SIMD-aware word length optimization for floating-point to fixed-point conversion targeting embedded processors

Ali Hassan El Moussawi

ANNÉE 2017

UNIVERSITÉ DE RENNES 1

UNIVERSITE BRETAGNE LOIRE

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

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale Matisse

présentée par

Ali Hassan El Moussawi

Préparée à l'Unité Mixte de Recherche 6074 – IRISA
Institut de recherche en informatique et systèmes aléatoires
UFR Informatique Electronique

SIMD-aware Word Length Optimization for Floating-point to Fixed-point Conversion targeting Embedded Processors

Thèse soutenue à Rennes
le 16/12/2016

devant le jury composé de :

David Defour
MCF - Université de Perpignan / Rapporteur

Lionel Lacassagne
PR - Université Pierre et Marie Curie / Rapporteur

Karine Heydemann
MCF - Université Pierre et Marie Curie / Examinatrice

Daniel Menard
PR - INSA Rennes / Examinateur

Tomofumi Yuki
CR - INRIA Bretagne Atlantique / Examinateur

Steven Derrien
PR - Université de Rennes 1 / Directeur de thèse

Certes, la science guide, dirige et sauve;
l'ignorance égare, trompe et ruine.

Remerciements

Steven a toujours été là pour me conseiller et me motiver, je tiens à le remercier infiniment pour tout ce qu'il a fait pour contribuer à la réussite de mes travaux de thèse.

Je remercie d'autant Tomofumi Yuki pour le temps qu'il a consacré pour discuter des divers aspects de cette thèse. Ses conseils avaient un impact important sur ces travaux.

Je tiens également à remercier Patrice Quinton pour sa contribution à la revue/correction de ce manuscrit.

Je remercie tous les membres du jury, qui m'ont fait l'honneur en acceptant de juger mes travaux, surtout Lionel Lacassagne et David Defour qui ont eu la charge de rapporteurs.

Je remercie tout le personnel de l'Inria/Irisa, notamment les membres de l'équipe CAIRN pour le chaleureux accueil qu'ils m'ont réservé. Je remercie, en particulier, Nadia Derouault qui s'est occupée de toutes les démarches administratives, Nicolas Simon et Antoine Morvan pour l'assistance avec les outils de développement, ainsi que Gaël Deest, Simon Rokicki, Baptiste Roux et tous les autres.

Last but not least, je tiens à remercier ma famille pour leur support durant toutes ces années.

Table of Contents

Résumé en Français **v**

List of Figures **xi**

List of Tables **xv**

Glossary **xvii**

1 Introduction **1**
1.1 Context and Motivations . . . 2
1.2 ALMA Project . . . 4
1.3 Timeline . . . 6
1.4 Contributions and Organization . . . 8

2 Background: Floating-point to Fixed-point Conversion **11**
2.1 Introduction . . . 12
2.2 Floating-point Representation . . . 12
2.3 Fixed-point Representation . . . 14
2.4 Floating-point vs. Fixed-point . . . 16
2.5 Floating-point to Fixed-point Conversion Methodologies . . . 19
2.5.1 Integer Word Length Determination . . . 19
2.5.2 Word Length Optimization . . . 21
2.5.3 Fixed-point Code Generation . . . 22
2.6 Automatic Conversion Tools . . . 23
2.6.1 MATLAB Fixed-point Converter . . . 23
2.6.2 IDFix . . . 24
2.7 Conclusion . . . 27

3 Background: Single Instruction Multiple Data **29**
3.1 Introduction . . . 30
3.2 SIMD Instruction Sets . . . 31
3.2.1 Conventional Vector Processors . . . 31
3.2.2 Multimedia extensions . . . 32

3.3 Exploiting Multimedia extensions . . . 36
3.3.1 Manual Simdization . . . 37
3.3.2 Automatic Simdization Methodologies . . . 38
3.4 Conclusion . . . 42

4 Superword Level Parallelism 45
4.1 Introduction . . . 46
4.2 Related work . . . 47
4.2.1 `Original SLP` . . . 48
4.2.2 `Original SLP` Extensions and Enhancements . . . 49
4.3 `Holistic SLP` . . . 50
4.3.1 Overview . . . 51
4.3.2 Shortcomings . . . 59
4.4 Proposed `Holistic SLP` Algorithm Enhancements . . . 64
4.4.1 Pack Flow and Conflict Graph (PFCG) . . . 65
4.4.2 Group Selection Procedure . . . 66
4.4.3 Algorithm complexity . . . 72
4.4.4 Pseudo-code . . . 74
4.5 Experimental Setup and Results . . . 78
4.5.1 Implementation . . . 78
4.5.2 Target processors . . . 79
4.5.3 Benchmarks . . . 81
4.5.4 Tests Setup . . . 81
4.5.5 Results . . . 82
4.6 Conclusion . . . 88

5 SLP-aware Word Length Optimization 91
5.1 Introduction . . . 92
5.2 Motivations and Related Work . . . 94
5.3 Joint WLO and SLP extraction . . . 96
5.3.1 Overview and Intuition . . . 97
5.3.2 Processor Model . . . 98
5.3.3 SLP-aware WLO algorithm . . . 101
5.3.4 Accuracy-aware SLP extraction algorithm . . . 106
5.3.5 SLP-aware Scalings Optimization . . . 109
5.4 Source-to-source Compilation Flow . . . 112
5.4.1 Flow Overview . . . 112
5.4.2 Integer Word Length determination . . . 114
5.4.3 Accuracy evaluation . . . 115
5.4.4 Select and sort Basic-blocks for SLP . . . 115
5.4.5 Fixed-point/SIMD Code Generation . . . 115
5.5 Experimental evaluation and Results . . . 116

5.5.1 Experimental Setup . . . 116
5.5.2 `WLO-then-SLP` source-to-source flow . . . 117
5.5.3 Benchmarks . . . 118
5.5.4 Target Processors . . . 119
5.5.5 Results . . . 119
5.5.6 Floating-point vs. Fixed-point . . . 121
5.5.7 `WLO+SLP` vs. `WLO-then-SLP` . . . 121
5.6 Conclusion . . . 123

6 Conclusion **125**

A Target Processor Models **129**
A.1 XENTIUM . . . 129
A.2 ST240 . . . 129
A.3 KAHRISMA . . . 130
A.4 VEX . . . 130

Publications **133**

Bibliography **135**

Résumé en Français

Les processeurs embarqués sont soumis à des contraintes strictes de coût, consommation électrique et performance. Afin de limiter leur coût et/ou leur consommation électrique, certain processeurs ne disposent pas de support matériel pour l'arithmétique à virgule flottante.

D'autre part, les applications dans plusieurs domaines, tel que le traitement de signal et la télécommunication, sont généralement spécifiées en utilisant l'arithmétique à virgule flottante, pour des raisons de simplicité. Cette implémentation prototype est ensuite adaptée/optimisée en fonction de l'architecture cible.

Porter de telles applications sur des processeurs embarqués sans support matériel pour l'arithmétique à virgule flottante, nécessite une émulation logicielle, qui peut sévèrement dégrader les performances de l'application. Pour éviter cela, l'application est convertie pour utiliser l'arithmétique à virgule fixe, qui a l'avantage d'être plus efficace à implémenter sur des unités de calcul entier. Les deux representations, à virgule flottante et à virgule fixe sont illustrées dans la fig. 1. La conversion de virgule flottante en virgule fixe est une procédure délicate qui implique

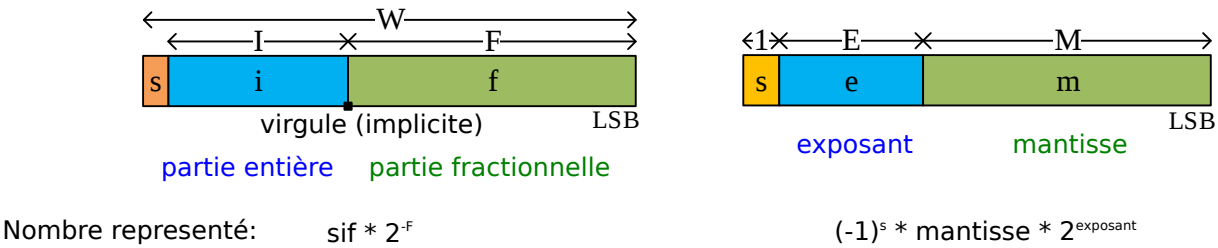


Figure 1 – Comparaison des representations à virgule flottante (droite) et à virgule fixe (gauche).

des compromis subtils entre performance et précision de calcul. Elle permet, entre autre, de réduire la taille des données au prix d'une dégradation de la précision de calcul. En effet, utiliser des opérations à virgule fixe, tout en gardant la précision complète des résultats, nécessite une augmentation considérable des tailles de mots des données. Par exemple, le résultat exact d'une multiplication entre deux nombres de taille w, nécessite une taille de $2w$. Cette augmentation de la taille de mots peut dépasser la taille maximale supportée par le processeur cible, nécessitant ainsi une émulation logicielle d'opérateurs de taille plus grande, qui peut aussi bien dégrader la performance de l'application. Afin d'éviter cela, les tailles des données (et des résultats des opérations) sont réduites en appliquant des quantifications, qui correspondent à éliminer les bits de poids faibles. Ces opérations de quantification introduisent des erreurs de calcul, appelées

erreurs de quantifications, qui se propagent dans le système et peut engendrer une erreur importante en sortie, dégradant ainsi la précision du résultat. En règle générale, plus la quantification est grande (i.e. plus la taille des données est réduite), plus la précision est faible mais meilleur est la performance. Il existe donc un compromis entre précision de calcule et performance.

Par ailleurs, la plupart des processeurs embarquées fournissent un support pour le calcul vectoriel de type *SIMD* ("Single Instruction Multiple Data") afin d'améliorer la performance. En effet, cela permet l'exécution d'une opération sur plusieurs données simultanément, réduisant ainsi le temps d'exécution. Cependant, il est généralement nécessaire de transformer l'application pour exploiter les unités de calcul vectoriel. Cette transformation de vectorisation est sensible à la taille des données; plus leurs tailles diminuent, plus le taux de vectorisation augmente. Il apparaît donc un autre compromis entre vectorisation et tailles de données.

En revanche, la vectorisation ne conduit toujours pas a une amélioration de performance, elle peut même la dégrader ! En fait, afin d'appliquer une opération vectorielle, il faut d'abord agréger ou *compacter* les données de chacun des opérandes pour former un vecteur, qui correspond en générale à un registre SIMD. De même, il faut décompacter les données afin de les utiliser séparément, comme le montre la fig. 2. Ces opérations de (dé)compactage peuvent engendrer

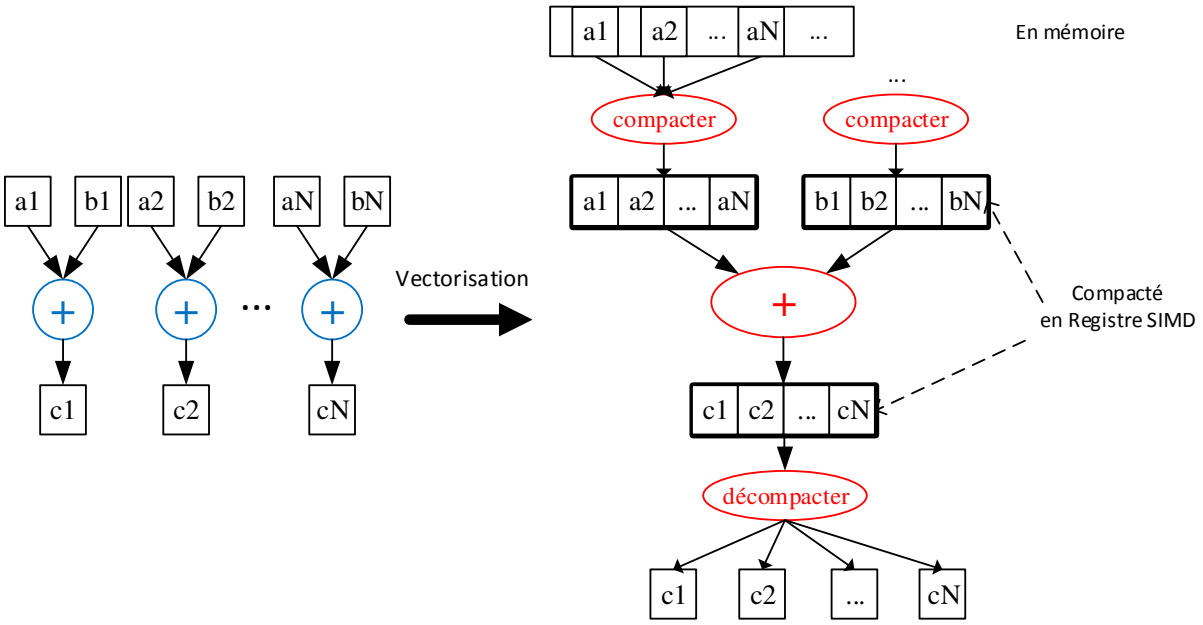


Figure 2 – Illustration de la transformation de vectorisation remplacant N opérations scalaires par une seule opération SIMD plus les opérations de (dé)compactage.

un surcoût important dans le cas où les données sont mal organisées en mémoire. Du coup, afin d'améliorer efficacement la performance, la transformation de vectorisation doit prendre en considération ce surcoût.

La conversion de virgule flottante en virgule fixe et la vectorisation sont deux transformations délicates qui nécessitent un temps de développement très élevé. Pour remédier à ça et réduire les délais de mise sur le marché des applications, une multitude de travaux ont ciblé l'automatisation

(complète ou partielle) de ces transformations. Dans l'état de l'art, on trouve des méthodologies permettant la conversion de virgule flottante en virgule fixe, tel que [51, 26, 64, 85]. Ces méthodologies comportent en générale trois parties principales :

— La détermination des tailles des parties entières. En se basant soit sur des simulations, ou en utilisant des méthodes analytiques telle que l'arithmétique d'intervalles et l'arithmétique affine.

— La détermination des tailles des mots. Cela fait en générale l'objet d'une optimisation exploitant le compromis entre précision et performance, connue sous le nom "*Word Length Optimization*" ou WLO (optimisation de taille de mots). Pour ce faire, des méthodes permettant l'estimation de la précision de calcule et la performance, d'une implémentation à virgule fixe, sont nécessaires. Plusieurs méthodes ont été proposées.

— La génération de code à virgule fixe [50].

D'autre part, on trouve également des méthodologies permettant l'exploitation des unités de calcul SIMD, entre autre, les techniques d'extractions du parallélisme au niveau du bloque de base, connues sous le nom "*Superword Level Parallelism*" ou SLP (Parallélisme au niveau du super-mot) introduit en 2000 par Larsen et Amarasinghe [69]. Ces méthodes ont pour objectif de trouver des groupes d'opérations, dans un bloque de base, qui peuvent être replacer par des opérations SIMD. Un tel groupe, appelée groupe SIMD, doit contenir des opérations, indépendantes, du même type (addition, multiplication, ...) et traitant des données de même taille. Le but des algorithmes d'extraction du SLP [69, 125, 78] est de trouver la « meilleure » solution de groupage qui permet d'améliorer la performance en tenant en compte le surcoût lié aux opérations de (dé)compactage.

Cependant, dans l'état de l'art, ces deux transformations sont considérées indépendamment, pourtant elles sont fortement liées. En effet, WLO détermine la taille des données qui affectent directement l'espace de recherche du SLP, et par conséquence la performance de la solution trouvée. Si WLO n'est pas conscient des contraintes d'extraction du SLP et du surcoût associé, il sera incapable d'estimer correctement l'impacte de ces décisions sur la performance finale de l'application (après avoir appliquer la conversion en virgule fixe et l'extraction du SLP). Par conséquence, il sera incapable d'exploiter efficacement le compromis enter précision et performance. Afin de mieux exploiter ce compromis, WLO doit prendre en considération SLP, alors que ce dernier ne peut pas procéder sans avoir une connaissance sur les tailles des données. Ce problème d'ordonnancement de phase est illustré par la fig. 3.

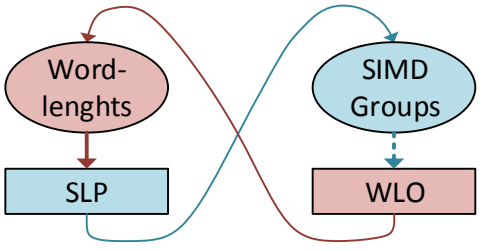


Figure 3 – Problème d'ordonnancement de phase entre WLO et SLP.

Dans ce contexte, on propose dans un premier temps un algorithme amélioré pour l'extraction du SLP. On se base sur un algorithme de l'état de l'art proposé en 2012 par Liu et al, on l'analyse soigneusement pour déterminer ses faiblesses puis on propose des améliorations pour y remédier. L'algorithme proposée est ensuite implémenté dans une plateforme de compilation source-à-source, Generic Compiler Suite (GeCoS)[38], ainsi que l'algorithme de référence, afin de valider notre approche. Les résultats expérimentaux, extraits sur un ensemble de neuf applications de tests et ciblant plusieurs processeurs embarqués, montrent une amélioration claire apportée par notre algorithme.

