Runtime Optimization of Binary Through Vectorization Transformations

Nabil Hallou

To cite this version:

Nabil Hallou. Runtime Optimization of Binary Through Vectorization Transformations. Other [cs.OH]. Université de Rennes 1 [UR1], 2017. English. NNT : 2017REN1S120 . tel-01795489v1

HAL Id: tel-01795489
https://inria.hal.science/tel-01795489v1

Submitted on 23 Dec 2017 (v1), last revised 18 May 2018 (v2)

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.

N^o d'ordre : 00000 ANNÉE 2017

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Nabil HALLOU

préparée à l'unité de recherche INRIA
Institut National de Recherche en Informatique et en Automatique
Université Rennes 1

Runtime Optimization of Binary Through Vectorization Transformations

Thèse soutenue à Rennes le 18 Décembre 2017

devant le jury composé de :

Mme Isabelle Puaut
Professeur à l'Université de Rennes 1 / *Présidente*

M Denis Barthou
Professeur à l'INP de Bordeaux / *Rapporteur*

M Felix Wolf
Professeur à l'Université Technique de Darmstadt / *Rapporteur*

Mme Alexandra Jimborean
Maître de conférence à l'Université d'Uppsala / *Examinatrice*

M Erven Rohou
Directeur de recherche INRIA, Equipe PACAP, à Rennes / *Directeur de thèse*

M Philippe Clauss
Professeur à l'Université de Strasbourg / *Co-directeur de thèse*

Acknowledgment

This research study is dedicated to my beloved mother, sister, and the rest of my familly members.

I would like to express my sincere thanks and gratitude to Dr. Erven Rohou and Dr. Philippe Clauss for giving me the opportunity to learn and work under their supervision. Their continuous guidance, help, support, and kindness throughout the course of my research was a propeling force without which I would not be able to propose this humble contribution.

Contents

Table of contents 1

Résumé 5
- 0.1 Optimisation dynamique de code binaire 5
- 0.2 Brève introduction à la vectorisation 6
- 0.3 La re-vectorisation du code binaire 7
  - 0.3.1 Conversion des instructions SSE en équivalents AVX 8
  - 0.3.2 La contrainte d'alignement 8
  - 0.3.3 L'ajustement du compteur 9
  - 0.3.4 L'ajustement du nombre total d'iterations 9
    - 0.3.4.1 Nombre total d'iterations connu au moment de la compilation 9
    - 0.3.4.2 Le nombre total d'iterations seulement connu à l'exécution 9
- 0.4 Vectorisation automatique 10
  - 0.4.1 McSema 10
  - 0.4.2 La vectorisation et la compilation à la volée 12
    - 0.4.2.1 Le problème des variables globales 12
    - 0.4.2.2 Le marquage des variables globales 12
- 0.5 Résultats 13
  - 0.5.1 Re-vectorisation 13
    - 0.5.1.1 Résultats 13
  - 0.5.2 Vectorisation automatique 14
    - 0.5.2.1 Résultats 14
- 0.6 Conclusion 16

1 Introduction 17
- 1.1 Context 17
- 1.2 Problem definition 19
- 1.3 Proposed solution 19
- 1.4 List of publications 20

2 Dynamic Binary Optimization 23
- 2.1 Software-based profiling 23
  - 2.1.1 Just-In-Time (JIT) approach 24

2.1.2 Probing approach 25
2.1.3 Overhead 25
2.2 Hardware-based profiling 26
2.2.1 Hardware performance counters at a glance 26
2.2.2 Useful features of hardware performance counter for profiling . . 27
2.2.3 Overhead vs accuracy 27
2.3 Padrone infrastructure 28
2.3.1 Profiling 28
2.3.2 Analysis 29
2.3.3 Code cache 30
2.4 Conclusion 31

3 Dynamic Re-vectorization of Binary Code 33
3.1 Background 33
3.1.1 Vectorization at a glance 33
3.2 Re-Vectorization of Binary Code 34
3.2.1 Principle of the SSE into AVX translation 34
3.2.2 Converting instructions from SSE into AVX 35
3.2.3 Register liveness 36
3.2.4 Induction variables 37
3.2.5 Loop bounds 37
3.2.5.1 Loop bound known at compile-time 38
3.2.5.2 Loop bound known only at run-time 38
3.2.6 Aliasing and Data Dependencies 40
3.2.6.1 Overview of aliasing 40
3.2.6.2 Issue of translating a loop with data dependencies . . . 40
3.2.6.3 Static interval-overlapping test 40
3.2.6.4 Dynamic interval-overlapping test 41
3.2.7 Alignment constraints 41
3.2.8 Reductions 43
3.2.8.1 Issue of translating a reduction 44
3.2.8.2 Subclass of reduction supported by the translator . . . 44
3.3 Conclusion 47

4 Dynamic Vectorization of Binary Code 49
4.1 Principles of the polyhedral model 50
4.1.1 Static control part (SCoP) 50
4.1.2 Perfect and inperfect loop nest 50
4.1.3 Iteration domain 51
4.1.4 Access function 52
4.1.5 Execution order between instruction 53
4.1.6 Data dependence 54
4.1.7 Loop transformations 58
4.1.7.1 Unimodular vs polyhedral transformations 58

4.1.7.2 Polyhedral transformation 61
4.2 Vectorization of Binary Code 64
4.2.1 Principle of scalar into vector optimization 64
4.2.2 Binary into intermediate representation using McSema 65
4.2.2.1 McSema 65
4.2.2.2 Integration of Padrone with McSema 66
4.2.2.3 Adjusting McSema to produce a suitable LLVM-IR . . 66
4.2.3 Vectorization of loops in LLVM-IR using Polly 68
4.2.3.1 Canonicalization 70
4.2.3.2 ScoP Detection 71
4.2.3.3 Scop Extraction 72
4.2.3.4 Dependence analysis 72
4.2.3.5 Scheduling 72
4.2.3.6 Vectorization 72
4.2.4 LLVM JIT tuning 76
4.2.4.1 Handling global variables 76
4.2.4.2 Marking the IR operands 76
4.3 Conclusion 76

5 Experimental Results 79
5.1 Re-Vectorization experimental results 79
5.1.1 Hardware/Software 79
5.1.2 Benchmarks 79
5.1.3 Performance Results 80
5.1.4 Overhead 84
5.2 Vectorization experimental results 85
5.2.1 Hardware/Software 85
5.2.2 Benchmarks 85
5.2.3 Performance Results 85
5.2.4 Overhead 86
5.3 Conclusion 86

6 Related work 87
6.1 Compilers' auto-vectorization 87
6.2 Thread level speculation systems 91
6.3 Binary-to-binary auto-vectorization and auto-parallelization 92
6.4 Optimizations at the level of a virtual machine or using dynamic binary translation tools 94
6.5 Conclusion 95

7 Conclusion 97
7.1 Perspectives 98

Index 99

Bibliography 108

List of Figures 109

Résumé

Depuis la sortie du processeur Intel 80286 en 1982, la tendance des fabricants de processeurs est de respecter la contrainte de compatibilité ascendante. En bref, elle permet aux programmes compilés pour un ancien processeur de s'exécuter sur une version plus récente de la même famille. Son inconvénient est de restreindre le logiciel à utiliser uniquement les fonctionnalités déjà existantes dans l'ancienne version du processeur dans sa nouvelle version. Cette sous-utilisation des ressources est en corrélation avec une faible performance. Cela se produit à cause de la la non-disponibilité du code source. Pour donner quelques exemples concrets, pour certaines raisons l'industrie continue d'utiliser des logiciels patrimoniaux ou hérités sur des nouvelles machines, et les programmes de calcul intensifs sont lancés pour tourner dans des clusters sans connaissance du matériel sur lesquels ils s'exécutent.

Pour répondre à ce problème de sous-utilisation des ressources, notre recherche porte sur l'optimisation du code binaire pendant l'éxécution. Nous ciblons les régions fréquemment exécutées en utilisant une méthode de profilage très légère. Une fois que ces dernières ont été détectées, elles sont automatiquement optimisées et le nouveau binaire est injecté dans le tas du programme cible. Les appels sont redirigés vers la nouvelle version optimisée. Nous avons adopté deux optimisations : premièrement, la re-vectorisation est une optimisation binaire-à-binaire qui cible des boucles déjà vectorisées pour une ancienne version du processeur (SSE) et les convertit en versions optimisées (AVX) pour maximiser l'utilisation des ressources. Deuxièmement, la vectorisation est une optimisation indépendante de l'architecture qui vectorise des boucles scalaires. Nous avons intégré des logiciels libres pour : (1) traduire dynamiquement le binaire x86 vers la représentation intermédiaire du compilateur LLVM, (2) abstraire et vectoriser les boucles imbriquées dans le modèle polyédrique, (3) les compiler à la volée en utilisant le compilateur LLVM Just-In-Time.

Dans la section 1, nous présentons Padrone, un outil utilisé pour optimiser dynamiquement les programmes et qui offre des fonctionnalités de profilage, d'analyse et d'injection de code. Les sections 2 et 3 présentent la re-vectorisation et la vectorisation des boucles. La section 4 présente les résultats expérimentaux.

0.1 Optimisation dynamique de code binaire

L'optimisation dynamique de code binaire vise à appliquer des transformations d'optimisation au moment du chargement du programme ou pendant son exécution sans accès

au code source ou à toute forme de représentation intermédiaire. Dans cette étude, nous utilisons la plate-forme Padrone [RRC+14], qui intègre plusieurs techniques d'analyse et de manipulation binaires génériques et spécifiques à l'architecture. Les services de Padrone peuvent être divisés en trois grandes catégories: 1) profilage, 2) analyse, et 3) optimisation, qui sont décrits brièvement dans les paragraphes qui suivent.

Le composant *profilage* utilise les appels système Linux perf_event afin d' accéder aux compteurs de performance matériels. La technique d'échantillonnage à bas coût fournit une distribution des valeurs du compteur programme, qui peuvent ensuite être utilisées pour localiser les points chauds, autrement dit, les régions de code fréquemment exécutées.

Le composant *analyse* accède au segment text du processus, désassemble et analyse le code binaire pour créer un Graphe de Flot de Contrôle (GFC) et localise les boucles dans les fonctions dès que possible. La re-vectorisation se produit sur ce GFC reconstruit. Ce dernier est encore élevé dans la representation intermédiaire pour l'auto-vectorisation. Padrone a les mêmes limitations que les autres décodeurs du jeu d'instructions x86. En particulier, le désassemblage du code binaire n'est pas toujours réalisable. Les raisons incluent les sauts indirects, le code obscurci, ou la présence d'octets étrangers, c'est-à-dire de données au milieu du segment text (code).

Le composant *optimisation* fournit des fonctionalitées de manipulation de code binaire, ainsi qu'un mécanisme d'injection de code-cache. Dans cette étude, ce composant a été utilisé principalement pour régénérer le code à partir de la représentation interne d'une boucle vectorisée après transformation. Ce composant s'occupe du calcul des adresses en mémoire relatives après le repositionnement du code optimisé dans le tas.

Padrone fonctionne comme un processus distinct, qui interagit avec le programme cible grâce à l'appel système `ptrace` fourni par le noyau Linux et à d'autres fonctionnalités Linux telles que le système de fichiers `/proc`. L'optimiseur est capable de manipuler les applications en cours d'exécution, ce qui ne nécessitent pas de recommencer l'exécution du programme depuis le début. Il surveille l'exécution du programme, détecte les points chauds, sélectionne la boucle vectorisée SSE ou scalaire et fournit un GFC correspondant. Après une nouvelle vectorisation ou une auto-vectorisation, Padrone est responsable de l'injection d'une version optimisée et de la réorientation de l'exécution.

0.2 Brève introduction à la vectorisation

Un opérande vectoriel est capable de stocker un tableau d'éléments de données indépendants du même type. La taille d'un vecteur varie en fonction de la technologie. Par exemple, un opérande SEE (Intel) mesure 128 bits, ce qui signifie qu'il peut être composé de quatre nombres flottants simple précision (32 bits) ou de deux nombres flottants double précision (64 bits). Une instruction vectorielle est une instruction capable d'effectuer simultanément la même opération sur chaque élément du tableau de données stocké dans l'opérande vectoriel.

La Figure 1 illustre le pseudo-code des versions séquentielles et vectorisées d'une

Version scalaire

```
int A[], B[], C[];
...
for(i=0; i<n; i++) {
  a = A[i];
  b = B[i];
  c = a+b;
  C[i] = c;
}
```

Version vectorisée

```
int A[], B[], C[];
...
/* Boucle vectorisée */
for(i=0; i<n; i+=fv) {
  va = A[i..i+fv[;
  vb = B[i..i+fv[;
  vc = padd(va, vb);
  C[i..i+fv[ = vc;
}
/* épilogue */
for( ; i<n; i++) {
  /** Les itérations restantes
   ** dans le cas où n n'est pas
   ** un multiple de fv
  **/
}
```

Figure 1: Exemple de vectorisation

addition de matrices C = A + B. Les variables va, vb, et vc désignent des vecteurs, padd designe une addition vectorielle. Le nombre d'éléments traités en parallèle est le facteur de vectorisation (fv). Dans l'exemple de la Figure 1, les éléments sont de type int, c'est-à-dire de 32 bits de large, les vecteurs SSE sont de 128 bits, le facteur de vectorisation est fv = 128/32 = 4. Puisque chaque iteration traite simultanément quatre éléments, le nombre d'iterations est divisé par quatre.

0.3 La re-vectorisation du code binaire

Notre objectif est de transformer les boucles vectorisées en SSE en versions AVX. Étant donné que le binaire est déjà vectorisé, nous ne sommes concernés que par la conversion des instructions de SSE en AVX, et des vérifications qui garantissent la légalité de la transformation. Nous nous concentrons sur les boucles internes, avec des accès de mémoire contigus. L'avantage principal de l'utilisation du jeu d'instructions AVX par rapport à SSE est que la taille de l'opérande vectoriel double de 128 bits à 256 bits. Par conséquent, le nombre d'éléments de données qui peuvent entrer dans l'opérande SSE double avec AVX. Par conséquent, dans le scénario parfait, une boucle AVX exécute la moitié du nombre total d'itérations d'une boucle SSE.

La Figure 2 montre les versions vectorielles SSE et AVX du pseudo-code de la boucle de la Figure 1. Dans la version originale en SSE, la première instruction (ligne 2) lit 4 éléments du tableau A en $xmm0$. La deuxième instruction (ligne 3) rajoute en parallèle 4 éléments de B aux valeurs qui résident dans $xmm0$, et la troisième instruction (ligne

(a) Version originale en SSE	(b) Version re-vectorisé (en AVX)

<pre>1 .L2:
2   movaps A(rax),xmm0
3   addps   B(rax),xmm0
4   movaps xmm0,C(rax)
5   addq $16,rax
6   cmpq $4096,rax
7   jne   .L2</pre>
	<pre>.L2:
	   vmovaps A(rax),xmm0
	   vinsertf128 1,A(rax,16),ymm0
	   vaddps B(rax),ymm0
	   vmovaps ymm0,C(rax)
	   addq $32,rax
	   cmpq $4096,rax
	   jne .L2</pre>

Figure 2: Exemple de conversion des instructions du SSE vers l'AVX

4) enregistre les résultats dans le tableau C. Le registre *rax* représente la variable d'induction. Cette dernière est alors incrémentée de 16 octets : la largeur des vecteurs SSE (ligne 5). La conversion du corps de la boucle est relativement simple. Mais pour garantir la légalité de la transformation, une certaine comptabilité est nécessaire.

0.3.1 Conversion des instructions SSE en équivalents AVX

Pour simplifier l'implémentation du processus de la conversion, nous avons construit une table qui mappe une instruction SSE à une ou plusieurs instructions équivalentes en AVX. Il est nécessaire que l'instruction AVX soit capable de faire le double travail de l'instruction SSE tout en conservant la même sémantique. Par exemple, *movaps* est une instruction de mouvement de données SSE qui déplace 16 octets de données de/vers la mémoire et qui sont alignées sur des limites de 16 octets. Lors de la conversion des instructions, lorsque les données à déplacer appartiennent à un alignement de 32 octets, une instruction qui déplace le vecteur de 32 octets de données à la fois est sélectionnée. En supposant que le tableau C, dans la Figure 2.a est aligné sur 32 octets, *movaps* de la ligne 4 dans la version SSE est converti en *vmovaps* (ligne 5) dans l'AVX résultant. Par conséquent, les registres *xmm* de taille 128 bits sont convertis à leurs equivalents *ymm* de plus grandes tailles (258 bits).

0.3.2 La contrainte d'alignement

Une instruction SSE peut avoir plusieurs alternatives en AVX. Par exemple, lorsque les données à déplacer sont alignées sur 16 octets, elles sont remplacées par deux instructions. L'instruction *vmovaps* nécessite que les données soient alignées sur 16 octets et ne déplace que la moitié inférieure de l'opérande vectoriel. L'instruction $vinsertf128$ déplace la moitié supérieure. Nous supposons que le tableau A est aligné sur 16 octets dans la Figure 2.a. Par conséquent, l'instruction de la ligne 2 dans la version SSE est traduite en les instructions des lignes 2 et 3 dans l'AVX résultant.

0.3.3 L'ajustement du compteur

Dans certains cas la variable d'induction sert à déterminer l'accès à la mémoire à chaque itération. Par exemple, dans la Figure 2.a, le registre *rax* dans les lignes 2, 3 et 4, est utilisé comme indice par rapport aux adresses de base des tableaux A, B et C à partir desquels les lectures et les écritures doivent être effectuées. Étant donné que les instructions de mouvement de données SSE fonctionnent sur des opérandes de 16 octets, le registre est incrémenté de 16 octets à la ligne 5 (Figure 2.a). Lorsque le code est traduit en AVX, la taille de l'opérande devient 32 octets. Par conséquent, l'instruction responsable de l'incrémentation de l'index est adaptée à la version AVX à la ligne 5 (Figure 2.b).

0.3.4 L'ajustement du nombre total d'iterations

Une itération AVX doit correspondre à deux itérations SSE. Par conséquent, le nombre total d'itérations devrait être réduit de moitié ou le compteur à l'intérieur de la boucle devrait doubler la valeur d'incrémentation. La conversion fait face à deux scénarios, le nombre total d'itérations est connu au moment de la compilation ou seulement connu à l'exécution.

0.3.4.1 Nombre total d'iterations connu au moment de la compilation

Le nombre d'itérations n de la boucle avec les instructions SSE peut être pair ou impair. Lorsque $n = 2 \times m$, l'optimiseur traduit simplement les instructions SSE en instructions AVX et double le rythme de changement du compteur afin que la boucle transformée itère m fois (ce qui se fait pendant le réglage de la variable d'induction pour l'accès à la mémoire). Cependant, lorsque le nombre d'itérations est impair, $n = 2 \times m + 1$, le code transformé est composé de deux blocs de base: primo, la boucle qui exécute les instructions AVX m fois. Secundo, les instructions SSE qui exécutent la dernière itération SSE.

0.3.4.2 Le nombre total d'iterations seulement connu à l'exécution

Supposons qu'une boucle séquentielle exécute un ensemble d'instructions n fois, et sa version vectorisée exécute des instructions équivalentes dans des itérations telles que : $n = m \times fv + r$, où m est le nombre d'iterations vectorisées et r est le nombre d'itérations séquentielles restantes ($r < fv$). Dans ces circonstances, le compilateur génère une boucle vectorisée qui itère m fois, et une séquence qui itère r fois. Les valeurs de m et r sont calculées au plus tôt, avant que le contrôle ne soit donné à l'une des boucles ou les deux selon la valeur de n découverte au moment de l'exécution. Nous modifions le code responsable de déterminer la valeur de m et r au moment de l'exécution en doublant la valeur de fv, pour que ces valeurs seront adaptées à la version AVX.

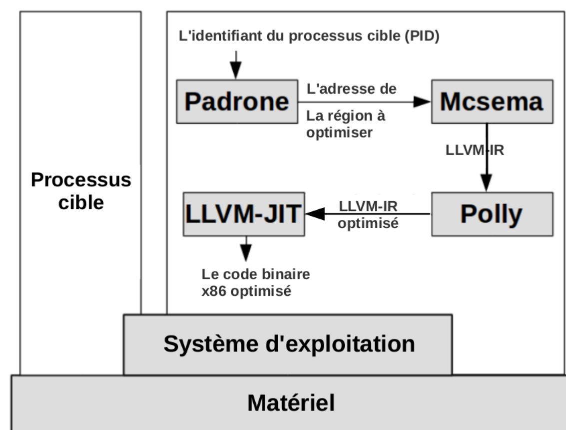


Figure 3: Le processus de la vectorisation automatique

0.4 Vectorisation automatique

Certaines applications contiennent des boucles scalaires qui n'ont pas été vectorisées par le compilateur, même si elles sont vectorisables. Une des raisons peut être que le code source a été compilé pour une architecture qui ne prend pas en charge les instructions SIMD. Une autre raison est que certains compilateurs ne sont pas capables de vectoriser certains types de codes [15], car ils n'intégrent pas d'analyse avancée des dépendances de données et de capacités de transformation de boucles, comme le modèle polyédrique [12]. Dans cette section, nous élargissons la portée de l'optimisation en auto-vectorisant les boucles scalaires. Nous utilisons le logiciel libre McSema [16], qui élève le code binaire vers la représentation intermédiaire (RI) de LLVM afin que les optimisations puissent être effectuées. De la RI, il est possible de déléguer le fardeau de la vectorisation à un autre outil, Polly [13], mettant en œuvre des techniques du modèle polyédrique pour vectoriser les boucles. Enfin, nous compilons la RI vectorisée en binaire en utilisant la compilation à la volée de LLVM (JIT LLVM). La Figure 3 résume le processus de vectorisation automatique.

0.4.1 McSema

McSema est un logiciel de décompilation dont la fonctionnalité est de traduire le code binaire vers la représentation intermédiaire de LLVM. McSema se compose de deux outils, comme on le voit sur la Figure 4 : Le bin_descend prend comme arguments d'entrée l'executable et le nom de la fonction cible, il construit un CFG de la fonction en assembleur, convertit ses structures de données vers un format stockable sur disque, et écrit ces dernières en la mémoire. L'outil cfg_to_bc lit les données du disque, reconstruit le CFG et traduit les instructions en représentation intermédiaire de LLVM. Nous avons modifié l'outil bin_descend pour prendre en compte l'adresse de la fonction fréquemment exécutée fournie par Padrone au lieu de son nom. En outre, nous avons

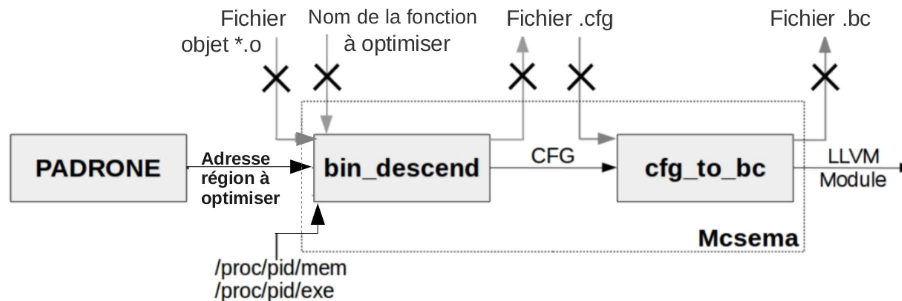


Figure 4: L'intégration de McSema dans Padrone

aussi modifié McSema de sorte que le CFG produit par bin_descend soit transmis directement à cfg_to_bc au lieu de l'écriture et la lecture de la mémoire. La Figure 4 montre l'intégration de Padrone avec McSema.

0.4.2 La vectorisation et la compilation à la volée

Polly [13] est une infrastructure d'optimisation de boucle statique pour le compilateur LLVM, capable d'exposer les opportunités de parallélisme pour les nids de boucles conformes au modèle polyédrique : les bornes de la boucle et les accès à la mémoire doivent être affines. L'objectif principal est de vectoriser le code séquentiel. Tout d'abord, des passes de canonisation sont exécutées, ce qui transforme la RI en une forme appropriée pour Polly. Deuxièmement, Polly détecte les sous-graphes des boucles avec un flot de contrôle statique. Troisièmement, ces régions sont abstraites dans le modèle polyédrique. Enfin, nous utilisons une méthode fournie par la passe d'analyse de dépendances pour vérifier si la boucle est parallèle. Nous avons développé un algorithme qui vérifie si les accès aux données sont consécutifs. Ensuite, nous transformons les instructions en instructions vectorielles et compilons à la volée à l'aide du compilateur LLVM juste-à-temps.

0.4.2.1 Le problème des variables globales

La décompilation du binaire en représentation intermédiaire consiste à générer les instructions et les adresses mémoire sur lesquelles elles opèrent. Par conséquent, McSema déclare des tableaux sur lesquels les instructions s'exécutent. Cependant, la compilation de la représentation intermédiaire génère des instructions et des allocations de tableaux dans l'espace de traitement de l'optimiseur. Par conséquent, l'injection de ce code dans le processus cible entraînerait de mauvaises références d'adresse.

0.4.2.2 Le marquage des variables globales

La solution adoptée consiste à marquer les opérandes soulevés par McSema avec leurs adresses physiques dans le binaire d'origine de sorte que, pendant la compilation, les

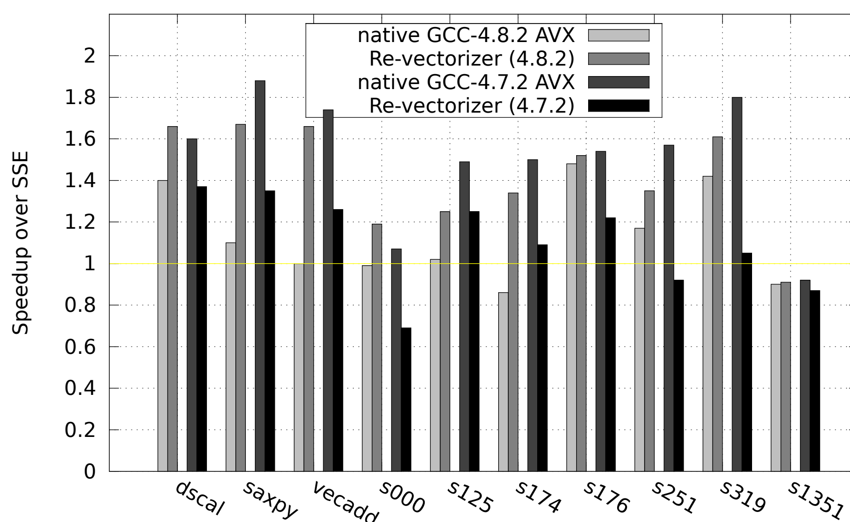


Figure 5: Les résultats de la re-vectorisation

adresses récupérées soient encodées dans les instructions binaires générées. La compilation de la représentation intermédiaire passe par plusieurs étapes. À chacune d'elles, les adresses récupérées par McSema sont transférées jusqu'à ce que les instructions générées soient encodées avec les adresses réelles de leurs opérandes.

