispredicted. When executing in the reversed order, the initial state of B[4] leads to a correctly predicted access for `A[B[i]]` and the verification has to wait for the last iteration, where the non-linear pointer access happen, before failing. When the faulty pointer access is detected, not only the last iteration has to be replayed, but also the first one. In the worst case, the whole loop has to be re-executed when such a misprediction is detected.

More formally, consider memory access a whose actual address accessed differs from what has been speculated. Its speculated address was $@_{spec}$ while the correct address for this access is actually $@_{cor}$. We call A_{spec} the set of dependent memory accesses which use the memory element at address $@_{spec}$. Similarly, A_{cor} is the set of dependent memory accesses which use the memory element at address $@_{cor}$. If the parallel schedule is valid with regard to the speculated dependences, it ensures that a and the memory accesses in A_{spec} are executed in the same relative order than during original sequential execution. However, as a misprediction occurs, the access a has to be correctly ordered with the accesses in A_{cor}. Indeed, the accesses in A_{cor} are in a dependence relation with

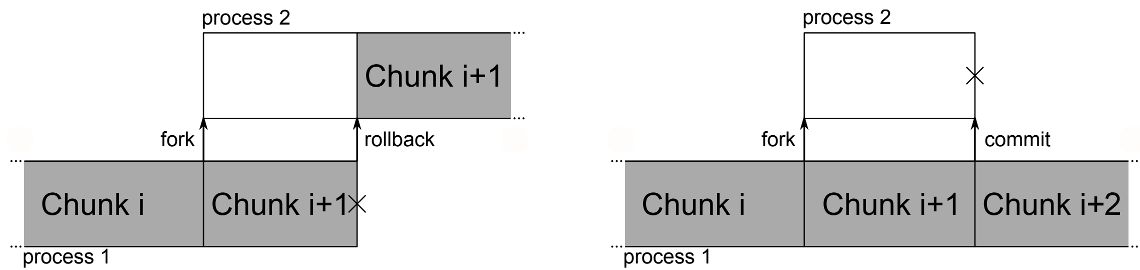

Figure 5.7: Execution of chunk i+1 cancelled (left) or commited (right).

a but has not been taken into account by the speculation. To guarantee the sequential semantics, the transactional system has to ensure that a and the accesses in A_{cor} are executed in the same relative order as during the sequential execution.

We recall to the reader that the chunking system is performed on the original outermost loop and that the chunks are executed in the sequential order, even if the iterations inside the chunks may be executed in a different order. This chunking strategy has a very interesting property: the iterations in the chunks preceding the current chunk necessarily precede the iterations of the current chunk during the sequential execution. Similarly, the iterations in the chunks executed after the current chunk are necessarily executed after those of the current chunk during the sequential execution. The sequential order is then imposed for two iterations in different chunks. Then, if the transactional system re-executes the current chunk using the sequential execution order, it can ensure that the relative order of an access in the current chunk with any other access is in fact equivalent to the sequential execution order. It means that, when a misprediction is detected, it is sufficient to re-execute the current chunk in the sequential order to ensure the correct ordering of the accesses in A_{cor} with a, and then the correct program semantics.

5.8.2 `fork`-based Transactional System

A simple mechanism can be implemented using the standard operating system facilities to perform the required back-tracking. Among the hundreds of tools offered by operating systems, are the processes. They are of particular interest because of the copy-on-write strategy used when duplicating processes in modern operating systems. When a process is being duplicated, typically using `fork`, the process and its copy use different memory spaces that have to be initialized to the same state. A naive approach is to copy the process memory when the process is duplicated. With the copy-on-write strategy, both processes initially share the same memory space, and before the first write to a shared memory page, the page is copied to ensure that the write is local to the process which performed it. This method is very efficient as only rewritten pages are duplicated, limiting the amount of memory which has to be copied.

We can take advantage of this mechanism to build a simple transactional system based on processes. In that implementation, the program is cloned using `fork` at the beginning of a every chunk. The new process simply waits for its parent termination. In the same time, the parent process executes the chunk and every modified memory slot is

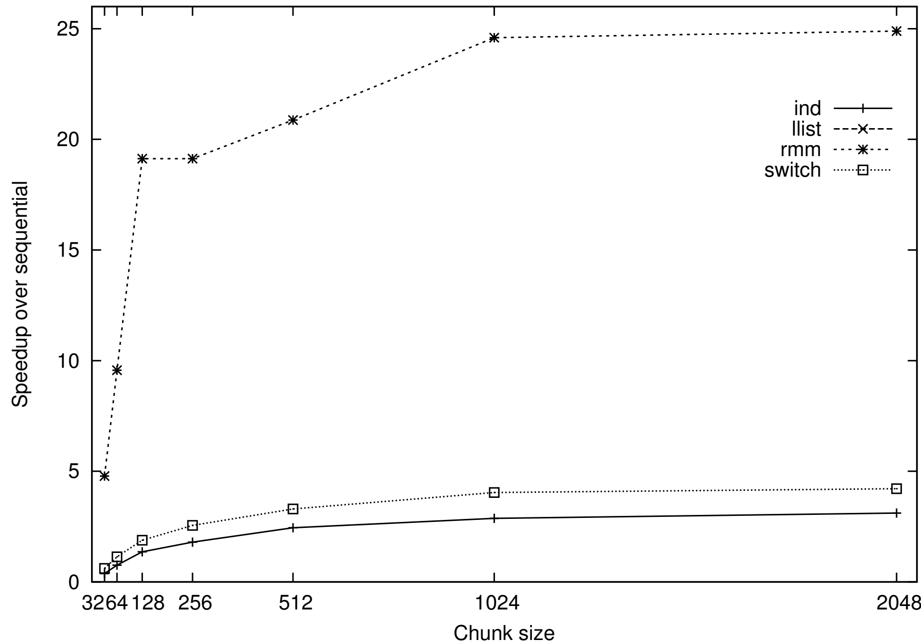

Figure 5.8: Speedup over the sequential version when using a `fork`-based transactional system.

automatically duplicated, thanks to the copy-on-write mechanism. If the speculation fails, the main process simply exits, awakening its son which can start again from the beginning of the chunk using another schedule. If the speculation succeeds, the son is simply terminated and the original process can continue. This is illustrated in Figure 5.7 with the execution of the chunk i+1 on process 1 which is either back-tracked or validated.