Ensuite, on propose une nouvelle technique permettant l'application conjointe, de la conversion de virgule flottante en virgule fixe et de l'extraction du SLP. Au contraire des méthodologies de l'état de l'art, cette nouvelle technique permet de mieux exploiter le compromis entre la précision de calcul et la performance d'une application, ciblant des processeurs embarqués avec jeux d'instructions SIMD sans support matérielle pour l'arithmétique à virgule flottante. Cette approche consiste à combiner un algorithme d'extraction du SLP conscient de la précision de calcul, avec un algorithme de WLO conscient des opportunistes de groupage SLP et du surcoût associé. Pour résoudre le problème d'ordonnancement de phases présenté précédemment, on a adapté l'algorithme d'extraction du SLP proposé, afin de relâcher les contraintes liées à la tailles des données. De cette façon, l'extraction du SLP peut désormais démarrer sans avoir à attendre le résultat du WLO. l'algorithme d'extraction du SLP est également conscient de la contrainte sur la précision de calcul, imposée par l'utilisateur. Cela permet d'éviter de sélectionner des groupes SIMD qui sont pas réalisable sans violer la contrainte de précision. Les groupes SIMD choisis, sont ensuite utilisés pour "guider" la sélection des tailles de mots par l'algorithme de WLO. La fig. 4 illustre cette approche. On implémente cette approche dans GeCoS sous forme d'un flot

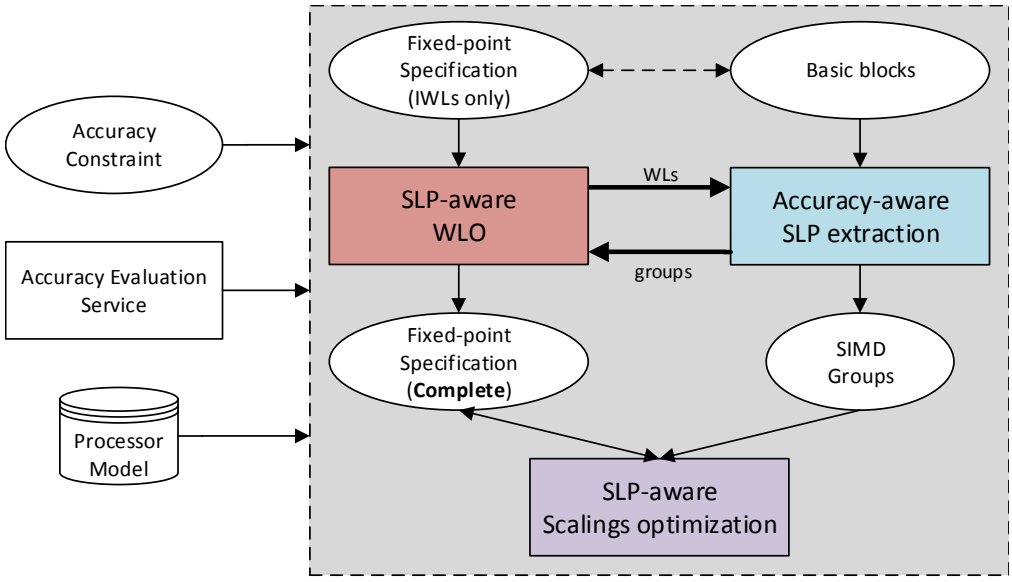


Figure 4 – Illustration de l'approche proposée.

de compilation source-à-source complètement automatisé. Afin de montrer la validité de notre

approche, on la compare contre une approche classique appliquant indépendamment, d'abord la conversion de virgule flottante en virgule fixe, ensuite l'extraction du SLP, qu'on implémente également dans GeCoS. On teste les deux flots sur plusieurs processeurs embarquées. Les résultats confirme l'efficacité de notre approche, dans l'exploitation du compromis entre performance et précision.

List of Figures

1 Comparaison des representations à virgule flottante (droite) et à virgule fixe (gauche). v
2 Illustration de la transformation de vectorisation remplacant N opérations scalaires par une seule opération SIMD plus les opérations de (dé)compactage. vi
3 Problème d'ordonnancement de phase entre WLO et SLP. vii
4 Illustration de l'approche proposée. viii

1.1 CPU trend over the last 40 years. Published by K. Rupp at www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data 2
1.2 ALMA tool-chain flow diagram. 5
1.3 Outline. 9

2.1 Binary representation of floating-point numbers in IEEE 754 standard. 12
2.2 Binary representation of a signed fixed-point number. 14
2.3 IEEE single-precision floating-point numbers precision vs. range. 17
2.4 Range propagation example. 20
2.5 IDFix flow diagram. 25

3.1 Illustration of an Single Instruction Multiple Data (SIMD) addition operation in contrast to scalar addition operation. 30
3.2 Illustration of Sub-word Level Parallelism (SWP) addition operator capable of executing 32-bit scalar addition (when propagate is 1) or 2x16-bit SIMD additions (when propagate is 0). 34
3.3 Illustration of some soft SIMD operations. 35
3.4 Example of Vectorization. 39
3.5 Illustration of vector memory load from an aligned stride-1 reference (middle) versus unaligned stride-1 (left) and aligned stride-2 (right), in case of a Multimedia extension with no support for unaligned accesses and with vector size of 2. 40
3.6 Example illustrating the difference between loop vectorization and Superword Level Parallelism (SLP). $[min : max]$ represents a vector memory access to consecutive array elements starting at offset min till offset max included. $< a, b >$ represents a packing/unpacking operation. 42

4.1 Example C code snippet (left) and its statements dependence graph (right). . . 47
4.2 Recall example of fig. 4.1. 51
4.3 `Holistic SLP` algorithm flow diagram. 52
4.4 Variable Pack Conflict Graph (VPCG) and Statement Grouping Graph (SGG) of the example in fig. 4.1. 53
4.5 $AG(\{S4, S5\})$ at the first selection iteration of the example of fig. 4.1. 56
4.6 Updated VPCG (left) and SGG (right) after the selection of $\{S4, S5\}$ of the example of fig. 4.1. 57
4.7 Example illustrating `holistic SLP` algorithm. It shows that VPCG and SGG do not represent cyclic dependency conflicts. The grouping solution is $\{c1, c2, c3\}$ which is not legal due to cyclic dependency between $c1$, $c2$ and $c3$. 60
4.8 Example showing how fake dependencies affect `holistic SLP` VP reuse estimation. 61
4.9 Example illustrating `holistic SLP` candidate benefit estimation. 63
4.10 SLP extraction framework overview. 65
4.11 Data Flow Graph (DFG) and Pack Flow and Conflict Graph (PFCG) of the example in fig. 4.1. Rectangular nodes in the PFCG represent pack nodes, whereas elliptic shaped ones represent candidate nodes. Undirected edges represent conflicts between candidates and directed edges represent Variable Pack (VP) Flow. Highlighted nodes (with orange) indicate a reuse. 67
4.12 Execution time improvement, over the sequential version, of various benchmarks obtained by applying `holistic SLP` for ten times. 69
4.13 Execution time improvement, over the sequential version, of various benchmarks after applying the proposed SLP extraction for ten times. 70
4.14 Recall the example of fig. 4.9. 71
4.15 PFCG of the example in fig. 4.14 at the first iteration using `prop-2`. 71
4.16 Execution time improvement of the SIMD code obtained by applying our proposed SLP extraction method, compared to the original (sequential) code, for different values of N considered for constructing the distance-N neighborhood of a candidate (sub-PFCG) in the PFCG. The targeted processor is KAHRISMA. . 73
4.17 SLP Extraction Framework implementation in GeCos. 79
4.18 Test procedure diagram. 82
4.19 Execution time improvement of the SIMD code obtained by `prop` vs. `hslp`, over `orig`. The test is repeated 10 times for each benchmark. A bar represent the mean value and a line segment represents the minimum and maximum values of the execution time improvement. 84
4.20 Execution time improvement of the SIMD code obtained by `prop` vs. `hslp`, over `orig`. The test is repeated 10 times for each benchmark. A bar represent the mean value and a line segment represents the minimum and maximum values of the execution time improvement. 85
4.21 Using `prop`. 87
4.22 Execution time improvement and Simdization time variation with respect to the batch size, for the **jacobi2d** benchmark running on KAHRISMA. 87

5.1 C code snippet. 93
5.2 Motivating example. 96
5.3 WLO/SLP phase ordering problem. 97
5.4 Overview of the joint Word Length Optimization (WLO) and SLP extraction approach 98
5.5 Fixed-point multiplication example. 100
5.6 Fixed-point multiplication example. 101
5.7 Representation of an operation node (op) and its predecessors/successor data nodes (d_x) in a fixed-point specification. $< i_x, f_x >$ represents a fixed-point format with i_x representing the Integer Word Length (IWL) and f_x the Fractional Word Length (FWL). Elliptical nodes represent scaling operations; a negative amount corresponds to a left shift by the absolute value. 103
5.8 Scaling example. f_x represents the FWL. 111
5.9 SLP-aware WLO source-to-source compilation flow diagram. 113
5.10 Test setup 117
5.11 `WLO-then-SLP` source-to-source compilation flow diagram. 118
5.12 Speedup of SIMD code version obtained by `WLO+SLP` flow, over the original `float` version, for different accuracy constraints expressed in dB (higher values (to the left) are less accurate) 120
5.13 Speedup (higher is better) comparison between SIMD versions of `WLO-then-SLP` and `WLO+SLP` vs. accuracy constraint expressed in dB (higher values(to the left) are less accurate). The baseline is the execution time of the (non SIMD fixed-point version of `WLO-then-SLP`. 122

List of Tables

2.1 Exact fixed-point operations. 15
2.2 Floating-point vs. Fixed-point. 19

4.1 Test Benchmarks. 81
4.2 Number of operations and memory accesses in benchmarks. CAA represents the number of contiguous array access candidates. 81

5.1 Benchmarks 118
5.2 Target processors supported operations. 120

A.1 XENTIUM Model. 129
A.2 ST240 Model. 130
A.3 KAHRISMA Model. 130
A.4 VEX Model. 131

Glossary

API Application Programming Interface.

DAG Directed Acyclic Graph.

DFG Data Flow Graph.

DLP Data Level Parallelism.

DSP Digital Signal Processor.

FWL Fractional Word Length.

GeCos Generic Compiler Suite.

HLS High-Level Synthesis.

HP Hawlett-Packard.

IR Intermediate Representation.

ISA Instruction Set Architecture.

IWL Integer Word Length.

KIT Karlsruhe Institute of Technology.

LSB Least Significant Bit.

LTI Linear Time-Invariant.

MIS Multimedia Instruction Set.

MMX Matrix Math eXtension.

MSB Most Significant Bit.

PFCG Pack Flow and Conflict Graph.

SCoP A loop nest in which all loop bounds, conditionals and array subscripts are affine functions of the surrounding loop iterators and parameters.

SFG Signal Flow Graph.

SGG Statement Grouping Graph.

SIMD Single Instruction Multiple Data.

Simdization The process of converting scalar instructions in a program into equivalent SIMD instructions.

SLP Superword Level Parallelism.

SQNR Signal-to-Quantization-Noise Ratio.

SWP Sub-word Level Parallelism.

TI Texas Instruments.

VLIW Very Long Instruction Word.

VP Variable Pack.

VPCG Variable Pack Conflict Graph.

WLO Word Length Optimization.

Chapter 1

Introduction

Contents

1.1 Context and Motivations 2
1.2 ALMA Project 4
1.3 Timeline 6
1.4 Contributions and Organization 8

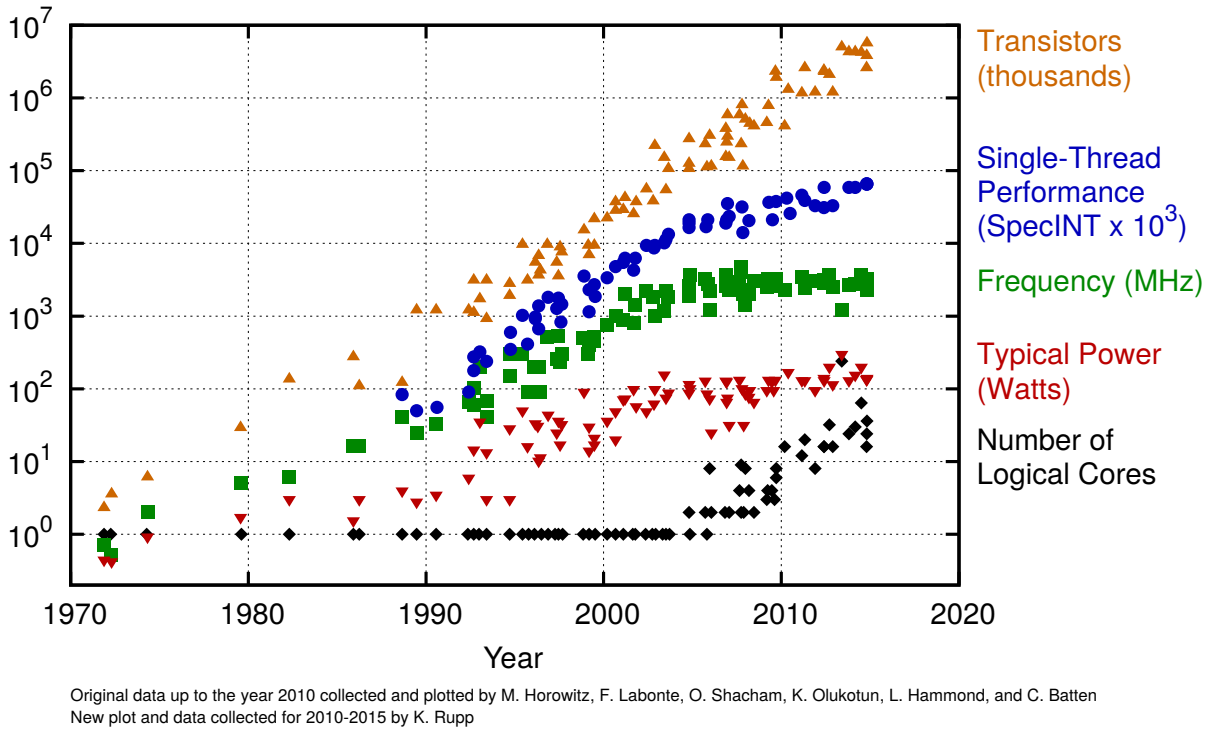


Figure 1.1 – CPU trend over the last 40 years. Published by K. Rupp at www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data

1.1 Context and Motivations

Ever since the first transistor computers appeared in the late '50s, manufacturing technologies kept continuously improving, allowing a steady exponential growth of the transistor count that can be integrated into a single die. This improvement rate was early observed by Gordon Moore in the mid '60s, when he predicted that the transistor density in an integrated circuit would double every two years. This has later led to the famously known Moore's law.

The transistor density growth, up to the early 2000s, was mostly invested in improving single core CPU performance, as shown in the graph of fig. 1.1. This was essentially achieved by increasing the core operational frequency, up to a point where the power density became too high for the generated heat to be practically dissipated. Limited by this *Power Wall*, frequency increase has stalled (mid 2000s) while the transistor density kept on increasing exponentially, causing a shift in focus toward multi-core parallelism. Though, other forms of (intra-core) parallelism have been exploited since the '60s, including pipelining, superscalar execution and Single Instruction Multiple Data (SIMD), which contribute to the continuous improvement of single-core performance.

Unlike general purpose (mainstream) processors, embedded ones are subject to stricter design constraints including performance, cost and power consumption. Indeed, they are destined to be

used in a wide variety of domain-specific applications with, for instance, a limited power source. In many application domains, such as signal processing and telecommunication, real numbers computation is employed. Since exact real numbers are not practically possible to represent in a processor, designers/developers resort to alternative, approximative representations of real numbers, which should be accurate enough while satisfying performance, power consumption and cost constraints of the targeted application. Among the most commonly used such representations are *floating-point* and *fixed-point*. Floating-point representation has the advantage to be very easy to use but it requires dedicated hardware support which increase the cost. On the other hand, fixed-point is cheaper since it is based on integer arithmetic but it is more complex to use, which increase the development time. So, the choice between these possibilities is mainly a tradeoff between cost (and power consumption) and ease of programmability.

Many embedded processors nowadays, such the ARM cortex-A family, provide hardware support for floating-point arithmetic, however a good number of ultra low power embedded processors, such as ARM cortex-M0/1/3, TI TMS320 C64x and Recore XENTIUM, do not. This comes at the cost of restraining programmability to the use of fixed-point arithmetic, while application prototyping generally employs floating-point the for sake of simplicity. This means that implementing the application on such processors requires either a software emulation of floating-point arithmetic or a conversion of floating-point into fixed-point. While feasible, software emulation of floating-point results in very poor performance. Alternatively, fixed-point implementation, if specified carefully, can achieve much (1 to 2 order of magnitude) better performance. However, this cannot always be done while keeping the same numerical accuracy, as it would require the use of increasingly large word-lengths, which unless supported by the target processor would also require software emulation, thus compromising performance. Instead, quantizations are applied to limit word-length growth at the cost of introducing *quantization errors* which alter the computation accuracy. This performance/accuracy tradeoff can be exploited during floating-point to fixed-point conversion in a process known as Word Length Optimization (WLO).

When targeting a processor that can only operate on data with a fixed word-length (the word-size, 32-bit in general), WLO does not make much sense. In fact, employing smaller data sizes does not necessarily benefit the application performance. On the one hand, it may require additional operations to perform data-type conversions, since all integer operations will eventually be performed on operands converted to the native word-size anyway. But on the other hand, it may reduce the memory footprint, which can improve performance. All in all, it is generally better to use the native word-size when targeting such processors.

The story changes when targeting processors with support for SIMD operations. In this case, smaller data word-lengths can be exploited to perform an operation on several (packed) data simultaneously, using SIMD instructions. In principle at least, this helps reducing the number of operations, thus ultimately improving performance. This can be exploited during WLO to explore the performance/accuracy tradeoff. Most embedded processors, such as XENTIUM and ARMv7, provide support for SIMD with various levels. However, taking advantage of SIMD capabilities to improve performance is a challenging task.

Automated methodologies for floating-point to fixed-point conversion and Simdization [1] are essential to reduce development cost and cut down time-to-market. None of the existing work tackles both transformations simultaneously, despite the strong relation between them. Typically, the floating-point to fixed-point conversion is performed first and then Simdization is (optionally) applied on the resulting fixed-point code.

In this thesis, we argue that considering both transformations independently yields less efficient solutions. We propose a new methodology to combine WLO with Simdization.

1.2 ALMA Project

From a broader perspective, this thesis took place in the context of the European project ALMA [2]. As we mentioned previously, applications in many domains, such as signal processing, are prototyped or specified using floating-point arithmetic and without too much worry about the target architecture characteristics, such as parallelism, for the sake of simplicity. More often than not, the application prototyping is done using high-level numerical languages or frameworks such as MATLAB or SCILAB. Once the specification is ready, the implementation phase aims at providing an accurate and optimized implementation for the target architecture. In the case of embedded multi-core processors with no hardware support for floating-point arithmetic, this generally involves three main steps:

— MATLAB (or SCILAB) to C/C++ conversion.

— Floating-point to fixed-point conversion.

— Coarse and fine -grained Parallelization.

Each of these steps is time consuming and error prone, which greatly increases the development time.