0.5 Résultats

0.5.1 Re-vectorisation

Nos expériences ont été menées sur une station Linux Fedora 19 à 64 bits avec un processeur Intel i7-4770 Haswell cadencé à 3,4 GHz. Nous avons traité deux types de benchmarks. Le premier type consiste en quelques boucles vectorisables fabriquées à la main. Le deuxième type est un sous-ensemble de la suite TSVC [MGG + 11].

0.5.1.1 Résultats

Nous mesurons les accélérations du re-vectoriseur AVX obtenues par rapport à la vitesse d'exécution de la boucle SSE initiale. Ensuite, nous les comparons au accélérations natives obtenues à partir des vitesses d'éxécution des codes AVX et SSE générées uniquement par deux versions de GCC : GCC-4.7.2 et GCC-4.8.2. Dans le cas de notre re-vectoriseur, nous rapportons comment il se compare au compilateur natif ciblant AVX. Ces chiffres sont représentés graphiquement à la Figure 5. À titre d'exemple, la première ligne (dscal) montre que le code AVX produit par GCC est supérieur de 1,4× à la version SSE. Le code produit par notre re-vectoriseur s'exécute 1.66× plus

rapidement que la version SSE, soit une amélioration de 19% par rapport à la version AVX native.

Avec GCC-4.8.2, notre re-vectoriseur est capable d'améliorer les performances des boucles jusqu'à 67%. Étonnamment, nous continuons à surpasser GCC pour AVX, jusqu'à 66% dans le cas de vecadd. L'une des raisons est que lors du ciblage d'AVX, GCC-4.8.2 génère un prologue de la boucle parallèle pour garantir l'accés aligné d'un des tableaux. Malheureusement, la boucle ne profite pas de l'alignement et s'appuie sur des accès de mémoire non alignés. Lors du ciblage de SSE, il n'y a pas de prologue et la boucle repose sur des accès de mémoire alignés sur 16 octets. En fait, le code AVX généré par GCC-4.7.2 est plus simple, sans prologue, et similaire à notre propre génération de code, et les performances correspondantes sont également corrélées.

Avec GCC-4.7.2, notre re-vectoriseur dégrade parfois la performance globale par rapport au code SSE. Nous observons que cela se produit lorsque le même registre (ymm0) est utilisé à plusieurs reprises dans le corps de la boucle pour manipuler différents tableaux. Cela augmente considérablement le nombre d'écritures partielles dans ce registre, un phénomène connu provocant des pénalités de performance [Int14a]. Ceci est particulièrement vrai dans le cas de s125. En dépit de ces résultats, puisque notre optimiseur fonctionne à l'exécution, nous avons toujours la possibilité de revenir au code d'origine, ce qui limite la pénalité à un délai court (et réglable). Contrairement à GCC-4.8.2, la performance du binaire converti est systématiquement pire que l'AVX natif. Cela est prévu car le compilateur a souvent la capacité de forcer l'alignement des tableaux à 32 octets selon les besoins. Puisque l'optimiseur ne traite que de le code compilé en SSE, nous n'avons que la garantie de l'alignement sur 16 octets. Dans cette situation, la conversion en AVX donne le même nombre d'instructions d'accès à la mémoire que dans le SSE original (comme si la boucle avait été déroulée), mais le nombre d'instructions arithmétiques est réduit de moitié. Notez cependant que nous améliorons le code SSE dans de nombreux cas, et nous avons la capacité de revenir au code original sinon.

0.5.2 Vectorisation automatique

Les expériences ont été réalisées à l'aide d'une machine équipée d'un processeur Intel Core i5-2410M basé sur l'architecture Sandy Bridge cadencée à 2,3 GHz. En outre, les benchmarks ont été compilés avec GCC 4.8.4 avec les options -O3 -fno-tree-vectorize qui désactivent la vectorisation et maintiennent le reste des optimisations.

0.5.2.1 Résultats

Nous évaluons les performances de la vectorisation en mesurant les accélérations des boucles optimisées pour SSE et AVX contre les boucles non vectorisées. Les résultats sont présentés à la Figure 6. Tout d'abord, nous observons que l'auto-vectorisation améliore la performance par le facteur vectoriel. Dans le cas de s1351, la version SSE surpasse sa concurente scalaire de 1,95×. En ce qui concerne dscal, la version AVX s'execute 3,70× plus rapidement. Deuxièmement pour saxpy, nous obtenons même

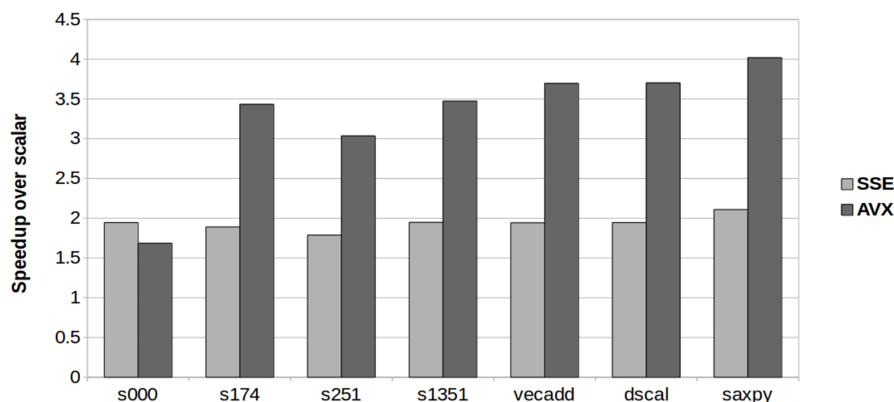


Figure 6: Les résultats de la vectorisation automatique

une accélération super-liénaire : 2,10× plus rapide. Nous comparons cette valeur à l'accélération obtenue avec GCC 4.8.4 qui est de 2,04×. Cette petite différence est due aux instructions générées par GCC 4.8.4 et le compilateur JIT LLVM qui ne sont pas similaires.

0.6 Conclusion

La tendance actuelle est de progresser vers l'intégration de plusieurs coeurs et des unitées vectorielles dans les processeurs. Cela nous a orienté à choisir les transformations liées à la vectorisation. Ces optimisations ont la capacité d'accélérer l'exécution du programme afin de compenser les frais de la transformation pendant l'exécution et d'obtenir une performance significative. Les optimisations adoptées incluent la re-vectorisation des boucles déjà vectorisées et la vectorisation automatique des boucles scalaires.

D'une part, une table de traduction contient des instructions SSE avec leurs correspondances en AVX utilisée pour re-vectoriser les boucles tout en garantissant la légalité de la transformation. En outre, les informations dynamiques sont exploitées pour améliorer les performances. Par exemple, lorsque les adresses des tableaux sont découvertes au moment de l'exécution, elles sont utilisées pour choisir les instructions alignées au lieu des instructions non alignés lorsque cela est possible. Les expériences ont montré que les coûts sont minimaux et que les accélérations sont conformes à celles d'un compilateur natif ciblant AVX.

D'autre part, les boucles scalaires sont auto-vectorisées. La méthode décompile le binaire vers sa représentation intermédiaire. À ce niveau, qui masque les détails de la machine cible, nous avons montré qu'il est possible d'optimiser les boucles en utilisant le modèle polyédrique. Enfin, elles sont compilées à l'aide du compilateur LLVM et injectées dans le processus cible. Les résultats montrent l'efficacité de l'approche et les accélérations sont significatives.

Chapter 1

Introduction

The expansion of the computing market is strongly coupled with complex applications developed for research and industrial purposes; such as, modeling and simulation algorithms in various fields like weather forecast, seismic and molecules analysis, aircraft design to overcome friction, etc. Such programs require high performance machines. The market need propel the computer architecture and compilation communities toward the enhancement of performance and better utilization of computers' resources.

This chapter is composed of three parts. First, it presents the CPU design trend which metamorphoses toward integrating multi-cores and vector processing units on single chips. Second, it defines the thesis problem statement by spotting the light on the failure of full use of CPUs' features. This issue mainly stems from the principle of backward compatibility adopted by CPU industries which guaranties scalability of programs' execution in newer CPUs; albeit, their prior compilation targeted older ones of the same familly. Third, it proposes a runtime-optimization-based solution which metigates vector processing under-utilization. The decision of choosing optimizations geared toward the subclass of parallel computers are: First, the current widespread and continuous development of these kinds of architectures by various processors vendors. Second, the return-on-investment which ensures accelerating the running program to cover the expense of runtime fine tuning of the binary and provide significant speed-up with regards to non-optimized version.

1.1 Context

Since the 1970s the designers have exhausted several areas of research to enhance CPU performance. For instance, instruction pipelining, which breaks the instruction cycle into multiple stages permiting them to operate simultaneously, leads to a higher throughput. Moreover, the instruction Level Parallelism (ILP) continued cherishing improvements thanks to the passage from the in-order to out-of-order execution which allows the execution of instructions in different order while preserving data dependence. As a consequence, it minimizes the pipeline stalls by giving instructions with no dependence higher priority for execution regardless of their order in the instrunction's

stream. Furthermore, on-chip cache, that integrates the first and sometimes the second cache level into the processor, bridges the gap between the unequilibrated growing speed paces of processor and main memory. Undeniably, data cache allows low latency access to memory operands compared to hundreds of cycles latency for the main memory. Besides, between the 1990s and the beginning of the 2000s CPU designers concern was increasing the processor clock-speed.

The continuous of improvement of some of the former CPU design areas faced a dead-end. Enthusiasm geared toward ILP improvements was reoriented toward multicore designs. Although parallel computing was already a research area prior to the 2000s, a growing preoccupation to industrialize these kinds of architectures, that allow Thread Level Parallelism (TLP) programming paradigms, becomes an alternative to pushing the ILP performance into its edge.

Two main reasons lay behind this: (1) A research effort [HP02] in finding optimimum pipeline depth confirm that their increase sublinearly increases the frequency; however, deepening beyond some treshold results with counter-productive effects. (2) Increasing the clock-speed is jeopardized by the physical limit of silicon better known as the power-wall. This issue is a good example of *the law of conservation of energy* which says:

> Energy is neither created nor destroyed

Indeed, the energy consumed by the integrated circuit is dissipated in the form of heat. As a matter of fact, rising the frequency results with a significant power consumption of complementary metal-oxide-semiconductor (CMOS) transistors in the CPU [KAB+03, Mud01]. As a consequence, over-clocking leads to a higher power consumption which causes a thermal over-stress, which by itself can cause CMOS components to fail. It is crystal clear that the statistical failure rate of processors should not exceed some treshold to lessen the expenses of budgets for product's insurance and satisfy the expectations of clients. These reasons orient vendors toward parallel processing design which is guaranteed by the estimation of Moore's law.

CPU manufacturers enhance the computational performance by integrating vector units into their chips. For instance, in 1997 Intel incorporated MMX instructions to its Pentium MMX, and in 1999 they added Streaming SIMD extensions (SSE) that include floating point vector operations to their subsequent processors starting from the shipment of Pentium III. The vector unit extends the processors with instructions that operate simultaneously on multiple data. The constraint on vectorization is independence of data in the vector. Vectorization promotes CPU performance based on data parallelism and by exploiting instruction-level parallelism (ILP). On one hand, the data parallelism allows to operate on multiple data at the same time, which reduces the number of instructions; hence, it cuts-down the amount of time spent in the fetch and decode stages in the pipeline. On the other hand, the SIMD instructions can also exploit ILP as its counterparts instructions that operate on individual data. Moreover, they are flexible to undergo execution in architecture with deep pipeline stages; however, the latter might lead to go back to square one which is an increase in the clock-speed and hence the power-wall issue.

1.2 Problem definition

Automatic code optimizations have traditionally focused on source-to-source and compiler Intermediate Representation (IR) transformation tools. Sophisticated techniques have been developed for some classes of programs, and rapid progress is made in the field. However, there is a persistent hiatus between software vendors having to distribute generic programs, and end-users running them on a variety of hardware platforms, with varying levels of optimization opportunities. The next decade may well see an increasing variety of hardware, as it has already started to appear particularly in the embedded systems market. At the same time, one can expect more and more architecture-specific automatic optimization techniques.

Unfortunately, many “old” executables are still being used although they have been originally compiled for now outdated processor chips. Several reasons contribute to this situation:

- commercial software is typically sold without source code (hence no possibility to recompile) and targets slightly old hardware to guarantee a large enough base of compatible machines;
- though not commercial, the same applies to most Linux distributions[1] – for example Fedora 16 (released Nov 2011) is supported by Pentium III (May 1999)[2];
- with the widespread cloud computing and compute servers, users have no guarantee as to where their code runs, forcing them to target the oldest compatible hardware in the pool of available machines;
- use of compilers with less capabilities of performing optimizations due to various reasons that can be limitation of use for small software companies, and the prices of license, etc.

To sum up, all of the binaries shipped or used in the presented circumstances share a common similarity which is the use of the CPUs’ design constraint that is backward-compatibility. This latter, mainly contribute in the under-utilization of processors features which correlate with low performance.

1.3 Proposed solution

All this argues in favor of binary-to-binary and binary-to-IR-to-binary optimizing transformations. Such transformations can be applied either statically, i.e., before executing the target code, or dynamically, i.e., while the target code is running. Dynamic optimization is mostly addressing adaptability to various architectures and execution environments. If practical, dynamic optimization should be preferred because it eliminates

[1] with the exception of Gentoo that recompiles every installed package

[2] `http://docs.fedoraproject.org/en-US/Fedora/16/html/Release_Notes/sect-Release_Notes-Welcome_to_Fedora_16.html`

several difficulties associated with static optimization. For instance, when deploying an application in the cloud, the executable file may be handled by various processor architectures providing varying levels of optimization opportunities. Providing numerous different adapted binary versions cannot be a general solution. Another point is related to interactions between applications running simultaneously on shared hardware, where adaptation may be required to adjust to the varying availability of the resources. Finally, most code optimizations have a basic cost that has to be recouped by the gain they provide. Depending on the input data processed by the target code, an optimizing transformation may be or not profitable.

In this work we show that it is possible, at runtime, to automatically convert SSE-optimized binary code to AVX as well as (2) auto-vectorize binary code whenever profitable. The key characteristics of our approach are:

- we apply the transformation at runtime, i.e. when the hardware is known;
- we only transform hot loops (detected through very lightweight profiling), thus avoiding useless work and minimizing the overhead;
- for SSE loops, we do not implement a vectorization algorithm in a dynamic optimizer. Instead, we recognize already statically vectorized loops, and convert them to a more powerful ISA at low cost;
- for scalar (unvectorized) loops, we integrated some open source frameworks to lift the binary code into the IR form of the LLVM compiler, auto-vectorize them on the fly, and compile them back using the LLVM Just-In-Time (JIT) compiler.

The thesis is organized as follow: Chapter 2 reviews the necessary background on key technologies at play: Software and hardware-based profiling techniques, binary optimization tools, and Padrone the platform developed within the team to profile, analyze, and inject binary. Chapter 3 presents our first contribution of translating on-the-fly SSE-optimized binary code into AVX. Chapter 4 addresses our second contribution of runtime auto-vectorizing of originally unvectorized loops. Our experiments are presented in chapter 5. Chapter 6 discusses related work. Chapter 7 concludes and draws perspectives.

1.4 List of publications

The list of published contributions are:

- N. Hallou, E. Rohou, P. Clauss. Runtime Vectorization Transformations of Binary Code. International Journal of Parallel Programming, Dec 2016, Springer US.
- N. Hallou, E. Rohou, P. Clauss, A. Ketterlin. Dynamic Re-Vectorization of Binary Code. International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), Dec 2015, IEEE, Agios Konstantinos, Greece.

- E. Riou, E. Rohou, P. Clauss, N. Hallou, A. Ketterlin. PADRONE: a Platform for Online Profiling, Analysis, and Optimization. International workshop on Dynamic Compilation Everywhere (DCE), Jan 2014, Vienna, Austria.

Chapter 2

Dynamic Binary Optimization

Dynamic binary optimization aims at applying optimization transformations with no access to source code or any form of intermediate representations during program's load or execution times. Online-profiling helps to understand the dynamic behavior of a process and to identify its bottlenecks that might not be detected by a static compiler. In this research, performed optimizations are based on 80-20 rule, which states that 80% of the execution time is spent in 20% section of code. In other words, transformations are confined only to the frequently executed sections of code identified by profiling and which we refer to as hot-codes.

This chapter discusses three main topics. It explains the use of dynamic binary translation as a software-based mechanism for code diagnosis and highlights its major drawbacks in the dynamic context. It suggests an alternative method that relies on the hardware performance counters. Finally, it presents PADRONE [RRC$^+$14], a framework developed within the team, which enbales the detection of process bottlenecks [1] and carries out code transformations.

2.1 Software-based profiling

One of the main objectives is to identify the regions of program appropriate for the optimizations at runtime. Dynamic binary translation is a large field of study that fulfills this aim. As a matter of fact, it refers to the approach of rewriting the binary instructions at the time of execution of a program. It has gained a wide popularity during the last decades as a consequence of its serviceability for a variety of purposes such as program instrumentation, optimization, and security. This section presents how instrumentation is used to observe a running program; and hence, it exposes its bottlenecks.

Instrumentation consists of adding extra instructions to a program to gather data about its execution. For instance, at the binary level, it is possible to count the number of instructions executed by inserting one, which increments one of the CPU's registers,

[1] The bottlenecks are regions of the program whose execution trigger a high occurence of events, such as cache miss, branch misprediction, pipeline stalls, etc, which deaccelerate the progress of execution.

after each instruction of the program. In general, instrumentation can be performed at different stages of the software production such as source, library, and binary levels. In this research, the constraint of not having the source code is a major factor to restrict our investigation on binary level instrumentation methods and limitations. At this level, a wider range of instrumentation granularities are offered which are instruction, basic block, and trace which is nothing but a sequence of instructions which ends by an unconditional control transfer such as a branch, a call, or a return. The literature

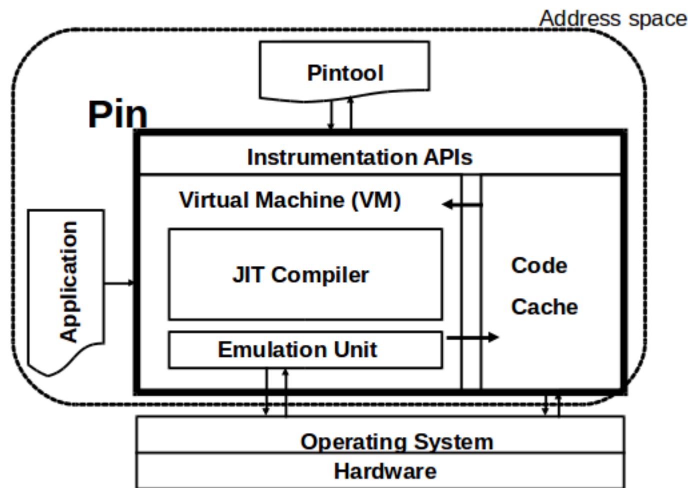


Figure 2.1: PIN's internal software architecture. *Reprinted from Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation (page 3), PLDI, 2005, New York, NY, USA, by ACM*

on instrumentation shows a variety of approaches. Some of them are integrated to host virtual machines and capable of instrumenting OS kernels [BL07]. We explore only the ones that support the user space applications, and which adopt the just-in-time and probe approaches.

2.1.1 Just-In-Time (JIT) approach

Previous researches have documented the just-in-time approach of instrumentation and they proposed both tools and APIs such as PIN [LCM+05], DynamoRio [BGA03], Valgrind [NS03]. The common similarity between these tools is that the target process to be instrumented runs on the top of a virtual machine. Consequentely, both of them share the same address space. This section presents an overview of the design of such approach and highlights its overhead.

Figure 2.1 portrays Pin's internal software architecture. Pin consists of a virtual machine (VM), a code cache and Pintool. The virtual machine orchestrates the execution of the pintool and the recompiled code of the target program. At first, Pin takes control either by launching the target program as an argument to pin or by attaching to a running process, then it instructs the JIT compiler to read a set of instructions

from the target process. This set of instructions is referred to as a trace. Once the trace is read, Pintool decides the insertion points along with the appropriate code to inject. Then, Pin performs a binary-to-binary compilation of the trace, injects it into the code cache and invokes it. The end of the compiled trace has a stub that redirects the execution to the virtual machine so as the next trace is instrumented.

The method comes at a price of performance degradation compared to the execution of the original code on bare-metal. First, alternating the execution of pintool, pin, and the compiled trace involves context switches which require saving and restoring the content of the processor's registers. Second, it is necessary to use registers for counting events such as instructions executed and branch mispredicted. The problem is that the number of general purpose registers is limited, and an interference of registers used by the code to be monitored and the instrumentation instructions is frequent. The instrumentation tools use different techniques to tackle this problem. Register spilling allows saving and restoring the values of registers into and from memory. This technique is used in conjunction with register liveness analysis which allows to choose the best point of insertion to minimize the occurence of spilling data. Register re-allocation is used by other tools to overcome the same issue as long as register allocator does not run out of registers. To alleviate the JIT overheads, whenever a trace is jitted, it is kept in the code cache, and it gets connected to its successor compiled traces. This reduces the recompilation and context switches overheads for previously compiled traces. However, nothing guarantees the use of same registers for event counting between traces, which requires a reconciliation from a trace to another.

2.1.2 Probing approach

The common similarity between tools and APIs that adopt the probe approach such as SPIRE [JYW+03] is that the target process and the one that instruments the code reside on different address spaces. Probing works in a way that a call to a set of instructions refereed to as a base trampoline is inserted in the instrumentation point; hence, the call replaces one or few instructions at that point. The base trampoline resides in the heap and part of its functionality is to conserves the altered bytes in the code section. The base trampoline starts by branching to a mini-trampoline which saves the CPU's registers state, it executes the event counting code, it restores the CPU's state, and branches back to the base trampoline. This latter executes the instructions that were corrupted by the insertion of the call to the base trampoline. The drawback of this method is that the profiled process performs branching and spilling for each event counting which slows down its speed.

2.1.3 Overhead

Research seems to agree that instrumentation suffers from performance problems by imposing a heavy overhead to ensure correct behavior. A study [RAH08] that evaluates the impact of dynamic binary translation on hardware cache performance found that on

average DynamoRio and Pin have 1.7x and 2.5x more instruction/trace cache[2] misses in comparison to the native execution of SpecInt 2006 on a 32 bit machine. This is due to register spilling, the stubs that redirect the execution flow between traces, and the size of the memory image of the instrumented code and the virtual machine on which it executes.

A study that proposes a technique to reduce the overhead of instrumentation developed a prototype PIPA [ZCW08] based on the parallelization of DynamoRio. They claim that they reduced the slowdown of cachegrind and pin dcache respectively from 100x and 32x into 10x. Both of the latters are tools that count the cache events of SPEC CPU 2000 suite of benchmarks on multi-core systems.

2.2 Hardware-based profiling

2.2.1 Hardware performance counters at a glance

Many CPU firms incorporate into their processors hardware performance units. This processor feature first appeared on Intel family of processors and then gained popularity among other RISC processors such as ARM, Power, and SPARC. The unit is nothing but an on-chip sequential circuit. The circuit consists of registers that counts hardware-related activities, also known as hardware events, such as retired instruction, memory accesses, pipeline stalls, and branch predictions etc.

Most modern processors provide registers that count the occurrence of hardware events. For instance, Intel provides Model Specific Registers that serve also for debugging. They include three categories: First, the Time Stamp Counter TSC which increments at each cycle providing the number of cycles. Second, the Control and Event Select Register CESR that tells the processor the specific event to be counted. Third, the CTR which saves the event value collected.

The CESR is first filled with an appropriate value that selects one of the counters CTR0, CTR1, depending on the number provided by the architecture, and sets up the circuit to count a certain event; for instance, a branch prediction. For the user, this request can be done through a system call which sets up the CESR through a device driver. As a function of the input value in CESR, the circuit isolates the branch miss as input, and rejects the remaining input such as data read operations, cache misses, etc. Now on, each time the processor misses the prediction of a branch, the value in the selected CTR increments. The retrieval of the value in the CTR register is also done through a system call.

The processor manufacturer gives a set of instructions that manipulates these registers. Some of them execute in the kernel mode while others in the user mode as well. For instance, the RDMSR and WRMSR execute in the privileged mode; consecutively, it loads a 64 bit MSR into EDX:EAX, and it moves the content of EDX:EAX into an MSR. Conversely, the RDPMC, which reads a selected performance counter, executes

[2] A micro-architecture structure which stores streams of the fetched and decoded instructions [RBS96]

in non-privileged mode without incurring the expense of a system call. Finally, RDTSC reads the value in the time stamp counter.

2.2.2 Useful features of hardware performance counter for profiling

Hardware profiling supports both counting and sampling modes. The counting mode sets up the event to be monitored before the execution of the workload, and reads the counter at the end of its execution. As a consequence, it consumes the CPU resources with minimal overhead precisely preceding and succeeding the workload execution. The sampling mode sets up the architectural statistics to be monitored and captures the profile data multiple times. The time-based one interrupts the running program at regular times. The event-based one relies on triggering the interrupt based on the occurrence of a specific event for a predefined number of times in order collect a sample.

The hardware monitoring unit focuses on the operations of the processor regardless of the application or thread that executes. In other words, it does not distinguish between processes being executed. The unit is assisted by the operating system that saves and restores the data of the registers during the context switch so as it isolates the performance information between processes and threads.

Hardware performance units enable the simultaneous measurement of multiple events. Multiplexing refers to the case when the number of events exceeds the number of counters that the hardware affords. This can be possible by the assistance of the operating system which virtualizes the counters so as the measurements can be done concurrently. Each of the events are counted for some period of time and switched out with the waiting events in the queue. In [MC05], they propose an algorithm that statistically estimate occurrences of events based on the concurrent collection of data which improves the accuracy in a timesharing of counters by events.

Operating systems allow to access hardware counters. Most known operating system interfaces are perfmon [Era06], perfctr [Pet99], perf_event [per]. The implementation of these interfaces is important for reasons mentioned earlier such as the event counting by the hardware does not differentiate between processes and multiplexing.

2.2.3 Overhead vs accuracy

Several studies have been conducted to measure the overhead of performance counters. In [BN14], they measure the cost for both counting and sampling modes. Their experiments come into a conclusion that the counting mode cost is usually negligible. However, the overhead of the sampling mode is a function of multiple variables such as the events selected, the frequency of sampling since it requires the execution of the interrupt handler in the kernel mode, the number of events to be counted since it requires virtualization when the number of events exceeds the number of physical counters, and finally the number of processes to be concurrently monitored. Their research states that the overhead can reach as much as 25%. At the same time, when these factors are minimal the overhead is negligible.