The main issue with that implementation is its time overhead. We have evaluated it on our reference computer and found that only a few processes can be created during the program execution while maintaining decent performance. This is directly related to the chunk size as it impacts the number of chunks required to execute the full iteration domain, and then the number of processes being created. We present our measurements in Figure 5.8 where the speedup of each program using the transactional system is related to the chunk size. The speedups are referring to the sequential execution. During that experiment, the process is duplicated at the beginning of each chunk and its copy is terminated at the end of the chunk, simulating a commit after every chunk. By comparing this figure with Figure 5.3, one can notice that the minimal chunk size required to perform an efficient execution is larger than without any transactional mechanism. The most important point is in fact the missing data for the `llist` program. Those data are missing because our operating system was not able to handle the process management required for that specific application. Indeed, with the tested chunk sizes, several thousands of processes are successively created and destroyed in a few milliseconds, provoking some malfunctions. The only way of using this implementation with `llist` is actually to use huge chunks, larger than 500,000 iterations. This is specific to `llist` because it is the only loop nest with such a large

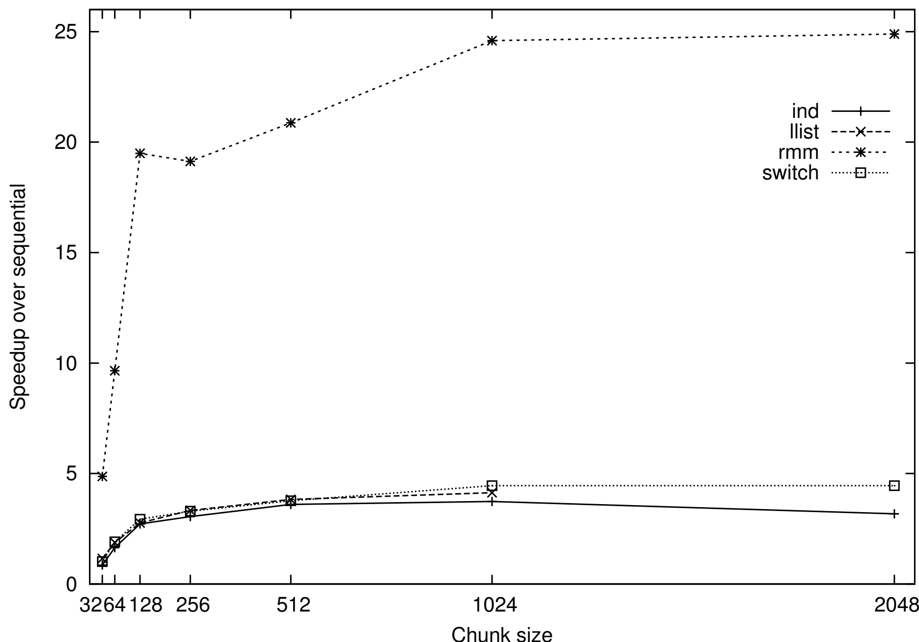

Figure 5.9: Speedup over the sequential version when using a `memcpy`-based transactional system.

number of outermost loop iterations.

5.8.3 `memcpy`-based Transactional System

We propose another implementation exploiting the polyhedral model. If the behavior of the loop nest is correctly predicted, the addresses reached by every memory reference and the loop bounds can be expressed as linear functions of the enclosing loop indices, including the while loops. In that context, the memory space reached by each memory reference can be exactly defined. Thus, we can precisely determine the address range modified by every write reference for the current chunk. This knowledge can be exploited to perform a backup of the modified memory elements before executing the chunk. In case of misprediction, this backup is restored and the chunk can be re-executed, otherwise, it is just overwritten by the backup of the next chunk.

Tools such as ISL can represent the memory space reached by a reference, and compute the minimal and maximal address reached depending on the chunk bounds. In case of a strided access, the impacted memory space is over-approximated but is contiguous, avoiding to perform expensive strided copies when saving the memory. The cost of this address space computation remains small as it needs to be performed only once when new speculative linear functions are profiled. The result of this method is a memory copy per memory write, saving the modified memory elements before executing the chunk, and the corresponding restoration copy in case of misprediction. It is typically implemented using the optimized `memcpy` function call. Notice that, in case of misprediction, the incorrect memory access has not to be performed as it will reach a memory slot which may not have been saved.

We have evaluated this method by copying the modified memory before running each chunk in parallel. The results are presented in Figure 5.9. We can see that, with that method, the program performs well even when small chunks are used. Moreover, `llist` does not cause any trouble, illustrating how several thousands of copies can be performed while still reaching good performance. A data is missing for `llist` with chunks of 2048 iterations. This is due to the high memory use of the original program: nearly all the memory is used by the original application and not enough space is left to backup the data with such large chunks.

5.8.4 Interrupting the Threads

In case of misprediction, the thread which detects the misprediction has to stop all the other threads as soon as possible before restoring the memory. Different designs can be considered to interrupt the other threads. First, the other threads can simply be killed. This is fast as the threads are immediately stopped, however, it requires to destroy and create new threads at every misprediction. This thread destruction and creation is very costly as many complex operations have to be performed by the operating system every time a thread is created or destroyed. Second, a signal can be used to notify to a thread that it has to stop. This approach allows one to reuse the threads in case of misprediction, but the signal handling procedure is then extremely complex to implement. Indeed, when a thread receives a signal, it has to enter in a waiting state until a new task is assigned to it, and in the mean time, it has to exit the signal handler in order to resume a normal execution. Executing simultaneously both operations requires a very complex implementation whereas simpler solutions exist. The third approach assigns a flag to every thread. Those flags are set by the thread that detects a misprediction. Each threads frequently checks for its flag and stops whenever it is set. The main overhead of the approach is an additional test, performed typically at each iteration. This overhead can be reduced by checking less frequently the flag value, for example only at every iteration of the outer loops. As each thread has its own flag, they should be located on different cache lines, avoiding unnecessary cache conflicts.

5.8.5 Rollback Strategies

Once the threads are stopped, the current chunk is canceled, and the memory is restored in its state preceding the chunk execution. Then, the mispredicted chunk has to be re-executed using the sequential schedule. One could envisage to take into consideration the new dependence caused by the mispredicted access. Considering this dependence, one could try to use any valid parallel schedule. This method leads to very specific issues and is not trivially usable. It is left for future research.