To address this problem, MATHWORKS provides an automatic C code generator, from a subset of the MATLAB language. However, the generated C code uses library calls to MATLAB special functions, for which the source code is not provided. This makes the code difficult to optimize in a later stage. Besides, MATHWORKS tools are proprietary and open source alternatives, like SCILAB, do not provide a similar functionality. Hence, the main motivations behind the ALMA project is to provide an alternative solution to this problem.

The ALMA project [2] aims at addressing the aforementioned problems by providing a complete tool-chain targeting embedded multi-core systems. An overview of the proposed tool-chain flow is depicted in fig. 1.2.

Starting from a SCILAB code, the tool aims, in a first place, at automatically converting it into an annotated C code. The latter then undergoes a multitude of optimizations, mainly performing:

— Coarse-grain parallelization: to exploit the multi-core nature of the target processors.

1. Simdization is the process of converting scalar instructions in a program into equivalent SIMD instructions.

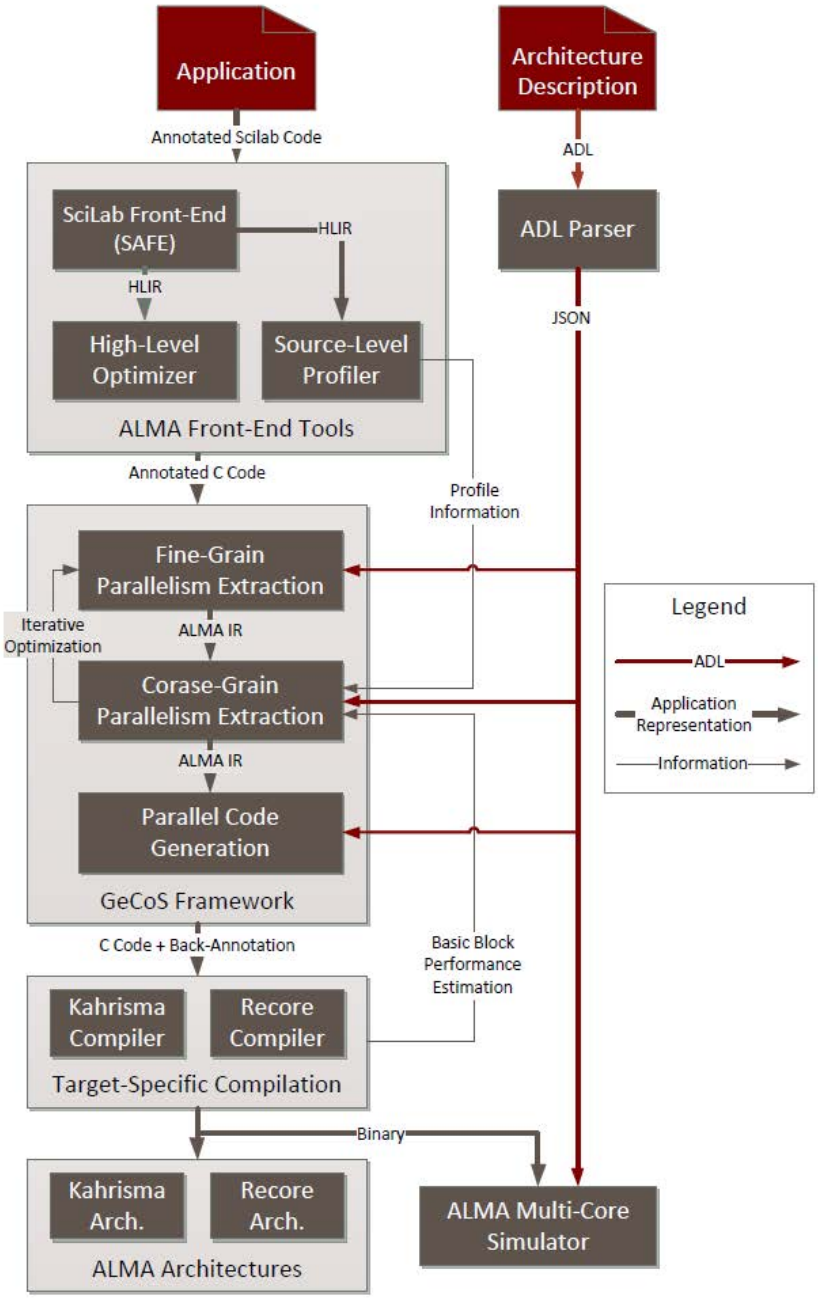


Figure 1.2 – ALMA tool-chain flow diagram.

— Floating-point to fixed-point conversion: to avoid performance degradation due to floating-point simulation.

— Simdization: to take advantage of the SIMD capabilities of the target processor cores.

The tool finally generates a parallel C code using a generic MPI (message passing interface), SIMD and fixed-point Application Programming Interfaces (APIs).

ALMA targeted two multi-core architectures from Recore Systems and Karlsruhe Institute of Technology (KIT), based on the XENTIUM [104] and KAHRISMA [58] cores respectively. None of which support floating-point arithmetic, but they provide subword SIMD capabilities.

1.3 Timeline

In this section, we present a brief timeline of the work done during this thesis, in order to help understanding the context, choices and contributions made during this work.

In the context of ALMA, we mainly had to:

— Implement a floating-point to fixed-point conversion, since both ALMA targets do not provide hardware support for floating-point arithmetic [2].

— Implement an automatic Simdization, since compilers of ALMA targets do not perform this transformation.

— Explore the performance/accuracy tradeoff using WLO and taking into account SIMD opportunities.

So, we explored the state-of-the-art for floating-point to fixed-point conversion targeting embedded processors (cf. chapter 2). We found that most approaches are similar in the way they address the problem:

1. First, the Integer Word Lengths (IWLs) are determined based on dynamic range values, which can be obtained using simulation or analytical methods.
2. Then, the word-lengths are specified, either by simply using a default word-length (generally the native word-size e.g. 32-bit), or by performing a WLO under and accuracy constraint.

The different approaches differ in the way dynamic ranges are obtained and/or the WLO algorithm and/or the accuracy estimation procedure.

Integrating such transformations into the target compilers is not a trivial task, but most importantly it should be done for each different target to be supported by the flow. Instead, we decided to implement this conversion at source code level using a source-to-source compiler flow. For this matter, we used the source-to-source compilation framework, Generic Compiler Suite (GeCoS) [38], which already integrates a floating-point to fixed-point conversion tool, IDFix [3], providing automatic dynamic range and analytical accuracy evaluation methods. Besides, this choice is also motivated by the fact that GeCoS/IDFix provides an extension mechanism allowing for "simple" integration of different methods for range evaluation and accuracy estimation without affecting our work. However, IDFix was initially designed for targeting High-Level Synthesis (HLS) using C++ fixed-point libraries. Since not all embedded processor compilers support C++ and in order to avoid the performance overhead introduced by using such libraries, we implemented a fixed-point C code generator using native integer data-types and operations.

2. Besides, this is also the case for many embedded processors.

Similarly, we decided to implement Simdization at source code level so that it can be easier to extend in order to support other targets. We investigated the different ways of performing Simdization. The existing techniques can be categorized into two groups:

— Loop-level vectorization.

— Basic-block level, also known as Superword Level Parallelism (SLP).

We decided to go with SLP, since it can exploit more opportunities than loop vectorization without the need for "complex" dependency analysis and loop transformations. We investigated the state-of-the-art of SLP extraction algorithms and we decided to implement the algorithm proposed by Liu et al [78] in 2012. However, during the implementation we found many shortcomings, so we came up with an improved SLP extraction algorithm, that we present in chapter 4. We implemented it in addition to the aforementioned algorithm by Liu et al, so that we can compare them. We integrated the SLP extraction algorithms as well as a SIMD C code generator into the same source-to-source compilation framework, GeCoS.

At this point, we had at our disposal a complete source-to-source flow capable of automatically generating a fixed-point SIMD C code for different embedded processors. With all that out of the way, we started exploiting the interaction between WLO on the one side and SLP on the other. In the literature, few existing floating-point to fixed-point conversion approaches target embedded processors with SIMD capabilities. Existing work though, do not consider Simdization while performing WLO; they simply assume that selecting narrower word-lengths would eventually increase the SIMD opportunities, and improve performance consequently. However, this assumption is very optimistic since the WLO in unaware of the SIMD opportunities and the associated cost overhead, which can result in a very high performance degradation [3].

Using the source-to-source compilation flow we already implemented, we integrated a typical WLO strategy that aims essentially at reducing data word-lengths without considering Simdization. In order to test how well such strategy can perform, we applied floating-point to fixed-point conversion (using the aforementioned WLO strategy), followed by SLP extraction, on some benchmarks for XENTIUM, KAHRISMA and two other embedded processors. The results showed that such an approach is not very efficient for targeting SIMD processors; the observed speedup due to Simdization varies inconsistently, supporting our hypothesis about the fact that, simply minimizing word-lengths without taking into account the Simdization problem would yield inefficient solutions.

In order to solve this problem, we propose a new SIMD-aware floating-point to fixed-point conversion approach based on a joint WLO and SLP extraction algorithm. We also integrate the proposed joint WLO and SLP algorithm to the source-to-source compilation flow in order to test it validity compared to prior typical approach. Using our approach, we obtain more efficient overall solutions; it enables a better exploitation of the performance/accuracy tradeoff when targeting embedded processors.

3. mainly due to data packing/unpacking operations

1.4 Contributions and Organization

More specifically the contributions of this work are the following ones:

— A new Intermediate Representation (IR) for SLP extraction (cf. chapter 4).

— A new SLP extraction algorithm (cf. chapter 4).

— A new approach for floating-point to fixed-point conversion considering, jointly, WLO and SLP extraction (cf. chapter 5).

— A fully automated source-to-source compilation flow [4] for SLP extraction and floating-point to fixed-point conversion, together with a fixed-point and SIMD C code generator with support for several embedded processors.

In the remainder of this manuscript, we first present some contextual background on floating-point and fixed-point representations and the conversion methodologies, in chapter 2. Then we present existing techniques for exploiting SIMD parallelism, in chapter 3.

In chapter 4, we present a thorough analysis of the state-of-the-art of SLP extraction algorithms and we propose a new enhanced algorithm. We implement the proposed algorithm as a source-to-source compilation flow and we compare it against a state-of-the art SLP extraction algorithm.

In chapter 5, we investigate the interactions between floating-point to fixed-point conversion and SLP extraction and we propose a new SLP-aware WLO algorithm. We implement it as a source-to-source compilation flow and we compare it against a typical approach performing floating-point conversion first, then SLP extraction.

4. using the compilation framework Gecos [38]

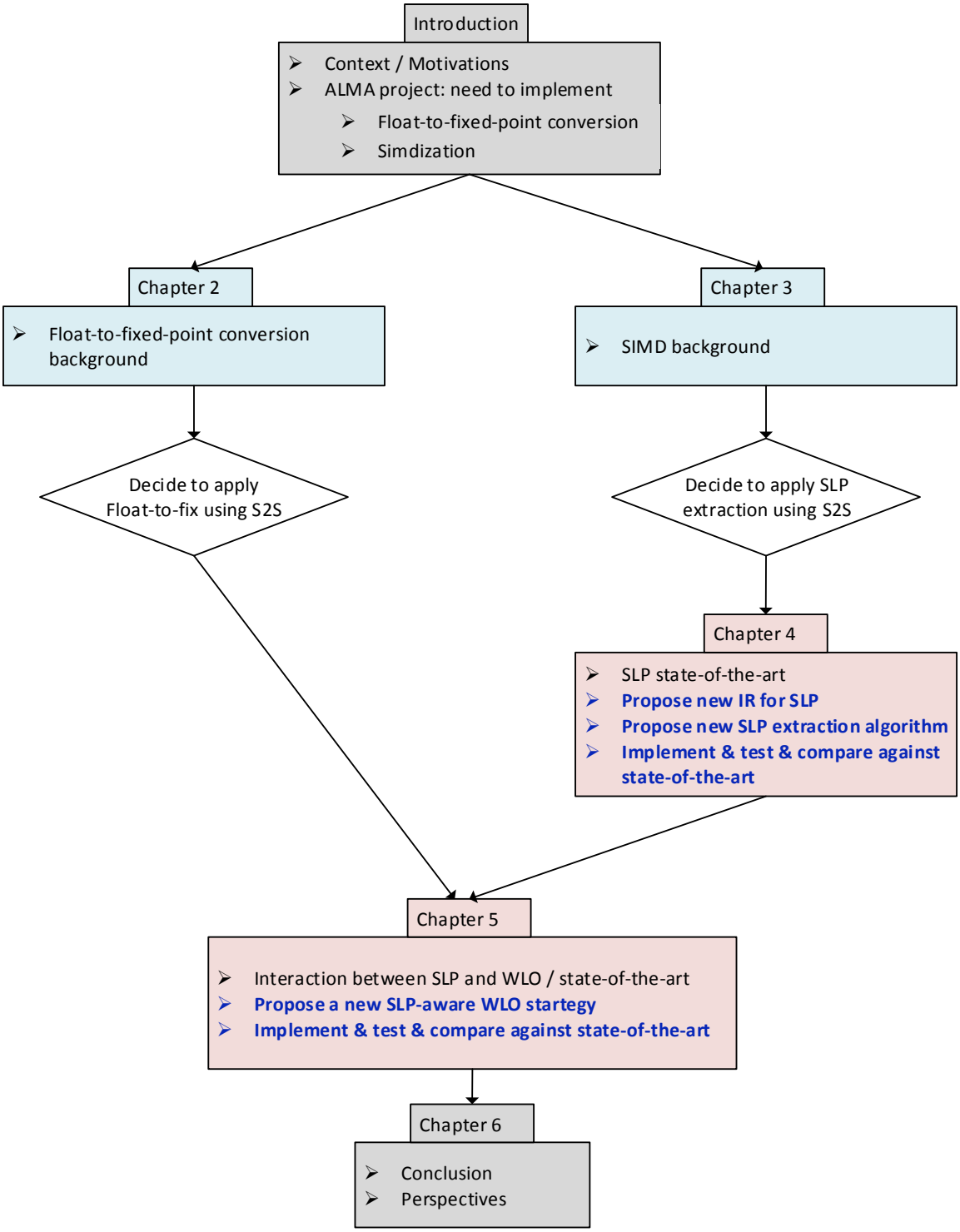


Figure 1.3 – Outline.

Chapter 2

Background: Floating-point to Fixed-point Conversion

Contents

2.1 Introduction 12
2.2 Floating-point Representation 12
2.3 Fixed-point Representation 14
2.4 Floating-point vs. Fixed-point 16
2.5 Floating-point to Fixed-point Conversion Methodologies 19
2.5.1 Integer Word Length Determination 19
2.5.2 Word Length Optimization 21
2.5.3 Fixed-point Code Generation 22
2.6 Automatic Conversion Tools 23
2.6.1 MATLAB Fixed-point Converter 23
2.6.2 IDFIX 24
2.7 Conclusion 27

2.1 Introduction

Real number computations are employed in many application domains, such as digital signal processing. Exact representation for most real numbers, like π for instance, require unlimited precision, thus it is impossible to represent them explicitly. However, they can be represented with virtually unlimited precision using implicit representations instead, such as functions [17]. But such representations are generally not practical and require lots of computing labor. Besides, for most applications, limited precision real arithmetic approximations are good enough. In the following, we will consider two of the most commonly used real number approximations, namely floating-point and fixed-point representations.

The goal of this chapter is mainly to explore existing solutions for floating-point to fixed-point conversion. In sections 2.2 and 2.3 we introduce floating-point and fixed-point representations, then we compare them in section 2.4. Finally, we discuss existing methodologies for floating-point to fixed-point conversion in section 2.5 and we present some existing tools for automatic conversion in section 2.6.

2.2 Floating-point Representation

Floating-point representation is an approximation of real numbers using a limited precision mantissa (or significand), scaled by a **variable** factor specified by a limited precision exponent:

$$mantissa \times base^{exponent} \tag{2.1}$$

It is hence similar to scientific notation. The base is common to all numbers in a defined system, so it is implicit and not represented in the number. In addition to the base, the precision and format (interpretation) of the mantissa and the exponent define a floating-point representation.

Since multiple floating-point representations of a real number are possible, it is very hard to maintain portability between different architectures. To overcome this problem, IEEE has defined a standard representation for binary floating-point numbers. This standard, namely IEEE 754, defines the format of floating-point numbers in base two. The floating-point approximation $FL(x)$ for a given real number x is represented as follows:

$$FL(x) = (-1)^s \times |mantissa| \times 2^{exponent} \tag{2.2}$$

The mantissa is represented in *sign-magnitude* representation where the sign bit is s, as depicted in fig. 2.1. The mantissa magnitude is normalized to the range $[1, 2[$. Only the fractional part is

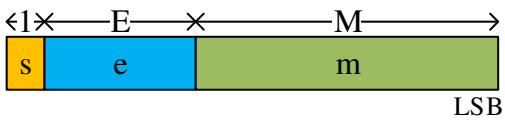


Figure 2.1 – Binary representation of floating-point numbers in IEEE 754 standard.

stored in the number as m on M bits, as depicted in fig. 2.1, and the leading integer bit, set to 1, is implicit.

$$mantissa = (-1)^s \times (1.m) \tag{2.3}$$

The exponent is a signed integer represented in *excess* $(2^{E-1} - 1)$ representation. The biased exponent is stored in the number as e on E bits. The true exponent value is obtained from e by adding the bias $2^{E-1} - 1$:

$$exponent = e - (2^{E-1} - 1) \tag{2.4}$$

The exponent value ranges in $[-(2^{E-1} - 1), 2^{E-1}]$. The minimal value, $-(2^{E-1} - 1)$, indicates an *Underflow*. In this case, the mantissa in not normalized (denormalized mode), the implicit leading bit is 0 and the exponent value is set to $-2^{E-1} - 2$. The values ± 0 are represented in this mode with $m = 0$. Whereas, the exponent maximal value, 2^{E-1}, represents two special cases:

— $\pm\infty$ for $m = 0$,

— NaN (Not A Number) for $m \neq 0$.

When the exponent value exceeds 2^{E-1} an *Overflow* occurs.

The IEEE 754 standard defines two main binary floating-point types, among others:

— 32-bit single precision for $M = 23$ and $E = 8$,

— 64-bit double precision for $M = 52$ and $E = 11$.