In [Wea15], they compare the overhead of the diverse implementations that manage

the PMU among perf_event, perfmon2, and perfctr. They measure the overhead of counting events for concurrent processes. It reveals that the overhead can increase up to 20% for perf_event due to saving the MSRs at the context switch and only 2% for perfctr and perfmon2 which are implemented for older Linux versions.

Limit [DS11] is an implementation that bypasses the operating system to set and read event counters. They claim that their tool is 23x faster than perf_event and consumes 12 ns per access.

In [intel book], they state that the event counters should be used with caution because of the non-accurate information that they can provide. The reason behind is that the delay between the overflow of the event and the delivery of the interrupt is not immediate. For example, suppose a user sets the HPC to an event-based mode to identify a loop suffering from branch misprediction. Once the predefined treshhold of misprediction event has been reached, there is a time delay to deliver the interrupt during which the loop might have exited, and a second one has started execution. In this case, the non-awareness might lead the user to handle the event by recording the current Program Counter (PC), which reflects an address within the second loop; even though, it is the first loop which overflows the misprediction event.

In [MTS02], they highlight the possibility of identifying the instruction that causes the event with static analysis; however, for modern out-of-order processor it is difficult or even impossible due to the complex design of the architecture.

Moreover, sampling involves collecting data at time intervals which extrapolates the results. Therefore, it is up to the user to trade-off between speed and accuracy.

2.3 Padrone infrastructure

Padrone is a library that provides a dynamic binary optimization API. It can be linked to a program that performs optimizations. Its main functionalities, portrayed in Figure 2.2, are: first, profiling which allows to observe the dynamic behavior of a program to be optimized; second, analysis which constructs an information from the profiled data, for instance the addresses of frequently executed sections of code, or the spots of code that generate caches misses, etc; third, injection of code that allows replacing some original function with an optimized version. In a similar way as debuggers, Padrone allows to attach and detach to a running process. This capability is required in the profiling and code injection phases.

2.3.1 Profiling

The profiling functionalities of PADRONE make use of services provided by the perf_event Linux kernel interface. To interact with the hardware performance unit resources, the kernel interface provides perf_event_open system call. This call allows to set either the counting or the sampling mode. Since we are interested in dynamic optimizations, we focus on the sampling mode.

The perf_event_open system call allows to choose the kind of event the user wants to monitor, the process PID to which the monitoring process will attach, and the

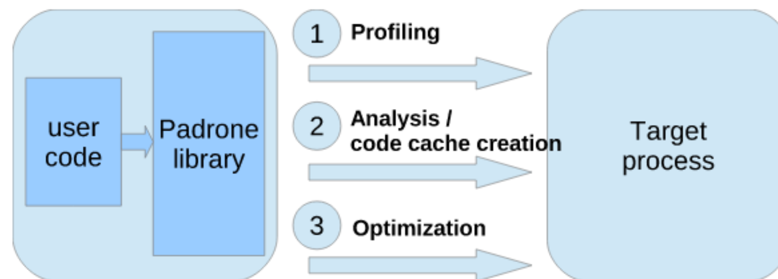


Figure 2.2: Padrone's functionalities.

sampling frequency. It returns a file descriptor that allows to read the event sampling statistics from a ring-buffer in the kernel space. The ring buffer can be mapped to the user space with mmap system call. This allows to share the buffer between the user and kernel spaces to avoid the copy overhead. When a sample is written into the buffer a wake up signal is sent to the user program. It can be captured by a poll, a select, or a handler of SIGIO signal.

2.3.2 Analysis

The dynamic optimization targets coarse grain structures. Specifically, we are interested in optimizing most frequently executed regions of the code. The profiling step collects samples with the address of the instruction being executed at the moment of sampling. The analysis part transforms these data into a histogram of addresses and their relative occurrence.

Each of the sample carries the related call chain as depicted in Figure 2.3. The yellow square contains code segment. The grey square is augmented with series of calls information, that lead to the instruction at the moment of the sample's capture and which resides in the address 4006bc in the code section. Padrone uses the call chain array to identify the beginning of the function that englobes the sampled instruction. The array contains the address of the successor instruction with regards to the caller, which is 400724. Padrone retrieves the address of the hot-function from the operand of the previous instruction (caller) at the address 40071f. The operand has the value 40069c which is the address of the function `foo`.

In order to find the end of the function, Padrone constructs a CFG starting from the beginning of the function. It starts disassembling from the first instruction of the function, builds the basic blocks, by exploring all paths until it reaches a return instruction.

Padrone has the same limitations as other binary re-writers working on x86 instruction sets. In particular, disassembling binary code may not always be possible. Reasons include indirect jumps, presence of foreign bytes, i.e. data in the middle of code sections, self rewriting code, or exotic code generation schemes, such as those used

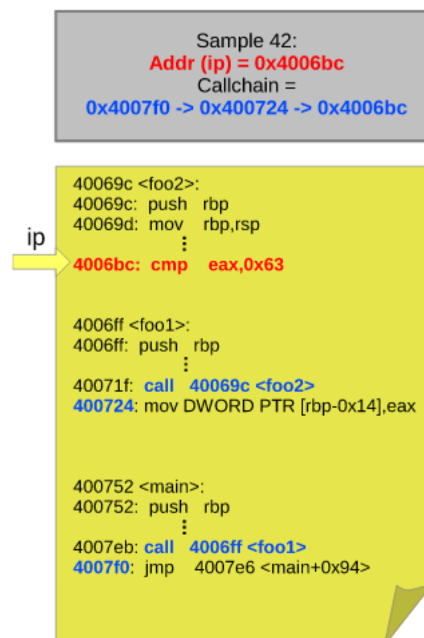


Figure 2.3: Profiling sample

in obfuscated code. Errors are of two kinds: parsing errors and erroneous disassembly. We easily handle the former by aborting the transformation. The latter is beyond the scope of this chapter. Quite a few techniques have been proposed in the literature, interested readers can refer to a survey in [Val14]. The nature of our target program structures (vectorized loops) are almost never subject to these corner cases

2.3.3 Code cache

One of the important questions in dynamic binary translation is where one can place the optimized code. The simplest solution that comes to mind is to replace it in the code section of the process. However, some problems rises with this solution. First, replacing an instruction by another is valid only when they have the same size. When the original one is larger it is also valid only if a padding is added. However, when the optimized one is larger this would lead to overwriting the next instruction. Similarly, for coarse-grain structures like function, when the optimized function is larger, in place modification is impossible. Second, there is no guaranty that an address inside the original code is a target of a jump. Third, it is more difficult to make an in place replacement, since all of the instructions of the function to be replaced should exit the pipeline before replacement.

To avoid all of the mentioned problems, Padrone creates a code cache in the process's heap, where it injects the optimized code. This is achieved by a ptrace system call that allows to observe and control a process execution. The steps are as follow: first, the

process is stopped. Second, the contents of the registers are copied. Third, a set of instructions coming after program counter, where the process is stopped, are stored and replaced by both instructions that allocates space in the heap and one that trap. The trap allows to restore the original instructions and the registers states, so as the process resumes execution correctly.

2.4 Conclusion

This chapter presented some dynamic binary translation tools and APIs along with their profiling methods. PIN, DynamoRio, Valgrind, and SPIRE are not appropriate for our optimization purposes mainly because of their overhead. PIN, DynamoRio, and Valgrind are not considered since compute-intensive programs, which demand high performance resources, usually do not execute on the top of VMs due to the cost of interpretation. Even though, SPIRE allows the target program to execute on bare-metal, its profiling and injection methods incurs branching and register spilling which makes it less attractive for on-the-fly optimizations. On the contrary, the HPC are more suitable alternative for profiling at runtime. This chapter introduced Padrone, which adopts the HPC technique for profiling, and it explained its analysis and code injection functionalities.

Chapter 3

Dynamic Re-vectorization of Binary Code

In this chapter, we target SIMD ISA extensions and in particular the x86 SSE and AVX capabilities. On the one hand, AVX provides wider registers compared to SSE, new instructions, and new addressing formats. AVX has been first supported in 2011 by the Intel Sandy Bridge and AMD Bulldozer architectures. However, most existing applications take advantage only of SSE and miss significant opportunities of performance improvement. On the other hand, even if in theory it could take advantage of vectorization, a loop may be left unvectorized by the compiler. This might happen when using an outdated version of a compiler which does not support vectorization, or when the compiler is unable to analyze the data dependencies or transform the code for ensuring correct vectorization.

3.1 Background

3.1.1 Vectorization at a glance

The incorporation of vector units into modern CPUs extended the instruction set with SIMD instructions. These instructions operate on vector operands containing a set of independent data elements. They include wide memory accesses as well as so-called packed arithmetic. In brief, the same operation is applied in parallel to multiple elements of an array. Figure 3.1 depicts the pseudo code of sequential and vectorized versions of an addition of arrays C=A+B. Variables `va`, `vb`, and `vc` denote vectors, `padd` stands for packed add, a simultaneous addition of several elements. The number of elements processed in parallel is the vectorization factor (vf). In the example of Figure 3.1, elements are of type `int`, i.e. 32-bit wide, SSE vectors are 128 bits, the vectorization factor is $vf = 128/32 = 4$.

The vectorized loop is faster because it executes fewer iterations (the scalar loop iterates n times; meanwhile, the vectorized one iterates $\lfloor \frac{n}{vf} \rfloor$ times), fewer instructions, and fewer memory accesses (accesses are wider but still fit the memory bus width: this

Scalar version

```
int A[], B[], C[];
...
for(i=0; i<n; i++) {
  a = A[i];
  b = B[i];
  c = a+b;
  C[i] = c;
}
```

Vectorized version

```
int A[], B[], C[];
...
/* vectorized loop */
for(i=0; i<n; i+=vf) {
  va = A[i..i+vf-1];
  vb = B[i..i+vf-1];
  vc = padd(va, vb);
  C[i..i+vf-1] = vc;
}
/* epilogue */
for( ; i<n; i++) {
  /* remaining iterations
     if n not multiple of vf */
}
```

Figure 3.1: Vector addition (pseudo code)

is advantageous).

Over time, silicon vendors have often developed several versions of SIMD extensions of their ISAs. The Power ISA provides AltiVec, and the more recent VSX. Sparc defined the four versions VIS 1, VIS 2, VIS 2+, and VIS 3. x86 comes in many flavors: starting with MMX, ranging to different levels of SSE, and now AVX, AVX2, and AVX-512.

Compared to SSE, AVX increases the width of the SIMD register file from 128 bits to 256 bits. This translates into a double vectorization factor, hence, in ideal cases, double asymptotic performance.

3.2 Re-Vectorization of Binary Code

3.2.1 Principle of the SSE into AVX translation

Our goal is to transform loops that use the SSE instruction set to benefit from a CPU that supports AVX technology. Since the binary is already vectorized, we are concerned only with the conversion of instructions from SSE into AVX, and some bookkeeping to guarantee the legality of the transformation.

At this time, our focus is on inner loops, with contiguous memory accesses. Future work will consider outer-loop vectorization and strided accesses.

The main advantage of using AVX over SSE instruction set is that the size of the vector operand doubles from 128 bits into 256 bits. Therefore, the number of data elements that fits into the SSE vector operand doubles in the AVX one. Hence, in the perfect scenario, an AVX loop runs half the number of iterations of a SSE loop.

As we operate in a dynamic environment, we are constrained to lightweight manipulations. The key idea is to leverage the work already done by the static compiler, and to tweak the generated SSE code and adjust it to AVX, while we must guarantee that the generated code is correct and semantically equivalent to the original code.

```
1  .L2:
2      movaps  A(rax),xmm0
3      addps   B(rax),xmm0
4      movaps  xmm0,C(rax)
5      addq    $16,rax
6      cmpq    $4096,rax
7      jne     .L2
```

(a) Original SSE

```
.L2:
    vmovaps  A(rax),xmm0
    vinsertf128  1,A(rax,16),ymm0
    vaddps   B(rax),ymm0
    vmovaps  ymm0,C(rax)
    addq     $32,rax
    cmpq     $4096,rax
    jne      .L2
```

(b) Resulting AVX

Figure 3.2: Body of vectorized loop for vector addition

Figure 3.2 illustrates the loop body of the vector addition (from Figure 3.1) once translated into SSE assembly, and how we convert it to AVX. Register `rax` serves as a primary induction variable. In SSE, the first instruction (line 2) reads 4 elements of array A into `xmm0`. The second instruction (line 3) adds in parallel 4 elements of B, and the third instruction (line 4) stores the results into 4 elements of C. The induction variable is then incremented by 16 bytes: the width of SSE vectors (line 5).

Converting the body of the loop is relatively straightforward. But to guarantee the legality of the transformation, some bookkeeping is necessary. It is detailed in the enumeration below, items 2 to 6, and further developed in Sections 3.2.3 to 3.2.7. The key parts of our techniques are the following:

1. convert SSE instructions into AVX equivalents (cf. Section 3.2.2);
2. restore the state of `ymm` registers in the (unlikely) case they are alive (cf. Section 3.2.3);
3. adjust the stride of induction variables (cf. Section 3.2.4);
4. handle loop trip counts when not a multiple of the new vectorization factor (cf. Section 3.2.5);
5. enforce data dependencies (cf. Section 3.2.6);
6. handle alignment constraints (cf. Section 3.2.7);
7. handle reduction (cf. Section 3.2.8).

3.2.2 Converting instructions from SSE into AVX

The optimization consists of translating a packed SSE SIMD instruction into an AVX equivalent, i.e. an instruction (or sequence of instructions) that has the same effect, but applied to a wider vector. When it is impossible to decide whether the conversion maintains the semantics, the translation aborts.

In most practical cases, there is a one-to-one mapping: a given SSE instruction has a direct AVX equivalent. These equivalents operate on the entire 256 bits vector

Table 3.1: SSE-to-AVX Instruction Conversion Map

SSE	AVX
movlps	vmovaps (only when combined with movhps, and data are aligned on 16 bytes)
movlps	vmovups (when data are not aligned)
movhps	vinsertf128 (only when combined with movlps, from memory to register)
movhps	vextractf (movement from register to memory)
movaps	vmovaps (when data are aligned on 32 bytes)
movaps	vmovaps (aligned on 16 bytes) + vinsertf128
shufps	vshufps + vinsertf128
xorps	vxorps
addps	vaddps
subps	vsubps
mulps	vmulps

operand. However, in some cases, an SSE instruction needs a tuple of instructions to achieve the same semantics. These equivalents operate on halves of the 256-bit vector operand.

Furthermore, an SSE instruction might have several alternatives of equivalent instructions because of alignment constraints, notably the SSE aligned data movement instruction `movaps`. It moves a 16-byte operand from or into memory that is 16-byte aligned. The conversion of this proposes two alternatives: on one hand, when data to be moved is 16-byte aligned, it is replaced by two instructions. Primary `vmovaps` requires data to be aligned on 16 bytes and moves only the lower half of the vector operand. Secondary, a `vinsertf128` that moves the upper half. Figure 3.2 shows that the instruction in line 2 in the original SSE is translated into instructions in lines 2 and 3 in the resulting AVX. We assume that array A is 16-byte aligned. On the other hand, when the data to be moved happens to be 32-byte aligned, an aligned instruction `vmovaps` that moves the whole vector operand at once is selected. Assuming that array C, in Figure 3.2, is 32-byte aligned, the 16-byte memory access (in line 4) is converted into a 32-byte `vmovaps`. Accordingly, the registers are converted from `xmm` to `ymm`. Finally, the translator encodes the translated instructions. Table 3.1 summarizes our instruction conversion map from SSE to AVX.

Finally, there are also opportunities to map several SSE instructions to a single AVX instruction. For example, AVX provides a fused multiply-add instruction that provides additional benefits (and also semantics issues due to floating point rounding). The new three-operand format also opens the door to better code generation. These elaborated translations are left for future work.

3.2.3 Register liveness

A register is said to be alive at a given program point if the value it holds is read in the future. Its live range begins when the register is first set with a specific value and ends

when the value is last used. Registers can be alive for long sequences of instructions, spanning the whole loop nests we are interested in. Liveness analysis requires a global data flow analysis.

Our optimizer only impacts the liveness of `ymm` registers. For all other registers, we simply update an operand of instructions, such as the stride of induction variables, which has no impact on live ranges. Registers `ymm` must be handled in a different way. They are not used in the loops we consider, and SSE code is unlikely to make any use of them. Still, there might be situations where SSE and AVX code coexist (such as hand written assembly, use of third party libraries, or ongoing code re-factoring – see §11.3 in [Int14a]). Our optimizer cannot run any interprocedural liveness analysis and has only a local view of the program. We cannot make any assumption on the liveness of these registers. Our straightforward solution is to spill the registers we use in the preheader of the loop (created as needed), and restore them after the loop.

The situation is actually slightly more complicated because `ymm` is an extension of `xmm`, where `xmm` denotes the 128 least significant bits of `ymm.` We only restore the 128 most significant bits to guarantee a consistent state.

3.2.4 Induction variables

In the scenario where arrays are manipulated, a register is typically used as an index to keep track of the memory access of the iteration. For instance, the vector addition example in Figure 3.2 (a) makes use of register `rax` for two purposes: first, as a counter of iterations and second as an array index. In lines 2, 3, and 4 it is used as an index with respect to the base addresses of arrays A, B, and C from and to which the reads, and writes should be performed. Since the SSE data movement instructions operate on 16 bytes operands, the register is incremented by 16 bytes in line 5.

When the code is translated into AVX, the operand's size changes. Hence, the instruction responsible for incrementing the index should be adapted to the AVX version. In other words, the pace of change becomes 32 bytes instead of 16 bytes. Line 5 in Figure 3.2 (b) illustrates the modification performed by the translator for adapting indices to the optimized version.

3.2.5 Loop bounds

Replacing vector instructions by their wider instructions requires an adjustment of the total number of loop iterations. In the case of translating SSE into AVX SIMD instructions, the size of the vector operands is doubled. Hence, a single AVX iteration is equivalent to two consecutive SSE iterations.

The translator handles loop bounds by classifying vectorized loops into two categories: loops with a number of iterations known at compile-time; and loops where the bounds are only known at run-time.

3.2.5.1 Loop bound known at compile-time

When the total number of iterations is known at compile-time, the translator has two alternatives: either replace the total number of iterations by its half value or double the increment of the loop counter. It opts for the second choice, since the loop counter may serve as an array index at the same time, whose pace of change is doubled as discussed in the previous subsection. Line 5 in Figure 3.2 (b) illustrates the transformation.

The number of iterations of the loop with SSE SIMD instructions can be either even or odd. When it is even, $n = 2 \times m$, the optimizer simply translates SSE into AVX instructions and doubles the pace of change of the counter so that the transformed loop iterates m times. However, when the number of iterations is odd, $n = 2 \times m + 1$, the transformed code is composed of two basic blocks: primary, the loop that runs AVX instructions m times. Secondary, the SSE instructions that run the last SSE iteration. The latter instructions are vex.128[1] encoded to avoid SSE/AVX transition penalty.

3.2.5.2 Loop bound known only at run-time

This happens, for instance, when n is a function parameter, or when an outer loop modifies the bound value of the inner one. Suppose that a sequential loop executes a set of instruction n times, and its vectorized version executes equivalent instructions in m iterations such that: $n = m \times vf + r$, where r is the number of remaining sequential iterations ($r < vf$) that obviously cannot be performed at once.

Under such circumstances, the compiler generates a vectorized loop that iterates m times, and a sequential one that iterates r times. The values of m and r are calculated earlier, before control is given to one of the loops or both of them depending on the values of n discovered at runtime. It is also possible that the static compiler unrolls the sequential loop. Currently, this latter situation is not handled.

For a correct translation into AVX, an adjustment of the values of m and r is required. Let us consider the vectorized loop L2 (see Figure 3.3). The loop has a bound register that is compared to the counter in line 9. The bound register holds the value m, that is computed earlier in line 3. This instruction initializes the bound register to a value equal to $\lfloor n/vf \rfloor$. The translator modifies this instruction to initialize it to $\lfloor n/(2 \times vf) \rfloor$.

Regarding the sequential loop L5, it iterates r times. The compiler initializes the counter by a value equal to $n - r$ which is also equal to $vf \times m$ (in line 15). Therefore, the loop iterates from $n - r$ to n for a total of r iterations. The translator traces the counter register `r8d` (in line 18). It finds that it received the contents of `r9d` (in line 15). It continues tracing `r9d` until line 4 where this latter is initialized by the value $m \times vf$. The translator then changes the initialization into $2 \times m \times vf$.

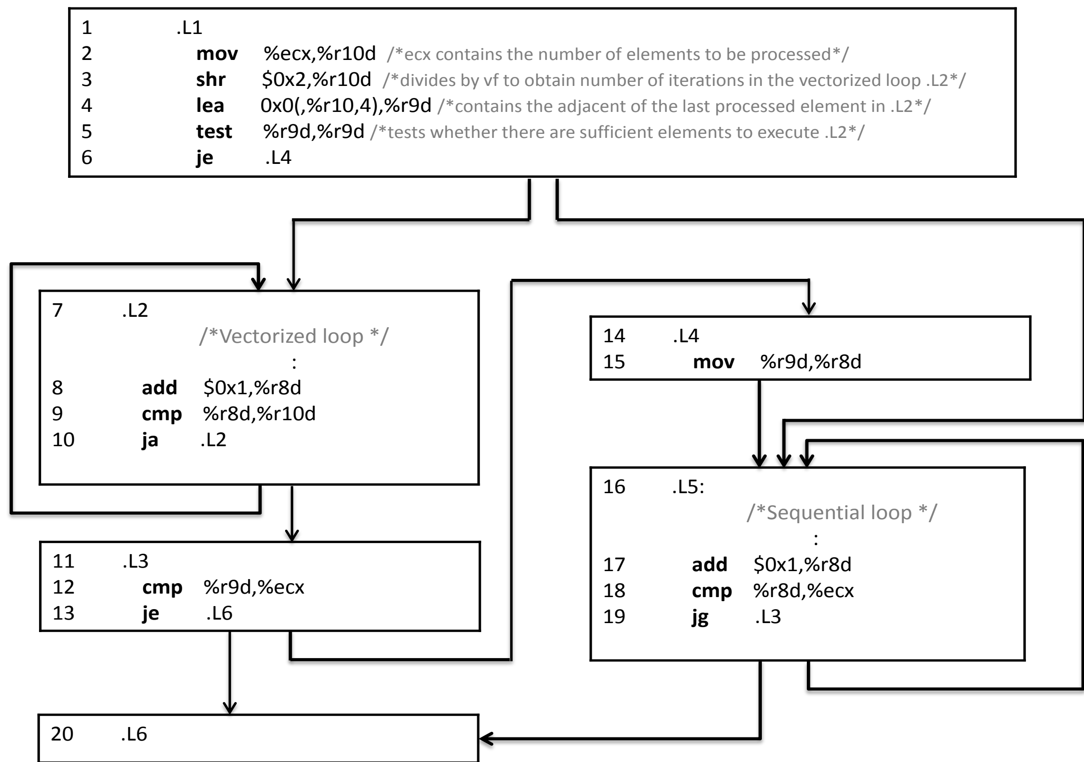


Figure 3.3: Pattern of code when the number of iterations is known only at runtime

```
for(i=0; i < n-4; i++)
  A[i+4] = A[i] + 1;
```

Figure 3.4: Example of loop-carried dependence

3.2.6 Aliasing and Data Dependencies

3.2.6.1 Overview of aliasing

Aliasing is the situation where multiple instructions access the same memory location. Dependence happens when at least one of the instructions is a write, as in the following code:

```
A[0] = B[0];   // writes into location A[0]
B[1] = A[0];   // reads from A[0]
```

There are three kinds of dependence: read-after-write, write-after-read, and write-after-write. The first one, also known as true dependence, forces the code to execute sequentially and the others allow code to be parallelized, subject to some slight code transformations. Sticking to the same example, it is not possible to execute both instructions at once; since, the second instruction that reads A[0] should wait until the first instruction writes into A[0].

A loop-carried dependence describes the situation where the dependence occurs between iterations of a loop. For instance, in Figure 3.4, a write into A[4] occurs at the first iteration, and a read from the same location occurs four iterations later. Considering the true dependence, the distance vector is the number of iterations between successive accesses to the same location. In this example, the distance is $d = (i + 4) - i = 4$. Therefore, executing up to four iterations in parallel is allowed; as well as, vectorizing up to four elements. However, vectorizing more than four elements violates the data dependence.

3.2.6.2 Issue of translating a loop with data dependencies

A blind widening of vectors from SSE's 128 bits to AVX's 256 bits might violate data dependencies. It may happen because this doubles the vectorizing factor, hence runs the risk to exceed the dependence distance.

3.2.6.3 Static interval-overlapping test

Static interval-overlapping test refers to a lightweight verification done by the translator at compile time (no code is generated) to ensure that the translation will not cause a dependence violation. The test consists of comparing the set of addresses touched by the new AVX SIMD instructions against all addresses of the remaining new AVX SIMD data movement instructions of the loop. A non-empty intersection signals an attempt to translate more elements than the dependence distance. Consequently, the translation process is aborted. The static test occurs in the following scenarios: when the addresses of the arrays are known at compile time, and the distance vector is constant during the execution of the loop.

For illustration, let us consider the original SSE code in Figure 3.2. The translator gathers SIMD instructions that involve an access to memory in lines 2, 3, and 4. The

[1]vex.128 instructions are enhanced versions of legacy SSE instructions that operate on lower half of `ymm` registers and which zero the upper half.

```
for(i=0; i < n ; i++) {
  for(j=4; i+j < n; j++) {
    A[i+j] = A[i] + 1;
  }
}
```

Figure 3.5: Example of aliasing with changing dependence distance

one in line 4 is a write; therefore, it is compared to the others. The set of addresses referenced by instruction in line 4, in the case it is translated into AVX, ranges between C and C+32. Similarly, for instruction in line 2 the range is between A and A+32. A non-empty intersection of intervals would stop the translation process. In this example, the optimizer proceeds. Likewise, the intersection test is done for instructions in line 3 and 4.

3.2.6.4 Dynamic interval-overlapping test

The dynamic interval-overlapping test refers to verification performed at runtime. This happens when the addresses of arrays manipulated are not known statically, or when the dependence distance in the inner loop is modified by the outer loop, as depicted in the example of Figure 3.5.

In these scenarios, the compiler generates the test as shown in basic block L1 in Figure 3.6. In case of empty intersection, the control flow is directed to the vectorized loop L2; otherwise, the sequential loop L3 is invoked. The basic block L1 contains the intersection test between the read and write of lines 13 and 16.

The test works as follow: an offset of 16 bytes is added to both `rdi` and `rdx` in lines 2 and 3, to verify whether the read interval [`rdi`, `rdi`+16] intersects with the write interval [`rdx`, `rdx`+16]. In order to adjust this code to work for the AVX version, our translator performs a pattern matching to identify the instructions in line 2 and 3, then it changes the offset from 16 (0x10) to 32 bytes (0x20).