Once the sequential re-execution has been performed, the behavior of the application may have durably changed, for example if the program has entered a new phase. Then, to distinguish between a local misprediction and a long-term phase change, the program can be profiled again to determine if new linear functions can be associated with the memory references. In that case, a new parallel schedule can be used to

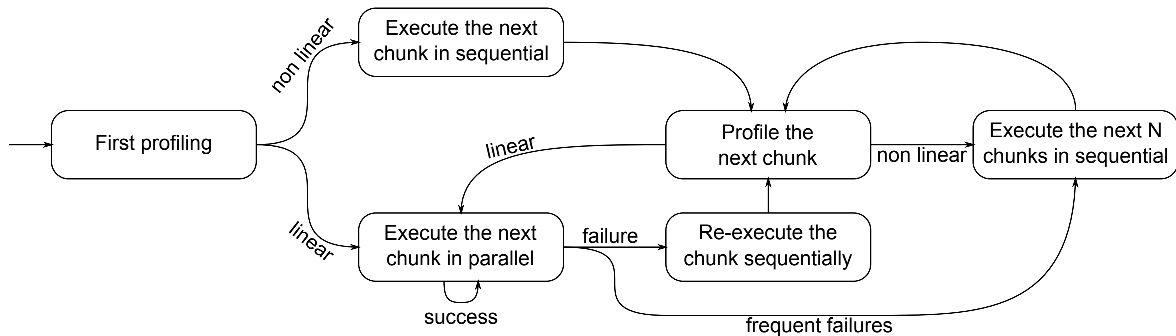

Figure 5.10: Runtime strategy.

execute the next chunks.

The number of rollbacks has to be minimized to limit the overhead of multiple failed parallelizations. Thus, in case of multiple successive mispredicted chunks, the system can execute several sequential chunks (or a larger sequential chunk) before trying again to profile and parallelize the loop nest. This is based on the observation that the program may have unpredictable phases, alternating with predictable ones.

5.8.6 Chosen Solution

We present in Figure 5.10 a suggested runtime strategy. It is designed to distinguish between two different kinds of mispredictions: rare local misspeculations and longer unpredictable phases. The former case can appear for instance with a linked list whose elements are allocated consecutively in memory. If this list is traversed whereas some elements in that list have been removed, some localized and rare misprediction occurs. The later case is related to program phases, considering that a loop nest can perform different operations on different data during the loop nest execution. Some of those phases may be predictable whereas others may not.

When the parallel execution fails, the chunk is re-executed sequentially and the next chunk is profiled. If a non-linearity is observed during profiling, several chunks are executed sequentially before profiling the loop nest again. Alternatively, a larger chunk can be executed to limit the control overhead. This long sequential execution can also occur if several consecutive parallel chunks are mispredicted, i.e. when a non-predictable phase is detected. Otherwise, while the behavior of the loop nest is linear and correctly predicted, it is executed in parallel. If the linear functions built when re-profiling the nest are incompatible with the currently selected parallel schedule, a different schedule is selected and the parallel execution continues with that other schedule.

We consider that two consecutive parallel failures define an unpredictable phase. Executing five sequential chunks to skip such a phase seems reasonable, allowing the system to adapt to short phase switches while limiting the overhead of frequent profiling. More dynamic strategies could be considered to limit this profiling overhead. For instance, the number or the size of sequential chunks skipping unpredictable phases can be increased at runtime if many successive unpredictable phases are detected.

Considering the presented results, the transactional system used to perform the rollback operation is implemented using the `memcpy`-based approach.

5.9 Putting it all Together

5.9.1 Overview

The decisions taken at every step are combined into one system. Before parallelizing the application, it is observed during an offline profiling run. Loop nests with mostly linear references are detected and some linear functions are built. Those linear functions characterize the addresses reached by the memory references, the values taken by the scalars used in address computation, and which cause inter-iteration dependences but which cannot be privatized, and the trip count of while loop inside the loop nest. A parallel schedule is selected using this speculative description of the program, assuming that it is representative of the overall program behavior. If several linear functions can be associated to the same memory reference, different schedules are generated to handle the different cases. For every schedule built from the profiling plus several generic schedules, the corresponding code is generated and is embedded in the application. Those codes contain the actual memory references, some initialization code for the predicted scalar references, and the verification code for the predicted scalar, memory references, and loop trip counts.

At runtime, the first few iterations of the sequential nest are profiled, the dependence are built from the observed linear functions and, if one of the available schedules is valid with regard to the speculative dependences, it is used to speculatively run the loop nest in parallel. During the execution, the verification is performed using conditional instructions. In case of failure, the threads are stopped using boolean flags, and the last executed chunk is reverted using a `memcpy`-based transactional system. The runtime strategy presented before is then applied.

The main drawbacks of the system are, first, that it is restricted to the schedules embedded with the application. Second, those schedules can heavily increase the binary program size. Third the transactional memory mechanisms adds a significant overhead to the parallelization. And fourth, this speculative system can only work in case of linear memory references. However, it is able to transform and parallelize complex applications which cannot be statically analyzed. As polyhedral transformations are applied on the loop nests, the parallelized code can outperform any existing speculative systems, which are not able to apply such advanced transformations. Moreover, thanks to its dynamic nature, our system can handle piecewise linear functions and apply different parallel schedules at different epochs of the execution. As our system restricts the memory references to be linear, we are also able to propose a fast software-only transactional system.

5.9.2 Evaluation

To evaluate our design decisions, we have simulated the effects of our system on the example programs. The programs are statically transformed using PLUTO to generate

program	parallel	speculative OK	sequential	speculative FAIL
`ind`	280 *ms* (×5.3)	600 *ms* (×2.5)	1,495 *ms*	620 *ms*
`llist`	495 *ms* (×6.3)	985 *ms* (×3.1)	3,100 *ms*	1,050 *ms*
`rmm`	330 *ms* (×31.3)	890 *ms* (×11.6)	10,330 *ms*	890 *ms*
`switch`	620 *ms* (×6.4)	1,400 *ms* (×2.8)	3,960 *ms*	1,500 *ms*

Table 5.8: Speculative system evaluation.

the parallel versions, assuming that a previous profiling step allowed us to build an equivalent version of the programs with linear memory references. The parallel versions are chunked, the transactional system is included, as well as the tests to validate the speculation and to kill the other threads.

We present the evaluation results in Table 5.8. The presented times are execution times of the full loop nests, sometimes accompanied by speedups over the original sequential execution time. In the table, “parallel” represents the program parallelized using PLUTO, assuming that it is able to analyze it, and where we guarantee that every memory reference is actually linear. The column “speculative OK” presents the execution time of the loop nest when no misprediction occurs, it can be considered as the best case for our system. The extra costs induced by profiling and dependence testing are considered: an extra 50 *ms* overhead is added to speculative measurements per profiling step. We show that our implementation is able to reach speedups about half of what can be performed by hand parallelization.