It also defines the operations on floating-point numbers, the exceptions and the different rounding modes.

Floating-point Addition/Subtraction is performed through the following steps:

1. Align the operand exponents to the maximal one, which is set as the result exponent, by right shifting the mantissa of the smallest exponent number by the difference of exponents.
2. Add/Sub aligned operand mantissas to obtain the result mantissa.
3. Normalize the result. If the mantissa magnitude is out of range $[1/2, 1 - 2^{-M-1}]$, shift it into range and increment/decrease the result exponent accordingly.
4. Round the result mantissa and adjust the exponent if necessary.

Floating-point Multiplication/Division requires fewer steps:

1. Mul/Div operand mantissas to get the result mantissa and Add/Sub exponents to obtain the result exponent.
2. Normalize the result.
3. Round the result.

In addition, all operators should check for overflow and other exceptions such as division by zero.

The rounding step introduces a rounding error. Various rounding methods are possible such as round to nearest even (default), toward 0 or toward ∞. Besides, operands alignment, for

add/sub, may result in a precision loss due to the right shifting. Guard bits are generally added to reduce these errors.

2.3 Fixed-point Representation

Fixed-point representation is an approximation of real numbers using a limited precision integer, scaled by a **fixed**, implicit factor. For a given real number, x, a binary fixed-point approximation, $FX(x)$, is represented as follows:

$$FX(x) = sif \times 2^{-F} \tag{2.5}$$

Where sif is a limited precision integer. It is the only information stored in the number using two's complement representation for signed numbers, as depicted in fig. 2.2. The Most Significant Bit (MSB) of the integer, s, represents the sign in case of signed numbers. The integer (signed or not) is interpreted as though it is multiplied by a scaling factor, 2^{-F}, specified by F which is referred to as the Fractional Word Length (FWL). F determines the position of the virtual binary point in respect to to the Least Significant Bit (LSB). The remaining I MSBs are referred to as the Integer Word Length (IWL). It can also be used to specify the position of the virtual binary point in respect to to the MSB. All three parameters, W, F and I are related by the following equation:

$$W = I + F \tag{2.6}$$

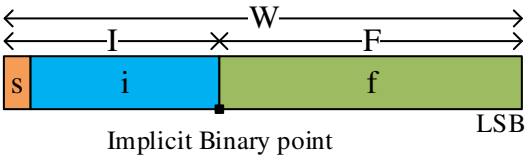


Figure 2.2 – Binary representation of a signed fixed-point number.

A fixed-point format is thus specified by at least two of the three parameters, I, F and W, in addition to the signedness. We will use the notation $<W, I, F>$ to refer to a fixed-point format, though we may skip one of the three parameters for brevity. In this case, the skipped parameter will be replaced by a '_' and can be simply obtained using eq. (2.6). In the case of signed numbers, the sign bit is accounted for in I.

Fixed-point Arithmetic

Fixed-point arithmetic is essentially integer arithmetic with proper handling of scaling factors. Let us consider two signed fixed-point numbers, f_1 and f_2, represented by the integers x_1 and x_2 with respective formats, $<W_1, I_1, F_1>$ and $<W_2, I_2, F_2>$.

For addition (and subtraction), the operands must first be aligned to the same scaling factor,

before applying the corresponding integer operation.

$$f_1 + f_2 = x_1 \times 2^{-F_1} + x_1 \times 2^{-F_2} = ((x_1 \times 2^{F-F_1}) + (x_2 \times 2^{F-F_2})) \times 2^{-F}, \quad F = \max(F_1, F_2) \quad (2.7)$$

To avoid precision loss, the operand with the smallest FWL is left shifted by $|F_1 - F_2|$, so that they both align to $\max(F_1, F_2)$. The operand word-length must also be increased by the same amount to avoid any potential overflow. Once the scaling factors are aligned, the underlying integers can be added/subtracted to obtain the result. This step may require sign extension. The format of the fixed-point result in this case is $<\_, \max(I_1, I_2) + 1, \max(F_1, F_2)>$

For multiplication, the underlying integer operand can be multiplied directly without aligning. The format of the fixed-point result is $<W_1 + W_2, I_1 + I_2, F_1 + F_2>$.

$$f_1 \times f_2 = (x_1 \times 2^{-F_1}) \times (x_1 \times 2^{-F_2}) = (x_1 \times x_2) \times 2^{-(F_1+F_2)} \quad (2.8)$$

Fixed-point operation	**Integer operations**	**Exact result format**
$f_1 + f_2$	Align to $max(F_1, F_2)$ $x_1 + x_2$	$<\_, \max(I_1, I_2) + 1, \max(F_1, F2)>$
$f_1 - f_2$	Align to $max(F_1, F_2)$; $x_1 - x_2$	$<\_, \max(I_1, I_2) + 1, \max(F_1, F2)>$
$f_1 \times f_2$	$x_1 \times x_2$	$<W_1 + W_2, I_1 + I_2, F_1 + F2>$

Table 2.1 – Exact fixed-point operations.

As can be seen in table 2.1, exact computations over fixed-point numbers require an eventual growth of the underlying integer word-lengths, specially in case of multiplication where the exact result requires $W1 + W2$ bits. Implementing such operations, when targeting a processor with predefined word-lengths, generally requires some sort of software emulation to support wider word-lengths, thus degrading performance. As a consequence, the fixed-point numbers are quantized to make them fit the target processor supported word-lengths.

Quantization

To convert a fixed-point number from a format $<W, \_, F>$ to $<W - k, \_, F - k>$, with $k > 0$, the k LSBs of the underlying integer should be eliminated by rounding the value of the number. This conversion is referred to as quantization.

Different rounding modes can be used, such as round toward zero (a.k.a. truncation) or round to the nearest value. Regardless of the rounding mode, the k LSBs are lost, resulting in potential precision loss. The error introduced due to quantization, known as *quantization error* (or noise), propagates in the computation system and may result in significant error at the system output.

Overflow and Saturation

The range of representable numbers by a signed fixed-point format $<W, \_, F>$ is:

$$range = [-2^{W-1}, 2^{W-1} - 1] \times 2^{-F} = [-2^{W-1-F}, 2^{W-1-F} - 2^{-F}] \quad (2.9)$$

This corresponds to the range of the underlying integer format (on W bits) scaled by a factor determined by the value of F.

An overflow occurs when a value goes out of range. In this case, the underlying integer value cannot fit on W bits. Consequently, the value of the underlying integer is *wrapped around* and the MSBs are lost, thus introducing a very large error. This overflow behavior (or mode) is known as *wrap around*.

However, the introduced error can be reduced by clipping to the maximal (or minimal) representable value on overflow. This overflow mode is known as *saturation*.

2.4 Floating-point vs. Fixed-point

In this section, we compare floating-point and fixed-point representations based on different criteria, including range, precision, implementation cost and ease of use.

Range and Precision

The precision of a fixed-point representation $<W, \_, F>$ is determined by the scaling factor or the unit-in-last-position (ulp). It is a constant given by:

$$ulp = 2^{-F} \tag{2.10}$$

In a IEEE floating-point representation, the mantissa magnitude can represent a set of 2^M different floating-point numbers in the range $[2^{exp}, 2^{exp+1} - 2^{exp-M}]$, for a given exponent value $exp \in [-(2^{E-1} - 2), 2^{E-1} - 1]$ (in normalized mode). The unit-in-last-position (or precision) is variable depending on the exponent value:

$$ulp = 2^{exp-M} \tag{2.11}$$

So all the numbers with same exponent have the same precision but numbers with higher exponent values are represented with lower precision as depicted in fig. 2.3. The range of representable floating-point positive numbers (normalized) is:

$$range = [2^{-(2^{E-1}-2)}, 2^{2^{E-1}} - 2^{2^{E-1}-1-M}] \tag{2.12}$$

Therefore, floating-point representation has the advantage to cover a much wider dynamic range but with adaptive precision, whereas fixed-point representation has a constant precision but covers a much narrower dynamic range.

Implementation

Hardware implementation of floating-point operators is expensive since it has to handle operands alignment, normalization, rounding and check for exceptions. This is mainly due to the fact that

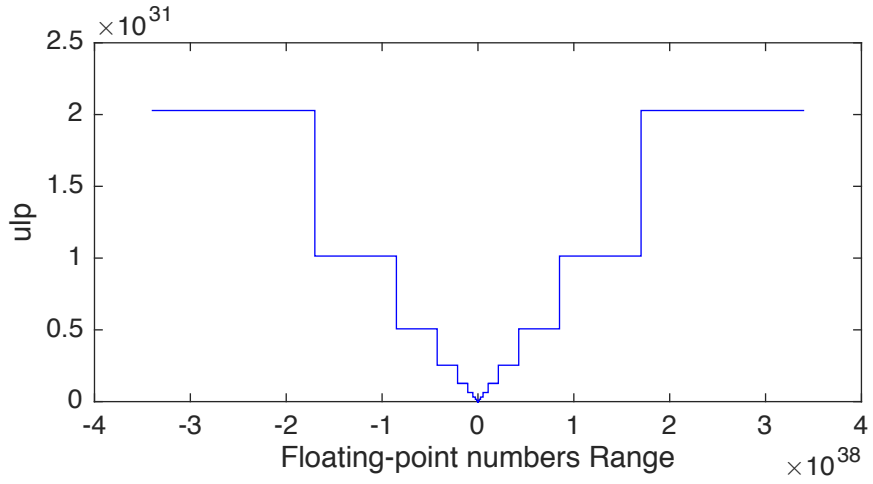


Figure 2.3 – IEEE single-precision floating-point numbers precision vs. range.

the alignment requires a right shifter and the normalization requires a left/right shifter for up to M positions. Implementing these shifters is a trade-off between execution speed and cost. For example shift registers are cheap to implement but require a variable number of cycles depending on the shifting amount. Barrel shifters on the other hand can perform any shift in constant time but are expensive. Generally multi-level barrel shifters are used.

On the other hand, since fixed-point essentially uses integer arithmetic, no special hardware implementation is required to support it.

For this reason, many embedded processors do not provide support for floating-point arithmetic, for the sake of reducing cost and/or power consumption; they only provide integer computation data-path. In order to perform real number computations on such processors, two options are possible: either emulating floating-point, or implementing fixed-point arithmetic using the integer data-path.

Floating-point arithmetic emulation has a great impact on performance since a single floating-point operation often requires tens of integer arithmetic, shift and branch operations to perform alignment, normalization and rounding as mentioned earlier. Also representing floating-point data requires either more memory, in case mantissas and exponents are stored separately, or extra computations to encode/decode them if they are stored on the same word. This overhead can be even greater depending on the accuracy of the emulation and its compatibility with IEEE 754; a full compliant simulation requires checking and handling of exceptions.

On the other hand, fixed-point arithmetic can be emulated much faster when limited to use the integer data types natively supported by the target. In this case most fixed-point operations can be computed directly using the corresponding integer operator with additional shift operations when scalings are needed (assuming truncation is used as quantization, and no saturation). However, since this method is limited to the precision of the native data-types, quantization should be applied to keep intermediate operation results fit. This procedure is tedious, error

prone and hard to debug since the programmer must keep track of the (implicit) fixed-point formats, for every variable and intermediate operation result, and perform necessary scalings accordingly.

Alternatively, using full-fledged fixed-point libraries, such as SystemC, can seamlessly emulate any fixed-point format with precisions higher than the native data types and can emulate operations with no precision loss. These libraries also support various quantization and saturation modes. But this is at the cost of much slower simulation speeds, by a factor of 20 to 400 as reported by KEDING et al [50].

Therefore the only viable option when seeking tight performance and power consumption goals is to use native data types to represent fixed-point numbers and perform quantizations to keep data fit. To enhance the performance and precision of this approach, some specialized processors provide fixed-point specific enhancements such as:

— Pre-scaling of the input operands and/or post-scaling of the result.

— Extended precision operators and registers.

— Hardware support for saturation and rounding.

Programmability

Most programming languages, like C/C++, provide native seamless support for standard floating-point data-types and operations but not for fixed-point; this is one of the reasons why most applications are developed using floating-point.

Floating-point is simpler to use, since the hardware does all the hard work providing an intuitive and straightforward interface. However, it can be very tricky in some cases. Floating-point immediate numbers are generally expressed in base 10 for the sake of simplicity. More often than not, these numbers cannot be exactly represented by the floating-point system being used (generally base 2), causing unintuitive behavior. For instance, comparing the result of operation 0.1×10 against 1 gives an unexpected result; both numbers are not equal as it might look like. Indeed, 0.1 is exactly representable in base 10 but not in base 2. In fact, C standard does not specify what base is used to represent floating-point data types, but in general it is base 2.

Conclusion

The characteristics of floating-point and fixed-point representations are summarized in table 2.2. Due to the high implementation cost of floating-point, fixed-point representation is often preferred in the context of ultra low power embedded applications. However, the development time is higher. Thus, automated floating-point to fixed-point conversion methodologies are required to cut down the time-to-market.

Representation	Range	Precision	Cost	Programmability	Development Time
Floating-point	very wide	variable	high	easy	low
Fixed-point	limited	constant	low	difficult	high

Table 2.2 – Floating-point vs. Fixed-point.

2.5 Floating-point to Fixed-point Conversion Methodologies

As discussed earlier, floating-point is not suitable when targeting low power embedded processors, and fixed-point is preferably used instead. Therefore, when applications are designed using floating-point, a floating-point to fixed-point conversion is required.

This conversion aims at attributing a fixed-point format to each floating-point data, and at replacing floating-point operations with fixed-point operations along with proper handling of scalings. This conversion may introduce computation errors due the overflows and/or quantizations. The conversion process must be aware of these errors and be able to estimate their effects in order to make sure that the computation accuracy remains within an "acceptable" limit specified by the developer, according to the application tolerance.

Overflows generally induce large errors. However, they can be prevented by evaluating the dynamic value range of each variable and intermediate result and deducing, for each, the minimum IWL required to represent its value range. In this way overflows are mostly avoided. Alternatively, overflows can be allowed for cases with low occurrence probability to allow the use of smaller word-lengths. In this case saturation can be used to clip any overflow to the maximal (or minimal) value. In this case the induced errors should be analyzed since they may have a great impact on the computation accuracy.

In contrast, quantization errors are relatively small, but they can get amplified when propagated in the computation system and may result in a significant error at the system output. Therefore, it is very important to evaluate their effect on the computation accuracy and to make sure the latter stays within the specified limit.

Floating-point to fixed-point conversion generally involves three steps:

1. IWL determination of each variable and operation intermediate result in the system.
2. Word-length determination, to complement IWL determination in order to fully specify the fixed-point formats. This generally makes the subject of an optimization, called Word Length Optimization (WLO).
3. Fixed-point code generation.

In the following, we discuss each of these steps.

2.5.1 Integer Word Length Determination

The IWL of a variable is determined based on its dynamic value range. The aim is to specify the binary point position in the fixed-point formats in such a way to avoid overflows. The dynamic

range can be obtained either using simulation-based methods [64] or analytical methods.

Simulation-based Methods

The floating-point code is instrumented to collect statistics on floating-point variables and operation results using simulations with representative input samples. The collected statistics are used to determine the dynamic ranges of the corresponding floating-point variables, which is then used to determine their IWL.

Simulation-based methods have the advantage to find tight ranges and therefore do not allocate unnecessary bits for the integer part. However, they do not guarantee the absence of overflows since the measured range depends on the tested input samples. thus, a large and representative input samples must be used to obtain accurate enough estimations of the dynamic range. Regardless, overflows may still occur and in this case saturation can be used to limit overflow errors.

Analytical Methods

Alternatively, analytical methods can be used to derive the dynamic range of each variable and intermediate result, in a given system. Range propagation can be achieved using interval arithmetic or affine arithmetic for instance. In this case the input ranges are propagated through operations by a applying a correspondent propagation rule.

For example, using interval arithmetic we can deduce the range of variable y, at the output of the system depicted in fig. 2.4, given the range of inputs a, b and c. Let $a \in [a_m, a_M]$, $b \in [b_m, b_M]$ and $c \in [c_m, c_M]$. The intermediate result of the multiplication is then $t \in [t_m, t_M]$, with:

$$t_m = min(a_m \times b_m, a_M \times b_M, a_m \times b_M, a_M \times b_m) \tag{2.13}$$

$$t_M = max(a_m \times b_m, a_M \times b_M, a_m \times b_M, a_M \times b_m) \tag{2.14}$$

Finally, $y \in [t_m + c_m, t_M + c_M]$.

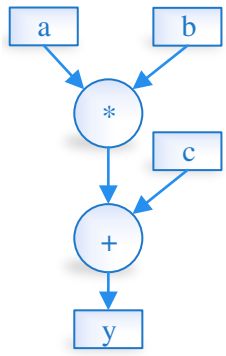


Figure 2.4 – Range propagation example.

Analytical methods have the advantage to be generally faster than simulation and results in

a certain range that ensures the absence of overflows. However, the obtained ranges may be over-estimated resulting in unnecessary bits being allocated for the integer parts.

2.5.2 Word Length Optimization

The aim of this step is to select the word-length of each variable and operation in the system. The FWLs are implicitly specified during this step following eq. (2.6), which directly affects the computation accuracy and performance. The word-length exploration space is generally specified depending on the target architecture. In the context of High-Level Synthesis (HLS) [65] targeting FPGA for instance, a wide range of custom word-lengths can be considered. However, when targeting an off-the-shelf processor, word-lengths are generally restricted to the ones natively supported (e.g. 8, 16 or 32 bits) by the processor. In order to explore this solution space for each variable/operation, a WLO algorithm [140, 85, 91] is generally used. It aims at selecting the best performance solution while maintaining accuracy within an "acceptable" limit. Therefore, this requires methodologies to evaluate the accuracy and the cost of a given fixed-point solution.

Cost Estimation

When targeting embedded processors, the goal is generally to optimize the execution time (although energy consumption could be also considered). Execution time can be estimated by running the application and collecting timing measurements. To do so, the fixed-point code implementing the solution to be tested should be generated, compiled and run on the target processor with representative input data samples. This has the advantage to give a precise estimation of the execution time. However, this method greatly increases the WLO time, since this needs to be done for each tested solution.

Alternatively, an estimation of the execution time can be obtained using static analysis or heuristic based on a cost model of the target architecture. Typically, a relative cost is associated to each operation type and size. In general, these methods do not lead to an accurate estimation of the execution time. However, it is often sufficient for comparing the cost between two different solutions.