3.2.7 Alignment constraints

Alignment is another issue faced during the translation of SIMD instructions. In a nutshell, the majority of x86 instructions are flexible with alignment; however, some of them demand data on which they operate to be aligned. From now on, we refer to these two categories respectively as unaligned and aligned instructions. Actually, the aligned SSE instructions require data to be aligned on 16 bytes. And, the aligned AVX instructions require data to be aligned on either 16 or 32 bytes [Int14b].

When the optimizer encounters an unaligned SSE instruction, it translates it to its equivalent unaligned AVX instruction. However, when it encounters an aligned SSE instruction, it must apply one of these three options:

- translate it into its equivalent 32-byte aligned AVX instruction;

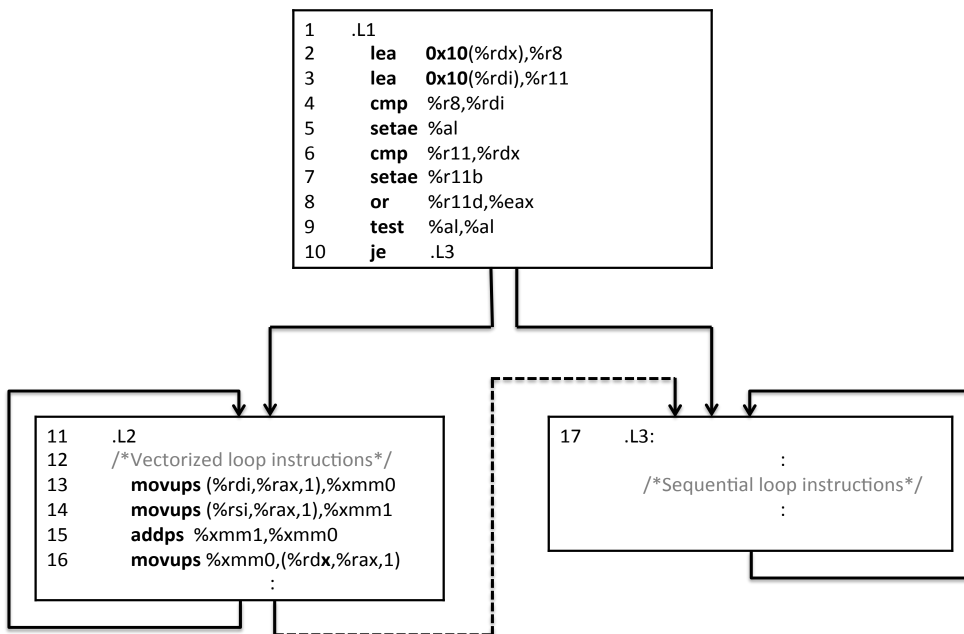


Figure 3.6: Pattern of code when array addresses are known only at runtime

```
float m = MIN;
for(i = 0; i < n; i++) {
  if (m < array[i]) /* max(m, array[i]) */
    m = array[i];
}
```

Figure 3.7: Search for the maximum element in an array of floats

- translate it into its multiple equivalent 16-byte aligned AVX instructions that operates on half of the vector operand;
- translate it into its equivalent unaligned AVX instruction.

When the address of data on which the aligned SSE instruction operates is explicitly or implicitly provided in the code section as a static value, the optimizer tests whether it is also aligned on 32 bytes. When the test succeeds, the equivalent 32 bytes aligned AVX instruction is eligible for use. Otherwise, since the SSE aligned instruction is by definition aligned on 16 bytes; the optimizer blindly translates it into the equivalent 16 bytes aligned AVX instructions.

The reason why we do not use the unaligned AVX instructions in the mapping of aligned SSE instructions, although they can be used instead of multiple aligned AVX instructions on 16 bytes, is: multiple aligned instructions, when they are independent, execute in parallel in an out-order-processor. When they are aligned on 16 bytes, they run faster than a single unaligned instruction that performs extra work of testing the cross-cache line access. The drawback of this solution is that multiple instructions occupy more issue slots, in comparison with the single unaligned AVX instruction.

There is another approach that we intend to use in the future to cope with alignment problem. In fact, some compilers extensively use it and it consists of looping over data elements sequentially until most of the manipulated data are aligned. At that moment the vectorized loop, which contains aligned instructions, is qualified for its use. While this implies more work at runtime, it also delivers additional performance.

To sum up, dynamic optimization allows us in some cases to check for data elements addresses to cope with alignment problem.

3.2.8 Reductions

Reduction is the process of reducing a set of values into a single one, for instance, summing the elements of a set.

Figure 3.7 depicts a reduction algorithm that searches for the largest floating point number in an array. The algorithm suffers from a loop-carried dependence whose distance is equal to 1 (read-after-write on m). In spite of the data dependence, the problem is prone to vectorization thanks to the associativity and commutativity of the `max` operation. Suppose we want to find the maximum value in the following set 5, 9, 3, 7. It is possible to execute simultaneously both operations $max(5, 9) = 9$ and $max(3, 7) = 7$, then execute $max(9, 7) = 9$.

```
1   .L2:
2           maxps   A(rax),xmm0
3           add     0x10,rax
4           cmp     rax,ARRAY_LIMIT
5           jle     .L2
6           movaps  xmm0,xmm1
7           psrld   8, xmm1 ; shift right logical
8           maxps   xmm1,xmm0
9           movaps  xmm0,xmm1
10          psrld   4,xmm1
11          maxps   xmm1,xmm0
12          movaps  xmm0,xmm1
```

Figure 3.8: Search for the largest element in an array, vectorized for SSE

3.2.8.1 Issue of translating a reduction

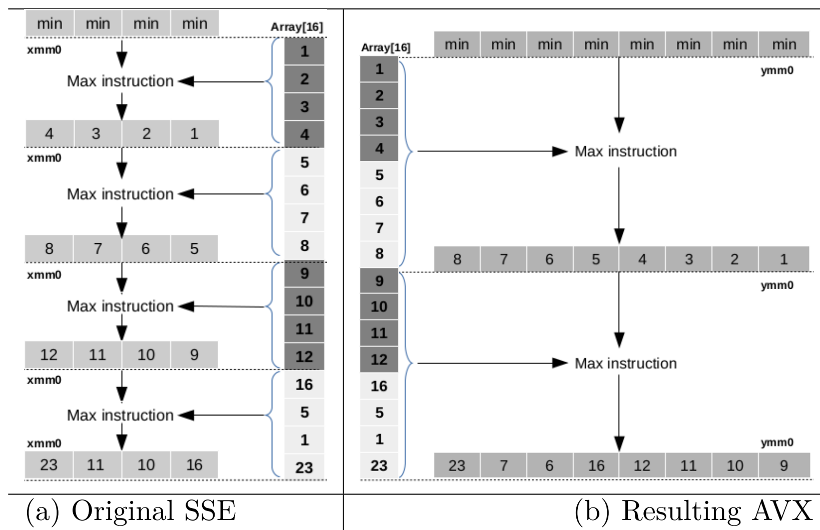
The code in Figure 3.8 depicts the assembly of Figure 3.7. The code has two sections. First, the loop in lines 2—5 searches simultaneously for four largest numbers in four different sections of the array. Precisely, the sections have indices that fall in $i \bmod 4$, $i \bmod 4 + 1$, $i \bmod 4 + 2$, and $i \bmod 4 + 3$. This yields the `xmm0` register with four greatest elements seized during the simultaneous traversal of these regions.

Figure 3.9 (a) delineates the execution of these lines on an array of sixteen elements. At each iteration, a pack of four elements is read from memory and compared to values stored in `xmm0`, which is initially loaded with minimal values. The register is updated based on the result of the comparisons. Second, since `xmm0` contains a tuple of elements, lines 6—12 resolve the largest one from the others by shifting it into the lowest part of the register. Figure 3.10 portrays the execution of this latter.

Translation of reductions requires special care. Suppose we translate the loop body of Figure 3.8 into AVX. Its execution yields register `ymm0` with eight sub-results as depicted in Figure 3.9 (b), as opposed to four in the case of SSE. The loop epilogue (lines 6—12 of Figure 3.8) “reduces” them to the final result. It must be updated accordingly to take into account the wider set of sub-results, i.e. the new elements in the upper part of `xmm0`. This is achieved by introducing a single step, between lines 5 and 6, to reduce the 256-bit `ymm` register into its 128-bit `xmm` counterpart, as produced by the original code. In our running example, we need an additional shift-and-mask pair, as shown in Figure 3.10 (b). Whenever the translator is not able to find a combination of instructions to restore the state of the register `xmm`, it simply aborts optimizing this loop.

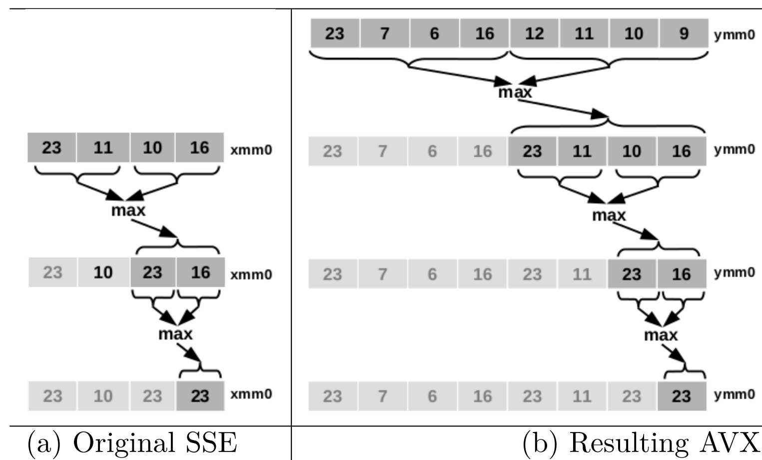
3.2.8.2 Subclass of reduction supported by the translator

This subsection shows a subclass of reduction for which it is possible to recover the same state of `xmm` as if the non-translated loop is executed. This subclass has a particular



(a) Original SSE (b) Resulting AVX

Figure 3.9: Body of vectorized loop for vector max



(a) Original SSE (b) Resulting AVX

Figure 3.10: Extracting the largest element from the SIMD register

characteristic described in the following recursive sequence form:

$$xmm_n = xmm_{n-1} \odot input_n \tag{3.1}$$

Where the operator $\odot$ is either an arithmetic or logical instruction that is both associative and commutative; and, the subscript n indicates the iteration number. Besides, the input is nothing but the memory read access of the same size as the `xmm` register.

Demonstration: Let us solve the recursive sequence in (3.1) that describes an SSE loop. In the last line of (3.2), the odd and even terms are separated since the operator is both associative and commutative (assuming n is even for the sake of simplicity):

$$\begin{aligned} xmm_n &= xmm_{n-1} \odot input_n \\ &= xmm_{n-2} \odot input_{n-1} \odot input_n \\ &= xmm_{n-3} \odot input_{n-2} \odot input_{n-1} \odot input_n \\ &= input_0 \odot input_1 \odot ... \odot input_{n-1} \odot input_n \\ &= [input_0 \odot input_2 \odot ... \odot input_n] \odot [input_1 \odot input_3 \odot ... \odot input_{n-1}] \end{aligned} \tag{3.2}$$

Since an AVX iteration is equivalent to two successive SSE iterations, there are two consequences. First, the number of iterations decreases by half. Second, a packed memory read in an AVX iteration is equivalent to the concatenation of memory reads of two consecutive iterations in SSE loop. In our notation, *input* represents a memory read for an SSE iteration and the same word in capital letters for an AVX iteration. Plus, $INPUT_L$ and $INPUT_H$ indicate the lower and higher halves of $INPUT$:

$$\begin{aligned} input_{2n} &= INPUT_{L_n} \\ input_{2n+1} &= INPUT_{H_n} \end{aligned} \tag{3.3}$$

Applying (3.3) in (3.2):

$$\begin{aligned} xmm_n &= [INPUT_{L_0} \odot INPUT_{L_1} \odot ... \odot INPUT_{L_{n/2}}] \odot \\ &\quad [INPUT_{H_0} \odot INPUT_{H_1} \odot ... \odot INPUT_{H_{n/2}}] \\ xmm_n &= [ymm_lowerhalf_{n/2-1} \odot INPUT_{L_{n/2}}] \odot \\ &\quad [ymm_higherhalf_{n/2-1} \odot INPUT_{H_{n/2}}] \\ &= ymm_lowerhalf_{n/2} \odot ymm_higherhalf_{n/2} \end{aligned} \tag{3.4}$$

Therefore, we conclude that applying the operator between the higher and lower halves yields the state of `xmm` as if it had executed the non-translated code.

3.3 Conclusion

This chapter covered a low overhead binary-to-binary transformation. Basically, SSE instructions are translated into their AVX equivalents which manipulate twice as many elements. Theoretically, the optimized code runs twice faster as fast as the original one thanks to loop trip count reduced by half. Besides the conversion of instructions, this method requires binary adjustments and validity tests with regards to several issues, namely: liveness, induction variable, trip count, data dependence, alignement, and reduction.

In the next Chapter, we address the runtime vectorization of binary codes that have not been originally vectorized.

Chapter 4

Dynamic Vectorization of Binary Code

The current trend toward multi-core and vector architectures rises challenges in the compiler community to maximize the use of hardware resources either during compilation, or by the use of a runtime for accelerating the execution on bare-metal or through interpretation. Many compute-intensive applications spend time in loops. The abstraction of complex constructs, such as loop nests enveloping conditionals, using the AST representation, is inappropriate to parallel and vector optimizations, which sometimes require a combination of advanced transformations such as inversion, interchange, skew, and strip-mine. This abstraction is not capable of exhibiting some data-dependences between statements belonging to interleaved iterations. A powerful and accurate alternative is the polyhedral model [FL11] which is a mathematical framework that allows the abstraction of program's instruction instances and constructs in a vector space. It allows to find appropriate transformations and generate optimized code.

This chapter addresses the runtime vectorization of loops in binary codes that were not originally vectorized. It scopes the theory of the polyhedral model and puts it into practice during the automatic vectorization. We use open source frameworks that we have tuned and integrated to (1) dynamically lift the x86 binary into the Intermediate Representation form of the LLVM compiler, (2) abstract hot loops in the polyhedral model, (3) use the power of this mathematical framework to vectorize them, and (4) finally compile them back into executable form using the LLVM Just-In-Time compiler. The auto-vectorizer is implemented inside a dynamic optimization platform; it is completely transparent to the user, does not require any rewriting of the binaries, and operates during program execution.

```
1  /**
2  * The tree parts of the outer loop are affine expressions:
3  * - Initialization: i⃗ = 0 × 0⃗ + 1⃗
4  * - Condition: 1 × i⃗ − n⃗ < 0⃗
5  * - Afterthought: i⃗ = 0 × i⃗ + 1⃗
6  */
7  for(i = 1; i < n; i++) {
8          /** Inner loop parts are affine expressions
9          * - Initialization: j⃗ = 0 × 0⃗ + 1⃗
10         * - Condition: 1 × j⃗ − 1 × i⃗ < 0⃗
11         * - Afterthought: j⃗ = 0 × j⃗ + 1⃗
12         */
13         for (j = 1; j < i; j++){
14                 /** Access functions are affine:
15                 *   - Write:1 × j⃗ + (n × i⃗ + 0⃗)
16                 *   - Read:1 × j⃗ + [((n − 1) × i⃗) − 1⃗]
17                 */
18                 A[i][j] = A[i-1][j-1] * 0.99;   // statement S1
19         }
20 }
```

Figure 4.1: Example of a valid Static Control Part

4.1 Principles of the polyhedral model

4.1.1 Static control part (SCoP)

A Static Control Part (SCoP) is a set of instructions in a program that is eligible for representation and linear transformation in the vector space. As from the name, it is a part of code whose control flow is static. In other words, the paths that are taken during the execution are known at compile time and not arbitrarily chosen at runtime based on dynamic data. Furthermore, a few more constraints have to be fully met. The memory access instructions, loop bounds (also known as local loop parameters), and conditionals have to be expressed as affine functions or affine inequalities of outer loop indices and global parameters. Besides, a loop must iterate with a single index which is expressed as an affine function as well. Figure 4.1 illustrates an example of a valid SCoP.

4.1.2 Perfect and inperfect loop nest

A loop nest is said to be perfect when all its statments reside exclusively in the inner loop as illustrated in Figure 4.1. Apart from that, it is considered imperfect.

4.1.3 Iteration domain

The execution of a statement at a particular iteration within a loop is called a dynamic instance. It is characterized by the values of the iterators in the loop nest at the moment of its execution. These values form an ordered sequence that is represented by an n-tuple or vector, where n is the depth of the loop nest. The iteration domain of an instruction is the set of all its dynamic instances executed during the life-time of the loop nest. For example, the statement $S1$, in line 16 and shown in Figure 4.1, resides within a loop nest whose outer and inner iterators are consecutively i and j. Binding them together, a 2-tuple or vector (i, j) is constructed. The values of iterators at the first execution of the statement are $(1, 1)$, which represents a dynamic instance. The iteration domain of the statement are the set of 2-tuples (i, j), where i and j are confined between a lower and upper bound values consecutively equal to 1 and n-1. In a formal way, the domain of $S1$ is:

$$D_{S1} = \{(i, j) | 1 \leq i \wedge i < n \wedge 1 \leq j \wedge j < i\} \tag{4.1}$$

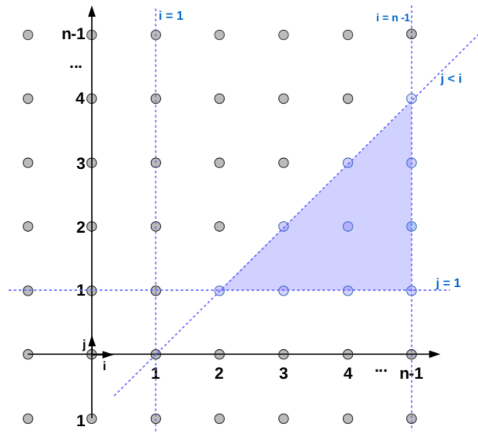


Figure 4.2: Iteration domain

Graphing the inequalities is represented by the shaded region in Figure 4.2. Because the polyhedral model considers SCoPs, which contain affine expressions and functions, for transformations, the iteration domain will always be defined as a set of linear inequalites. As a consequence, a matrix form can also stand for the constraints of the domain:

$$A \times \vec{v} \geq \vec{0} \tag{4.2}$$

Where $\vec{v} = \begin{pmatrix} \vec{x} \\ \vec{p} \end{pmatrix}$ is a vector that aggregates the iteration vector $\vec{x} = \begin{pmatrix} i \\ j \end{pmatrix}$, and the loop parameters vector $\vec{p} = \begin{pmatrix} 1 \\ n \end{pmatrix}$, and A is a matrix consisting of coefficients of the variables used in the inequalities.

For instance, the matrix form of $S1$ already defined in equation (4.1) is:

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

4.1.4 Access function

The access function of an instruction is an affine function that maps its instances to positions in an array. In other words, it allows to know the memory location accessed by an instruction at a given iteration. Its formal mathematical representation is:

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

Where, A is an $n \times m$ coefficient matrix and $\vec{x}$ is the iteration vector.

Example 4.1: Representation of the access function

The read access function in statement S1 in Figure 4.1

$$f_{S1read}\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ -1 \end{pmatrix} = \begin{pmatrix} i-1 \\ j-1 \end{pmatrix}$$

It is also possible to represent the access function with the following:

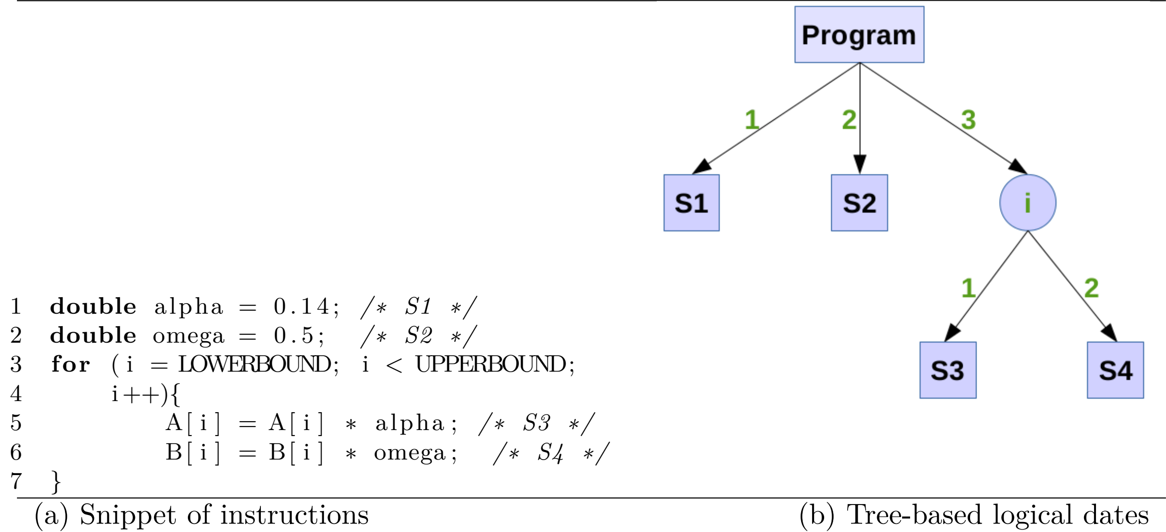
$$g(\vec{x}) = A' \times \vec{v}$$

Where, A is an $n \times (m + dim(\vec{v}))$ coefficient matrix; such that, $dim(\vec{v})$ is the dimension of the vector $\vec{v}$, which is nothing but a vector that aggregates loop's iterators and parameters, $\vec{v} = \begin{pmatrix} \vec{x} \\ \vec{p} \end{pmatrix}$. Throught this chapter the latter representation will be used.

Example 4.2: Alternative representation of the access function

The read access function in statement S1 in Figure 4.1

$$f_{S1read}\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & -1 & 0 \end{pmatrix} \times \begin{pmatrix} i \\ j \\ 1 \\ n \end{pmatrix} = \begin{pmatrix} i-1 \\ j-1 \end{pmatrix}$$



(a) Snippet of instructions (b) Tree-based logical dates

Figure 4.3: Logical date

4.1.5 Execution order between instruction

In general, a logical date associated to an instruction S is a vector that represents its time of execution. The order of execution of several instructions is determined by the lexicographic comparison between their logical dates. In [Bas04b], Bastoul describes a method based on AST to represent the original execution order of instructions in a program. Consider the code snippet shown in Figure 4.3.a, its associated tree-based schedule is depicted in Figure 4.3.b. For instance, the logical dates for instructions S_1 and S_2 in line 1 and 2 (Figure 4.3.a) are the vectors (1) and (2) retrieved from the edges seen in Figure 4.3.b. Instructions that reside in loops might have many instances; hence, each of the instances has its specific logical date. It is possible to represent all of the logical dates of instances of an instruction with a single vector whose elements contains the loop nest's iterators. For instance, the logical dates for all instances of instructions S_3 and S_4 in the same Figure are (3,i,1) and (3,i,2). It is easy to express parallelism with this representation by giving independent instances the same logical date.

The aim of optimization in the polyhedral model is orchestrating the time of execution of instances of instructions. For instance, in loop parallelization a group of instances can be given the same excution date. Hence, the need of a scheduling function, that maps the instances of an instruction S in the iteration domain into logical dates, rises. The function is formally defined as:

$$\Theta_S(\vec{x}) = A_S \times \vec{x} + \vec{b}$$

Where A is a matrix and $\vec{b}$ is a constant vector. The section 4.1.7.2 describes the method to find the scheduling function using the Farkas lemma and the Fourier-Motzkin elimination method.

4.1.6 Data dependence

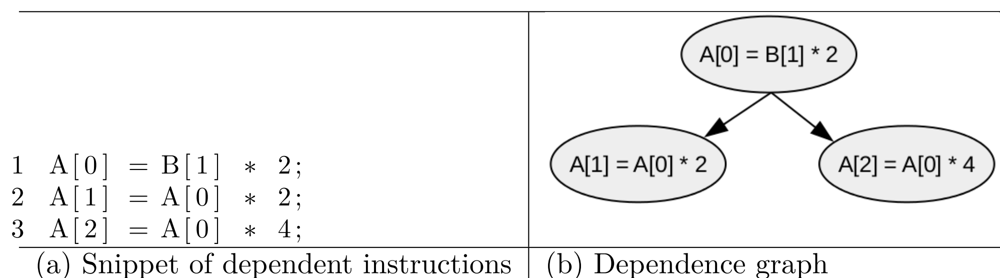
Data dependence analysis exposes the dependences between program statements. Such information contributes in applying parallelizing optimizations, since it determines when two operations, statements, or two iterations of a loop can be executed in parallel. It is crucial to find out the dependent instructions and abstract them in dependence relations which will be covered later in this section. The main reason is that applying some transformations without taking data dependences into account jeopardizes the correctness of the optimized program; meaning, the original and transformed programs may behave differently. As a rule of thumb, a transformation with dependence preservation implies that the transformed program preserves the semantics of the original one. This section addresses various issues: it defines the data dependences, furnishes the purpose of a transformation with regards to dependent instructions, provides some of the data analysis used in some transformations, highlights their shortcomings, and suggests the data dependency analysis. Finally, it provides both methods for abstracting dependences between instructions in the polyhedral model, namely the distance vector and the dependence polyhedron.

As it can be inferred, data dependence is a situation where multiple instructions reference the same memory location. An analysis to determine dependence between instructions is mandatory for some program transformations such as reordering of instructions as pipeline optimization, vectorization, and parallelization. Before going any further, let us first explore the conditions of dependence between instructions. The Bernstein conditions state that two statements are dependent when the three following conditions hold:

- First, they reference the same memory cell.
- Second, at least of the accesses is a write.
- Third, these statements are in the same path of the control flow.

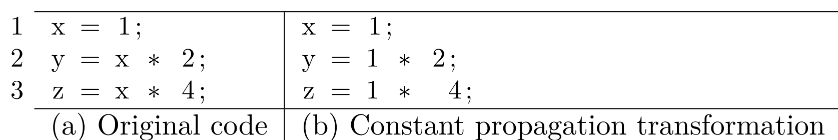
There are three categories of dependences: a read-after-write also called true dependence occurs when the first instruction updates a memory location that is later read by the second one. A write-after-read also called anti-dependence occurs when the first statement reads a location that is later updated by the second one. Third, a write-after-write also known as output dependence occurs when both instructions write the same memory location concurrently.