To evaluate the cost of a misprediction, we present in “speculative FAIL”, the execution time of the loop nest with a rollback after every chunk. Those chunks are not re-executed and the state of the memory after the execution is the same as before the loop nest execution. The only purpose of this measurement is to better evaluate the overhead of a misprediction. The worst case execution time of our system is about the sum of the sequential execution time plus a sixth of this entry. This worst case occurs when every chunk is profiled as being linear but they are all found non-linear at the very last iteration of every chunk, and sequentially re-executed by group of five chunks. This worst case overhead remains limited thanks to the optimizations applied in cases of misprediction. First, if the failure occurs earlier in the chunk execution, only a part of the chunk is executed in parallel before its sequential re-execution. Second, if the runtime profiling detects this non-linearity, the chunk is even not executed in parallel. And third, in case of consecutive failures, the system detects an unpredictable phase and immediately execute five sequential chunks.

5.10 Conclusion and Perspectives

We have presented in this chapter a speculative system able to apply polyhedral transformations on complex loop nests. Different designs are presented, and a full speculative

system is proposed and evaluated on four example programs. The main contributions of this work are twofold.

First, we detail each step required to build a speculative polyhedral system. For each step, we present several implementations, and evaluate them on four example programs, judging for their feasibility. The design choices are described, evaluated and discussed at each step to allow an informed decision.

Second, based on the discussions about the different phases, we propose a complete speculative system able to perform advanced transformations on complex loop nests. This design is confronted to the four example programs and we show that good performance can be expected from such a parallelization strategy.

In this chapter we have shown that advanced transformations can be applied when the code itself is complex. Thanks to the polyhedral model, major optimizations can be applied on programs which are even not analyzable by a static compiler. Our implementation is fully software, allowing it to be used on current general-purpose computers. It is not able to handle all the existing programs, however, alongside other (possibly speculative) systems, it can provide major performance enhancement on regular or partially regular loop nests even when no static analysis is possible.

The first perspective that comes in mind is a concrete implementation of this system. Indeed, the main proof of feasibility for such a system is its implementation. This could also allow us to evaluate the effects of this strategy on more complex programs to adjust the strategy if needed. Different other enhancements can be proposed. A clever incremental dependence analysis could be performed, testing the mispredicted accesses against the schedules at runtime, to check if it actually creates a new invalid dependence or if the parallel execution can continue. To limit the overhead of the verification and of the software transactional mechanism, the system could also be evaluated with hardware transactional memory. Finally, an implementation of an efficient runtime polyhedral parallelizer would be of major importance for our system as it could avoid restricting the set of possible transformations. This implementation could be progressive. The scheduling could be performed first on a restricted set of transformations. With the adequate restrictions, a faster code generation could be envisaged and adapted to the hybrid code generation we have presented. If this can be done efficiently, the runtime scheduling and code generation can be progressively extended until reaching an efficient runtime polyhedral scheduler.

Chapter 6

Conclusion and Perspectives

6.1 Contributions

We have proposed three systems which have the ambition of improving the state-of-the-art. All together, they form a consistent approach to enhance the parallelization capabilities of existing systems.

In Chapter 3, we present a binary parallelization system. This system provides the following improvements compared to existing systems.

- It is a static system, on the contrary to most of the existing approaches, using speculative parallelization to handle binary programs. We have shown that, some of the loops in executable codes can already be parallelized statically, and do not require to be handled at runtime. Thus, our system does not suffer from the runtime overhead associated to any dynamic tool. Moreover, it does not require any specific hardware to detect dependences between loop iterations. Although our system can only handle polyhedral loop nests, it is a perfect complement to any dynamic binary parallelization system. Such partnership provides an efficient solution for statically parallelizable loop nest as well as other more complex structures.

- Our approach is very modular: any source-to-source parallelizer can be used to transform and parallelize the loop nests extracted from the binary programs. Thus, it is not attached to any particular method to analyze the dependences, transform the program, or parallelize loops. We have shown, using two different back-end parallelizers, that different parallelization approaches can be considered while maintaining high performance compared to an equivalent parallelization from the original source code. This modularity allows our system to virtually benefit from many parallelization or transformation techniques which have been or will be developed, as soon as the skeleton extracted from the binary program suffice them.

- The static binary parallelization we propose is the first attempt to apply polyhedral transformations on an already compiled program. When using a polyhedral parallelizer, we show that advanced loop optimizations can be efficiently applied

on affine loop nests. Our decompilation phase is generally sufficient to extract enough information to allow such tools to perform well. The different steps performed to recompile the program are also efficient, leading to benefit from nearly all the gains of such advanced parallelization. The transformation applied can sometimes clearly outperform all the existing static parallelizers, which do not perform such loop transformation.

- We propose an extension to parallelize loop nests with non-linear memory references. This extension is mandatory to handle more complex binary programs. As it does not allow code transformations, we have also proposed a new parallelization strategy dedicated to binary code, where the original binary program can be parallelized without moving nor duplicating any code fragment.

This parallelization system is the first brick of our general approach and exploits the compilation time to parallelize the programs, even when their format is complex.

The second proposed system is hybrid, exploiting both the compilation time and the execution time to enhance the parallelization of programs. This code selection mechanism is presented in Chapter 4. It provides several enhancements to the current state-of-the-art:

- It targets parallel programs, while the existing code selections mechanisms are often restricted to sequential programs. Parallel programs induce specific challenges which have to be handled specifically. Thus, the existing selection methods can often not be directly ported to handle parallel programs. The approach we present specifically focuses on parallel programs and addresses the related challenges.

- On the contrary to most of the existing systems, our system can select an efficient code version since the first execution of the program. After a short profiling period performed at the program installation, our system selects an efficient version for every encountered execution context. Its high precision, coupled with a negligible runtime overhead, leads to a very efficient system, able to quickly adapt to changing execution contexts.

- Our selection system is fully automatic. It is interesting to note that most of the existing systems require a human intervention at some point. With our approach, the original program does not need to be rewritten in a specific language, it is not needed to describe the operations performed in the program, neither to build valid training data.

- The polyhedral approach used by this selection system is the warrant for its precision. Indeed, the stable behavior of affine loop nests guarantees the precision of the selection mechanism. The other mechanisms proposed until now cannot benefit from any such property, and can be easily tricked by unstable programs.

This is the second brick of our general approach: we show that, by exploiting both compilation and execution times, the parallelization performed on programs can be enhanced toward changing execution contexts. We show that an efficient dynamic

system, coupled with a static profiling step, can ensure an efficient selection of parallel versions of a program.

The last system we have presented is a speculative parallelizer using the polyhedral model. Despite it has not yet been totally implemented and evaluated, the first results allow us to subsume that several advances can be performed relatively to existing systems:

- As our system is the first allowing polyhedral transformations to be performed on non statically analyzable loop nests, it is also the first one able to benefit from the associated performance gains. It can then be expected that such a system would clearly outperform the existing systems which cannot transform a loop nest to enhance data locality nor to exhibit a parallel loop level when none is naturally exposed.