Accuracy Evaluation

Accuracy evaluation aims at quantifying or measuring the accuracy of a given fixed-point solution. Many metrics can be used for this matter, such as Bit Error Rate (BER), generally used in communication applications, or Signal-to-Quantization-Noise Ratio (SQNR) often used in image/signal processing applications.

Simulation-based methods [128, 51, 65, 113] can be used for accuracy evaluation. They mainly consist of simulating the fixed-point system, with representative input samples, and comparing it against a baseline floating-point specification. The results are compared to find errors and compute the accuracy value. Such methods suffer from scalability issues, since the simulation time may be very high, specially in the context of design space exploration in which each tested

fixed-point solution should be simulated with enough input samples to evaluate its accuracy. This results in impractically long optimization times. On the other hand however, these methods have the advantage to be simple and applicable with no restrictions on the underlying computation system.

In order to cut-down the accuracy evaluation time, analytical methods [89, 87, 81, 83, 22, 109], aim at computing a closed-form generic expression, representing the fixed-point system accuracy, as a function of the number of bits assigned to different variables and operations in the system. Once this expression is generated, it can then be used to quickly evaluate the accuracy of any fixed-point solution for the system, by simply plugging-in the correspondent values of number of bits, making it more suitable for design space exploration. However, analytical methods are restricted to some particular systems, mainly Linear Time-Invariant (LTI) and non-recursive non-LTI systems.

For an LTI system, Menard and Sentieys [89] proposed a method to automatically compute the quantization noise power expression based on a computation of the transfer function of the system, represented by its Signal Flow Graph (SFG). This is then used to determine the SQNR.

Menard et al [87] later extended this method to cover non-LTI non-recursive systems using only one floating-point simulation to determine statistical parameters of noise sources and signal. These techniques evaluate the first moments (mean and variance) of the quantization noise sources and propagate it in the system to the output. To do so, the system is represented by its SFG, which can be constructed from the corresponding C code, but this requires flattening of all the control structures. For large systems, this make the computation of the analytical accuracy evaluation expression slow.

To solve this problem, Deest et al proposed an alternative, more scalable, representation [30] of the system based on the polyhedral model.

Shi and Brodersen proposed an alternative method [121] using a model of the quantization noise based on perturbation theory. This method uses simulations to compute the value of some parameters needed to obtain the analytical expression of the output noise power.

Alternative methods [81, 137] use affine arithmetic to represent the quantization noise in order to compute an analytical expression of the accuracy.

2.5.3 Fixed-point Code Generation

As discussed earlier, fixed-point can be simulated using C++ libraries such as SystemC and Algorithmic C Datatypes [1]. This solution has the advantage to seamlessly represent arbitrary-precision fixed-point formats and supports different overflow and quantization modes. However, it introduces a significant performance overhead as shown by Keding et al [50].

As mentioned earlier in section 2.4, the best performance option, when targeting embedded processor, is to limit fixed-point word-lengths to the natively supported ones. In this case, the fixed-point code can be generated using only native integer data-types to represent fixed-point

data. Fixed-point operations are directly performed using integer operations; however, scaling operations should be carefully added to reflect the fixed-point quantizations. This option is much more tedious and error prone to implement. Using automatic code generation to produce such code can alleviate this problem.

Scalings are implemented using shift operations. This can therefore severely impact the performance depending on the support provided by the target processor. For this reason, floating-point to fixed-point conversion methodologies should minimize the cost of scaling operations. KUM et al proposed a scaling optimization algorithm [64] that minimizes a scaling cost function using integer linear programming, taking into account whether or not the target processor has a barrel shifter. MENARD et al proposed a method[85] to reduce the impact of scaling operations on the execution time. This optimization, performed under accuracy constraint, aims at moving the scaling operations whenever possible: when a scaling operation is brought outside a loop structure, for instance, its impact on the execution time is greatly reduced.

2.6 Automatic Conversion Tools

In order to reduce time-to-market, automatic or semi-automatic tools are crucial in order to explore the fixed-point design space while obeying to strict time-to-market constraints.

Many simulation-based semi-automatic conversion tools are commercially available like MATLAB Fixed-point converter and SYNOPSYS System Studio. Such simulation-based tools suffer from high conversion latency, which limits the design space exploration, but it has a wide code coverage.

Alternatively, analytical-based conversion tools suffer from very restricted code coverage. For this reason they are generally limited to research tools like IDFIX [3]. Nevertheless, given a compatible system, they allow for faster conversion and hence more efficient design space exploration.

2.6.1 MATLAB Fixed-point Converter

MATLAB is widely used for many domain specific applications design like signal processing. In MATLAB, fixed-point programming is supported; fixed-point data types can be created via the MATLAB `fi` function.

Besides explicit fixed-point programming, MATLAB provides a semi-automatic simulation-based tool to assist the conversion of floating-point MATLAB programs to fixed-point. This tool is referred to by MATHWORKS as *Fixed-point Converter*. This tool mainly functions as follows:

— It takes as inputs a floating-point MATLAB function in addition to a testing MATLAB script that calls the function and provides simulation input samples.

— The first step consists in instrumenting the input floating-point code, which is then run via the testing script to collect statistics about floating-point variables and operations. These

statistics mainly include the dynamic maximal, minimal and mean values.

— The simulation dynamic range is then used to infer the IWL for each variable. The user can specify an additional safety margin to the simulation range in order to minimize overflow occurrences. Alternatively, he can specify a static range for each, or some, variables which are used instead of the simulation ranges.

— The user can then either specify a default word-length (or FWL) which will be used to infer the FWLs (or word-lengths). Alternatively, he can manually specify the fixed-point format for each or some variables.

— The operations intermediate result formats are inferred based on a user-defined default behavior for each operation type. It indicates whether the operation should keep full-precision or perform quantization. In the latter case it specifies what quantization mode is to be used.

— Finally, the fixed-point MATLAB code is generated to reflect the selected fixed-point specification, using `fi` function to create fixed-point types and specify default behaviors.

— The tool also provides a way to automatically instrument and simulate the resulting fixed-point code in order to compare it against the original floating-point version to verify its accuracy. This eases the fixed-point design space exploration.

Since MATLAB code is not suitable to run on embedded processors, the developer must port the generated MATLAB code to C/C++ in general. This process is time consuming and error prone. However, MATLAB also provides a tool to automate this task for a subset of MATLAB syntax. But this has several disadvantages:

— The generated C code uses library calls to MATLAB special functions, for which the source code is not provided. This make the code difficult to optimize in a later stage.

— These tools are proprietary, and alternative open source languages like SCILAB does not provide similar functionalities.

The ALMA [2] project, presented earlier in section 1.2, aimed at addressing these limitations by providing a complete tool-flow starting from SCILAB code down to C code with coarse- and fine- grained optimizations and parallelism extraction. In this context, a contribution of the work of this thesis was to provide floating-point to fixed-point conversion, along with Simdization, to efficiently target embedded processors with no hardware floating-point support.

2.6.2 IDFIX

IDFIX is a research tool for automatic floating-point to fixed-point conversion using analytical methods, originally designed to target HLS. It is developed by the Cairn/Irisa team and is integrated in the source-to-source compilation framework, Generic Compiler Suite (GeCoS).

IDFIX flow diagram is depicted in fig. 2.5. It takes as input an annotated floating-point C code, an accuracy constraint and a model of the target architecture. It comprises several stages that are described below.

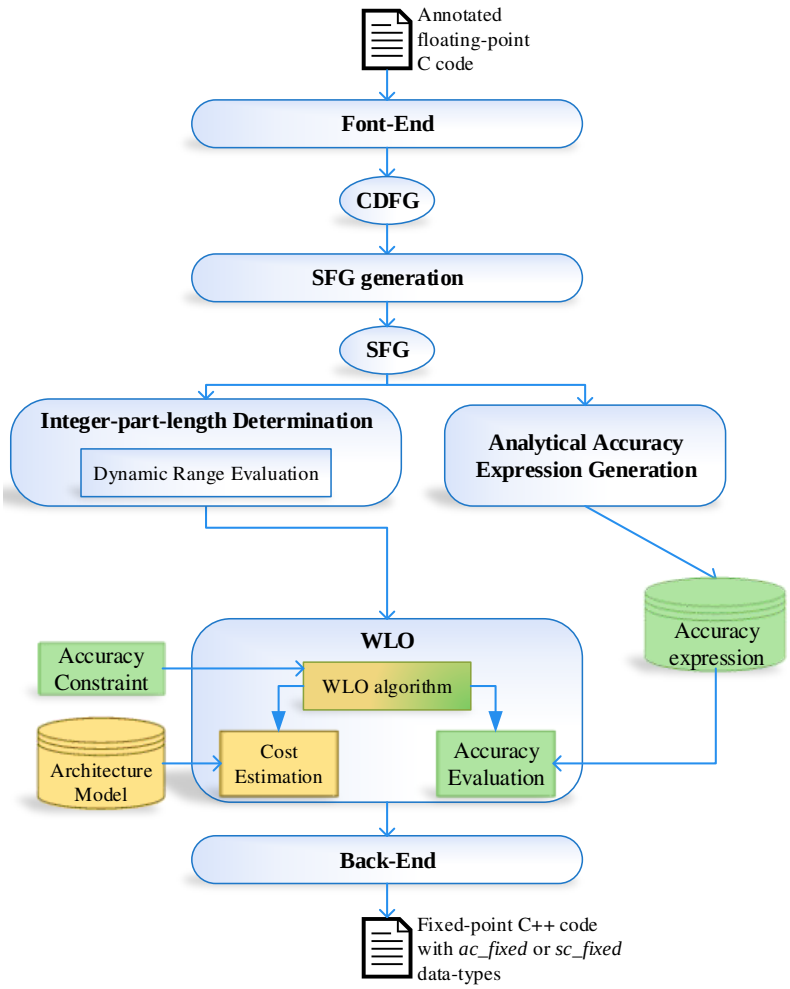


Figure 2.5 – IDFix flow diagram.

A Font-End parses the input floating-point C code and generate an Intermediate Representation (IR), which is basically a control and data flow graph (CDFG).

The input C code must satisfy several constraints, mainly:

— All control structures, such as loop bounds, must be statically evaluable.

— The system input variables must be annotated with a *pragma* annotation specifying their dynamic values range (`#pragma DYMANIC[min,max]`).

— The system outputs must be marked with a pragma annotation (`#pragma OUTPUT`).

— All variables used to represent a delay line[1] should be marked with a pragma annotation (`#pragma DELAY`).

1. a variable used to store previous values of a signal, often implemented using a circular buffer. For example, `X` represents a delay of the signal represented by `sample` in the following C code snippet: `int X[N]; X[0] = sample; ... for(int i=N-1; i>0; i--) X[i] = X[i-1];`

All control structures in the CDFG are then flattened, and loops are fully unrolled. The flattened CDFG is used to generate a SFG representing the computation system, which is later used to propagate dynamic range information and to compute the accuracy expression.

Dynamic Range Evaluation The user-specified input dynamic ranges are propagated in the SFG, using interval arithmetic, to infer the dynamic range of each variable and intermediate result (nodes). The dynamic range information is then used to determine, for each node, the minimum IWL such that no overflows can occur.

Accuracy Expression Generation The analytical accuracy expression of the system is generated as function of all node FWLs and their associated quantization modes. This is based on quantization error modeling and analytical SQNR evaluation presented in [89, 87], which only support LTI systems (and non-recursive non-LTI systems). The generated accuracy expression is later used to evaluate the accuracy for a given fixed-point solution during WLO.

Word Length Optimization IDFix provides several WLO algorithms to explore the performance/accuracy tradeoff. It uses a modular and extensible design to allow for simple implementation of additional WLO algorithms.

The word-length solution space is constrained by the operators supported by the target architecture, described using a simple model, which specifies:

— For each supported operation type (add, mul, ...), the supported operators that can perform it. An operator is simply represented by the word-lengths of the operands on which it can operate. For example `ADD[32 32 32]`, represents an operator capable of adding two 32-bit operands to produce a 32-bit result.

— For each operator, a static cost representing either its execution time, its area, or its energy consumption.

A WLO aims at finding the best solution, according to a specified criteria, while maintaining the system accuracy within an "acceptable" limit, provided by the user. It uses the analytical accuracy expression, generated earlier, to evaluate the accuracy of a tested fixed-point solution.

Cost Estimation based on the specified cost criteria (execution time, area, or energy consumption), the cost of a given fixed-point solution is automatically estimated by adding the cost of the operators assigned to each operation in the system. The operator cost is retrieved from the architecture model.

Back-End generates a fixed-point C++ code, corresponding to the fixed-point specification resulting from the WLO, using either SystemC or Algorithmic C Datatypes C++ libraries targeting HLS tools like Catapult C. The fixed-point code is simply obtained by changing the data-types

of all floating-point variable to the corresponding fixed-point class. Operations, scalings, saturations and quantizations are seamlessly handled by the underlying C++ library using operators overloading.

2.7 Conclusion

In this chapter, we have presented and compared floating-point and fixed-point representations. We have also discussed existing methodologies for floating-point to fixed-point conversion.

In summary, we have observed that:

— Many embedded processors do not provide hardware support for floating-point arithmetic.

— Floating-point software-emulation greatly degrades performance.

— Fixed-point is more suitable in this case; it provides a better performance at the cost of degrading computation accuracy.

— C++ fixed-point libraries are seamless to use, but they can introduce an overhead. Best performance can be achieved using only native integer data-type word-lengths.

— Floating-point to fixed-point conversion can exploit this performance/accuracy tradeoff through WLO.

— Automated conversion mainly requires a method for evaluating dynamic ranges and for evaluating the accuracy of a given fixed-point implementation.

— Simulation based methods are very slow, which make them not suitable for design space exploration.

Based on these conclusions, we decided to implement automatic floating-point to fixed-point conversion, in the context of the ALMA project, at source code level producing a fixed-point C code using only native integer data-types. We decided to use IDFix, since it already provides a framework for analytical accuracy evaluation and dynamic range evaluation. In addition, it is already integrated into a source-to-source compilation framework, namely GeCoS.

Later on, in chapter 5, we will present the source-to-source compilation flow that we implemented for floating-point to fixed-point based on IDFix. We will use this flow in conjunction with an automatic Simdization flow, which we will present in chapter 4, to explore the interaction between floating-point to fixed-point conversion and Simdization and their impact on performance/accuracy when targeting embedded processors. But first, in chapter 3, we will discuss another important performance-impacting factor when targeting embedded processors, namely Single Instruction Multiple Data (SIMD).

Chapter 3

Background: Single Instruction Multiple Data

Contents

3.1 Introduction 30
3.2 SIMD Instruction Sets 31
3.2.1 Conventional Vector Processors 31
3.2.2 Multimedia extensions 32
3.3 Exploiting Multimedia extensions 36
3.3.1 Manual Simdization 37
3.3.2 Automatic Simdization Methodologies 38
3.4 Conclusion 42

3.1 Introduction

In the previous chapter we presented the floating-point and fixed-point representations of real numbers. We discussed their impact on performance and accuracy, in the context of embedded processors. We concluded that fixed-point is more suitable for many embedded processors, specially in the context of the ALMA project. So, we explored existing methodologies for converting floating-point applications into fixed-point. In this context (as explained in section 1.3), another important performance-impacting factor is Single Instruction Multiple Data (SIMD), which we will discuss throughout this chapter.

Ever since the first transistor computer, manufacturing technologies continue to push miniaturization to its limit allowing more and more transistors to fit into a single die [116]. A big share of this growth has been invested in providing parallelism at various levels. In 1966, Flynn distinguished four classes of architectures according to the form and the grain of parallelism they provide. SIMD is one of them. It is the class of architectures capable of executing the same operation on several data simultaneously, thus providing a form of Data Level Parallelism (DLP). Figure 3.1 shows an illustration of a SIMD addition in contrast to its equivalent scalar operations. The SIMD operation performs the addition of all N data elements simultaneously, whereas N scalar additions are needed to perform the same operation.

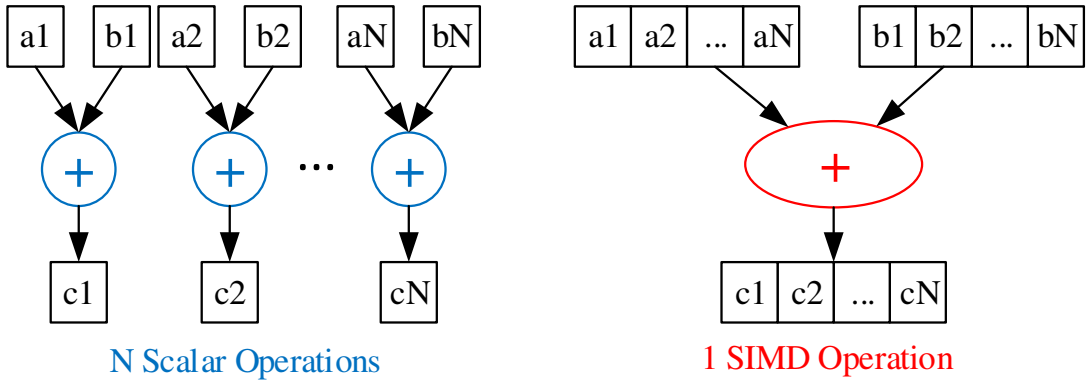


Figure 3.1 – Illustration of an SIMD addition operation in contrast to scalar addition operation.

Earliest form of SIMD architectures appeared in vector processors, used in supercomputers like the TI ASC and Cray-1 [115] in the 1970s. These supercomputers mainly targeted computation-intense scientific applications. With the increasing interest in multimedia applications for general public, SIMD architectures found their way into mainstream desktop processors, in mid 1990s, as an economical yet effective way to boost performance for such data parallel applications [72, 15]. These newer SIMD architectures are commonly known as Multimedia extensions or SIMD extensions.

The lack of native support for SIMD operations in common programming languages, in addition to the irregularities between different Multimedia Instruction Sets (MISs), made it very challenging to take full advantage of parallelism provided by Multimedia extensions. Program-

mers had to explicit SIMD parallelism in the application code, which is tedious and error prone. The natural cure to this problem is to rely on compilers to automatically extract and generate SIMD code.