One primary purpose of a transformation besides optimization is to preserve the semantics. It is sustained as long as the dependencies are not violated. Determining the independence of two statements means that they can execute in any orders and thus in parallel without corrupting the memory cells on which they operate. For instance, let us consider the example in Figure 4.4.a, statements in lines 2 and 3 read the same location written by statement in line 1. This situation exhibits true dependencies where both the second and third statements use the value written by the first one. The dependence graph of this code snippet is represented in Figure 4.4.b. The parent child relationship forces the order of execution giving the parent the priority, and siblings of a node are



(a) Snippet of dependent instructions | (b) Dependence graph

Figure 4.4: Instruction reordering



(a) Original code | (b) Constant propagation transformation

Figure 4.5: Constant propagation

subject to a reordered, concurrent, or parallel execution as long as there is no path that links them in the graph, so do statements in lines 2 and 3.

Sometimes dataflow analysis techniques such as constant propagation and alias analysis expose the possibility of optimizing some ambiguous instructions. The constant propagation algorithm replaces any constant variable by its value throughout the program. The replacement is done by a traverse of the control flow graph. Let us consider the example in Figure 4.5.a, running a constant propagation algorithm will replace the value of the constant variable x in lines 2 and 3 as shown in 4.5.b; as a consequence, the three instructions can be run in parallel. Alias analysis allows to know the different names that points to the same storage when the program uses pointers. Two instructions that use different pointer names but may refer to the same memory object cannot be optimized unless alias analysis is performed to uncover the ambiguity around the memory references.

Previous dataflow analysis techniques are necessary for optimizations but they are not sufficient for parallelization and vectorization. Alias analysis determines whether there is an access to the same storage or not for some section of code or the entire program. In some cases, a loop containing instructions operating on pointers that does not alias can be parallelized or vectorized based only on alias information. However, for loop carried dependences, this analysis is not capable of providing additional information such as the number of iterations that separate accesses to the same location.

In contrast to dataflow analysis that manages to identify the dependence between instructions, the dependence analysis is more precise since it considers the dependence between dynamic instances of instructions. Literature in the polyhedral model came with two main approaches which are the distance vector and the dependence polyhedron.

Distance vector abstraction was first described by [Lam74] and [Wol89], used to detect cross-iteration dependences and formalized as relations. Among the methods

```
1 for (i = LOWERBOUND; i < UPPERBOUND; i++){
2     A(c1*i+c2) = . . .        /* S1 */
3     . . .      = A(c3*i+c4)   /* S2 */
4 }
```

Figure 4.6: GCD test

that serves for that purpose is the greatest common divisor test. To illustrate this technique consider the loop in Figure 4.6, where $c1$, $c2$, $c3$, $c4$ are constants. When the greatest common divisor GCD ($c1$,$c3$) divides ($c4$-$c2$), then a dependence between S_1 and S_2 may exist.

When the GCD test holds, the distance vector is obtained by differentiating indices between the write and read instructions. For instance, the dependence vector for the SCoP in Figure 4.1 is:

$$\vec{d} = (i, j) - (i - 1, j - 1) = (1, 1).$$

When the GCD test does not hold, then there is no dependence between the statements.

The dependence polyhedron abstraction was introduced by Irigoin and Triolet in [IT87]. This abstraction method stems from the formal definition of dependence. The dependence is satisfied when the following conditions are met:

- 1. One of the operations is a write.
- 2. Both statements share a set of storage locations.

In other words, statements S_1 and S_2 are dependent if and only if for all the dynamic instances belonging to their respective iteration domains, the ranges of both access functions intersect. It means that equating the access functions has a solution which is not null. As a consequence, the dependence polyhedron represented as inequalities should encompass the iteration domains of both instructions D (in case of a loop fusion optimization, instructions can belong to different loops), their access functions (F), and the precedence relations (P) which determine the original lexicographic order for this pair of statements before transformation. Precisely, the precedence relation represents the relationship between the instruction's instance which produces a value and the instruction's instance that consumes it. The importance of this constraint is to allow keeping the same semantic after a program's transformation.

$$\boldsymbol{D} = \begin{pmatrix} D_{S_1} & 0 \\ 0 & D_{S_2} \\ F_{S_1} & -F_{S_2} \\ P_{S_1} & -P_{S_2} \end{pmatrix} \geq \vec{0}$$

Example 4.3: Dependence matrix

In this example, we construct the dependence matrix for the read S_{read} and write S_{write} statements within the SCoP in Figure 4.1.

Iteration domains

$$D_{Swrite} = \begin{pmatrix} 1 & 0 & -1 & 0 \\ -1 & 0 & -1 & 1 \\ 0 & 1 & -1 & 0 \\ 1 & -1 & -1 & 0 \end{pmatrix} \times \begin{pmatrix} i \\ j \\ 1 \\ n \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$D_{Sread} = \begin{pmatrix} 1 & 0 & -1 & 0 \\ -1 & 0 & -1 & 1 \\ 0 & 1 & -1 & 0 \\ 1 & -1 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} i' \\ j' \\ 1 \\ n \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Access functions

$$F_{Swrite} = \begin{pmatrix} i \\ j \\ 1 \\ n \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} i \\ j \\ 1 \\ n \end{pmatrix}$$

$$F_{Sread} = \begin{pmatrix} i' \\ j' \\ 1 \\ n \end{pmatrix} = \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & -1 & 0 \end{pmatrix} \times \begin{pmatrix} i' \\ j' \\ 1 \\ n \end{pmatrix}$$

Precedence constraints

At a given iteration (i', j'), the value consumed is produced at iteration (i-1, j-1). The constraint is:

$i' = i - 1$ and $j' = j - 1$

Hence,

$$P \begin{pmatrix} i \\ j \\ i' \\ j' \\ 1 \\ n \end{pmatrix} = \begin{pmatrix} 1 & 0 & -1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 & -1 & 0 \end{pmatrix} \times \begin{pmatrix} i \\ j \\ i' \\ j' \\ 1 \\ n \end{pmatrix}$$

Dependence polyhedron

$$\mathbf{D}\begin{pmatrix} i \\ j \\ i' \\ j' \\ 1 \\ n \end{pmatrix} = \left(\begin{array}{cccccc} 1 & 0 & 0 & 0 & -1 & 0 \\ -1 & 0 & 0 & 0 & -1 & 1 \\ 0 & 1 & 0 & 0 & -1 & 0 \\ 1 & -1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 1 \\ 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & -1 & 1 & 0 \\ \hline 1 & 0 & -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 & 1 & 0 \\ 1 & 0 & -1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 & -1 & 0 \end{array}\right) \times \begin{pmatrix} i \\ j \\ i' \\ j' \\ 1 \\ n \end{pmatrix} \begin{array}{c} \geq \\ \hline = \end{array} \vec{0}$$

$\mathbf{D}_{write}$, $\mathbf{D}_{read}$, $\mathbf{F}_{write}$, $-\mathbf{F}_{read}$, $-\mathbf{P}$

4.1.7 Loop transformations

4.1.7.1 Unimodular vs polyhedral transformations

Loop interchange, reversal and skewing are unimodular transformations. They correspond to a special type of scheduling functions that are expressed as unimodular square matrices whose determinants are equal to -1 or 1.

We have seen that loop characteristics such as its indices, parameters, access functions, and dependences between instructions are abstracted in the polyhedral model as vectors and matrices. The transformation process is relatively simple to understand, it can be seen as a mapping of the abstracted loop from a vector space to another one that exposes optimizations such as parallelization. A transformation is nothing but a multiplication of the transformation matrix to the indices of the loop nest. Moreover, combining optimizations is a product of transformation matrices. There are three unimodular transformations matrices:

- The interchange transformation matrix permutes two loops in the nest. It usually enhances the array access by decreasing the number of cache misses. For a 2-depth loop nest, the scheduling function maps iteration (i,j) to (j,i).

$$\begin{pmatrix} i & j \end{pmatrix} \times \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} j & i \end{pmatrix}$$

- The reversal transformation matrix reverses the iteration directions. The usefulness of this transformation lies in its capacity of enabling some optimizations. For instance, for a 2-depth loop nest, the scheduling function maps iteration (i,j) to (-i,j).

$$\begin{pmatrix} i & j \end{pmatrix} \times \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} -i & j \end{pmatrix}$$

- The skewing transformation modifies the bounds of the inner loop with a value that is dependent on the outer loop index. This optimization enables parallelization where the dependence prevails on the outermost loop. This would will be illustrated later in this section. For instance, the below matrix maps iteration (i,j) to (i+j, i)

$$\begin{pmatrix} i & j \end{pmatrix} \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} i+j & i \end{pmatrix}$$

A transformation matrix is only legal when the resulting dependences are lexicographically positive in the target space. Formally, let d be the distance vector in the original iteration space, T a linear transformation, and $T \times d$ the transformed distance vector. The transformation is only legal when the transformed distance vector is lexicographically positive: $T \times d > 0$

The parallelization of a loop requires that there is no dependence between different iterations. To illustrate the use of skew transformation to expose parallelism, we propose the loop nest in Figure 4.8.a, which performs a stencil computation. The loop dependence vectors are (0,1), (1,0), and (1,1), that are represented on top of Figure 4.8.a. The loop is not permutable unless it undergoes the skew transformation. Figure 4.7, shows the process of parallelization. At the first step, we apply the skew transformation on the dependences, which results with lexicographically positive vectors. This legal transformation exposes paralellism in the inner loop since it yields independent instances on each of the jth iteration. The second step determines the new iteration domain of the statement, by multiplying the old bounds with the inverse of the skew function. The third step determines the new access functions in a similar way by multiplying the access functions by the inverse of the skew function. The produced loop nest exposes a parallel inner loop. Now it is possible to replace the access functions, bounds, and the inner loop with a parallel for.

4.1.7.2 Polyhedral transformation

As opposed to unimodular transformations, affine transformations can be applied to imperfect loop nest. In addition to interchange, reversal, and skewing transformations, the affine transformations enable loop fission and fusion. The approach is discussed in [Fea93] and uses both Farkas lemma and Fourier-Motzkin elimination method to find a set of possible schedules for a SCoP.

The Farkas lemma states that for a non-empty polyhedron defined as a set of affine inequalities in the form $f(\vec{x}) = A \times \vec{x} + \vec{a} \geq \vec{0}$, there is a solution $\vec{x} \geq \vec{0}$ if and only if there exist a vector $\vec{\lambda}$, such that $\vec{\lambda}^T \times A \geq \vec{0}$ and $\vec{\lambda}^T \times \vec{a} \leq \vec{0}$. We can write:

$$f(\vec{x}) = A \times \vec{x} + \vec{a} = \lambda_0 + \vec{\lambda}^T \times (A \times \vec{x} + \vec{a})$$

such that the Farkas multipliers λ are positive ($\lambda_0 \geq 0$ and $\vec{\lambda} \geq \vec{0}$)

The aim is to find a set of schedules that does not violate the dependences between instances of dependent instructions. The key idea [Fea93] is that the difference between

Testing the legality of the transformation:

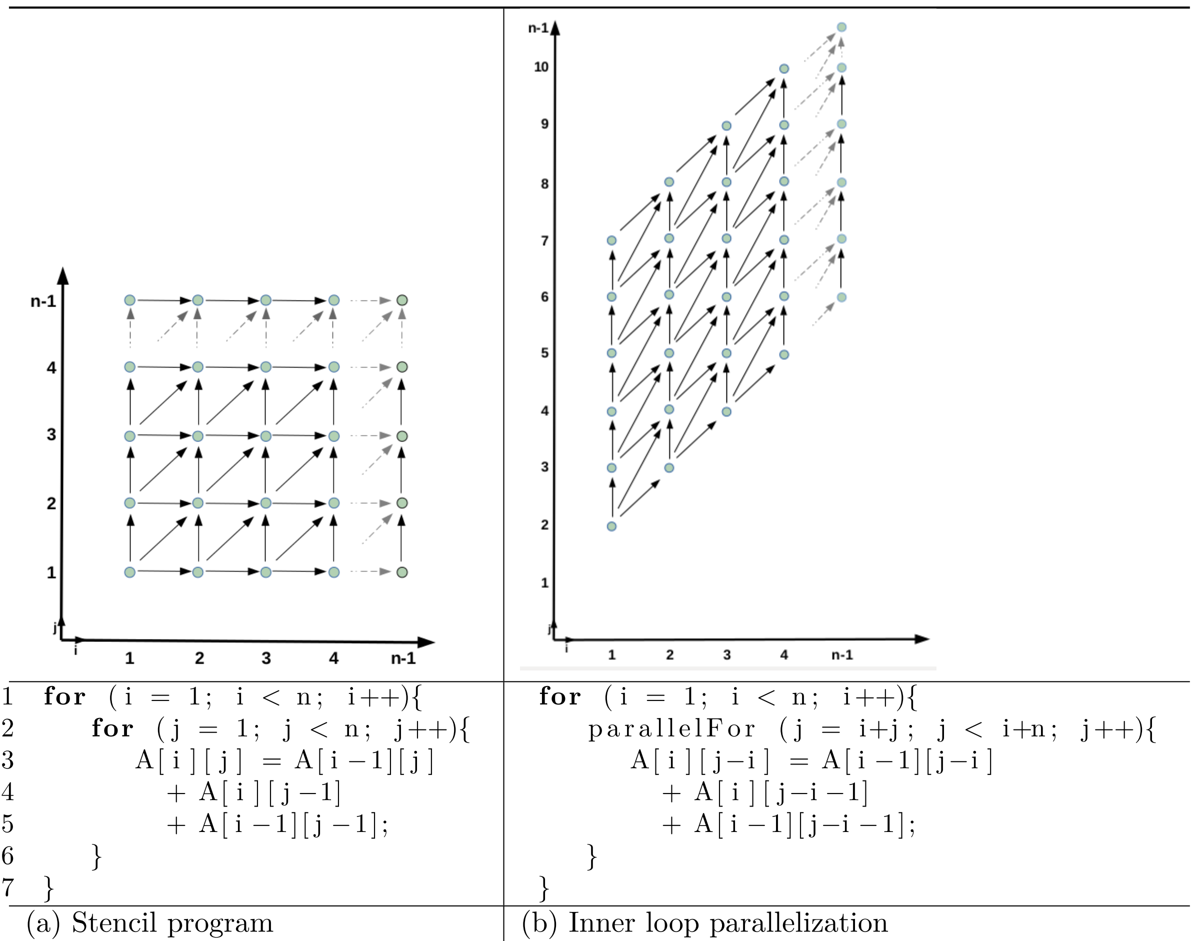
$$T_{skew} \times \vec{d_1} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \times \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$T_{skew} \times \vec{d_2} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$T_{skew} \times \vec{d_3} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

Original	Transformed
Original array bounds: - A $\times \begin{pmatrix} i \\ j \end{pmatrix} =$ $\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} -1 \\ n-1 \\ -1 \\ n-1 \end{pmatrix}$	Transformed array bounds: - A $\times T^{-1} \times \begin{pmatrix} i' & j' \end{pmatrix} =$ $\begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} \times \begin{pmatrix} i' \\ j' \end{pmatrix} \leq \begin{pmatrix} -1 \\ n-1 \\ -1 \\ n-1 \end{pmatrix}$
Original read accesses functions: - $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \end{pmatrix} =$ $\begin{pmatrix} i \\ j-1 \end{pmatrix}$ - $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ 0 \end{pmatrix} =$ $\begin{pmatrix} i-1 \\ j \end{pmatrix}$ - $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ -1 \end{pmatrix} =$ $\begin{pmatrix} i-1 \\ j-1 \end{pmatrix}$	Transformed read access functions: - $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} \times \begin{pmatrix} i' \\ j' \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \end{pmatrix} =$ $\begin{pmatrix} i'-1 \\ j'-i' \end{pmatrix}$ - $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} \times \begin{pmatrix} i' \\ j' \end{pmatrix} + \begin{pmatrix} -1 \\ 0 \end{pmatrix} =$ $\begin{pmatrix} i' \\ j'-i'-1 \end{pmatrix}$ - $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} \times \begin{pmatrix} i' \\ j' \end{pmatrix} + \begin{pmatrix} -1 \\ -1 \end{pmatrix} =$ $\begin{pmatrix} i'-1 \\ j'-i'-1 \end{pmatrix}$
Original write access function: - $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} =$ $\begin{pmatrix} i \\ j \end{pmatrix}$	Transformed write access function: - $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} \times \begin{pmatrix} i' \\ j' \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} =$ $\begin{pmatrix} i' \\ j'-i' \end{pmatrix}$

Figure 4.7: Unimodular skewing transformation: computation of new loop bounds and access functions




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

```
for (i = 1; i < n; i++){
    parallelFor (j = i+j; j < i+n; j++){
        A[i][j-i] = A[i-1][j-i]
            + A[i][j-i-1]
            + A[i-1][j-i-1];
    }
}
```

(a) Stencil program | (b) Inner loop parallelization

Figure 4.8: Unimodular skewing transformation

the schedules of dependent statements, which is an affine function, is positive. In order to generate the schedules, their associated coefficient matrices are considered as unknowns, their difference is equated to the second part of the Farkas lemma. Solving the equations and inequalities by eliminating the Farkas multipliers using Fourier-Motzkin, a set of constraints on the unknown coefficient matrices is generated which allows to define the schedules.

Example 4.4: Illustration of Farkas lemma and Fourier-Motzkin

Let θ_{S_1} be the general form of schedule function of S_1 in the SCoP of Figure 4.1:

$$\theta_{S_1}(\begin{pmatrix} x \\ y \end{pmatrix}) = v_0 \times x + v_1 \times y$$

where v_1 and v_2 are integers.

The dependence of write and read instances occuring in two different iterations can be expressed in the form:

$$\theta_{write}(\begin{pmatrix} i \\ j \end{pmatrix}) - \theta_{read}(\begin{pmatrix} i' \\ j' \end{pmatrix}) - 1 = \lambda_0 + A \times \begin{pmatrix} i \\ j \\ i' \\ j' \\ \vec{p} \end{pmatrix}, \vec{p} = \begin{pmatrix} 1 \\ n-1 \end{pmatrix}$$

Applying the first in the second equation yields:

$$(v_0 \times i + v_1 \times j) - (v_0 \times i' + v_1 \times j') - 1 = \lambda_0 + A \times \begin{pmatrix} i \\ j \\ i' \\ j' \\ 1 \\ n-1 \end{pmatrix}$$

Developing the right most part of the equality in the previous equation yields, by

replacing matrix A with the dependence polyhedron:

$$= \lambda_0 + \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \\ \lambda_5 \\ \lambda_6 \\ \lambda_7 \\ \lambda_8 \\ \lambda_9 \\ \lambda_{10} \\ \lambda_{11} \\ \lambda_{12} \end{pmatrix}^T \times \begin{pmatrix} 1 & 0 & 0 & 0 & -1 & 0 \\ -1 & 0 & 0 & 0 & -1 & 1 \\ 0 & 1 & 0 & 0 & -1 & 0 \\ 1 & -1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 1 \\ 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & -1 & 1 & 0 \\ \hline 1 & 0 & -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 & 1 & 0 \\ 1 & 0 & -1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 & -1 & 0 \end{pmatrix} \times \begin{pmatrix} i \\ j \\ i' \\ j' \\ 1 \\ n \end{pmatrix} \frac{\geq}{=} \vec{0}$$

Hence, we can write

$$(\ \mathrm{v}_0 \times i + v_1 \times j) - (v_0 \times i' + v_1 \times j') - 1 = \lambda_0 + (\lambda_1 - \lambda_2 + \lambda_4 + \lambda_9 - \lambda_{11}) \times i + (\lambda_3 - \lambda_4 + \lambda_{10} - \lambda_{12}) \times j + (\lambda_5 - \lambda_6 - \lambda_9 + \lambda_{11}) \times i' + (\lambda_7 - \lambda_8 - \lambda_{10} - \lambda_{12}) \times j' + (\text{-}\lambda_1 - \lambda_2 - \lambda_3 - \lambda_4 - \lambda_5 - \lambda_6 - \lambda_7 + \lambda_8 + \lambda_9 + \lambda_{10} - \lambda_{11} - \lambda_{12}) \times 1 + (\lambda_2 + \lambda_6) \times (n-1)$$

In order to find the constraints on v_0 and v_1, we equate accordingly the factors of $(i,j)^T$ and $(i',j')^T$ of both sides of the previous equation, which yields:

$$\begin{aligned} v_0 &= \lambda_1 - \lambda_2 + \lambda_4 + \lambda_9 - \lambda_{11} && (1) \\ v_1 &= \lambda_3 - \lambda_4 + \lambda_{10} - \lambda_{12} && (2) \\ -v_0 &= \lambda_5 - \lambda_6 - \lambda_9 + \lambda_{11} && (3) \\ -v_1 &= \lambda_7 - \lambda_8 - \lambda_{10} - \lambda_{12} && (4) \\ -1 &= \lambda_0 - \lambda_1 - \lambda_2 - \lambda_3 - \lambda_4 - \lambda_5 - \lambda_6 - \lambda_7 + \lambda_8 + \lambda_9 + \lambda_{10} - \lambda_{11} - \lambda_{12} && (5) \\ 0 &= \lambda_2 + \lambda_6 && (6) \end{aligned}$$

The addition of equation 1, 2, and 5 eliminates λ_9, λ_{10}, λ_{11} and λ_{12}:

$$v_0 + v_1 - 1 = \lambda_0 - (2 \times \lambda_2) - \lambda_4 - \lambda_5 - \lambda_6 - \lambda_7 + \lambda_8 + (2 \times \lambda_9) + (2 \times \lambda_{10}) - (2 \times \lambda_{11}) - (2 \times \lambda_{12})$$

The latter equation can be written as the following inequality:

$\lambda_0 - (2 \times \lambda_2) - \lambda_4 - \lambda_5 - \lambda_6 - \lambda_7 + \lambda_8 + (2 \times \lambda_9) + (2 \times \lambda_{10}) - (2 \times \lambda_{11}) - (2 \times \lambda_{12}) \geq v_0 + v_1 - 1$

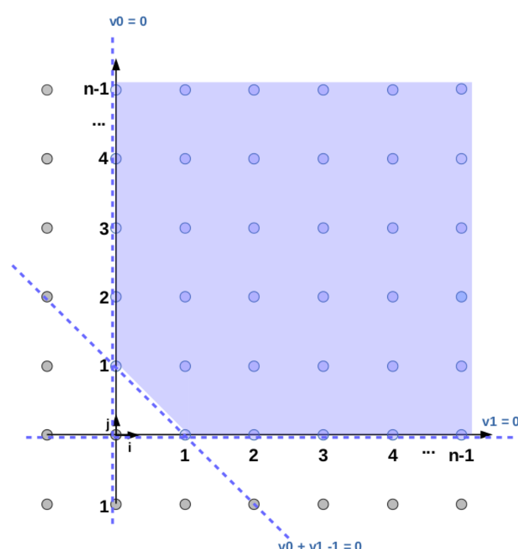
such that: $\lambda_i \geq 0$
Then:
$\lambda_0 + \lambda_8 + (2 \times \lambda_9) + (2 \times \lambda_{10}) \geq v_0 + v_1 - 1$

Hence:
$\lambda_0 + \lambda_8 + (2 \times \lambda_9) + (2 \times \lambda_{10}) \geq \max(0, v_0 + v_1 - 1)$

The constraints on v_0 and v_1 are:

$$\begin{aligned} v_0 + v_1 - 1 &\geq 0 \\ v_1 - 1 &\geq 0 \\ v_0 - 1 &\geq 0 \end{aligned}$$

The solutions are visualized bellow in the blue region. For instance, the solution (0,0) does not correspond to a valid schedule. Solution (1,0) results in a valid schedule. When multiplied with the dynamic instances in the domain, it gives instances belonging to each horizontal line the same logical date; hence, it parallelizes the outermost loop. Solution (0,1) gives the dynamic instances belonging to the vertical line the same execution date, which allows to parallelize the inner loop. Finally, solution (1,1) allows a wave front execution.



4.2 Vectorization of Binary Code

4.2.1 Principle of scalar into vector optimization

It may happen that some applications contain scalar codes that were not vectorized by the compiler, even when they contain loops that have the properties required for correct

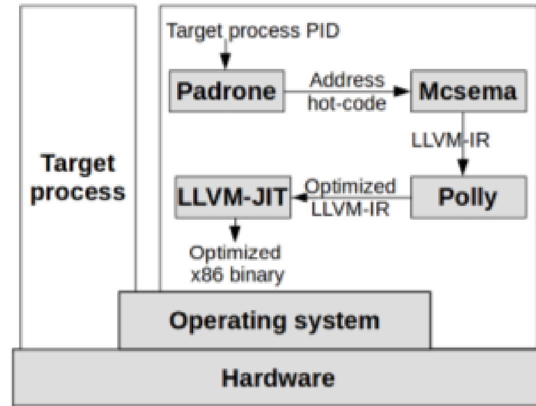


Figure 4.9: Process automatic vectorization

and beneficial SIMD processing. One reason may be that the source code has been compiled for an architecture that does not support SIMD instructions. Another reason is that some compilers are not able to vectorize some class of codes [15], because they do not embed advanced data dependence analysis and loop transformation capabilities, as provided by the polyhedral model [12]. In this section, we widen the optimization scope by auto-vectorizing long running inner loops which contain scalar instructions. We rely on an open source framework, McSema [16], that lifts binary code into the LLVM-IR so that higher level loop optimizations can be performed. From the IR, it is then possible to delegate the vectorization burden to another framework, Polly [13] of section 4.2.3, implementing techniques of the polyhedral model to vectorize candidate loops. Finally, we compile back the vectorized IR into binary using the LLVM JIT compiler. Figure 4.9 shows the auto-vectorization process. This approach constitutes a proof-of-concept for dynamic full vectorization using an ambitious mechanism. Note that it is not meant to be integrated with the re-vectorization presented in the previous chapter, but rather to assess feasibility and performance of such a technique. Indeed, additionally to vectorization, the presented mechanism may support general high-level loop optimizing transformations such as loop interchange, loop tiling, loop skewing and loop parallelization.

4.2.2 Binary into intermediate representation using McSema

4.2.2.1 McSema

McSema is an open source decompilation framework that statically translates native x86 code into LLVM IR. It is capable of translating integer, floating point, and SSE instructions. There are other decompilation tools and libraries such as Dragger, Fracture, and libbeauty capable of lifting one or multiple instruction sets compliant to x86, ARM, and PPC. The main tasks of McSema are: First, it parses and abstracts into internal data structures the various file formats specifically portable executable PE,

common object file format COFF, and executable and linkable format ELF. Second, it recovers the control flow graph of the binary. Third, it translates instructions into an LLVM intermediate representation.

To extract efficiently information from executables and object files McSema parses a variety of file formats among them the ELF format. Concerning the PE/Coff formats, it relies upon a standalone open source tool, whose name is PE parse. Then, McSema abstracts it into a data type that provides functions; such as, reading a byte at some virtual address, or getting the entry point of the base address of a program, etc. The other functionalities are listed in McSema's documentation page.