- The polyhedral model is often considered to be useful only for static parallelization. We show here that dynamic parallelization of non statically analyzable code can be performed using the polyhedral model. This strongly widen the class of programs which can be handled by this representation and its associated tools. In some sense, this work is then extending the different approaches which have been proposed to apply the polyhedral model on more complex programs.

- The constraint of linearity imposed on memory references helps to build an efficient speculative system where no hardware is required. This is rare for speculative parallelizers which often delegate the complex task of dependence analysis and memory management to dedicated hardware, despite the few commercialized hardware transactional mechanisms.

This system is the last brick where the runtime is considered at the moment to apply advanced parallelization techniques on non-statically analyzable loops. This third system complements the two others to form a global approach where every specific moment of the program lifetime is exploited to parallelize it or to enhance the performed parallelization. Those three approaches together provide concrete solutions to the automatic program parallelization problem.

From the polyhedral model point of view, we also show that, with no specific modification to the model itself, it is possible to extend its uses to programs where the format is complex (binary programs), where the execution context is complex, and where the code structure itself is complex. We show that, on the contrary to what is commonly assumed about this model, an efficient parallelization can be performed in all those different situations.

6.2 Future work

The static binary parallelizer we have presented shows that complex loop transformations can be applied statically at the binary level. The system itself can be enhanced and especially its robustness can be improved. The compilers that produce the sequential binary program can apply complex code transformations such as loop unrolling or

basic block fusion. This can disrupt the analysis performed on the binary program. If the system would be able to revert those transformations, it could more easily parallelize the programs in such cases. It may also be important to approximate the control flow if it is unanalyzable. For instance, data-dependent conditions could be ignored. This would lead to consider more dependences than there actually are, but would allow the parallelization of more complex programs. On the other hand, some programs are fully analyzable and a very precise high-level representation of the program can be reconstructed from the binary code. In that case, the translation to another architecture could be an interesting feature, allowing for instance to parallelize legacy x86 programs on GPUs. Moreover, our system is modular and can use any source-to-source parallelizer, we could exploit this modularity for other goals than parallelization. For instance code size, or energy consumption could be interesting alternative goals.

Our hybrid version selection system is restricted to regular, affine loop nests. Several other approaches have shown some improvements on more general codes. Then, it would be interesting to extend the system to a more general class of programs. The main issue is, as explained before, that the polyhedral model gives us a very strict frame in which we can ensure that the program behavior can be efficiently predicted. Other extensions could target more complex architectural contexts. For instance, performing this code selection on platforms with multiple parallel hardware is an interesting challenge. It would allow one to decide whether to execute a given kernel on a CPU or a GPU for instance. Interestingly, one could also notice that combining the code selection with the binary code approach could allow one to perform this selection at the binary level.

The speculative parallelization we have presented has not yet been completely implemented, on the contrary to the other presented systems. The first perspective is to achieve this implementation. It could lead to several adjustments on the presented techniques and strategies. From this implementation, a vast research area would be opened, with the final goal of reaching fully dynamic application of the polyhedral model. This would allow the polyhedral model to be applied on programs which are currently not in its scope. It is not clear up to what point the polyhedral model can be extended to this fully dynamic use, and several challenges have to be overcome, including the complexity of the core algorithms used in this model.

Those three systems altogether form a global answer to the parallelization problematic. This dissertation is obviously not a final answer to this problem, but provides solutions extending the automatic parallelization of programs towards a fully automatic system able to handle complex program formats, complex execution contexts, and complex code structures.

This parallelization problematic will probably become even more important in the next years as the hardware designers already announced processors with hundreds of cores. This will probably be accompanied by an increased complexity of the hardware with the appearance of specialized parallel hardware, or simpler cache coherency on processors for instance. Other problematics, related to those new hardwares, will probably emerge. For instance energy consumption or reliability will probably become more and more important. It creates a complex architectural environment, with numerous parallel resources that the softwares will have to correctly exploit. The automatic

parallelization performed by compilers is then the first step towards this challenging future.

Personal Bibliography

- Adaptive Runtime Selection of Parallel Schedules in the Polytope Model. Benoît Pradelle, Philippe Clauss and Vincent Loechner. In 19th High Performance Computing Symposium, ACM/SIGSIM, Boston, MA, USA, 2011.
- Context-Aware Runtime Selection of Parallel Loops on Multi-Core Platforms. Benoît Pradelle, Philippe Clauss and Vincent Loechner. Technical report ICPS/LSIIT. Submitted for publication in a journal. 2011.
- Transparent Parallelization of Binary Code. Benoît Pradelle, Alain Ketterlin and Philippe Clauss. In First International Workshop on Polyhedral Compilation Techniques, IMPACT 2011, in conjunction with CGO 2011, Chamonix, France, 2011.
- Polyhedral Parallelization of Binary Code. Benoît Pradelle, Alain Ketterlin and Philippe Clauss. In 7th HiPEAC conference, ACM, Paris, France, 2012.
- Polyhedral Parallelization of Binary Code. Benoît Pradelle, Alain Ketterlin and Philippe Clauss. To be published in Transactions on Architecture and Code Optimization, TACO, ACM, 2012.
- Adapting the Polyhedral Model as a Framework for Efficient Speculative Parallelization. Alexandra Jimborean, Philippe Clauss, Benoît Pradelle, Luis Mastrangelo and Vincent Loechner. Technical report ICPS/LSIIT. Submitted for publication. 2011.

Bibliography

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 295–305, 2006.

[2] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: a language and compiler for algorithmic choice. In *PLDI '09*, pages 38–49. ACM, 2009.

[3] A. W. Appel and M. Ginsburg. *Modern Compiler Implementation in C.* Cambridge University Press, 2004.

[4] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. A survey of adaptive optimization in virtual machines. volume 93, pages 449 –466, feb. 2005.

[5] Vishal Aslot, Max J. Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. Specomp: A new benchmark suite for measuring parallel computer performance. In *WOMPAT '01*, pages 1–10. Springer-Verlag, 2001.

[6] Hansang Bae, Leonardo Bachega, Chirag Dave, Sang-Ik Lee, Seyong Lee, Seung-Jai Min, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. In *Proc. of the 14th Int'l Workshop on Compilers for Parallel Computing (CPC'09)*, 2009.

[7] Riyadh Baghdadi, Albert Cohen, Cédric Bastoul, Louis-Noël Pouchet, and Lawrence Rauchwerger. The Potential of Synergistic Static, Dynamic and Speculative Loop Nest Optimizations for Automatic Parallelization. In Wei Liu, Scott Mahlke, and Tin fook Ngai, editors, *Pespma 2010 - Workshop on Parallel Execution of Sequential Programs on Multi-core Architecture*, Saint Malo, France, 2010.