Alternative solutions, such as supporting SIMD parallelism in programming languages, has been also investigated in order to efficiently leverage Multimedia extensions. However, a huge base of legacy applications, coded with sequential (scalar) languages like C and Fortran (prior to Fortran90), has already been established. This fact makes such solutions less attractive.

In this chapter, we present the characteristics of some SIMD instruction sets, as well as the different techniques of exploiting them. In section 3.2, we present an overview of the evolution of SIMD instruction sets. We then discuss the different ways of exploiting them in section 3.3.

3.2 SIMD Instruction Sets

In this section we present an overview of the evolution of most common SIMD instruction sets. In section 3.2.1, we briefly describe conventional vector processor architectures. We then present the evolution of Multimedia extensions in section 3.2.2.

3.2.1 Conventional Vector Processors

Vector processors are among the first models of SIMD architectures. Development of such processors started in the 1960s. They were mainly used in supercomputers, like the Texas Instruments (TI) Advanced Scientific Computer (ASC) and Cray-1 [115], targeting compute-intensive scientific applications. These processors support the execution of scalar and vector instructions. They have the ability to perform the same operation on several, say N, data elements, called vectors, by issuing a single vector instruction compared to N scalar instructions. Using vector instructions requires less instruction bandwidth, reduces loop control operations and reduces code size compared to scalar instructions.

Earlier vector processors, such as CDC's [1] STAR-100 and TI's ASC, used a memory-to-memory architecture. In such architectures, functional units operate on data directly from memory and write the results back to memory. The high latency associated with memory accesses is a limiting factor of such architectures.

Since Cray-1, this limitation is overcome by employing register-to-register architectures. The operands are first loaded to vector registers, on which vector instructions are applied. When needed, the results are stored back to memory. In order to do so, several vector registers are added, in addition to scalar registers. A vector memory access unit handles the transfer of data between the memory and vector registers, while functional units operates on registers.

In contrast to memory-to-memory architectures, the vector length is limited by the vector registers size (generally of several Kbits). Though, it can be changed dynamically through a

1. Control Data Corporation

Vector Length Register. Furthermore, vector masking allows the conditional execution of a vector instruction on certain elements of the vector operands.

Such architectures generally use one or more pipelined functional units (FU). A vector instruction is fetched and decoded once and then executed on all vector elements sequentially, though in case of multiple FUs, multiple vector elements are processed in parallel. Also, thanks to vector chaining (also called bypassing) a vector instruction can start executing as soon as the first elements of its vector operand - resulting from a previous vector instruction - are ready instead of waiting for the previous instruction to finish processing the entire vector.

Besides, these architectures have no restrictions on vector memory accesses. Scatter/gather accesses are supported and the accessed data is not required to be aligned on the vector size.

All these features, allow vector processors to have a seamless, complete and efficient support for vector operations.

3.2.2 Multimedia extensions

Despite their advantages, vector processors were very expensive [2] and not as efficient executing scalar instructions compared to conventional microprocessors, which was a limiting factor (Amdahl's law).

For these reasons, vector processors lost interest in the 1990s. Few years later, due to the increasing popularity of multimedia applications, another form of SIMD instruction sets, known as Multimedia extensions or small-scale SIMD, found its way to mainstream (general-purpose) and embedded processors. The main goal of these extensions was to provide a performance boost when targeting multimedia applications, by supporting a MIS, with little cost overhead.

Contrary to conventional vector processors, Multimedia extensions are based on a partitioned data-path that simultaneously operates on all vector elements. The vector representation in Multimedia extensions is different than in vector processors:

— Vector size is fixed and generally small (between 32 and 256 bits).

— Multiple vector element sizes are supported (generally 8, 16 and 32-bit).

— Vector elements must be packed into a register. Some Multimedia extensions use dedicated SIMD registers while others make use of the same scalar registers.

This means that for smaller element sizes, more elements can be processed simultaneously. Though, the number of cycles required to execute a SIMD instruction on a Multimedia extension is the same regardless of the number of elements and their size (generally the same as its scalar counterpart). This is generally not the case for conventional vector processors, where the maximum number of vector elements does not change with element size (the elements are not packed) and the number of cycles required to execute a vector instruction vary according to the number of elements. Consequently, when targeting Multimedia extensions the program must handles data packing and unpacking.

2. A Cray-1 supercomputer was sold for few million dollars.

Besides, the main difference between vector processors and Multimedia extensions is in the memory unit. Unlike vector processors, most Multimedia extensions do not support scatter/-gather memory accesses and require data to be contiguous and aligned to the vector size, in order to be directly loaded into a vector register. Otherwise, multiple memory accesses have to be performed and data have to be rearranged in vector registers. Multimedia extensions provide data permutation operations to allow the reordering of vector elements in registers. This, furthermore complicates the programming of Multimedia extensions.

MISs are designed to suit multimedia application needs. Since multimedia application domains are numerous and sometimes have different requirements, various MISs have been introduced by processor manufacturers with differences in vector size, supported element data-types and supported SIMD operations. Choosing among these criteria is generally a tradeoff between the coverage of the targeted application domain(s), the desired performance enhancement and the associated cost overhead.

Multimedia extensions have been constantly evolving ever since they were introduced. The vector size tends to increase with time, from 32-bit to 512-bit nowadays, and is expected to continue growing. Supported data-types are generally common powers of 2, starting from 8-bit for integers, in addition to single, double and, recently, half precision floating-point. The SIMD instructions set, also become richer with time by including more specialized instructions, composite instructions and/or horizontal SIMD instructions. All this led to many irregularities and variations between different MISs, which make them even harder to program.

All in all, Multimedia extensions generally suffer from various shortcomings that the programmer should handle:

— Vector elements packing and unpacking.

— Data permutations to handle unsupported memory operations (scatter/gather and alignment).

— Variations between different MISs.

In the remainder of this section we present an overview of the evolution of Multimedia extensions. We distinguish two main categories, *Subword* Multimedia extensions and *Superword* Multimedia extensions.

3.2.2.1 Subword Multimedia extensions

First Multimedia extensions were achieved by partitioning the existing scalar data-path, usually of 32-bit or 64-bit wide, into several smaller parts. This gives the processor the ability to operate on registers as a whole (in case of scalar instructions) or as aggregates of several, usually 8, 16 or 32-bit wide, independent subwords. In such architectures, the registers used by both scalar and SIMD instructions are thus the same.

The data-path is partitioned by introducing logic to handle propagation across subword boundaries. The same functional unit can be configured, by the instruction, to execute an operation

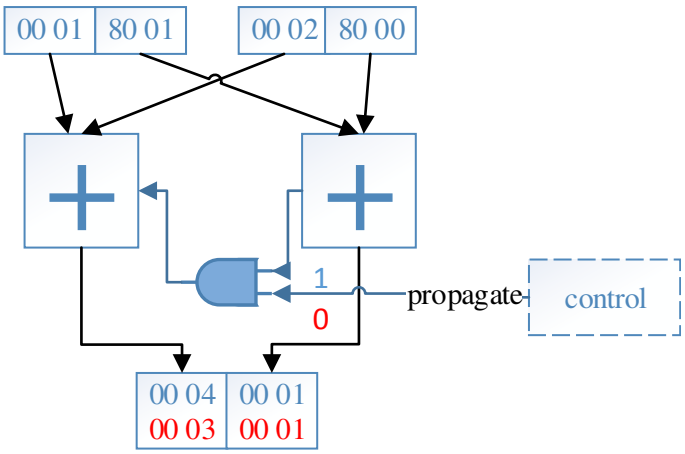


Figure 3.2 – Illustration of Sub-word Level Parallelism (SWP) addition operator capable of executing 32-bit scalar addition (when propagate is 1) or 2x16-bit SIMD additions (when propagate is 0).

between the entire words held in operand registers or between subwords independently. For instance, let us consider the unsigned addition operator in fig. 3.2. It can either perform the scalar 32-bit operation $0x18001 + 0x28000$ to get $0x40001$ as result, or the 2x16-bit SIMD operations $0x1 + 0x2$ and $0x8001 + 0x8000$ resulting in $0x3$ and $0x1$ respectively. Note that in the later case, the carry from the first addition is not propagated to the second one, the content of the destination register is $0x30001$.

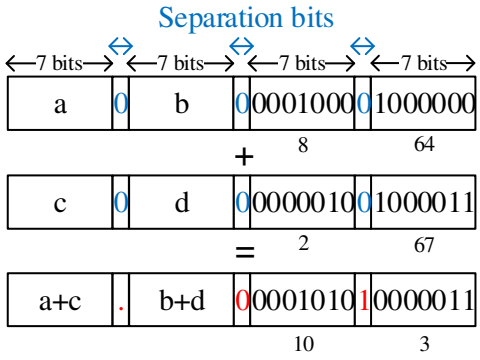
The Instruction Set Architecture (ISA) is extended by adding a MIS. It generally provides SIMD instructions for basic arithmetic operations (add, mul, ...) supporting different subword data-types. In addition, it also provides SIMD instructions for data packing, unpacking and permutation.

Such architectures provide small scale SIMD support, known as SWP, for very a modest cost overhead. For instance, MAX-1, the first such MIS proposed by Hawlett-Packard (HP) in 1994, provided real-time MPEG video decompression and occupied only 0.2 percent [72] of the PA-7100LC processor [57] die area. MAX-2 that was introduced 2 years later occupied even less than 0.1 percent [72] of the PA-8000 processor [66] die area.

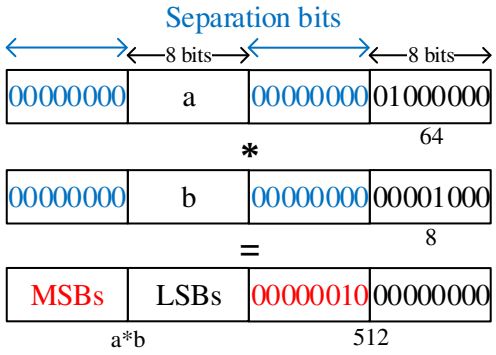
Soft SIMD To a certain extent, SWP can be exploited by software with no special hardware support, using but scalar instructions, this is known as Soft SIMD [37, 61].

Just like subword Multimedia extensions, this technique allows the execution of an operation on multiple subwords, packed into scalar registers, simultaneously. However, it can do this without any special hardware support by using only the existing scalar data-path. This basically consists of virtually partitioning the scalar data-path to perform SIMD operations by introducing separation bits between subwords in order to isolate them.

Figure 3.3a shows an example of a soft SIMD operation performing four 7-bit integer additions.



(a) Soft SIMD operation performing four 7-bit additions using scalar integer 32-bit adder. Operands are packed into 32-bit registers, each separated by one separation bit.



(b) Soft SIMD operation performing two 8-bit multiplications using scalar integer 32-bit multiplier. Operands are packed into 32-bit registers, each separated by eight separation bits.

Figure 3.3 – Illustration of some soft SIMD operations.

The operands of these additions are packed into 32-bit scalar registers with one separation bit between each two subwords. The potential overflow of one addition is caught by the separation bit on its Most Significant Bit (MSB) side, this way the result of one addition cannot pollute the result of the other. Since the addition of two 7-bit operands requires at most 8-bit to hold the result, only one separation bit is enough to avoid interferences. In case of multiplication, however, more separation bits are needed, as illustrated in the example of fig. 3.3b.

Soft SIMD requires careful management of separation bits in order to avoid subword operation interferences. This greatly limits the efficiency of such approach, especially in the case of chained operations, where the number of required separation bits rapidly grow. The more separation bits are needed the less subwords can be packed together. Nevertheless, soft SIMD can still be useful in some cases, especially when handling very small data sizes or irregular data-types (e.g. 10-bit) that are generally not supported by the hardware.

3.2.2.2 Superword Multimedia extensions

Later Multimedia extensions are based on special hardware extensions dedicated to support SIMD instructions. Similar to subword extensions, they are based on partitioned data-path. However this data-path is dedicated to SIMD and is generally separated from the scalar data-path. These extension units use special SIMD registers that are wider than the machine's native word size (hence the name superword). In contrast to subword, superword Multimedia extensions are more costly but offer higher vectorization factors. Also, since registers are not shared with the regular scalar data-path, register transfer instructions are generally added to move data between regular and SIMD registers.

One of the first such Multimedia extensions is Intel Matrix Math eXtension (MMX) [98] that equipped Pentium processors in 1997 providing support for 64-bit integer SIMD instructions

capable of operating on eight 8-bit, four 16-bit or two 32-bit packed data.

Ever since, most processor manufacturers continuously enhance and extend these MISs. SSE and later SSE2, 3 and 4 were successively proposed by Intel to support wider SIMD vector size, more subword data-types and/or more SIMD operations. As of today, Intel AVX-512 provides 512-bit SIMD support for all powers of 2 integer data-types starting from 8-bit, as well as single and double precision floating-point data-types.

Nowadays, superword Multimedia extensions are found in most general-purpose processors. They tend to have the same features as conventional vector processors by supporting wider vector sizes and less restricted memory accesses.

In low power embedded processors, it is more frequent to find support for SWP due to their near-zero cost and power consumption overhead. This is the case of many low power Very Long Instruction Word (VLIW) architectures such as Recore Systems Xentium[104], TI TMS320 C64x and STMicroelectronics ST240, as well as ARMv7 processors except those equipped with the NEON SIMD extension providing 128-bit SIMD support. In the context of this thesis, we mainly target this kind of architectures.

3.3 Exploiting Multimedia extensions

As mentioned in the previous section, the potential of SIMD architectures was recognized very early. Hardware support for SIMD has been provided through different architectures. Nowadays almost all processors, both general-purpose and embedded, provide a certain form of SIMD support through Multimedia extensions. In this section we discuss the different ways of exploiting the potential of these extensions to improve performance.

Since the hardware does not automatically make use of Multimedia extensions, the software must explicitly invoke SIMD instructions. However, most applications are generally coded in a scalar fashion [3]. Therefore, in order to make use of Multimedia extensions, a scalar code must be transformed to explicitly use SIMD instructions. We refer to this transformation as Simdization.

Simdization mainly requires two steps:

— First, exposing and uncovering SIMD parallelism in the scalar code. This generally requires code analysis and transformations, such as dependences analysis and loop transformations, to help exposing and enhancing SIMD parallelism.

— Converting groups of independent scalar instructions into equivalent SIMD instructions. This requires proper handling of data packing and unpacking taking into account memory access constraints.

In the following, we present the different ways of applying Simdization. We first discuss manual Simdization techniques, then we present various automatic Simdization methodologies.

3. using scalar instructions only. We will refer to this as scalar or sequential code.

3.3.1 Manual Simdization

When seeking maximal performance, hand-written optimized code is still the best bet. This is a difficult task, however. The programmer must have good knowledge of the target ISA, but also should have some skills in program transformations in order to efficiently expose and leverage parallelism provided by Multimedia extensions. This task is known to be tedious and error prone. Manual SIMD programming can be performed at various abstraction levels:

Inline Assembly

Inline assembly code can be used from within C/C++ code to explicitly invoke SIMD instructions. This is however very tedious and error prone. Furthermore, it is not portable since it is architecture-dependent. However, it allows fine-tuning to achieve best performance.

Intrinsics

Most SIMD processors provide a set of compiler-known intrinsics to invoke SIMD instructions. This allows a very similar expressibility as inline assembly programming, but has the advantage to be simpler, since it provides a function-like Application Programming Interface (API). However, it still mostly architecture-dependent and lacks portability.

SIMD Libraries

Another alternative consists of using pre-optimized libraries that implement certain functionalities using highly optimized SIMD code. This solution is very easy to use and more portable, but it is limited to few functionalities. It generally allows to achieve good performance, however, since the compiler is not aware of the optimized functions code, it cannot perform certain (inter-procedural) analysis and optimizations.

SIMD Programming Languages

Many high-level programming languages, with native support for vector operations, have been proposed as a mean to ease the use of SIMD hardware. Fortran 8x, later known as Fortran 90, C* [111] are early examples of such languages, which were introduced in the 80s to simplify programming vector processors. More recent languages, like Cilk Plus and SIMD directives of OpenMP, are more suitable for targeting Multimedia extensions.

Although SIMD programming languages provide a clean, efficient and portable way to target Multimedia extensions, they still require additional effort to figure out parallelism and express it correctly when designing the application, but most importantly they are of no use for already existing software. One solution to overcome this limitation consists in using source-to-source compilers to convert legacy (scalar) code to a SIMD language. An example of this approach is the Fortran-to-Fortran8x compiler [10].

3.3.2 Automatic Simdization Methodologies

As mentioned previously, manual Simdization is generally tedious, hard to maintain and lacks portability. SIMD programming languages can simplify these problems, but they are helpless for legacy software. The ultimate solution consists in relying on compilers to automatically detect and expose SIMD opportunities and efficiently generate SIMD code.

We can distinguish two main methodologies of automatic Simdization which emerged in harmony with the evolution of SIMD hardware architectures. The first, rather known as *Loop Vectorization*, was first introduced to target vector processors. It has later been adapted to target Multimedia extensions (mid 90s). And the second, basic-block level Simdization, appeared in 2000s as an alternative efficient way to target Multimedia extensions.

3.3.2.1 Loop Vectorization

Loop vectorization is a Simdization code transformation targeting loop nests. It aims at exploiting vector parallelism, a subset of loop-level parallelism. It basically consists of converting a "vectorizable" scalar loop into equivalent loop with vector instructions. A vectorizable loop is a loop that does not carry dependencies between its iterations (i.e. its body instruction instances can be executed in parallel) and that has a number of iterations equal to the vector size. Such loops are rarely present in a code, thus loop transformations allowing to exhibit them constitute a key part of loop vectorization techniques.

An auto-vectorizing compiler relies on instance-wise dependency analysis and loop transformations, such as loop interchange, fission, fusion, shifting, skewing and strip-mining to name a few, in order to detect and expose vectorizable loops.

Figure 3.4 illustrates a simple example of loop vectorization. The loop in the original code (fig. 3.4a) is dependence-free, but it has N iterations which may be different from the vector size V. Applying loop peeling and strip-mining in this case allows to expose a vectorizable loop (fig. 3.4b), which can be then replaced by equivalent vector code (fig. 3.4c).