The first stage of decompilation is disassembly. This issue is already discussed in literature and it turns out that it hides a spectrum of problems that varies from simple to difficult to solve. Among these problems, distinguishing code and data that reside on the same address space such as the alignment bytes that usually precede loops or generally branch targets in order to increase performance, variable-length instructions commonly used in CISC architectures rises the difficulty of extracting instruction sequences, and indirect control transfer instructions. The two well known static techniques for disassembly are linear sweep and the recursive traversal disassembly. The linear sweep proceeds by disassembling linearly an instruction after the other starting from the first byte of the code section. This straightforward approach is used by the GNU utility objdump. The shortcoming of the scheme is the alignment bytes, which can be misinterpreted and considered as instructions, this would influence the disassembly of the remainder of the binary resulting with wrong translation. The recursive traversal overcomes the problem by linearly disassembling the code but taking into account the control flow graph. In the sense, it starts to disassemble and when branch is met it recognizes the different targets and disassembles them recursively. Hence, the unreachable alignment bytes found before branch targets are never disassembled. This main barrier of the scheme are the indirect jumps. McSema adopts this method to recover the CFG and the disassembly in a concurrent way.

The second stage of decompilation is lifting the assembly into LLVM IR. Since the control flow structure is already recovered in the previous step, it is a matter of translation while preserving semantics. The execution of some instructions modifies the status registers on which other instructions relie for their execution. In order to maintain the semantic during translation McSema lifts the status registers and then translates the instructions that depend on them accordingly.

4.2.2.2 Integration of Padrone with McSema

McSema is a decompilation framework whose functionality is to lift code from binary into LLVM-IR. First, the tool disassembles the x86 executable and creates an according control flow graph (CFG). Second, it translates the assembly instructions into LLVM-IR. Basically, McSema operates on x86-64 executable files. The bin descend tool takes as arguments the binary file and the name of the function to disassemble, creates a CFG of assembly and marshals this data structure. The cfg to bc tool demarshals the file, and translates assembly instructions into equivalent LLVM-IR form that is written

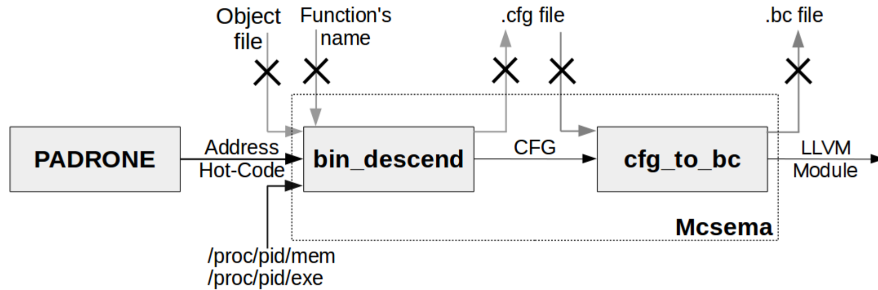


Figure 4.10: Integration of Padrone with McSema

to a bit-code file. We tuned McSema to take as input the address of the frequently executed function provided by Padrone and the processed image. Moreover, we avoid the overhead of writing into data storage by skipping the marshaling/demarshaling process so as the data structure produced by bin descend is passed directly to cfg to bc. Figure 4.10 shows the integration of Padrone with McSema.

4.2.2.3 Adjusting McSema to produce a suitable LLVM-IR

The code lifting from binary to IR obviously requires to preserve the semantics of the binary instructions. FPU and arithmetic instructions usually alter the FPU and arithmetic status registers. Hence, McSema lifts the status registers as well to maintain the state of the processor. It is important to keep track of these bits since subsequent instructions like conditional jumps, bitwise or shift-rotate operations, etc. not only depend on their operands, but also on these flags.

The consequent issue is that the translation of these instructions ends up in generating a large amount of corresponding LLVM-IR instructions. These instructions confuse the loop analyzer of Polly and thus prevent it from optimizing the bit-code. For instance, Table 4.1 depicts the translation of an assembly loop into LLVM-IR. The comparison instruction is a signed subtraction of its operands which alters the AF, SF, OF, PF, CF, ZF flags as well. As a consequence, McSema translates the comparison in line 3 of the first column into lines 6—17 in the second column, where the subtraction is performed (line 9) and the states of the flags are set accordingly. Furthermore, the conditional jump depends on the flags to direct the flow of execution. The Jump if Not Equal (JNE) tests whether the ZF flag is equal to zero. Correspondingly, line 4 of the first column is translated into lines 19—21. To determine the number of the loop iteration, Polly requires a single comparison instruction whose arguments are the induction variable and the loop trip count. However, McSema produces several comparison instructions free from the loop trip count operand. Thus, we prevent McSema from generating the instructions that alter the state of the flags, and we keep track of the subtraction's arguments. Depending on the conditional jump, an appropriate predicate is fed into the comparison instruction. In this example, the comparison takes three arguments, a predicate Signed Less or Equal (SLE), the induction variable and

Table 4.1: Adjusting McSema to produce a suitable LLVM-IR

x86 assembly	LLVM-IR produced by McSema	Adjusted LLVM-IR
1. L1:	1. %RAX_val = alloca i64	1. %RAX_val = alloca i64
2. . . .	2. %ZF_val = alloca i1	2. %ZF_val = alloca i1
3. cmp $1024, %rax	3.	3.
4. jne L1	4. %block_L1	4. %block_L1
	5. . . .	5. . . .
	6. /* instructions modify the	6. %156 = load i64* %RAX_val
	7. state of ZF */	7. %157 = icmp sle i64 %156, 4096
	8. %117 = load i64* %RAX_val	8. br i1 %157, label %block_L1, label %block_L2
	9. %118 = sub i64 %117, 1024	9. %block_L2
	10. %127 = icmp eq i64 %118, 0	
	11. store i1 %127, i1* %ZF_val	
	12.	
	13. /* instructions modify the	
	14. states of the remaning flags	
	15. AF, SF, OF, PF, CF depending	
	16. on the subtract instruction */	
	17. . . .	
	18.	
	19. %136 = load i1* %ZF_val	
	20. %137 = icmp eq i1 %136, false	
	21. br i1 %137, label %block_L1, label %block_L2	
	22.	
	23. %block_L2	

the loop bound. Lines 6—8 in the third column is the produced semantically equivalent code.

4.2.3 Vectorization of loops in LLVM-IR using Polly

Polly [13] is a static loop optimization infrastructure for the LLVM compiler, capable of optimizing data locality and exposing parallelism opportunities for loop nests that are compliant with the polyhedral model. When the loop bounds and memory accesses are detected as affine, it creates their representation in the polyhedral model, and reschedules the accesses while respecting the data dependencies. The main objective in the current step is to vectorize sequential code. We make use of the compiler passes provided by Polly. First, the canonicalization passes are run, which transform the IR into a suitable form for Polly. Second, the Static Control Parts (SCoPs), which are subgraphs of the CFG defining loops with statically known control flow, are detected. Basically, Polly is able to optimize regions of the CFG, namely loops, with fixed number of iterations and conditionals defined by linear functions of the loop iterators (i.e., induction variables). Third, the SCoPs are abstracted into the polyhedral model. Finally, the data dependencies are computed in order to expose the parallelism in the SCoPs. At

C code		Original LLVM-IR	Produced IR (Mem2reg pass)
	1		
	2	`int test() {`	
	3		`int test() {`
	4	`entry:`	
	5	`    result = alloca int`	`entry:`
	6	`    a = alloca int`	`    a = alloca int`
`int test() {`	7	`    b = alloca int`	`    b = alloca int`
`    int a, b;`	8	`    store int 5, int* a`	`    store int 5, int* a`
`    a = 5;`	9	`    store int 3, int* b`	`    store int 3, int* b`
`    b = 3;`	10	`    var.1 = load int* a`	`    var.1 = load int* a`
`    return a + b;`	11	`    var.2 = load int* b`	`    var.2 = load int* b`
`}`	12	`    var.3 = add int var.1,`	`    var.3 = add int var.1,`
	13	`            var.2`	`            var.2`
	14	`    store int var.3,`	`    br label return`
	15	`            int* result`	
	16	`    br label return`	`return:  ; preds = entry`
	17	`return:`	`    ret int var.3`
	18	`    var.4 = load int* result`	`}`
	19	`    ret int var.4`	
	20	`}`	

Figure 4.11: Applying memory to register pass

the time being, our work is confined only to inner loops. We use a method provided by the dependence analysis pass to check whether the loop is parallel. An algorithm that checks whether the data accesses are consecutive along iterations has been developed. We cast the LLVM-IR loads and stores into vector loads and stores, so that the JIT generates SIMD instructions. The induction variable is modified depending on the vector factor as well.

4.2.3.1 Canonicalization

After lifting the assembly, the LLVM-IR is adjusted to conform to the standards allowing its manipulation in Polly as described previously. At this stage, the IR is optimized using a set of passes available in the LLVM and Polly frameworks. The following briefly describe the passes based on the documentation shipped in the source code:

- LLVM Promote Memory To Register Pass: This optimization replaces a memory reference by a use of register references. This reduces overhead of reading and writing into the stack frame. Figure 4.11 illustrates the transformation, the stack allocation, storing the final value into memory, and reading it again in lines 5, 14, and 18 in the original LLVM-IR (second column) are bypassed by the use of a register in the produced IR (third column).

- Instruction Combining Pass: This optimization reduces the number of instructions by combining some which share an algebraic nature.

```
int %test() {

entry:
        result = alloca int
        a = alloca int
        b = alloca int
        store int 5, int* a
        store int 3, int* b
        var.1 = load int* a
        var.2 = load int* b
        var.3 = add int var.1, var.2
        store int var.3, int* result
        br label return
return:
        var.4 = load int* result
        ret int var.4
}
```

Figure 4.12: Example memory to register pass

- LLVM CFG Simplification Pass: This optimization eliminates dead code by removing unreachable blocks. Besides, it merges basic blocks; for instance, a bloc that is connected to a single successor which by its turn has only one predecessor.
- LLVM Tail Call Elimination: This optimization transforms some recursive functions into a loop specifically the ones whose callee does not interfere with the caller stack frame. The algorithm supports functions that does not fit to the optimization; for instance, functions that contain associative and commutative expression are transformed to a reduction variable.
- LLVM Reassociate: This optimization transforms arithmetic statements with commutitative nature in a way that constant propagation pass performs efficiently. The idea is that it reorders the operands in way that the constants are associated.
- Polly Induction Variable Pass: This optimization simplifies the computation of the induction variable into a canonical form. In other words, the computation is in the form of an increment by a constant amount.

4.2.3.2 ScoP Detection

As explained earlier, the Static Control Part (ScoP) is subtree of the Abstract Syntax Tree of the program whose associated control flow is static. Its detection is less tedious once the previous optimizations already described such as constant propagation, loop normalization, canonicalization of induction variables, and dead code elimination are

already run on the hot section of code by Polly. The ScoP detection pass proceeds by checking each of the largest loops and narrowing down to the inner one until a static control subregion is found in a recursive fashion. The criterias checked by Polly during the validation whether the subregion has a static control are: The subregion has a single entry and exit. The natural loops along the conditions are nested perfectly and whose predicates are affine linear expressions. The loop bounds are described as affine linear functions in the direct enclosing loop iterators or invariant variables. The memory accesses use affine expressions of parameters and loop counters in pointing to memory locations.

4.2.3.3 Scop Extraction

Once the ScoPs are detected, information are collected from the LLVM IR by TempScopInfo class that is later translated into the ScoP class which abstracts the polyhedral model representation. TempScopInfo pass makes use of the LLVM ScalarEvolution pass to extract loop properties. For example, it is capable of recognizing the induction variable of the loop, representing it as a scalar expression, and analyzing it to finally obtain the trip count. Polly adopts the same polyhedral representation used by graphite compiler and other tools such as ClooG and Pluto. The ScopInfo pass is responsible of building Polly IR which is nothing but the polyhedral representation of the static control region. This latter is abstracted in the Scop class which is an aggregation of the instructions executed in the Scop and constant global parameters which are static scalar integer values during the execution but their values are know at runtime such as loop bounds. Each of the instructions is a tuple that consists of its domain, its array references where data is written into, its array references where data is read from, and its affine schedule. For more detailed information on the abstraction and extraction of SCoPs, iteration domains, access relations and initial schedule refer to [VG12]

4.2.3.4 Dependence analysis

The dependence analysis pass makes use of Interger Set Library (ISL) [Ver10] to compute dependencies. It is an implementation of data flow algorithm published in [VNS13] which is a variation of algorithms focusing on static affine programs [Fea91], [Mas94] and [PW94]. In a nutshell, the algorithm seeks to find out the relations that describe the number of iterations between last source accesses that feed each of the sink accesses.

4.2.3.5 Scheduling

This pass delegates the work of calculating an optimized schedule adequate to parallelism and tileability to the ISL libary [Ver10]. The algorithm is based on the affine form of Farkas lemma and Fourier-Motzkin. ISL and Pluto implementations are variations of the same algorithm published by [BHRS08a]. The method was first suggested in [LL97] that finds a schedule which divides the instances of a domain into independent partitions for parallel execution, and their independence would minimize communica-

Pointer arithmetic recurrence	Array subscript

int A[];
 ...
 for(i=0; i<n; i++) {
   *A++;
 }
	int A[];
	 ...
	 for(i=0; i<n; i++) {
	   A[i];
	 }

Figure 4.13: Difference between pointer arithmetic recurrence and array subscript

tion. [BHRS08a] extends this latter [LL97] by minimizing communication using tiling transformation.

4.2.3.6 Vectorization

Once Polly's dependence pass certifies that statements in an inner loop are independent, an algorithm is developed to analyze whether the operations in the loop involve operands that have consecutive memory accesses. When the test is positive scalar loop is converted into vector one; otherwise, the vectorization process is not appropriate and hence cleared of its transformation responsibility.

Detection of consecutive memory access and use of constants :

The consecutive memory access detection is a pattern-matching algorithm. During the canonicalization phase of the program, the induction variable simplification pass transforms loop into a form in which it iterates over a single canonical index and sets its initial value to 0. Moreover, it solves the pointer arithmetic recurrence in memory access statements in a way that they use array subscripts. The difference between these memory accesses is shown in Figure 4.13.

As long as strength reduction optimization is not run on the code, the consequence of the transformation of memory accesses is that the intermediate representation would use indexed addressing that has the following form:

$$Base + index * scale$$

Where the base is the beginning of the array, index is an affine function of the induction variable and the scale is a constant.

For a consecutive access, since the index increments only by one, the scale should absolutely be equal to the size of the data type of the access; otherwise, the access is not consecutive.

The abstraction of a program in the LLVM adopts both a double linked list that links each instruction to its successor and predecessor; besides, a DAG abstraction in which instructions point to their operands. The algorithm iterates over the instructions residing within the loop. When a memory write instruction is found, the DAG is traversed by the Depth First Search (DFS) algorithm to search for the index and scale. It verifies that the index is nothing but the induction variable and that it increments by 1. It checks that the scale has a value that is equal to the size of the data type of the pointer where data will be written to. The algorithm continues the traversal until

```
for(i = 0; i < n; i++)
   A[i] = A[i] + 1;
```

Figure 4.14: C program which illustrates consecutive memory access

```
1  block0x80483f7 :           ; preds = block0x804883e0 ,
2                                block0x80483f7
3
4    EAXval.0 = phi i32 (0, block0x804883e0),
5                  (64, block0x80483f7)
6    61 = mul i32 EAXval0, 4
7    62 = getelementptr @data0x804a020,
8            i32 0, i32 0, i32 61
9    63 = bitcast i8* 62 to 32*
10   64 = load i32* 63, align 4
11   65 = add i32 64, 1
12   store i32 65, i32* 63, align4
13   66 = add i32 EAXval.0, 1
14   exitcond = icmp ne i32 66, 1023
15   br i1 exitcond, label block0x80483f7, label block
```

Figure 4.15: LLVM IR which illustrates consecutive memory access

it reaches memory reads (if any) and does similar verifications on the index and scale to confirm the consecutive accesses. The memory access are marked as visited during the graph traversal, so as during the walk over the instructions located within the loop, the marked memory accesses are not checked again.

To illustrate the test, consider the example in Figure 4.14. The binary of this program is lifted into an LLVM IR using McSema which can be seen in 4.15. This latter is abstracted into the polyhedral model using Polly passes, and the dependence pass confirmed that the loop is parallel. The check of consecutive memory access is positive since as easily seen in the Direct Acyclic Graph (DAG) in Figure 4.16 the unit stride is equal to four bytes for the write instruction and one of the memory reads operands used in the addition. The second operand in the memory read is a detected to be a constant. Hence, this code is eligible for the conversion of scalar into vector instructions.

Conversion of instructions and loop bound

The linked list that connects the inner loop's instructions is traversed. Whenever a store instruction is met, the algorithm investigates whether the operand that it stores involve an arithmetic or logical instruction; besides, it checks whether this latter have at least a memory access whose operands is written as a function of the induction variable and a global variable (which is the array's begining). Moreover, it checks whether the pointer to memory of the write instruction is a function of both a base address and the induction variable. In this case, all of the store, arithmetic, and load can be converted into vector instructions. For instance, the write instruction in line 12 Figure 4.15,

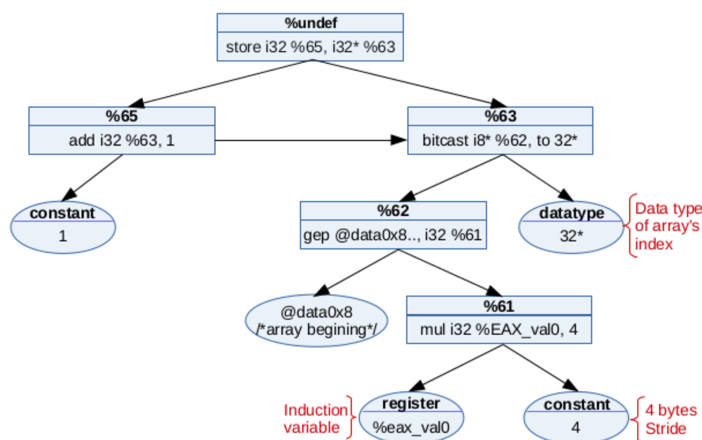


Figure 4.16: DAG which illustrates consecutive memory access

stores the results of the arithmetic instruction in line 65 whose left operand involves the addition of the arrays begining in line 7 with an index calculated in line 6 which is multiplication of an integer data type of size four with the induction variable.

The conversion of scalar into vector rises the modification of index to memory considered for computation at a specific iteration. For instance, the program shown in Figure 4.15 requires the modification of the instruction in line 6 so as chuncks of memory are read and written to. The computation involves the multiplication of the induction variable by the size of the data type and by the vector factor.

Trip count modification :

The trip count becomes the ceiling function of the old number of iteration divided by the vector factor. When the old trip count is not a multiple of the vector factor, the body of the scalar loop is inserted into the rear of the vector loop to compute the remaining scalar iterations starting from the largest multiple of the vector factor which is strictly less than the trip count associated with the scalar version of the loop.

Alignement constraint

At the LLVM IR level of abstraction the JIT is responsible of assigning aligned or non aligned instructions in the case the arrays are aligned on multiples of 16 bytes. For a better performance a loop unroll is performed for the first needed number of iterations so as many as possible of instructions in vector loop can profit from a translation into aligned instructions. This is done by checking whether the base address of arrays used in the loop are aligned on 32 or 64 bytes depending whether the loop is vectorized into SSE or AVX.

4.2.4 LLVM JIT tuning

4.2.4.1 Handling global variables

The lifting of binary into LLVM-IR consists of both lifting the instructions and the memory addresses on which they operate. Hence, McSema declares arrays on which the IR instructions perform their operations. However, compiling the IR yields instructions and array allocations in the optimizer process space. The generated instructions perform their operations on the initiated arrays. Therefore, injecting this code into the target process would result in bad address references.

4.2.4.2 Marking the IR operands

As a solution, we mark the operands lifted by McSema with the physical addresses in the original binary; so as, while JITting, the recovered addresses are encoded in the generated instructions. The compilation of LLVM-IR into binary goes through multiple stages. At the beginning, the IR instructions form an Abstract Syntax Tree (AST) that is partially target independent. At each stage, the instruction in the AST is reincarnated into a different data type which is decorated with more target dependent information. At each of these phases, the addresses recovered, while McSema lifts the original binary, are transferred until the generated instructions are encoded

4.3 Conclusion

This chapter focuses on the vectorization of scalar loops. The first section is devoted to polyhedral model's theory which abstracts regions of the process whose control flow is static and performs transformations in the vector space. The second section puts the theory into practice by interconnecting open-source frameworks and tuning them to coherently work together. Basically, the hot-code is detected using Padrone which communicates the information to McSema that lifts it into LLVM-IR. The latter is cleaned to become Polly friendly. Then it is abstracted in the polyhedral model, that allows extracting dependence relations between instructions and checking whether innerloops are parallel. Moreover, we show a technique to test consecutive memory accesses. When the loop can be vectorized, scalar instructions are transformed to their vector counterparts. Finally, the IR is compiled using the LLVM Just-In-Time compiler.

Chapter 5

Experimental Results

In this section, we present our experimental apparatuses and the results obtained for both dynamic vectorization transformations adopted in our work. We show for each of the approaches the hardware and software environments, the benchmarks, as well as the metric and experimental results.

5.1 Re-Vectorization experimental results

5.1.1 Hardware/Software

Our experiments were conducted with a 64-bit Linux Fedora 19 workstation featuring an Intel i7-4770 Haswell processor clocked at 3.4 GHz. Turbo Boost and SpeedStep were disabled in order to avoid performance measure artifacts associated with these features.

We observed that different versions of GCC produce different results. We made our experiments with two relatively recent version of GCC: GCC-4.7.2 (Sep. 2012) and GCC-4.8.2 (Oct. 2013) available on our workstation. ICC-14.0.0 was used whenever GCC was not able to vectorize our benchmarks.

5.1.2 Benchmarks

We handled two kinds of benchmarks. The first kind consists in a few hand-crafted loops that illustrate basic vectorizable idioms. The second kind is a subset of the TSVC suite [MGG+11]. Table 5.1 summarizes the main features for each benchmark. All TSVC kernels manipulate arrays of type `float`. We also manually converted them to `double` to enlarge the spectrum of possible targets and assess the impact of data types.

We compiled most of our benchmarks with GCC using flags `-O3 -msse4.2` (`-O3` activates the GCC vectorizer). Only `s311` and `s314` were compiled with ICC because both versions of GCC were unable to vectorize them.

We decouple the analysis of the optimizer overhead, and the performance of the re-vectorized loops.

Table 5.1: Experimental Kernels

Name	Short description
vecadd	addition of two arrays
saxpy	multiply an array by a constant and add a second
dscal	multiply an array by a constant
s000	addition and multiplication of arrays
s115	triangular saxpy
s125	product and addition of two-dimensional arrays
s174	addition of an array with a part of the second array storing in an other part of the latter
s176	convolution
s251	scalar and array expansion
s311	sum of elements of a single array (reduction)
s314	search for maximum element in an array (reduction)
s319	sum of elements of multiple arrays
s1351	addition of two arrays using `restrict` pointers

Each benchmark (both hand-crafted and TSVC) essentially consists in a single intensive loop that accounts for nearly 100 % of the run time. This is classical for vectorization studies as it shows the potential of the technique (its asymptotic performance). As per Amdahl's law [Amd67], the speedup on a given application can easily be derived from the weight of the loop in the entire application: let α be the weight of the loop, and s the speedup of this loop, the overall speedup is given by:

$$s' = \frac{1}{1 - \alpha + \frac{\alpha}{s}}$$

As expected, $\lim_{\alpha \to 1} s' = s$. This is the value we observe experimentally.

5.1.3 Performance Results

We assess the performance of our technique by means of two comparisons. First we measure the raw speedup, i.e. we compare the transformed AVX-based loop against the original SSE-based loop. Then, we also compare it against the native AVX code generated by GCC with flags `gcc -mavx -O3` (except `s311` and `s314` whose native AVX codes are generated by ICC with flags `icc -mavx -O3`). Table 5.2 reports the speedups of both native compiler for AVX and our re-vectorizer compared to SSE code. In the case of our re-vectorizer, we also report how it compares to the native compiler targeting AVX. These numbers are shown graphically in Figures 5.1 and 5.2 for data type `float` and `double` respectively. As an example, the first row (`dscal`) shows that the AVX code produced by GCC runs 1.4× faster than the SSE version. The code produced by our re-vectorizer runs 1.66× faster than the SSE version, that is a 19 % improvement over the AVX version.

We confirmed that the difference in the code quality between the SSE references produced by both compilers is small compared to the variations observed between SSE

Table 5.2: Performance Improvement over SSE

	Kernel	GCC 4.8.2 native AVX	GCC 4.8.2 our revect.	GCC 4.8.2 % vs. native	GCC 4.7.2 native AVX	GCC 4.7.2 our revect.	GCC 4.7.2 % vs. native
float	dscal	1.40×	1.66×	+19%	1.60×	1.37×	-14%
float	saxpy	1.10×	1.67×	+52%	1.88×	1.35×	-28%
float	vecadd	1.00×	1.66×	+66%	1.74×	1.26×	-28%
float	s000	0.99×	1.19×	+20%	1.07×	0.69×	-36%
float	s125	1.02×	1.25×	+23%	1.49×	1.25×	-16%
float	s174	0.86×	1.34×	+56%	1.50×	1.09×	-27%
float	s176	1.48×	1.52×	+3%	1.54×	1.22×	-21%
float	s251	1.17×	1.35×	+15%	1.57×	0.92×	-41%
float	s319	1.42×	1.61×	+13%	1.80×	1.05×	-42%
float	s1351	0.90×	0.91×	+1%	0.92×	0.87×	-5%
float	s115	translator aborts					
double	dscal	1.17×	1.11×	-5%	1.60×	1.31×	-18%
double	saxpy	1.01×	1.18×	+18%	1.57×	1.10×	-30%
double	vecadd	0.82×	1.34×	+63%	1.47×	0.97×	-34%
double	s000	0.95×	0.99×	+5%	0.98×	0.67×	-32%
double	s125	0.90×	0.91×	+0%	0.91×	0.91×	0%
double	s174	0.89×	1.33×	+51%	1.49×	0.98×	-34%
double	s251	0.94×	0.96×	+2%	0.97×	0.96×	-1%
double	s319	1.25×	1.33×	+6%	1.33×	0.90×	-33%
double	s1351	0.88×	0.91×	+3%	0.91×	0.90×	-1%
double	s115	translator aborts					
double	s176	translator aborts					

	Kernel	Reduction with icc native	Reduction with icc our revect.	Reduction with icc % vs. native
float	s311	1.80×	1.79×	-0.55%
float	s314	1.79×	1.79×	0%
double	s311	1.79×	1.79×	0%
double	s314	translator aborts		

+ native AVX: the execution time of the native SSE divided by native AVX.
+ our revect.: the execution time of native SSE divided by AVX generated by our revectorizer.
+ % vs. native: the percentage of improvement of "our revect" compared to "native AVX".