[8] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM.

[9] Utpal K. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Norwell, MA, USA, 1988.

[10] Muthu Manikandan Baskaran, Albert Hartono, Sanket Tavarageri, Thomas Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 200–209, 2010.

[11] 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*, pages 209–225, College Station, October 2003.

[12] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13*, pages 7–16, September 2004.

[13] Cédric Bastoul. *Improving Data Locality in Static Control Programs.* PhD thesis, University Paris 6, Pierre et Marie Curie, France, December 2004.

[14] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *ETAPS CC*, 2010.

[15] Jean Christophe Beyler. *Dynamic Software Optimization of Memory Accesses.* PhD thesis, University Louis Pasteur, Strasbourg, France, December 2007.

[16] Jean Christophe Beyler and Philippe Clauss. Performance driven data cache prefetching in a dynamic software optimization system. In *Proceedings of the 21st annual international conference on Supercomputing*, ICS '07, pages 202–209, New York, NY, USA, 2007. ACM.

[17] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28:135–151, 2001.

[18] William Blume and Rudolf Eigenmann. The range test: A dependence test for symbolic, non-linear expressions. In *Proceedings of Supercomputing '94, Washington D.C*, pages 528–537, 1994.

[19] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Journal of Parallel and Distributed Computing*, pages 207–216, 1995.

[20] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5), 1999.

[21] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O' Boyle, and Erven Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation*, Paris, France, 1998.

[22] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Proceedings of the Joint European Conferences on Theory and Practice of Software, 17th international conference on Compiler construction*, CC'08/ETAPS'08, pages 132–146, Berlin, Heidelberg, 2008. Springer-Verlag.

[23] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI '08*, pages 101–113. ACM, 2008. `http://pluto-compiler.sourceforge.net`.

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

[25] Derek L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation.* PhD thesis, Cambridge, MA, USA, 2004. AAI0807735.

[26] Pierre-Yves Calland, Alain Darte, Yves Robert, and Frédéric Vivien. On the removal of anti- and output-dependences. *Int. J. Parallel Program.*, 26:285–312, June 1998.

[27] C. Chen, J. Chame, and M. Hall. Chill: A framework for composing high-level loop transformations. Technical Report 08-897, University of Southern California, June 2008.

[28] Ding-Kai Chen, Josep Torrellas, and Pen-Chung Yew. An efficient algorithm for the run-time parallelization of doacross loops. In *Proceedings of the 1994 conference on Supercomputing*, Supercomputing '94, pages 518–527, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[29] Peng-Sheng Chen, Ming-Yu Hung, Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jenq Kuen Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '03, pages 25–36, New York, NY, USA, 2003. ACM.

[30] Marcelo Cintra and Diego R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '03, pages 13–24, New York, NY, USA, 2003. ACM.

[31] Ph. Clauss and I. Tchoupaeva. A Symbolic Approach to Bernstein Expansion for Program Analysis and Optimization. In *CC*, 2004.

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

[33] Philippe Clauss, Federico Javier Fernández, Diego Garbervetsky, and Sven Verdoolaege. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. *IEEE Trans. Very Large Scale Integr. Syst.*, 17, 2009.

[34] Albert Cohen, Sylvain Girbal, and Olivier Temam. A polyhedral approach to ease the composition of program transformations. In *in: Euro-Par'04, no. 3149 in LNCS*, pages 292–303. Springer-Verlag, 2004.

[35] Robert S. Cohn, David W. Goodwin, and P. Geoffrey Lowney. Optimizing alpha executables on windows nt with spike. *Digital Tech. J.*, 9:3–20, April 1998.

[36] Jean-François Collard. Automatic parallelization of while-loops using speculative execution. *International Journal of Parallel Programming*, 23:191–219, 1995. 10.1007/BF02577789.

[37] Jean-François Collard, Denis Barthou, and Paul Feautrier. Fuzzy array dataflow analysis. *SIGPLAN Not.*, 30:92–101, August 1995.

[38] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, 23, 2001.

[39] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4), 1991.

[40] Matthew DeVuyst, Dean M. Tullsen, and Seon Wook Kim. Runtime parallelization of legacy code on a transactional memory system. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 127–136, New York, NY, USA, 2011. ACM.

[41] Chen Ding and Ken Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *In Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 229–241, 1999.

[42] Lamia Djoudi, Denis Barthou, Patrick Carribault, Christophe Lemuet, Jean-Thomas Acquaviva, and William Jalby. Maqao: Modular assembler quality analyzer and optimizer for itanium 2. In *Workshop on Explicitly Parallel Instruction Computing Techniques*, 2005.

[43] Kemal Ebcioğlu and Erik R. Altman. Daisy: dynamic compilation for 100% architectural compatibility. *SIGARCH Comput. Archit. News*, 25:26–37, May 1997.

[44] Andrew Edwards, Hoi Vo, and Amitabh Srivastava. Vulcan binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft, 2001.

[45] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[46] P. Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, 1991.

[47] Paul Feautrier. Array expansion. In *In ACM Int. Conf. on Supercomputing*, pages 429–441, 1988.

[48] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part 1 : one dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, 1992.

[49] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part 2 : multidimensional time. *International Journal of Parallel Programming*, 21(6), 1992.

[50] Paul Feautrier. Automatic parallelization in the polytope model. In *Laboratoire PRiSM, Université de Versailles St-Quentin en Yvelines, 45, avenue des États-Unis, F-78035 Versailles Cedex*, pages 79–103. Springer-Verlag, 1996.

[51] Grigori Fursin, Albert Cohen, Michael O'Boyle, and Olivier Temam. A practical method for quickly evaluating program optimizations. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2005)*, pages 29–46. Springer Verlag, 2005.

[52] Grigori Fursin, Yuriy Kashnikov, Abdul Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher Williams, and Michael O'Boyle. Milepost gcc: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, 39:296–327, 2011.

[53] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996.

[54] Georgios Goumas, Maria Athanasaki, and Nectarios Koziris. An efficient code generation technique for tiled iteration spaces. *IEEE Transactions on Parallel and Distributed Systems*, 14:1034, 2003.

[55] Tobias Grosser. *Enabling Polyhedral Optimizations in LLVM.* PhD thesis, Passau, Germany, 2011.

[56] Armin Größlinger, Martin Griebl, and Christian Lengauer. Introducing non-linear parameters to the polyhedron model. In Michael Gerndt and Edmond Kereku, editors, *Proc. 11th Workshop on Compilers for Parallel Computers (CPC 2004)*, Research Report Series, pages 1–12. LRR-TUM, Technische Universität München, July 2004.

[57] Gautam Gupta and Sanjay Rajopadhye. The $\mathcal{Z}$-polyhedral model. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 237–248, New York, NY, USA, 2007. ACM.