First attempts of auto-vectorizing compilers were targeting vector processors, like Cray-1's Fortran compiler (CFT). The quality of such compilers is greatly limited by their ability to perform precise dependence analysis and advanced loop transformations. Padua and Wolfe highlight the importance of data dependence analysis in automatic loop parallelization compilers [96]. Nobayashi and Eoyang published a study [92] comparing auto-vectorizing compilers targeting conventional vector processors in 1989 and outlined the importance of loop transformations to expose vectorizable loops.

In this context, the polyhedral model is well fitted to provide such services. Indeed, the polyhedral model offers an instance-wise abstract representation of some compatible loop nests, known as Static Control Parts (SCoPs) or ACL (Affine Control Loop) [4]. This mathematical

4. Basically a loop nest in which all loop bounds, conditionals and array subscripts are affine functions of the surrounding loop iterators and parameters.

```
for (i = 0; i < N ; ++i)
   Y[i] = X[i] + C[i];
```

(a) Original code

```
//V: the vector size
for (ii = 0; ii < ⌊N/V⌋*V; ii += V)
   for(i = ii; i < ii+V; i++) //vectorizable
      Y[i] = X[i] + C[i];
for (; ii < N; ++ii)
   Y[i] = X[i] + C[i];
```

(b) After applying peeling and strip-mining

```
for (ii = 0; ii < ⌊N/V⌋*V; ii += V)
   //this is later replaced by SIMD instructions
   Y[ii:ii+V-1] = X[ii:ii+V-1] + C[ii:ii+V-1];
for (; ii < N; ++ii)
   Y[i] = X[i] + C[i];
```

(c) After applying vectorization. $[min : max]$ represents a vector access to contiguous array elements starting at offset min till offset max included.

Figure 3.4 – Example of Vectorization.

representation is used to derive a precise instance-wise and array element-wise dependence analysis [32], based on which the space of valid loop transformations (also known as schedules) can be explored. A loop transformation corresponds to a mathematical relation that reschedules the loop instances. It is valid if it does not violate any dependency [33, 34]. With the advance of polyhedral model techniques and tools, such as Pluto[19] and ISL [139], auto-vectorizing transformations become more robust, and easier to implement [60].

Most compilers nowadays, such as GNU GCC and LLVM, provide support for polyhedral transformations and automatic loop vectorization.

Adapting Loop Vectorization to Multimedia extensions

When Multimedia extensions emerged, due to their resemblance to vector processors, the trend was to adapt loop vectorization techniques in order to automatically generate SIMD code targeting Multimedia extensions. Indeed, the same loop analysis and transformations can be employed to expose vectorizable loops, which are then transformed into equivalent SIMD instructions from the target MIS.

However, unlike vector processors, MISs are irregular and vary significantly between different extensions. But most importantly, they do not provide a complete support for vector operations, especially memory accesses. This makes the conversion of vectorizable loops into SIMD code more complex as compared to vector processors, for which this conversion is straightforward. Fisher and Dietz discussed this problem and proposed to complement hardware SIMD instructions with software emulated ones [37] (soft SIMD) in order to have better support for vector operations.

On the other hand, vectors are represented differently in Multimedia extensions. Unlike vector processors, vector elements must be *packed* in SIMD registers in order to be processed by SIMD instructions. Thus, the compiler must be able to handle data packing and unpacking, in order

to load vector elements from memory and pack them into SIMD registers, and unpack elements to store them back into memory. This step is particularly sensitive especially in case of complex memory accesses (scatter/gather or unaligned accesses). Since most Multimedia extensions only support aligned [5] unit-stride [6] vector memory accesses, an important overhead is associated with the packing/unpacking operations, which are required in the case of complex memory accesses. Figure 3.5 illustrates an example of vector memory load for such Multimedia extensions. In case of unaligned (left) or non unit-stride memory access (right), additional operations are needed in order to pack vector elements into a register. This further complicates the task of auto

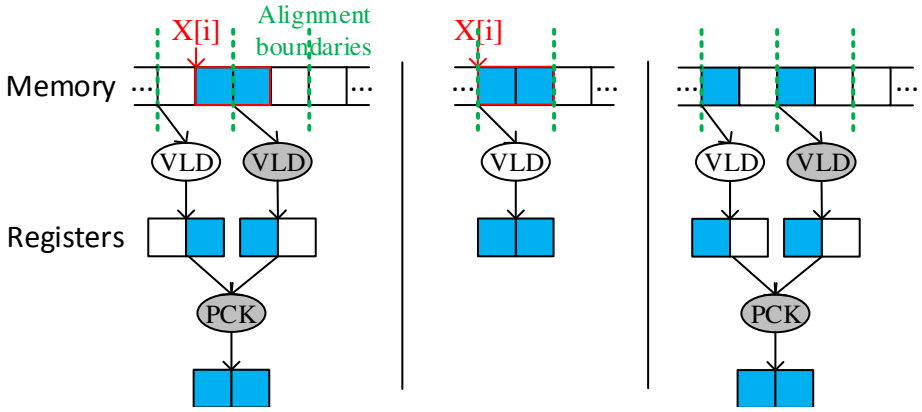


Figure 3.5 – Illustration of vector memory load from an aligned stride-1 reference (middle) versus unaligned stride-1 (left) and aligned stride-2 (right), in case of a Multimedia extension with no support for unaligned accesses and with vector size of 2.

vectorization. Additional code analysis and transformations are required in order to produce efficient SIMD code, such as memory reference alignment and pointer analysis [114, 101], handling of unaligned memory accesses [31, 141], data layout transformations [48, 44] and more advanced loop transformations [35, 60].

3.3.2.2 Basic-Block Level Simdization

Due to the complexity of loop vectorization techniques when targeting Multimedia extensions as discussed above, a new concept of Simdization emerged in the 2000s. It consists in exploiting SIMD parallelism at the basic-block level rather than loop-level.

In 2000, Leupers proposed a basic-block level Simdization technique [74]. It is performed at instruction selection stage in two main steps. First, the Data Flow Graph (DFG) representing a basic block is partitioned into Data Flow Trees (DFT), each of which is covered using tree pattern matching. Multiple alternative covers are permitted including sub-SIMD covers which represent a sub-operation (or element) in a SIMD instruction. The second step then uses integer linear programming to determine the best cover for the DFG while aiming at minimizing the

5. Access to memory at address aligned to the SIMD registers size.
6. Access to contiguous elements in memory (stride = 1).

implementation cost. Tree covers can be merged to form SIMD covers. This technique mainly lacks scalability with respect to the basic block size and SIMD data-path size. Also, this method does not consider data transfer, which often prevent the exploitation of SIMD instructions.

Tanaka et al proposed an extension [131] to the aforementioned work, taking data transfer operations into account.

Larsen and Amarasinghe proposed a new method for exploiting SIMD at the basic-block level, they referred to it as Superword Level Parallelism (SLP) [69]. They define SLP as a new type of parallelism, distinguishing it from conventional vector parallelism. SLP aims at finding in a basic block, similar independent statements that can be packed together and replaced by SIMD instructions. Thus, in contrast to loop vectorization, SLP does not have to deal with loop carried dependencies or employ sophisticated loop transformations to uncover parallelism. Hence, it is simpler to apply. Although they are not necessary, loop transformations can still be used to convert loop-level parallelism into SLP (using unroll-and-jam or register level tiling [49, 108] for example).

In fact, SLP is a superset of vector parallelism. The latter can only pack instances of the same statement in a vectorizable loop (inter-loop instances parallelism), whereas SLP can pack different statements in a basic block, which can be part of a loop body (intra-loop parallelism). SLP is, hence, able to exploit a finer level of parallelism, since loop-level parallelism is easily amenable to intra-loop parallelism by mean of loop transformations such as loop unrolling. This allows SLP to find more and/or better opportunities when targeting Multimedia extensions.

Note that SLP extraction techniques are applicable to Subword Multimedia extensions as well, since subword parallelism is actually a special case of SLP.

To illustrate SLP let us consider the example in fig. 3.6, targeting a 64-bit wide Multimedia extension supporting 32-bit integer data type as vector elements. The loop in the original code (fig. 3.6a) does not carry any dependency, thus all its iterations can be executed in parallel. The loop is partially unrolled by a factor of 2 (fig. 3.6b).

Using SLP, we can either:

— Pack statements $S0$ and $S1$ in the loop body as illustrated in fig. 3.6c.

— Or pack consecutive instances of each statement separately, after applying loop unrolling by a factor 2, as illustrated in fig. 3.6d.

The first solution is likely to give better performance than the second one since it requires less packing/unpacking operations (assuming vector memory accesses are aligned).

Using loop vectorization, only the second solution is possible.

As mentioned earlier, when targeting Multimedia extensions, the compiler must deal with data packing/unpacking and memory access alignment, which can represent a significant overhead hindering the Simdization benefits. This is not different in case of SLP. In fact, this is the main challenge for SLP extraction algorithms, which attempt to address it by adopting heuristics promoting the packing of statements that lead to less packing/unpacking overhead. The state-

```
   for (i = 0; i < N ; i++) {
S0:   Y[2i] = X[2i] + A[i];
S1:   Y[2i+1] = X[2i+1] + B[i];
   }
```

(a) Original code.

```
   for (i = 0; i < N ; i+=2) {
   //i
S0:   Y[2i] = X[2i] + A[i];
S1:   Y[2i+1] = X[2i+1] + B[i];
   //i+1
S0:   Y[2i+2] = X[2i+2] + A[i+1];
S1:   Y[2i+3] = X[2i+3] + B[i+1];
   }
```

(b) 2x unroll.

```
   for (i = 0; i < N ; i+=2) {
S0S1: Y[2i:2i+1] = X[2i:2i+1] + <A[i],B[i]>; //i
S0S1: Y[2i+2:2i+3] = X[2i+2:2i+3] + <A[i+1],B[i+1]>; //i+1
   }
```

(c) Packing $S0_i$ and $S1_i$ together. Note that the unrolling is not necessary in this case

```
  for (i = 0; i < N ; i+=2) {
S0:   <Y[2i],Y[2i+2]> = <X[2i],X[2i+2]> + A[i:i+1]; // <i,i+1>
S1:   <Y[2i+1],Y[2i+3]> = <X[2i+1],X[2i+3]> + B[i:i+1]; // <i,i+1>
   }
```

(d) Packing two consecutive instances of $S0$ ($S0_i$ and $S0_{i+1}$) and $S1$.

Figure 3.6 – Example illustrating the difference between loop vectorization and SLP. $[min : max]$ represents a vector memory access to consecutive array elements starting at offset min till offset max included. $< a, b >$ represents a packing/unpacking operation.

of-the-art of SLP is discussed later in chapter 4.

3.4 Conclusion

In this chapter we have presented the characteristics of some SIMD instruction sets and we have discussed the existing techniques for exploiting them.

In summary, we can observe that:

— Most embedded processors provide SIMD capabilities, mainly as subword Multimedia extensions.

— Simdization can be performed either at loop-level using loop vectorization techniques, or at the basic-block level using SLP extraction techniques.

— SLP can exploit more parallelism than loop vectorization. The latter can be amenable to SLP using simple loop transformations.

— SLP extraction is simpler since is it does not require advanced loop dependencies analysis and transformations to uncover parallelism.

Based, on these conclusions, we decided to implement SLP extraction to perform Simdization in the context of the ALMA project.

In the next chapter, we discuss the state-of-the-art of SLP and we propose an enhanced SLP extraction algorithm, which we then implement as a source-to-source transformation along with a SIMD C code generator, and compare it against a state-of-the-art algorithm.

Chapter 4

Superword Level Parallelism

Contents

4.1 Introduction 46
4.2 Related work 47
4.2.1 `Original SLP` 48
4.2.2 `Original SLP` Extensions and Enhancements 49
4.3 `Holistic SLP` 50
4.3.1 Overview 51
4.3.2 Shortcomings 59
4.4 Proposed `Holistic SLP` Algorithm Enhancements 64
4.4.1 Pack Flow and Conflict Graph (PFCG) 65
4.4.2 Group Selection Procedure 66
4.4.3 Algorithm complexity 72
4.4.4 Pseudo-code 74
4.5 Experimental Setup and Results 78
4.5.1 Implementation 78
4.5.2 Target processors 79
4.5.3 Benchmarks 81
4.5.4 Tests Setup 81
4.5.5 Results 82
4.6 Conclusion 88

4.1 Introduction

In the context of the ALMA project, as presented in sections 1.2 and 1.3, we needed to implement an automatic Simdization in order to exploit the Single Instruction Multiple Data (SIMD) capabilities of the targeted embedded processors, namely XENTIUM [104] provided by RECORE SYSTEMS and KAHRISMA [58] provided by KARLSRUHE INSTITUTE OF TECHNOLOGY (KIT). In fact, both embedded processors provide support for integer subword SIMD operations. They support 2x16-bit operations, in addition KAHRISMA supports 4x8-bit operations. However, none of the back-end compilers provided for each target implements a automatic Simdization optimization. Instead, they provide a set of intrinsics to access SIMD instructions. This is also the case of many other embedded processors, such as ST240 [4].

As we discussed in the previous chapter, performing Simdization manually using intrinsics is a tedious, error prone and time consuming task. Therefore, one of the goals of the ALMA project is to automate this transformation. This can be achieved by integrating an automatic Simdization transformation into the back-end compiler of each one of the target processors. However, this is not a practical solution since it should be done for each new target processor to be supported. In order to avoid this, we decided to implement Simdization at source code level using a source-to-source compilation flow capable of generating a C code with SIMD intrinsics. In the previous chapter, we discussed different methodologies for performing Simdization, mainly loop vectorization and Superword Level Parallelism (SLP). We decided to use SLP since it does not require advanced loop dependencies analysis and transformations, in addition it can exploit more parallelism.

SLP concept was first defined and presented by LARSEN and AMARASINGHE in 2000 [69]. In the same work they also presented an algorithm allowing to extract SLP in a given basic-block. In the remainder of this chapter we will refer to this algorithm as `original SLP`. The statements grouping strategy used in `original SLP` aims at reducing data packing/unpacking cost. However, it is based on a local heuristic ignoring some packing possibilities. A more detailed analysis of `original SLP` is presented later on in section 4.2.1. Several work [124, 125, 36, 46] aimed at enhancing `original SLP`, we present some of them in section 4.2.2.

Recently a new SLP extraction algorithm was introduced by LIU et al in 2012 [78] to address the shortcoming of `original SLP`. Unlike `original SLP`, the statements grouping strategy they propose is based on a *global* estimation of the data packing/unpacking cost, taking into account all (size-two) packing possibilities in the basic-block. We will refer to this method as `holistic SLP` and we will discuss it thoroughly later on in section 4.3.

Based on the advantage of `holistic SLP` over `original SLP` [1], we decided to go with `holistic SLP` for implementing Simdization in the ALMA flow. However, after a thorough analysis of the `holistic SLP` algorithm we discovered several shortcomings. In section 4.3.2, we present an elaborate explanation of these shortcomings. In short, we noticed that `holistic SLP` yields

1. experimental results presented in [78], shows a performance improvement as high as 15% over `original SLP`

poor performance without applying data layout transformation, mainly due to the grouping candidate benefit estimation method they use. In fact, data layout transformations are not always applicable (the method they propose is only applicable in the context of the polyhedral model), and they increase the memory footprint (due to arrays replication). This is not suitable when targeting embedded platforms with limited memory resources, such is the case in the context of ALMA. Furthermore, the Intermediate Representation (IR) they propose in order to represent SLP grouping solutions, is bulky and does not provide necessary information for rapid estimation of the amount of reuse between different statement grouping candidates.

In section 4.4, we propose several improvements to solve these problems. More specifically, we propose:

— A new IR for SLP extraction capturing grouping opportunities with conflicts and superword reuse information. It is more expressive and compact compared to `holistic SLP`.

— A new grouping candidate benefit estimation together with a new group selection method, allowing to achieve up to 40% performance improvement compared to `holistic SLP` without the need for data layout transformations.

Finally, in section 4.5 we present the implementation and some experimental results comparing our proposed SLP extraction algorithm against `holistic SLP` on a set of nine benchmarks targeting four different embedded processors.

4.2 Related work

In this section we discuss the state-of-the-art of SLP algorithms. We start by presenting `original SLP`, we highlight its shortcomings and then we present an overview of the work improving them.

```
S1:   v1 = a + b;
S2:   v2 = c + d;
S3:   v3 = e + v2;
S4:   v4 = v3 - b;
S5:   v5 = v1 - d;
```

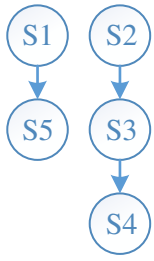


Figure 4.1 – Example C code snippet (left) and its statements dependence graph (right).

Définition 4.1 (Statement) *A statement is a root instruction in a basic block. It contains one or more operations. Two statements are isomorphic if they both contain the same operations in the same order.*

Example 1 *$S1, S2, ..., S5$ in the C code example of fig. 4.1 are all statements. $S1, S2$ and $S3$ are isomorphic statements, and so are $S4$ and $S5$. Statement $S3$ depends on $S2$.*

Définition 4.2 (Superword Statement) *A superword statement is a list of isomorphic independent statements that can be implemented using one or more SIMD instructions. Superword statement's operands are superwords (or vectors).*

4.2.1 `Original SLP`

`Original SLP` algorithm aims at extracting SIMD parallelism at basic block level. Given a list of a basic block's statements, it aims to combine several isomorphic, independent statements into superword statements. This is known as statements packing or grouping. The superword statements are then implemented using SIMD instructions.

Statements packing is a combinatorial optimization problem. Using exhaustive search is often not realistically feasible. Therefore, `original SLP`, and all other SLP extraction algorithms for this matter, use a heuristic to find a satisfying solution within a reasonable time.

The heuristic employed by `original SLP` exploits two observations:

— Accesses to aligned contiguous memory references is efficiently implemented in Multimedia extensions. N individual memory access operations can be performed using one SIMD access which transfers packed data directly to/from a SIMD register with no need for additional packing/unpacking or shuffling operations. This roughly saves $N-1$ memory accesses along with their address calculation operations.

— No packing/unpacking or shuffling operation is needed when a SIMD instruction's operand has been produced by another SIMD instruction since it is already packed in a SIMD register (assuming enough registers are available).