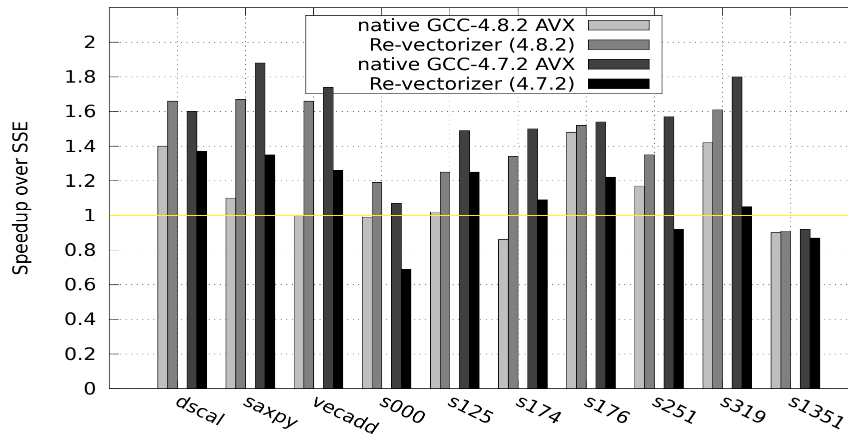


Figure 5.1: Speedups for type `float`

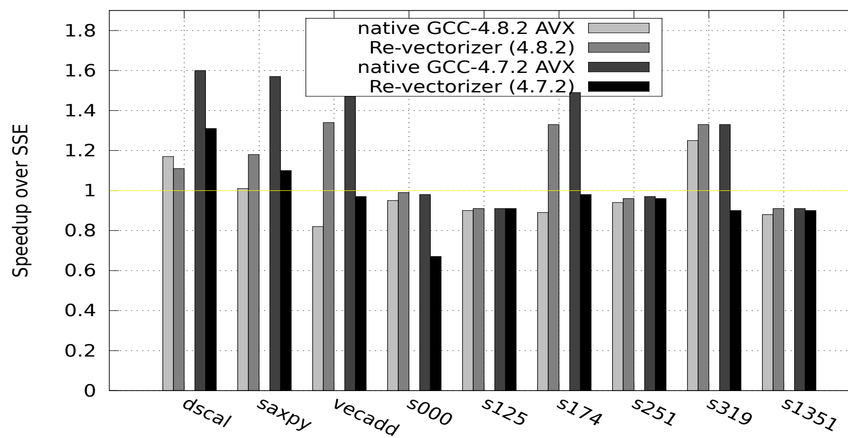


Figure 5.2: Speedups for type `double`

and AVX.

GCC-4.8.2 As a general trend, our re-vectorizer is able to improve the performance of eligible loops, up to 67 %. Surprisingly, we also constantly outperform GCC for AVX, up to 66 % in the case of `vecadd`. The reasons are:

1. When targeting AVX, GCC-4.8.2 generates a prologue to the parallel loop to guarantee alignment of one of the accessed arrays. Unfortunately, the loop does not take advantage of the alignment and relies on unaligned memory accesses (`vmovups` followed by `vinsertf128` when a single `vmovaps` sufficed). When targeting SSE, there is no prologue, and the loop relies on 16-byte aligned memory accesses. In fact, AVX code generated by GCC-4.7.2 is more straightforward, without prologue, and similar to our own code generation, and corresponding performance also correlates.

2. GCC-4.8.2 tries to align only one of the arrays. Testing for alignment conditions of all arrays and generating specialized code for each case would result in excessive code bloat. The static compiler hence relies on unaligned memory accesses for all other arrays. Because we operate at runtime, we have the capability to check actual values and generate faster aligned accesses when possible.

In the case of `s115`, we correctly identified the vectorized hot loop, but we were not able to locate its sequential counterpart (the loop at label L5 in Figure 3.3) in the function body, needed to execute a few remaining iterations when the trip count is not a multiple of the new vectorization factor (as described in Section 3.2.5.2). The reason is that the native compiler chooses to fully unroll this epilogue. Our optimizer simply aborted the transformation.

The only negative effect occurs with `s1351`, with a 9% slowdown. Note that the native AVX compiler also yields to a 10 % slowdown. In this example, the compiler generates unaligned packed instructions. Precisely, a combination of two instructions that move separately the low and high 128 bits of operands between memory and registers. This degrades the performance of AVX. To verify that it is an alignment issue, we made a manual comparison between SSE and AVX versions of `s1351` and forced data to be 32-byte aligned. As consequence, the compiler generates aligned instructions. Under this circumstance, the AVX version outperforms SSE.

Performance of our re-vectorizer as well as native GCC AVX is generally lower when applied to kernels operating on type `double`. The reason is that arrays with the same number of elements are twice larger, hence increasing the bandwidth-to-computation ratio, sometimes hitting the physical limits of our machine, as well as increasing cache pressure. We confirmed that halving the size of arrays produces results in line with the `float` benchmarks.

Three benchmarks failed with type `double`: `s115`, `s176`, and `s314`. This is due to a current limitation in our analyzer: the instruction `movsd` may be translated in two different ways, depending on the presence of another instruction `movhpd` operating on

the same registers. Our analyzer currently considers instructions once at a time, and must abort. Future work will extend the analysis to cover such cases.

GCC-4.7.2 With GCC-4.7.2, our re-vectorizer sometimes degrades the overall performance compared to SSE code. We observe that this happens when the same register (`ymm0`) is used repeatedly in the loop body to manipulate different arrays. This increases significantly the number of partial writes to this register, a pattern known to cause performance penalties [Int14a]. This is particularly true in the case of `s125`. Despite these results, since our optimizer operates at runtime, we always have the option to revert to the original code, limiting the penalty to a short (and tunable) amount of time.

As opposed to GCC-4.8.2, we systematically perform worse than native AVX. This is expected because the native AVX compiler often has the capability to force alignment of arrays to 32 bytes when needed. Since we start from SSE, we only have the guarantee of 16-byte alignment, and we must generate unaligned memory accesses. The net result is the same number of memory instructions as SSE code, while we save only on arithmetic instructions. Note, though, that we do improve over SSE code in many cases, and we have the capability to revert when we do not.

ICC-14.0.0 For both of `s311` and `s314`, our re-vectorizer produces codes that run almost 1.80× faster than native SSE, and they have almost the same performance as the native AVX.

5.1.4 Overhead

The overhead includes profiling the application to identify hot spots, reading the target process' memory and disassembling its code section, building a control flow graph and constructing natural loops, converting eligible loops from SSE to AVX, and injecting the optimized code into the code cache. Profiling has been previously reported [RRC+14] to have a negligible impact on the target application. With the exception of code injection, all other steps are performed in parallel with the execution of the application.

On a multicore processor, running the re-vectorizer on the same core as the target improves communication between both processes (profiling, reading original code, and storing to the code cache) at the expense of sharing hardware resources when both processes execute simultaneously. The opposite holds when running on different cores. Since our experimental machine features simultaneous multi-threading (Intel Hyperthreading), we also considered running on the same physical core, but two different logical cores.

In our results for all configurations, the time overhead remains close to the measurement error. Table 5.3 reports the overhead in milliseconds of re-vectorizing loops, for each benchmark. We ran the experiment ten times and we report the average and the standard deviation.

On the application side, storing and restoring the `ymm` registers represent a negligible overhead, consisting in writing/reading a few dozen bytes to/from memory.

name	s000	s125	s174	s176	s319	s1351	vecadd	saxpy	dscal
average	1.4	1.0	1.0	1.4	1.5	1.1	0.8	1.3	1.2
stddev	0.4	0.3	0.2	0.2	0.05	0.03	0.1	0.03	0.05

Table 5.3: Re-vectorization overhead (ms). Average and standard deviation over 10 runs.

5.2 Vectorization experimental results

5.2.1 Hardware/Software

The experiments were carried using a machine equipped with an Intel Core i5-2410M processor based on the Sandy Bridge architecture. The clock rate is 2.3 GHz. As before, the Turbo Boost and SpeedStep were disabled. The hardware resources are managed by an Ubuntu 14.04.1 LTS operating system. Furthermore, the benchmarks were compiled with GCC 4.8.4.

5.2.2 Benchmarks

The integration of the complex pieces of software which are Padrone, McSema, and Polly, is not yet finalized at the time of writing, thus forcing us to investigate only a subset of the benchmarks addressed previously with re-vectorization. The kernels operate on arrays of double-precision floating point elements which are 32-bit aligned. The benchmarks were compiled with flags `-O3 -fno-tree-vectorize` which disable the vectorization and maintain the rest of the optimizations. Besides, the methods are declared with attribute `noinline` to disable function inlining.

5.2.3 Performance Results

We assess the performance of vectorization by measuring the speedup of the optimized loops (vectorized for SSE or AVX) against the unvectorized loops. Results are shown in Figure 5.3.

First, we observe that auto-vectorization improves the performance by factors on par with the vector width. The SSE version outperforms the scalar version up to 1.95× in the case of `s1351`. As for `dscal`, the AVX version runs 3.70× faster.

Second for `saxpy`, we even obtain a superlinear speedup: it runs 2.10× faster. We compare this value to the speedup obtained with GCC 4.8.4 which is 2.04×. This small difference is due to the instructions generated by GCC 4.8.4 and the LLVM's JIT which are not similar.

Finally, for `s000`, we notice that the AVX version's performance is slightly less than the one of SSE. The reason is that `s000` is prone to a bottleneck between the register-file and cache. In other words, the microbenchmark intensively accesses memory, and the limitation of bandwidth results in introducing stalls into the CPU pipeline. Hence, AVX data movement instructions commit with extra cycles with regards to SSE.

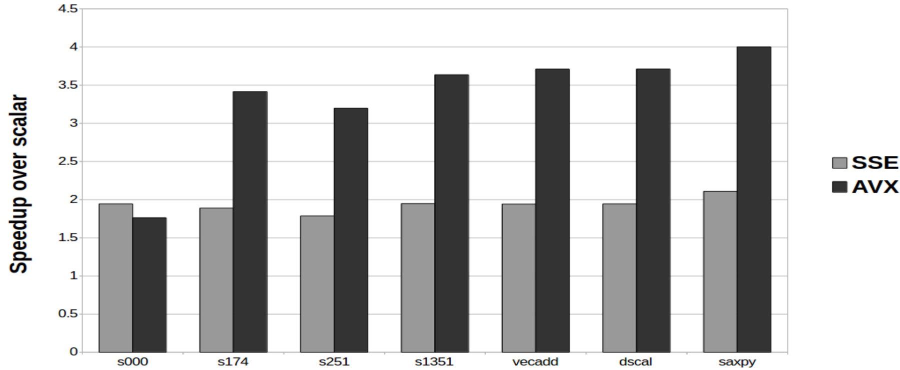


Figure 5.3: SSE and AVX Autovectorization Speedups

name	s000	s174	s251	s1351	vecadd	dscal	saxpy
average	205.4	112.0	162.5	113.8	106.9	104.6	123.3
stddev	7.5	5.7	4.5	4.5	3.0	3.4	3.3

Table 5.4: Vectorization overhead (ms). Average and standard deviation over 10 runs.

5.2.4 Overhead

The overhead of fully vectorizing loops is reported in Table 5.4. As expected, the values are much higher than in the case of re-vectorization, typically two orders of magnitude. This is due to the complexity of vectorization, to the approach based on lifting the code to LLVM-IR and applying the polyhedral model, and partly the fact that we connect building blocks whose interfaces have not been designed for efficiency at runtime.

5.3 Conclusion

Assessment of the re-vectorization and auto-vectorization techniques is done through the computation of speed-ups and time overheads. The re-vectorization transformation reaches a maximum speed-up of 67 %. The auto-vectorization has a speed-up that is sometimes close to the number of elements that fit within the vector operand. For both optimizations, the overhead of profiling, analysis, and transformation is negligible.

Chapter 6

Related work

Our work is based on performing two-fold vectorization transformations at runtime. In this context, both of automatics re-vectorization and vectorization transformations are considered as architecture dependent optimizations. The former is a straightforward binary-to-binary translation that maps instructions to their counterparts to maximize the use of vector unit. Even though, the latter is binary-to-IR-to-binary optimization which involves target independent transformations, but runtime information direct the dynamic compilation to use vector factor appropriate to vector processing unit. It is clear that our contributions are related to research efforts in binary translation, polyhedral transformations, and dynamic compilation. We managed to synthesize the closest work into the following: (1) compilers auto-vectorization, (2) thread level speculations systems, (3) binary-to-binary auto-vectorization and auto-parallelization, (4) optimizations at the level of the virtual machine and using the dynamic binary translation tools.

6.1 Compilers' auto-vectorization

Most of the works in this section are compilers' related. Although we share a similar aim that is the automatic vectorization, methodologies and techniques differ. The similarity with some of their techniques is the use the polyhedral model in the automatic vectorization. The differences are: first, some of these research efforts use of abstract interpretation methods and inter-procedural analysis during the vectorization and parallelization. Second, most of them perform source-to-source transformations. Third, some of them target different details such as outermost loop vectorization, transformation necessitating reordering of elements, and control flow with conditional constructs. Finally, our work is done at run-time targeting hot-loops, while all of them are off-line transformations.

Numerous source-to-source tools, like Pluto [BHRS08a, BHRS08b], PoCC [Pou13], LooPo [MG12], and the open-source compiler GCC with its graphite extension use the polyhedral model to represent and apply loop transformations on loop nests. Some of them rely on ClooG [Bas04a] to generate the transformed representation into a source code as Pluto or GCC. SUIF [WFW$^+$] is a compiler that parallelize C and Fortran

programs; precisely, it makes use of the uni-modular transformations in the polyhedral model.

Another source-to-source transformation framework is PIPs [IJT91]. It is geared toward optimizations for parallel machines by transforming sequential DO loops of Fortran77 into DOALL constructs. In PIPs, the polyhedral model cooperates with hierarchical control flow graph and inter-procedural analysis for optimizations. Moreover, PIPs uses abstract interpretation methods that obviously trade-off accuracy with number of tests to detect dependence for inter-procedural parallelization.

Par4All [ACE+12] is a source-to-source compiler based on PIPs and PoCC which transforms C and Fortran sequential programs into OpenMP, cuda or OpenCL code. PIPs allows the identification of optimizable regions and PoCC performs the memory accesses optimizations.

Due to the large productions of target-machines, diverse vector instruction sets prevail. These latter hamper the communities in compilation to come up with efficient hardware dependent vectorizing compilers. For this reason SAC [Gue10], a subproject of PIPs, emerges. It is a source-to-source tool that generates vector SIMD intrinsics for vectorizable functions. Their main idea is to create a set of non-dependent functions that unifies the instruction sets, and delays the back-end assignment of the SIMD intrinsics. SAC supports SSE and AVX for x86 and Neon for ARM processors.

Nuzman and Zaks explore in [NZ08] the benefits of vectorization of the outer-loop and present their technique. Several reasons motivating the outer-loop vectorization are provided in their research paper. First, when the trip count of the outer loop is greater than the inner loop. Second, it enhances the spatial locality when the memory access of the outer loop is unit-stride. Third, it may decrease the required memory bandwidth by increasing register reuse. Fourth, when the inner loop performs a reduction computation, it is beneficial to proceed in an outer-loop vectorization which eliminates the repeated vector to scalar operations. Fifth, the inner loop involves initialization and finalization operations such as permutations to handle alignment which are computationally expensive. The research surveys the prevailing outer loop vectorization methods such as loop interchange combined with vectorization of the innermost loop and introduces a technique which upgrades the vectorization of the innermost loop to support the outer one. Their method is implemented in gcc 4.3 and has a speedup factor that is significant compared to the inner-loop vectorization.

The interdependence between optimizations leads to a difficult transformation ordering problem. Trifunović, Nuzman, Cohen and Zaks suggest in [TNC+09] a model that estimates the performance impact of transformations orders targeting vectorization. Their cost model is integrated within the polyhedral representation which assesses the benefits of different optimizations orders. The factors upon which they base their model depend on the trip count, reduction operations, the memory access strides and alignment.

Nuzman, Rosen and Zaks in [NRZ06] address the issue of vectorizing code with interleaving data accesses. The aim is to automatically recognize the interleaved pattern, estimate its cost, and generate an optimized vectorized version of code. The interleaved access pattern is a sub-case of the scatter-gather problem which requires data reorder-

ing using indirection tables that helps to correctly permute and combine elements from different vectors. Their method is confined to regular stride accesses since irregular accesses lead to a higher overhead. Their solution is architecture dependent and relies on instruction set features provided by the target machine, since different architectures provide different mechanisms to support non-consecutive accesses.

Eichenberger, Wu and O'Brien [EWO04] address the challenge of vectorizing loops with misaligned memory references and consecutive access. Such problems are common and can be illustrated with operations on arrays with different indexes at a specific iteration. The basis of their technique relies on the automatic reorganization of data within registers. Since the reordering operation induces some overhead, they came up with several techniques that cut it down. The method achieves its aim using a graph augmented with reordering operations. The code generation handles a loop with several statements related to compile and run-time alignment information and unknown trip count.

[LA00] introduces the patterns of Super-word-Level Parallelism (SLP) and methods of extracting it. The SLP is a subclass of ILP whose statements have the same operations in identical order. Moreover, these scalar instructions can be combined into a single vector by merging the source and result operands into vector ones. The vectorization of such patterns involves packing and unpacking which varies in cost according to the target architecture. The paper introduces an SLP recognition which is based on basic blocks rather than loop nests. Their method exposes the SLP through loop unrolling since SLP is a subclass of ILP. The unroll factor is calculated according to the vector size and the operands' data type. Data dependence analysis allows them to ensure the safety of packing. They perform an alignment analysis to amortize the memory access overheads.

In [PJ15], Porpodas and Jones propose vectorization throttling which maximizes vectorization for a set of vectorization opportunities neglected in Superword-Level Parallelism (SLP). SLP is a vectorization technique for pattern of code and usually target straight-line code. The core of its algorithm is based on a bottom-up traversal of the data dependence graph from stores until reachable loads. It groups possible consecutive writes, scans the scalars that can be grouped into vectors, and verifies consecutive read accesses. Afterwards, it checks whether the vector conversion is worthwhile; otherwise, it keeps the instructions in the scalar form. The decision of the transformation depends on a cost-effectiveness test. Some patterns of code do not fall into the SLP form for various reasons. For instance, during the traversal, all operations can be vector grouped but the reads may turn-out to be non consecutive. The throttled SLP is less rigid, it allows vectorization as long as it is beneficial without considering the loads. Sometimes the vectorization of some graph regions results with performance degradation in comparison with the scalar form. Their technique improves the performance by 9% percent on average and 14% as a peak speedup in the best scenarios.

The motivations for auto-vectorization is that the short vector (SIMD) instructions significantly propel the speed of programs, cause a lower energy consumption, and increase the resources' utilization. In [KK05], Kudriavtsev and Kogge cover the SIMDization of computations from regular code and mainly addresses the data re-

ordering with the aim to minimize the permutation instructions. The vectorization algorithm is based on tree matching patterns that involve grouping individual memory operations into packed operation based on the effective addresses which accesses adjacent location, with similar data types, and which are independent. When the pattern forms groups whose size satisfies the size of the SIMD operand, the nodes of the scalar trees are merged into SIMD trees. Concerning data reordering, initially they start by grouping memory operations based on the effective addresses, bearing in mind that the permutation is not in the mathematical sense since elements might be replicated or omitted depending on the computation. Hence, the reordering is a subsequent phase that depends on the latter computations which involves the prior accesses. This needs a propagation of ordering nodes in the graph. The algorithm chooses orderings that reduces the cost of permutation based on the instruction set provided by the target architecture.

Nuzman describes in [Nuz06] the design and implementation of loop's SIMDization in GCC. The method differs from classic vectorization which is based on data dependences analysis, focuses on array-based Fortran scientific programs, and targets the vector machines of the 1970's. They claim that the classic vectorization theory is not appropriate to SIMD machines for various reasons: first, the proliferation of pointer use in programs; second, the architectural difference between SIMD and vector machines in which the former support contiguous memory accesses and aligned boundaries which are not constraints for the latter. The limitation of dependence analysis which uses data dependence graph is that in some cases it prevents vectorization for dependences carried across loop iteration. Their auto vectorization method for SIMD architectures scopes the automatic detection and transformation steps of scalar into vector loops. It operates on the gimple intermediate representation based on tree data structure, and applies passes such as scalar evolution to enhance the recognition of induction variables and the calculation of trip count. When the loop iterations are countable, they analyze the memory references which are required for the analyses of memory-dependence, access pattern, and alignment. The vectorization determines the vector factor, strip mines the loop, modifies the loop bounds, and transforms instructions into their packed form.

Shin addresses the problem of vectorizing code in the presence of control flow in [Shi07]. Previous works on the subject take advantage of predication which is known as an alternative to conditional branch instructions. In essence, some instructions are converted into vector instructions guarded by vector predicates. The shortcoming of the method is that the vectorized version executes intructions in all the paths to merge values which result with slower execution speed compared to the scalar version in some cases. This paper proposes a solution to overcome this issue by extracting relations between predicates, inserting branches-on-superword-condition-codes to bypass some vector instructions.

6.2 Thread level speculation systems

The thread level speculation systems optimize codes with memory access patterns and control flow that are not known at compile-time and variate at runtime. The similarity with our work is performing transformation at runtime. However, they trade-off between aggressive optimization and roll-back in case of wrong estimations; in our case, we make use solely of information known at runtime but do not variate at the course of execution for purposes such as alignment and manipulating last elements of arrays.

Most compilers focus on static analysis which leads to conservative optimizations. For instance, a regular compiler could never possibly parallelize a loop with changing access patterns at compile time. Many researches introduced the Thread Level Speculation Systems which analyses and optimizes the code at runtime. In [RAP95], they propose a technique that parallelizes the code at runtime. In a nutshell, it generates an inspector code that accompanies a target loop for optimization by analyzing its access patterns and computing cross iteration dependences. The inspector builds a data structure that contains the references to array elements by order of the iteration number. Based on a cost/profit model, a scheduler finds an optimal wave front schedule. Finally, the executor launches the wave front found iterations. Moreover, they use other techniques to eliminate the anti and output dependences by privatizing variables and detecting reductions which help in increasing the potential parallelism of loops.

An approach introduced by Rauchwerger and Padua in [RP94], hypothetically supposes any loop is fully parallel and executes its iterations concurrently, but at runtime it assesses any dependence anomalies. The method deviates from the inspector/executor tactics of speculative parallelization, which finds a parallel schedule of execution. The apprehension is that inspector/executor risks to delay launching the next parallel iteration specially when data computation is consumed by address computation. As a matter of fact, the novel technique narrows down the possibility of parallelization to loops without cross iteration dependences. Privatization transformation is used to filter the anti and output dependences and allows to optimize a wider range of loops. The doall test evaluates the full parallelism based on a consideration that array's elements are privatizable, proceeds marking the array's locations read from and written to, and depending on the order of these operations some locations are considered as non-privatizable. The analysis phase intersects these data structures to decide whether the loop has cross iteration dependences or not, and accordingly whether is it is fully parallel or not.

Automatic speculative POLyhedral Loop Optimizer abbreviated to APOLLO [SR16] is a TLS runtime that uses the polyhedral model to speculatively parallelize dynamic SCoPs on the fly. It supports various kinds of loops including while constructs, and non-linear loop nests; moreover, pointer and indirections accesses are handled as well. It profiles the serial loop for a short laps of time to collect memory addresses and loop count. From these latters, the equations and inequalities are formed. The runtime delegates the task of solving the equations to PLUTO which results with suggesting possible transformations and generating code for a number of iterations. This parallel version contains also additional instructions that verifies the memory access predictions

against the current reached values. When the speculation is violated, operations are squashed and the serial version is fired. Otherwise, it continues execution until the end of the current slice, backs-up the state, and executes another slice with the same estimations.

A research [ALGE12] extended Open-MP library with clauses that allows speculation about variables. They permit the programmer to write parallel loops without clue of an eventual dependence violation at runtime. The speculative load gets an up-to-date state of the location accessed. The speculative store writes the value into a copy associated to the processor on which the writing thread executes, and it ensures, that in the next iteration, requesting threads for this location do not consume an out-dated datum; otherwise, a dependence violation is signaled; and hence, the thread is halted and restarted. When the thread finishes executing a commit or discard function is executed according to the dependence violation occurrence. Clearly, this extension to Open-MP exempts the programmer to conclude whether a variable is private or shared.

Yang, Skadron, Soffa, and Whitehouse [YSSW11] developed a prototype optimizer which automatically parallelize trace in the binary program at runtime. The candidate traces, sequence of basic blocks coalesced into a single block, are detected through their frequency of execution, parallelized regardless of their control dependency, and cached for future use. When the sequential process reaches a location which has cached traces, it gets suspended, and the parallelized versions are launched on their own copies of the program in the remaining cores. The traces with side exits are aborted and the longest trace which completes is committed. Then, the sequential program proceeds execution from the end of the selected trace. The choice of using copy or discard mechanism over of roll-back on failed speculation stems from the fact that the latter enforces buffering the output of all speculative instructions which can be expensive. This novel method has a peak average speed up of 1.96x. In the course of parallelization, the code is transformed into an SSA form which guarantees the eliminations of anti and output dependencies. Plus, the trace undergoes a set of optimizations which increases the parallelization opportunities such as constant propagation, value propagation, redundancy and dead code elimination. They use a variant version of critical-path algorithm [WG90] to schedule instructions among the cores and synchronize the memory access order.

6.3 Binary-to-binary auto-vectorization and auto-parallelization

This section presents the works that share almost the same objectives as in our contributions. Their optimizations target the binary as well. The key differences are: in some of them, the target program and optimizer share the same address space while in our contributions they do not. Moreover, the profiling method is done through instrumentation, while we use lightweight hardware performance counters. Furthermore, our contributions are purely dynamic while some of them are static.

SecondWrite [KAS$^+$10] is a tool that optimizes x86 binaries. It incorporates an automatic parallelization implementation within a binary rewriter. The binary is trans-

lated into the LLVM IR representation which allows the use of the LLVM framework. The canonicalization passes and other passes namely Scalar Evolution and Induction Variable Analysis are a burden relief for their work prior deriving induction variables, and detecting affine loops. The calculation of distance vectors is based on their implementation of the GCD test to solve linear diophantine equations recovered from the binary. Two transformations are present in their work reductions and strip-mining; however, interchange, fusion, fission or skewing are not present. The code generation makes use of Posix-Threads.