[58] A. Hartono, M.M. Baskaran, J. Ramanujam, and P. Sadayappan. Dyntile: Parametric tiled loop generation for parallel execution on multicore processors. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, april 2010.

[59] Albert Hartono, Muthu Manikandan Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 147–157, New York, NY, USA, 2009. ACM.

[60] Ben Hertzberg and Kunle Olukotun. Runtime automatic speculative parallelization. *Code Generation and Optimization, IEEE/ACM International Symposium on, CGO*, 0:64–73, 2011.

[61] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE'01*, pages 54–61. ACM Press, 2001.

[62] Jeffrey Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. In *In Scalable High Performance Computing Conference*, pages 841–850, 1994.

[63] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Speculative thread decomposition through empirical optimization. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 205–214, New York, NY, USA, 2007. ACM.

[64] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14:563–590, July 1967.

[65] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In *Frontiers'95: the 5th symposium on the Frontiers of Massively Parallel Computation*, pages 332–341, 1994.

[66] Wayne Anthony Kelly. *Optimization within a unified transformation framework*. PhD thesis, College Park, MD, USA, 1996. AAI9707628.

[67] Alain Ketterlin and Philippe Clauss. Recovering the Memory Behavior of Executable Programs. In *10th IEEE Working Conference on Source Code Analysis and Manipulation, SCAM*, Timisoara, Romania, September 2010. IEEE Computer Society Press.

[68] DaeGon Kim and Sanjay V. Rajopadhye. Parameterized tiling for imperfectly nested loops. Technical Report CS-09-101, Colorado State University, February 2009.

[69] DaeGon Kim, Lakshminarayanan Renganarayanan, Dave Rostron, Sanjay Rajopadhye, and Michelle Mills Strout. Multi-level tiling: M for the price of one. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 51:1–51:12, New York, NY, USA, 2007. ACM.

[70] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. Sd3: A scalable approach to dynamic data-dependence profiling. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 535–546, Washington, DC, USA, 2010. IEEE Computer Society.

[71] Thomas Kistler and Michael Franz. Continuous program optimization: Design and evaluation. *IEEE Trans. Comput.*, 50:549–566, June 2001.

[72] T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, and H. A. G. Wijshoff. Iterative compilation in program optimization. *Computing Systems*, 2000.

[73] Xiangyun Kong, David Klappholz, and Kleanthis Psarris. The I Test: An improved dependence test for automatic parallelization and vectorization. *IEEE Transactions on Parallel and Distributed Systems*, 2(3), July 1991.

[74] Aparna Kotha, Kapil Anand, Matthew Smithson, Greeshma Yellareddy, and Rajeev Barua. Automatic parallelization in a binary rewriter. MICRO '43, 2010.

[75] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. PEBIL: Efficient static binary instrumentation for linux. In *ISPASS*, 2010.

[76] Christian Lengauer and Martin Griebl. On the parallelization of loop nests containing while loops. In *Proc. 1st Aizu Int. Symp. on Parallel Algorithm/Architecture Synthesis (pAs'95)*, pages 10–18. IEEE Computer Society Press, 1995.

[77] Shun-Tak Leung and John Zahorjan. Improving the performance of runtime parallelization. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 83–91, New York, NY, USA, 1993. ACM.

[78] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. In *Parallel Computing*, pages 201–214. ACM Press, 1998.

[79] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: a TLS compiler that exploits program structure. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 158–167, New York, NY, USA, 2006. ACM.

[80] Shun Long and Grigori Fursin. A heuristic search algorithm based on unified transformation framework. In *In ICPPW '05: Proceedings of the 2005 International Conference on Parallel Processing Workshops (ICPPW'05)*, pages 137–144. IEEE Computer Society, 2005.

[81] Jiwei Lu, Howard Chen, Pen-Chung Yew, and Wei chung Hsu. Design and implementation of a lightweight dynamic optimization system. *Journal of Instruction-Level Parallelism*, 6:2004, 2004.

[82] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa, and Reddi Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, pages 190–200. ACM Press, 2005.

[83] Jason Mars and Robert Hundt. Scenario based optimization: A framework for statically enabling online optimizations. In *CGO '09*, pages 169–179. IEEE Computer Society, 2009.

[84] Erik Meijer and John Gough. Technical overview of the common language runtime. *http://research.microsoft.com/en-us/um/people/emeijer/papers/CLR.pdf*, 2000.

[85] John Mellor-Crummey, Robert J. Fowler, Gabriel Marin, and Nathan Tallent. Hpcview: A tool for top-down analysis of node performance. *J. Supercomput.*, 23:81–104, August 2002.

[86] Michael Philippsen, Nikolai Tillmann, and Daniel Brinkers. Double inspection for run-time loop parallelization. In *Proceedings of the 24th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2011)*, 2011.

[87] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce, Irvin Karen, L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tools. *IEEE Computer*, 28:37–46, 1995.

[88] Ravi Mirchandaney, Joel H. Saltz, and Doug Baxter. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40, 1991.

[89] Nicholas Mitchell, Larry Carter, and Jeanne Ferrante. Localizing non-affine array references. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, PACT '99, Washington, DC, USA, 1999. IEEE Computer Society.

[90] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[91] Robert Muth, Saumya Debray, Scott Watterson, and Koen De Bosschere. alto: A link-time optimizer for the compaq alpha. *Software - Practice and Experience*, 31:67–101, 1999.

[92] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[93] Andy Nisbet. Gaps: A compiler framework for genetic algorithm (ga) optimised parallelisation. In *In HPCN Europe*, pages 987–989, 1998.

[94] K. Ootsu, T. Yokota, T. Ono, and T. Baba. Preliminary evaluation of a binary translation system for multithreaded processors. In *Innovative Architecture for Future Generation High-Performance Processors and Systems, 2002. International Workshop on*, pages 77 – 84, 2002.

[95] OpenMP Application Program Interface. `http://www.openmp.org`.

[96] Eunjung Park, Louis-Noël Pouchet, John Cavazos, Albert Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. In *9th IEEE/ACM International Symposium on Code Generation and Optimization (CGO'11)*, Chamonix, France, April 2011. IEEE Computer Society press.

[97] Paul M. Petersen and David A. Padua. Static and dynamic evaluation of data dependence analysis. In *Proceedings of the 7th international conference on Supercomputing*, ICS '93, pages 107–116, New York, NY, USA, 1993. ACM.

[98] PLAT0 library website. `http://www.cs.utsa.edu/~plato/lib/platolib.html`.

[99] PLuTo website. `http://pluto-compiler.sourceforge.net/`.

[100] Polybenchs 1.0. `http://www-rocq.inria.fr/pouchet/software/polybenchs`.

[101] Polylib. `http://icps.u-strasbg.fr/PolyLib`.

[102] R. Ponnusamy, J. Saltz, and A. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Supercomputing '93, pages 361–370, New York, NY, USA, 1993. ACM.

[103] Louis-Noël Pouchet. *Iterative Optimization in the Polyhedral Model*. PhD thesis, University of Paris-Sud 11, Orsay, France, January 2010.

[104] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *PLDI'08*, pages 90–100. ACM Press, June 2008.

[105] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *CGO'07*, pages 144–156. IEEE Computer Society press, March 2007.

[106] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Conference on Supercomputing (SC'10)*, New Orleans, LA, November 2010. IEEE Computer Society Press.

[107] PPL: The parma polyhedra library. `http://www.cs.unipr.it/ppl/`.

[108] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 4–13, New York, NY, USA, 1991. ACM.

[109] Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 269–279, New York, NY, USA, 2005. ACM.

[110] Fabien Quillere, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28:2000, 2000.

[111] Easwaran Raman, Neil Va hharajani, Ram Rangan, and David I. August. Spice: speculative parallel iteration chunk execution. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 175–184, New York, NY, USA, 2008. ACM.

[112] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–120, 1992.

[113] Lawrence Rauchwerger, Nancy M. Amato, and David A. Padua. Run-time methods for parallelizing partially parallel loops. In *Proceedings of the 9th international conference on Supercomputing*, ICS '95, pages 137–146, New York, NY, USA, 1995. ACM.

[114] Lawrence Rauchwerger, Nancy M. Amato, and David A. Padua. A scalable method for run-time loop parallelization. *IJPP*, 26:26–6, 1995.

[115] Lawrence Rauchwerger and David Padua. The privatizing doall test: A run-time technique for doall loop identification and array privatization. In *Proceedings of the 1994 International Conference on Supercomputing*, pages 33–43, 1994.

[116] Lawrence Rauchwerger and David Padua. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 218–232, New York, NY, USA, 1995. ACM.

[117] Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay Rajopadhye, and Michelle Mills Strout. Parameterized tiled loops for free. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 405–414, New York, NY, USA, 2007. ACM.

[118] Lakshminarayanan Renganarayanan and Sanjay V. Rajopadhye. Positivity, posynomials and tile size selection. In *SC'08*, 2008.

[119] Reservoir Labs. R-stream ® - high level compiler. `https://www.reservoir.com/rstream`.

[120] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and optimization of win32/intel executables using etch. In *In Proceedings of the USENIX Windows NT Workshop*, pages 1–7, 1997.

[121] Silvius Rus, Maikel Pennings, and Lawrence Rauchwerger. Sensitivity analysis for automatic parallelization on multi-cores. In *Proceedings of the 21st annual international conference on Supercomputing*, ICS '07, pages 263–273, New York, NY, USA, 2007. ACM.

[122] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. *Int. J. Parallel Program.*, 31:251–283, August 2003.

[123] Joel H. Saltz and Ravi Mirchandaney. The preprocessed doacross loop. In *International Conference on Parallel Processing*, pages 174–179, 1991.

[124] Joel H. Saltz, Ravi Mirchandaney, and Kathleen Crowley. The doconsider loop. In *Proceedings of the 3rd international conference on Supercomputing*, ICS '89, pages 29–40, New York, NY, USA, 1989. ACM.

[125] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[126] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. Plto: A link-time optimizer for the intel ia-32 architecture. In *In Proc. 2001 Workshop on Binary Translation (WBT-2001)*, 2001.

[127] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. pages 196–205. ACM, 1994.

[128] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 91–102, New York, NY, USA, 2003. ACM.

[129] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), 2005.

[130] Nathan Thomas, Gabriel Tanase, Olga Tkachyshyn, Jack Perdue, Nancy M. Amato, and Lawrence Rauchwerger. A framework for adaptive algorithm selection in stapl. In *PPoPP '05*, pages 277–288. ACM, 2005.

[131] Chen Tian, Min Feng, and Rajiv Gupta. Speculative parallelization using state separation and multiple value prediction. In *Proceedings of the 2010 international symposium on Memory management*, ISMM '10, pages 63–72, New York, NY, USA, 2010. ACM.

[132] Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. Copy or discard execution model for speculative parallelization on multicores. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 330–341, Washington, DC, USA, 2008. IEEE Computer Society.

[133] Kai Tian, Yunlian Jiang, Eddy Z. Zhang, and Xipeng Shen. An input-centric paradigm for program dynamic optimizations. In *OOPSLA '10*, pages 125–139. ACM, 2010.

[134] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In *Proceedings of the international symposium on Code generation and optimization*, pages 204–215. IEEE Computer Society, 2003.

[135] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. In *GCC Research Opportunities Workshop (GROW'10)*, Pisa, Italie, January 2010.

[136] L. Van Put, D. Chanet, B. De Bus, B. De Sutler, and K. De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *Signal Processing and Information Technology*, 2005.

[137] Nicolas Vasilache, Albert Cohen, and Louis-Noel Pouchet. Automatic correction of loop transformations. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 292–304, Washington, DC, USA, 2007. IEEE Computer Society.

[138] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *Proceedings of the Third international congress conference on Mathematical software*, ICMS'10, pages 299–302, Berlin, Heidelberg, 2010. Springer-Verlag.

[139] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 48(1):37–66, June 2007. DOI: 10.1007/s00453-006-1231-0.

[140] Michael J. Voss and Rudolf Eigemann. High-level adaptive program optimization with ADAPT. In *PPoPP '01*, pages 93–102. ACM, 2001.

[141] R. Clinton Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[142] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 30–44, New York, NY, USA, 1991. ACM.

[143] Michael Joseph Wolfe. *Optimizing supercompilers for supercomputers*. PhD thesis, Champaign, IL, USA, 1982. AAI8303027.

[144] J. Yang, K. Skadron, M.-L. Soffa, and K. Whitehouse. Feasibility of dynamic binary parallelization. `http://www.usenix.org/event/hotpar11/poster.html`, may 2011.

[145] Efe Yardımcı and Michael Franz. Dynamic parallelization and mapping of binary executables on hierarchical platforms. In *Proceedings of the 3rd conference on Computing frontiers*, CF '06, pages 127–138, New York, NY, USA, 2006. ACM.

[146] Hongtao Zhong, Mojtaba Mehrara, Steven A. Lieberman, and Scott A. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *HPCA*, pages 290–301, 2008.

[147] Chuan-Qi Zhu and Pen-Chung Yew. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Trans. Softw. Eng.*, 13:726–739, June 1987.