A VM-like runtime layer [YF06], whose objective is the parallelization and vectorization of binary, amends the parallelism optimizations by targeting non-affine constructs and memory accesses. It prospects the executing program by a lightweight profiling, by instrumenting solely the basic blocks that dominate super-blocks. A static analysis is performed over feasible loops whose execution frequency is high to determine non potential cross iteration data dependences. Cross-iteration dependences are conservatively taken into account. They tune the size of tiles depending on the communication overhead of the system. For instance, separate processing units over chips with low communication latencies are assigned a large tile size. The criterion used for vectorization is the lowest number of iterations in a loop. The vectorization strategy is pattern oriented, they claim optimizing loops with the following models: load-execute-store, load-compare, or shift contents.

Selftrans [NMO11] is a tool that offers an architecture specific optimization of binary code at runtime. It performs a dynamic automatic vectorization that pertains to x86 machine code. The tool is composed of two modules. Selftrans monitor is responsible of choosing the candidate process for optimization. Then Selfrans analyzer is injected into the target process. At that moment, it proceeds profiling to identify streams of binary that needs optimization. The region is disassembled and a control flow graph is constructed which exhibits loop using the dominance relationship between the basic blocks. The CFG is elevated into an SSA intermediate representation used for dependency analysis. The vectorization targets loops with super word level parallelism [LA00] or data level parallelism. The main differences with regard to our method: first, selftrans chooses the process to be optimized; whereas, padrone takes the program's name as input. Second, their profiling, analysis, and code amendment region of code resides within the target code. Padrone performs all these steps in a separate address space. Third, Padrone is complementary to their work by allowing the re-vectorization. Fourth, Padrone vectorization demands an LLVM IR that represents the program in the form of functions; while, they lift only the loop to be reformed. Their vectorization opportunities is limited; for instance, vectorization that needs transformations of the code such loop interchange, reverse, or skewing are not feasible.

In [PKC11], Pradelle, Ketterlin, and Clauss describe an off-line method of parallelizing an x86-64 binary. This pure static approach first parses the binary and extracts high-level information. From this information, a C program is generated. This program captures only a subset of the program semantics, namely, loops and memory accesses. The eligible functions for transformation are parallelized using the polyhedral source-to-source compiler Pluto [Plu]. The transformed functions are compiled into a

dynamic library which is loaded by the OS. Their tool chain consists of a runtime that allows inserting break points into sequential loops and whenever a breakpoint is met the runtime component redirects the flow into the parallel version in the dynamic library. Their work includes all the tedious work of extracting functions, disassembling the binary, constructing CFG, computing the dominator tree, identifying loops, amending into an SSA intermediate representation, and lifting to a C code. The key differences are: their parallelization process is static while ours is applied at runtime. They target their optimization to the whole application; meanwhile, we allow the optimization only for hot-code. Their optimized versions are dynamically loaded; whereas, we make use of a code cache. Their tool chain circulates the lifted code to a source-to-source polyhedral compiler; while, lifting into source is bypassed using polly that optimizes an SSA intermediate representation.

6.4 Optimizations at the level of a virtual machine or using dynamic binary translation tools

This section presents related research efforts that perform dynamic binary optimizations. The optimizers reside either in virtual machines or make use of dynamic binary translation tools. Some of them require hardware assistance or target different optimizations such as memory prefetching.

Vectorization has been initially proposed as a purely static code optimization which equips all industrial-grade compilers. Retargetable compilers have introduced the need to handle several targets, including various levels of SIMD support within a family [NH06].

Liquid SIMD [CHY+07], from Clark et al., is conceptually similar to our approach. A static compiler auto-vectorizes the code, but then scalarizes it to emit standard scalar instructions. The scalar patterns are later detected by the hardware, if equipped with suitable SIMD capabilities, resurrecting vector instructions and executing them. The difference is that our method requires no additional hardware.

Recent work integrates initial automatic vectorization capabilities in JIT compilers for Java [NCL+10] [ESEMEN09]. The former paper addresses a tree pattern-based method whose algorithm identifies similar operations on adjacent locations and transforms them into vector instructions. The limitations of the method are its support to consecutive accesses and the multidimensionality of the arrays. This paper also provide a programming interface which exhibits data parallelism. The latter paper introduces an automatic vectorization algorithm that overcomes the limitations of the GCC vectorizer [NRZ06] which requires the access stride to be a power of 2. Our focus is on increasing the performance of native executables.

Attempting to apply binary translation technology to migrate assembly code, including SIMD instructions, over to markedly different architectures at runtime suffers from several difficulties stemming from lack of type information [LZXH06]. Instead, our proposal considers a single ISA, and migrates SIMD instructions to a higher level of features, targeting wider registers, but retaining the rest of the surrounding code.

Vapor SIMD [RDN+11] describes the use of a combined static-dynamic infrastructure for vectorization, focusing on the ability to revert efficiently and seamlessly to generate scalar instructions when the JIT compiler or target platform do not support SIMD capabilities. It was further extended [NDR+11] into a scheme that leverages the optimized intermediate results provided by the first stage across disparate SIMD architectures from different vendors, having distinct characteristics ranging from different vector sizes, memory alignment and access constraints, to special computational idioms. In contrast, our work focuses on plain native executables, without any bytecode, JIT compiler, or annotated (fat) binary. Vapor SIMD also automatically exploits the highest level of SSE available on the target processor. We only consider SSE as a whole vs. AVX, however it could easily be extended to support various versions of SSE.

In [RRC+14], we dynamically modify a running executable by substituting an SSE hot loop by a statically compiled version targeting AVX. In this thesis, we dynamically generate code for the new loop, and do not rely on a statically prepared patch.

As a binary optimizer, ADORE [CLHY04] uses hardware counters to identify hotspots and phases to apply memory prefetching optimizations. Similar goals are addressed in [BC07] where a dynamic system inserts prefetch instructions on-the-fly where it has been evaluated as effective by measuring the load latency and modeling memory strides using Markov chains. Both approaches focus on reducing the memory latency of memory instructions.

Various tools have been developed with DynamoRIO [BGA03]. The framework we used keeps all the client and toolbox in a separate address space and modifies as little as necessary the original code, while DynamoRIO links with the target application and executes code only from the code cache.

Dynamic binary translation also operates at run-time on the executable, but translates it to a different ISA. It has been implemented several ways, such as Qemu [Bel05], FX!32 [CHH+98], or Transmeta's Code Morphing System [DGB+03]. We generate code for the same ISA (only targeting a different extension). This gives us the ability to avoid much of the complexity of such translators, and to execute mostly unmodified application code, focusing only on hotspots.

6.5 Conclusion

To sum up, the key similarities and differences are organized as follows: even though some of work efforts in the compiler's auto-vectorization use the polyhedral model as in our contribution, their optimizations are static while ours are dynamic. The TLS systems operate also at runtime but speculate about memory accesses while we do not. Most of the binary-to-binary optimizations are static. And finally, conducted researches related to optimizations using virtual machines and binary translation tools target different optimizations.

Chapter 7

Conclusion

The continuous computer architecture's performance improvements extends the ISA with instructions that exploit the newer CPU features. The capability of the latter to execute programs compiled for previous CPU versions, such as legacy softwares, leads to an underutilization of the additional resources. On-the-fly binary translation is a efficient alternative to overcome this undesired behavior. The contributions of the thesis are based on a lightweight binary rewriting framework. PADRONE allows the detection and transformation of hot-loops with negligible interruption overhead.

The current design trend moving toward integration of multi-cores and SIMD processing within processors biased our decision of choosing the optimizations. SIMDization has the ability to fast-forward while execution (1) paying-off the expenses of binary translation and (2) resulting in higher speed-ups. The adopted optimizations comprise a binary-to-binary and binary-to-ir-to-binary transformations.

On the one hand, eligible loops that were compiled for the SSE SIMD extension are converted to AVX at runtime. The method is based on the use of a translation table, which contains SSE instructions along with their AVX equivalents, leverages the effort of the static vectorizer, and only extends the vectorization factor, while guaranteeing that the transformation is legal. Experiments showed that the overhead is minimal, and the speedups are in line with those of a native compiler targeting AVX. Moreover, runtime information such as actual alignment of memory accesses, can also result in substantial performance improvements.

On the other hand, scalar loops are auto-vectorized. The method lifts the binary into the LLVM intermediate representation. At this level, which hides the target machine details, we have shown that it is possible to handle the loops using the polyhedral model. They are then vectorized when possible. Finally, they are compiled into executable form using the LLVM JIT compiler. The results show the effectiveness of the approach and the speedups are significant.

To sum up, the re-vectorizer and auto-vectorizer are implemented inside a dynamic optimization platform; it is completely transparent to the user, does not require any rewriting of the binaries, and operates during program execution.

7.1 Perspectives

The binary-to-binary contribution neglected patterns of code with exotic instructions such as the shuffle and horizontal adds to avoid correctness issues. These binaries merit finding a method to translate them. The most convenient way and lightweight in terms of speed is to find a combination of AVX instructions equivalent to the SSE ones; plus, either (1) a mathematical proof is required to show that the transformations would always yield a semantically correct equivalents optimized binary as it was done for the reduction part in the thesis, or (2) an algorithm to check the validity of the transformation. Otherwise, a radical method to overcome this issue is to convert the code into its scalar version, lift into the LLVM-IR, and use the polyhedral model for the vectorization.

Additionally to what has already been addressed in the thesis's contributions, the alignment issue could be enhanced. Currently, our work shifts the loop by a number of iterations that gives as many as possible instructions the advantage of aligned memory accesses while the prior iterations are executed in scalar fashion. This number is identified according to the majority of statements with common arrays' access address modulo vector's operand size. The yielded remainder shifts the loop. This method is inefficient in some scenarios, when the instructions within the loop access various indices of arrays excluding several ones from taking advantage of using aligned vector instructions. A proposed solution would be a loop's software pipelining coupled with reordering of data within registers. It is known that data permutation involves packing and unpacking instructions whose overhead is architecture dependent. It is worthwhile to investigate this technique and check the data-permutations expense that could be traded-off by providing more instructions the elegibility of alignment access.

A runtime automatic parallelization of binary is a promising approach due to the widespread of multi-core architectures. Amdhal's law guarantees a significant speed-up when the serial part of the program is small. In our case, the target loop nest needs an execution's time that exceeds the one needed for the transformation, with the constraint that its parallel version would pay-off the expense of the transformation's overhead. A better scenario, is multiple calls of the function which encompasses the loop nest. An interesting and more aggressive optimization would be the parallelization of outer-loops and vectorizing inner ones. The risk could be the contention on the bus due to combined vectorized memory accesses and the required synchronizations between the parallel threads.

Index

Access function, 50

Code cache, 29

Decompilation, 63
Dependence polyhedron, 54

Hardware performance counters, 24

Instrumentation, 21
Iteration domain, 49

Polyhedral transformation, 57
Probing, 23

Reduction, 41
Re-vectorization, 32

SCoP, 48

Unimodular transformation, 56

Vectorization, 31

Bibliography

[ACE+12] Mehdi Amini, Béatrice Creusillet, Stéphanie Even, Ronan Keryell, Onig Goubier, Serge Guelton, Janice Onanian Mcmahon, François-Xavier Pasquier, Grégoire Péan, and Pierre Villalon. Par4All: From Convex Array Regions to Heterogeneous Computing. In IMPACT 2012 : Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012, Paris, France, January 2012. 2 pages.

[ALGE12] Sergio Aldea, Diego R. Llanos, and Arturo González-Escribano. Support for Thread-Level Speculation into OpenMP, pages 275–278. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In Spring Joint Computer Conference, AFIPS '67 (Spring), pages 483–485. ACM, 1967.

[Bas04a] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques, pages 7–16, Juan-les-Pins, France, September 2004.

[Bas04b] Cédric Bastoul. Improving Data Locality in Static Control Programs. PhD thesis, University Paris 6, Pierre et Marie Curie, France, December 2004.

[BC07] Jean Christophe Beyler and Philippe Clauss. Performance driven data cache prefetching in a dynamic software optimization system. In ICS'07, pages 202–209. ACM, 2007.

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In Usenix ATC, Freenix Track, pages 41–46, 2005.

[BGA03] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, pages 265–275, 2003.

[BHRS08a] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[BHRS08b] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[BL07] P. Bungale and C.K. Luk. Pinos: A programmable framework for whole-system dynamic instrumentation. Proceedings of the 3rd international conference on Virtual execution environments, pages 137–147, 2007.

[BN14] G. Bitzes and A. Nowak. The overhead of profiling using PMU hardware. CERN openlab report, 2014.

[CHH+98] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. FX!32: A profile-directed binary translator. IEEE Micro, 18(2):56–64, 1998.

[CHY+07] Nathan Clark, Amir Hormati, Sami Yehia, Scott Mahlke, and Krisztian Flautner. Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping. In HPCA, 2007.

[CLHY04] Howard Chen, Jiwei Lu, Wei-Chung Hsu, and Pen-Chung Yew. Continuous adaptive object-code re-optimization framework. In Advances in Computer Systems Architecture, volume 3189 of LNCS. 2004.

[DGB+03] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In CGO, 2003.

[DS11] J. Demme and S. Sethumadhavan. Rapid identification of architectural bottlenecks via precise event counting. Proceedings of the 38th annual international symposium on Computer architecture, pages 353–364, 2011.

[Era06] S. Eranian. Perfmon2: a flexible performance monitoring interface for linux. Proc. Ottawa Linux Symposium, pages 169–288, 2006.

[ESEMEN09] Sara El-Shobaky, Ahmed El-Mahdy, and Ahmed El-Nahas. Automatic vectorization using dynamic compilation and tree pattern matching technique in Jikes RVM. In ICOOOLPS, 2009.

[EWO04] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for simd architectures with alignment constraints. In Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, PLDI '04, pages 82–93, New York, NY, USA, 2004. ACM.

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. International Journal of Parallel Programming, 20(1):23–53, 1991.

[Fea93] P. Feautrier. Toward automatic partitioning of arrays on distributed memory computers. In Proceedings of the 7th international conference on Supercomputing, pages 175–184, 1993.

[FL11] Paul Feautrier and Christian Lengauer. Polyhedron model. In Encyclopedia of Parallel Computing. 2011.

[Gue10] S. Guelton. Sac: An efficient retargetable source-to-source compiler for multimedia instruction sets. 2010.

[HP02] A. Hartstein and Thomas R. Puzak. The optimum pipeline depth for a microprocessor. SIGARCH Comput. Archit. News, 30(2):7–13, May 2002.

[IJT91] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In Proceedings of the 5th International Conference on Supercomputing, ICS '91, pages 244–251, New York, NY, USA, 1991. ACM.

[Int14a] Intel 64 and IA-32 Architectures Optimization Reference Manual, 2014.

[Int14b] Intel 64 and IA-32 Architectures Software Developer's Manual, 2014.

[IT87] François Irigoin and Remy Triolet. Computing dependence direction vectors and dependence cones with linear systems. Technical report, Ecole des Mines de Paris, Fontainebleau, France, 1987.

[JYW+03] N. Jia, C. Yang, J. Wang, D. Tong, and K. Wang. Spire: improving dynamic binary translation through SPC-indexed indirect branch redirecting. VEE, pages 1–12, 2003.

[KAB+03] Nam Sung Kim, Todd Austin, David Blaauw, Trevor Mudge, Krisztián Flautner, Jie S. Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. Leakage current: Moore's law meets static power. Computer, 36(12):68–75, December 2003.

[KAS+10] Aparna Kotha, Kapil Anand, Matthew Smithson, Greeshma Yellareddy, and Rajeev Barua. Automatic parallelization in a binary rewriter. In

Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '43, pages 547–557, Washington, DC, USA, 2010. IEEE Computer Society.

[KK05] Alexei Kudriavtsev and Peter Kogge. Generation of permutations for SIMD processors. In Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES '05, pages 147–156, New York, NY, USA, 2005. ACM.

[LA00] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI '00, pages 145–156, New York, NY, USA, 2000. ACM.

[Lam74] Leslie Lamport. The parallel execution of do loops. Commun. ACM, 17(2):83–93, February 1974.

[LCM+05] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, Geoff Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pages 190–200, 2005.

[LL97] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '97, pages 201–214, New York, NY, USA, 1997. ACM.

[LZXH06] Jianhui Li, Qi Zhang, Shu Xu, and Bo Huang. Optimizing dynamic binary translation for SIMD instructions. In CGO, 2006.

[Mas94] Vadim Maslov. Lazy array data-flow dependence analysis. In Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '94, pages 311–325, New York, NY, USA, 1994. ACM.

[MC05] W. Mathur and J. Cook. Improved estimation for software multiplexing of performance counters. 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, pages 23–32, 2005.

[MG12] Christian Lengauer Martin Griebl. Polyhedral loop parallelization: Loopo. `https://www.infosun.fim.uni-passau.de/trac/LooPo/`, 1993–2012.

[MGG+11] Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. An evaluation of vectorizing compilers. In PACT'11, pages 372–382, 2011.

[MTS02] M. E. Maxwell, P. J. Teller, and L. Salay. Accuracy of performance monitoring hardware. In Proc. LACSI Symposium, Sante Fe, 2002.

[Mud01] Trevor Mudge. Power: A first-class architectural design constraint. Computer, 34(4):52–58, April 2001.

[NCL+10] Jiutao Nie, Buqi Cheng, Shisheng Li, Ligang Wang, and Xiao-Feng Li. Vectorization for Java. In NPC, 2010.

[NDR+11] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. Vapor SIMD: Auto-vectorize once, run everywhere. In CGO, 2011.

[NH06] Dorit Nuzman and Richard Henderson. Multi-platform auto-vectorization. In CGO, 2006.

[NMO11] T. Nakamura, S. Miki, and S. Oikawa. Automatic vectorization by runtime binary translation. In 2011 Second International Conference on Networking and Computing, pages 87–94, Nov 2011.

[NRZ06] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for SIMD. SIGPLAN Not., 41(6):132–143, June 2006.

[NS03] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In Third Workshop on Runtime Verification, 2003.

[Nuz06] Dorit Nuzman. Autovectorization in GCC – two years later. In GCC Developer's summit, June 2006.

[NZ08] Dorit Nuzman and Ayal Zaks. Outer-loop vectorization: Revisited for short simd architectures. In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, PACT '08, pages 2–11, New York, NY, USA, 2008. ACM.

[per] The perfevent interface.

[Pet99] M. Pettersson. The perfctr interface. 1999.

[PJ15] Vasileios Porpodas and Timothy M. Jones. Throttling automatic vectorization: When less is more. In 2015 International Conference on Parallel Architecture and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015, pages 432–444, 2015.

[PKC11] Benoit Pradelle, Alain Ketterlin, and Philippe Clauss. Transparent Parallelization of Binary Code. In First International Workshop on Polyhedral Compilation Techniques, IMPACT 2011, in conjunction with CGO 2011, Chamonix, France, April 2011. Christophe Alias, Cédric Bastoul.

[Plu] Pluto: An automatic parallelizer and locality optimizer for multicores.

[Pou13] Louis-Noël Pouchet. The Polyhedral Compiler Collection package, Feb 2013.

[PW94] William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences, pages 546–566. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.

[RAH08] A. Ruize-Alvarez and Kim Hazelwood. Evaluating the impact of dynamic binary translation systems on hardware cache performance. Proceedings of the IEEE International Symposium on Workload Characterization, 2008.

[RAP95] Lawrence Rauchwerger, Nancy M. Amato, and David A. Padua. A scalable method for run-time loop parallelization. International Journal of Parallel Programming, 23(6):537–576, 1995.

[RBS96] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 29, Paris, France, December 2-4, 1996, pages 24–35, 1996.

[RDN+11] Erven Rohou, Sergei Dyshel, Dorit Nuzman, Ira Rosen, Kevin Williams, Albert Cohen, and Ayal Zaks. Speculatively vectorized bytecode. In HiPEAC, 2011.

[RP94] Lawrence Rauchwerger and David Padua. The privatizing doall test: A run-time technique for doall loop identification and array privatization. In Proceedings of the 8th International Conference on Supercomputing, ICS '94, pages 33–43, New York, NY, USA, 1994. ACM.

[RRC+14] Emmanuel Riou, Erven Rohou, Philippe Clauss, Nabil Hallou, and Alain Ketterlin. Padrone: a platform for online profiling, analysis, and optimization. In Dynamic Compilation Everywhere, 2014.

[Shi07] Jaewook Shin. Introducing control flow into vectorized code. In 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007), Braşov, Romania, September 15-19, 2007, pages 280–291, 2007.

[SR16] Aravind Sukumaran-Rajam. Beyond the Realm of the Polyhedral Model: Combining Speculative Program Parallelization with Polyhedral Compilation. PhD thesis, INRIA CAMUS, ICube Lab. Univ. of Strasbourg, France, 2016.

[TNC+09] Konrad Trifunović, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. Polyhedral-Model Guided Loop-Nest Auto-Vectorization. In The 18th International Conference on Parallel Architectures and Compilation Techniques, Raleigh, United States, September 2009.

[Val14] Cédric Valensi. A generic approach to the definition of low-level components for multi-architecture binary analysis. PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, 2014.

[Ver10] 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.

[VG12] Sven Verdoolaege and Tobias Grosser. Polyhedral extraction tool. In Second International Workshop on Polyhedral Compilation Techniques, Paris, France, 2012.

[VNS13] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. On Demand Parametric Array Dataflow Analysis. In Armin Größlinger and Louis-Noël Pouchet, editors, Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques, pages 23–36, Berlin, Germany, January 2013.

[Wea15] V.M. Weaver. Self-monitoring overhead of the Linux perfevent performance counter interface. International Symposium on Performance Analysis of Systems and Software, pages 102–111, 2015.

[WFW+] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih wei Liao, Chau wen Tseng, Mary Hall, Monica Lam, and John Hennessy. An overview of the SUIF compiler system. Technical report.

[WG90] M. Y. Wu and D. D. Gajski. Hypertool: a programming aid for message-passing systems. IEEE Transactions on Parallel and Distributed Systems, 1(3):330–343, Jul 1990.

[Wol89] Michael Wolfe. Optimizing Supercompilers for Supercomputers. The MIT Press, Cambridge, Massachusetts, 1989.

[YF06] Efe Yardimci 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.

[YSSW11] Jing Yang, Kevin Skadron, Mary Lou Soffa, and Kamin Whitehouse. Feasibility of dynamic binary parallelization. 2011.

[ZCW08] Q. Zhao, I. Cutcutache, and W.F. Wong. Pipa: Pipelined profiling and analysis on multi-core systems. Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization., pages 185–194, 2008.

List of Figures

1 Exemple de vectorisation 7
2 Exemple de conversion des instructions du SSE vers l'AVX 8
3 Le processus de la vectorisation automatique 10
4 L'intégration de McSema dans Padrone 11
5 Les résultats de la re-vectorisation 13
6 Les résultats de la vectorisation automatique 15

2.1 Pin internal design 24
2.2 Padrone's functionalities 29
2.3 Profiling sample 29

3.1 Vector addition (pseudo code) 34
3.2 Body of vectorized loop for vector addition 35
3.3 Pattern of code when the number of iterations is known only at runtime 39
3.4 Example of loop-carried dependence 39
3.5 Example of aliasing with changing dependence distance 41
3.6 Pattern of code when array addresses are known only at runtime 42
3.7 Search for the maximum element in an array of floats 43
3.8 Search for the largest element in an array, vectorized for SSE 44
3.9 Body of vectorized loop for vector max 45
3.10 Extracting the largest element from the SIMD register 45

4.1 Example of a valid Static Control Part 50
4.2 Graph linear inequalities 51
4.3 Logical date 53
4.4 Instruction reordering 55
4.5 Constant propagation 55
4.6 GCD test 56
4.7 Unimodular skewing transformation: computation of new loop bounds and access functions 60
4.8 Unimodular skewing transformation 61
4.9 Process automatic vectorization 64
4.10 Integration of Padrone with McSema 67
4.11 Applying memory to register pass 70
4.12 Example memory to register pass 71

4.13 Difference between pointer arithmetic recurrence and array subscript . . 73
4.14 C program which illustrates consecutive memory access 74
4.15 LLVM IR which illustrates consecutive memory access 74
4.16 DAG which illustrates consecutive memory access 75

5.1 Speedups for type `float` 82
5.2 Speedups for type `double` 82
5.3 SSE and AVX Autovectorization Speedups 86

Abstract

In many cases, applications are not optimized for the hardware on which they run. This is due to backward compatibility of ISA that guarantees the functionality but not the best exploitation of the hardware. Many reasons contribute to this unsatisfying situation such as legacy code, commercial code distributed in binary form, or deployment on compute farms. Our work focuses on maximizing the CPU efficiency for the SIMD extensions. The first contribution is a lightweight binary translation mechanism that does not include a vectorizer, but instead leverages what a static vectorizer previously did. We show that many loops compiled for x86 SSE can be dynamically converted to the more recent and more powerful AVX; as well as, how correctness is maintained with regards to challenges such as data dependencies and reductions. We obtain speedups in line with those of a native compiler targeting AVX. The second contribution is a runtime auto-vectorization of scalar loops. For this purpose, we use open source frameworks that we have tuned and integrated to (1) dynamically lift the x86 binary into the Intermediate Representation form of the LLVM compiler, (2) abstract hot loops in the polyhedral model, (3) use the power of this mathematical framework to vectorize them, and (4) finally compile them back into executable form using the LLVM Just-In-Time compiler. In most cases, the obtained speedups are close to the number of elements that can be simultaneously processed by the SIMD unit. The re-vectorizer and auto-vectorizer are implemented inside a dynamic optimization platform; it is completely transparent to the user, does not require any rewriting of the binaries, and operates during program execution.

Résumé

Dans plusieurs cas, les applications ne sont pas optimisées pour le matériel sur lequel elles s'exécutent. De nombreuses raisons contribuent à cette situation insatisfaisante, comme les logiciels hérités, ou code commercial distribué sous forme binaire, ou le déploiement des prgrammes dans des fermes de calcul. On se concentre sur la maximisation de l'efficacité du processeur pour les extensions SIMD. Dans la première contribution, nous montrons que de nombreuses boucles compilées pour x86 SSE peuvent être converties dynamiquement en versions AVX plus récentes et plus puissantes; Ainsi que la façon dont l'exactitude est maintenue en ce qui concerne les défis tels que les dépendances de données et les réductions. Nous obtenons des accélérations conformes à celles d'un compilateur natif ciblant AVX. La deuxième contribution est la vectorisation en temps réel des boucles scalaires. Nous avons intégrés logiciels libres pour (1) soulever dynamiquement le binaire x86 vers la forme de représentation intermédiaire du compilateur LLVM, (2) abstraire les boucles fréquement executé dans le modèle polyédrique, (3) utiliser la puissance de ce model mathématique pour les vectoriser, et (4) enfin les compiler en utilisant le compilateur Just-In-Time de LLVM. Les accélérations obtenues sont proches du nombre d'éléments pouvant être traités simultanément par l'unité SIMD. Notre plate-forme d'optimisation est dynamique; elle ne nécessite aucune réécriture des binaires et fonctionne pendant l'exécution du programme.