This heuristic consists of starting by packing statements that contain contiguous memory accesses. And then find additional statement packs by following the def-use and use-def chains of the already formed ones. At first, only groups of two statements are packed together (to reduce the search space) and then multiple groups are merged to obtain larger superword statements to better utilize the SIMD data-path.
This imposes that each superword statement contains either a vector access to contiguous region in memory or/and reuses superwords resulting from other superword statements directly, which minimizes the need for overhead packing/unpacking and permutation operations. But still, some packed statements might have an important overhead that can degrade performance. To avoid such cases, a cost model is used during statements packing process. It estimates the speedup of each potential packed statements and discards those with negative effects. This estimation takes into account the necessary packing/unpacking operations as well as reuses of already existing superwords. It is also used to select between conflicting statements packing possibilities.

This heuristic aims at minimizing the overhead of packing/unpacking operations by maximizing superwords reuse, starting from and/or leading to, in-memory packed and aligned superwords. Statement packs are selected based on their superword reuse with previously decided statement packs, leading to a local optimization that is strongly affected by the previously made packing decisions. Furthermore, it completely ignores statement packing possibilities which are not in a

superword reuse chain coming from, or leading to, an in-memory packed superword.

This main shortcoming has been identified and addressed by Liu et al who recently proposed an alternative SLP statements packing heuristic [78] based on a global, or holistic, superword reuse estimation. This work is thoroughly discussed later in section 4.3.

4.2.2 `Original SLP` Extensions and Enhancements

Shin et al introduced superword-level locality [124, 122], a technique allowing the use of SIMD register file as a compiler-controlled local memory. They propose superword replacement transformation, which substitutes vector memory accesses to arrays by accesses to superword temporaries allocated in SIMD registers. In addition, they employ unroll-and-jam loop transformation to enhance locality and shorten superword reuse distance, along with index set splitting to increase the amount of aligned memory accesses.

Besides, they also proposed an extension of `original SLP` allowing it to be applied beyond simple basic block, in the presence of control flow structures [125, 122]. To do so, they perform if-conversion prior to extracting SLP in order to convert control structures into predicated instructions. After superword packing, they restore control structures to be able to target Multimedia extensions with no support for predication.

Kudriavtsev and Kogge [63] proposed an alternative packing strategy to extract SLP. Unlike `original SLP`, it allows the exploration of superword reordering to efficiently generate and optimize permutation operations. It borrows from the work of Leupers [74] the fact of leveraging tree pattern matching to find covers for data flow trees – obtained by partitioning the basic block's Data Flow Graph (DFG) – and then find a cover for the entire DFG using SIMD instructions.
They start by grouping contiguous memory accesses to form SIMD groups [2], then they find further SIMD groups by traversing the DFG starting from memory operations appearing in SIMD groups. Then, in the DFG, they replace the nodes of each SIMD group by a node representing the group. Finally they try to find an order, for the elements of each SIMD group, that minimizes the cost. For doing so, they use order propagation starting from memory access SIMD groups, which have a fixed order, to obtain the set of possible permutations per SIMD group, then they use Integer Linear Programming to find the best ordering assignment per group.

Ren et al proposed a standalone algorithm to optimize data permutations [107] in a given basic block that can be used in complement to other Simdization transformations. First it converts all permutations in a source code into a generic representation. Then it tries to optimize the permutations inside each basic block before regenerating the corresponding target-dependent permutation instructions.

2. A SIMD group is a set of operations that can be implemented with a SIMD instruction.

4.3 `Holistic SLP`

The statements packing heuristic employed by `original SLP` aims at maximizing superword reuse by following def/use and use/def chains starting from, or leading to, in-memory packed superwords. However, it does not consider statement packing options unless they are in a superword reuse chain seeded by one or more unit-stride (and aligned) memory accesses pack. Also, it does not consider superword permutations (or shuffling). In fact, it only qualifies as superword reuse, superwords containing the same elements in the *same* order. This ignores the possibility of employing shuffling operations to reuse a superword containing the same elements but in a different order. Shuffling operations are generally much less costly then packing/unpacking operations.

This results in a local optimization since statement packs are selected based on their superword reuse with already formed statement packs only, and not across the whole basic block.

To address this shortcoming, Liu et al proposed an alternative statement packing heuristic [78] based on a global, or holistic, estimation of superword reuse across the entire basic block. We refer to it as `holistic SLP`.

In section 4.3.1 we present the `holistic SLP` extraction algorithm, we then analyze it and discuss several shortcomings in section 4.3.2, that we try to address in section 4.4.
But, first we start by giving several related definitions:

Définition 4.3 (SIMD Group Candidate) *A SIMD group candidate is an unordered pair (pair set) of isomorphic and independent statements, yielding superwords compatible with the SIMD data-path size. All elements in a superword must have the same size, that need to be supported by the target SIMD Instruction Set Architecture (ISA), and the total size should not exceed the SIMD vector size.*

Définition 4.4 (Variable Pack) *A Variable Pack (VP) is an unordered pair (pair set) of operands from same position of both statements in a SIMD group candidate. Two VPs are* equivalent *if they contain the same elements (regardless of their order). We use the symbol ≡ to mark equivalence.*

Définition 4.5 (Candidate Conflict) *Two candidates are considered in conflict if, they either share a common statement or there exist a cyclic dependency between them. We refer to the first case as* common statement conflict *and to the second as* cyclic dependency conflict.
We distinguish two types of cyclic dependency conflict:

- *Direct: occurs between two candidates only.*
- *Indirect: occurs between more than two candidates.*

Example 2 *Consider the example of fig. 4.1 with 32-bit data-types and targeting a 64-bit wide SIMD data-path.*

```
S1:   v1 = a + b;
S2:   v2 = c + d;
S3:   v3 = e + v2;
S4:   v4 = v3 - b;
S5:   v5 = v1 - d;
```

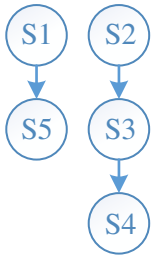


Figure 4.2 – Recall example of fig. 4.1.

— $\{S1, S2\}$ *is a SIMD group candidate, and so are* $\{S1, S3\}$ *and* $\{S4, S5\}$. $\{S2, S3\}$ *is not a candidate, though* $S2$ *and* $S3$ *are isomorphic.*

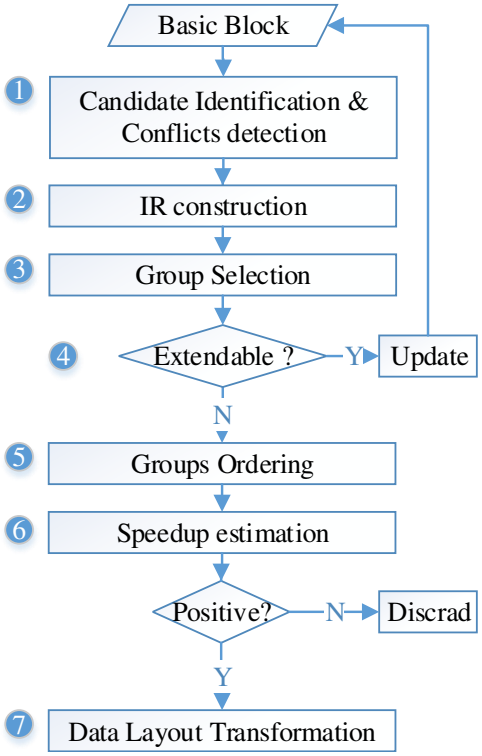
— *The VPs of the candidate* $\{S1, S3\}$ *are* $\{v1, v3\}$, $\{a, e\}$ *and* $\{b, v2\}$. *VPs* $\{v1, v3\}$ *from candidate* $\{S1, S3\}$, *denoted* $\{v1, v3\}_{\{S1,S3\}}$, *and* $\{v3, v1\}$ *from* $\{S4, S5\}$ *are equivalent i.e.* $\{v1, v3\}_{\{S1,S3\}} \equiv \{v3, v1\}_{\{S4,S5\}}$.

— *Candidates* $\{S1, S2\}$ *and* $\{S1, S3\}$ *have a common statement conflict. Figure 4.7 shows an indirect cyclic dependency conflict case.*

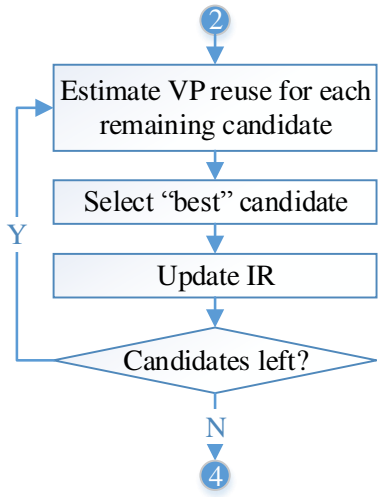
4.3.1 Overview

`Holistic SLP` extraction algorithm overview is depicted in fig. 4.3. It consists of several steps that we illustrate in this subsection based on its presentation by Liu et al in [78].

In a nutshell, `holistic SLP` aims at finding as much statement packs (or groups) as possible while maximizing superword reuse throughout the entire basic block. Exploiting the fact that, increasing superword reuse reduces the need for data packing/unpacking operations, which generally introduce an important overhead that might eclipse the potential benefit of SLP and degrade performance. To do so, it considers all statement grouping candidates available in the basic block (step 2), among them, it then selects the ones that are more likely to bring the most superword reuse and discard those conflicting with them (step 3).
Since considering all possible statement packing combinations, with different sizes and different statements order, yields a very high number of possibilities, `holistic SLP`, in fact, only considers statement groups of size two and ignores the order of statements inside a group. Such statements group is referred to as SIMD group candidate (definition 4.3) and represents two statement packs possibilities.
In order to extend the statement groups size beyond two and fully utilize the target SIMD datapath, the selected SIMD groups at step 3 are integrated in the basic block before starting a new iteration, as shown in step 4.
The order of statements inside each selected group is then specified, at step 5, with the objective of maximizing the direct superword reuse and hence minimizing the need for permutation operations.
At last, `holistic SLP` optionally applies a data layout transformation (step 7) to optimize superwords placement in memory, when possible, in order to minimize the overhead associated



(a) Algorithm overview.



(b) Group Selection procedure overview.

Figure 4.3 – `Holistic SLP` algorithm flow diagram.

with accessing them.

Step 1: SIMD Group Candidates and Conflicts Identification

This step aims at constructing the set of all possible SIMD group candidates as well as the set of all conflicts between them. SIMD group candidate and conflict definitions are given by definition 4.3 and definition 4.5 respectively.

Step 2: Intermediate Representation (IR) Construction

To capture all the information required by the algorithm, an intermediate representation is build. This representation consists of two undirected graphs:

— Statement Grouping Graph (SGG): It is a weighted undirected graph. Its nodes are the set of all statements present in at least one candidate. An edge represents a possible SIMD group candidate; so that two nodes are connected by an undirected weighted edge if they belong to the same candidate. The weight of each edge represents the benefit, described later on, associated with the selection of the corresponding candidate. All edge weights are computed in the next step.

— Variable Pack Conflict Graph (VPCG): This graph captures VP reuses and conflicts across all candidates. It is used to ease the computation of VP reuse benefit of each candidate as we will explain in the next step.
VPCG is an unweighted undirected graph. Its nodes are the variable packs (VPs), as defined by definition 4.4, from each SIMD group candidate. In order to distinguish equivalent VPs belonging to two different candidates, each VP node is tagged by the candidate to which it belongs. Two VP nodes are connected by an undirected edge in VPCG if their corresponding candidates conflict with each other.

Example 3 *In the ongoing example of fig. 4.1:*
The set of SIMD group candidates is:

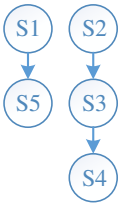
$$C = \{c1 = \{S1, S2\}, c2 = \{S1, S3\}, c3 = \{S4, S5\}\} \tag{4.1}$$

There is only one conflict, a common statement conflict in fact, between the candidates. The conflicts set is:

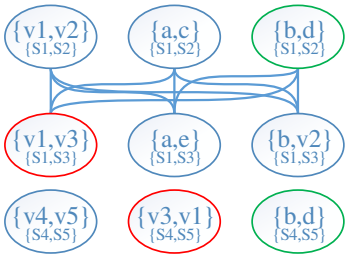
$$\{\{c1, c2\}\}$$

The corresponding VPCG and SGG are shown in fig. 4.4b and fig. 4.4c respectively.

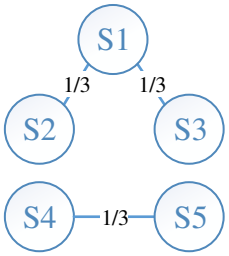
```
S1:   v1 = a + b;
S2:   v2 = c + d;
S3:   v3 = e + v2;
S4:   v4 = v3 - b;
S5:   v5 = v1 - d;
```



(a) Recall of example in fig. 4.1.



(b) Corresponding VPCG. Equivalent VP nodes are highlighted with same color.



(c) Corresponding SGG. The edge weights computation is described in Step 3.

Figure 4.4 – VPCG and SGG of the example in fig. 4.1.

Step 3: SIMD Group Candidate

The SGG constructed in step 2 represents all the possible SIMD group candidates, not all of which can coexist together since some conflict with others. The aim of this step is to select the most profitable (in term of VP reuse) conflict-free subset of candidates.

The selection procedure employed by `holistic SLP`, illustrated in fig. 4.3b, is based on a global (holistic) estimation of the amount of VP reuse each SIMD group candidate can bring to the entire basic block. It operates as follow:

— First, for each of the remaining group candidates (edges in SGG), VP reuse across the entire basic block is estimated assuming the candidate at hand get selected. This VP reuse estimation is attributed as weight to the corresponding edge in SGG. The method used for estimating VP reuse is presented later on.

— Then, the candidate whose selection brings the maximum VP reuse, that is the edge in SGG with highest weight, is selected to be part of the solution. In case multiple candidates yield the same highest VP reuse, only one of them is selected randomly.

— Finally, both SGG and VPCG are updated before starting the next selection iteration until no more candidates remain (SGG has no edges anymore). SGG is updated by removing both nodes belonging to the selected group candidate as well as all nodes/edges connected to any of them [3]. Similarly, VPCG is updated by removing all nodes belonging to the selected group candidate as well as all nodes/edges connected to any of them.

Variable Pack Reuse Estimation

The SIMD group selection process is iterative. In order to decide which candidate should be selected at a given iteration, all remaining candidates are evaluated. For each of them, the average VP reuse among the entire basic block is estimated. The aim is to find the candidate that, if selected, is more likely to bring the highest VP reuse across the whole basic block.
Since the actual VP reuse across the basic block cannot be computed before the selection process is completed and the final grouping solution is obtained, the authors of `holistic SLP` propose a heuristic to estimate this reuse. The intuition is to consider the context in which the tested candidate, c, get selected. In this context, the actual VP reuse among already selected groups can be computed, but this does not include the potential contribution of the remaining candidates (from which, some would be selected in the next selection iterations to become part of the final solution). In order to estimate the potential contribution of these candidates, a greedy heuristic is employed. It aims at finding a conflict free subset of the remaining candidates which have reuses with c. This is achieved by considering all VPs, from the remaining candidates, that are equivalent [4] to any of c's VPs. These VPs may be conflicting with each other, which means they cannot coexist in any possible grouping solution. Therefore, these conflicts are solved by, iteratively, eliminating the VP which has the most conflicts, until no conflicts remain. The number of remaining VPs represent an estimation of the amount of reuse of c's VPs by future selected candidate in case c get selected.
All in all, VP reuse among already established part of the grouping solution as well as remaining group candidates are considered, making this approach more *holistic* that `original SLP`.

To formally define the benefit evaluation of an SIMD group candidate, let:

3. SGG nodes connected to any of both nodes of the selected candidate are part of conflicting candidates.
4. Two VPs are equivalent if they contain the same elements (see definition 4.4). Equivalent VPs indicate a reuse.

— G be the set of already selected groups.

— C be the set of remaining candidates.

To estimate the benefit associated with $c \in C$, denoted $VPR(c)$, c is assumed to be selected. So:

— $G' = G \cup \{c\}$ is the set of selected groups.

— $C' = C \setminus (\{c\} \cup conf(c))$ is the set of remaining group candidates, if c get selected. Where $conf(c)$ is the set of candidates in C conflicting with c.

$VPR(c)$, as described in `holistic SLP` [78], is computed as follows:

$$VPR(c) = \frac{|V(G')| - |P(G')| + |R(c)|}{|P(G')|} \tag{4.2}$$

The definition of V, P and R is provided below, intuitively:

— $|P(G')|$ represents the number of packing/unpacking operations required by all groups in G'.

— $|V(G')| - |P(G')|$ represents the actual number of VP reuses between the selected groups in G'.

— $|R(c)|$ is the estimation of reuses of c's VPs by the remaining candidates (in C').

Where:

— V is a function that, given a set, S, of SIMD group candidates, returns the set of all their VPs, each candidate's VPs are distinguished from others by tagging them with the candidate to which they belong so that, equivalent [5] VPs from distinct candidates are distinguishable.

$$V(S) = \bigcup_{c \in S} \{v_c,\ v \in c\} \tag{4.3}$$

Where v_c is the variable pack v of the candidate c, tagged by c. So that, $v_{c1} \equiv v_{c2} \equiv v$ but $v_{c1} \neq v_{c2}$ i.e. $\{v_{c1}\} \cup \{v_{c2}\} = \{v_{c1}, v_{c2}\}$ (assuming $c1 \neq c2$).

— P is a function that, given a set, S, of SIMD group candidates, returns the set of VPs of all group candidates in S. Note that in contrast to V, the returned VPs are not tagged so that, equivalent VPs from different candidates are not distinguishable i.e. $\{v\ from\ c1\} \cup \{v\ from\ c2\} = \{v\}\}$

$$P(S) = \bigcup_{c \in S} \{v \in c\} \tag{4.4}$$

— $R(c)$ is a conflict-free subset of, the set of VPs of candidates in C' which are equivalent to (i.e. reuses) any of c's VPs. In `holistic SLP`, $R(c)$ is obtained by:

1. First, constructing an *auxiliary graph* (as they refer to it in `holistic SLP`), associated with c, denoted $AG(c)$. It is a subgraph of VPCG, built by extracting all nodes (and all edges connected to them) equivalent to, but not connected to those of c. $AG(c)$ nodes are therefore, all the tagged VPs from C' that are equivalent to any VP from c.

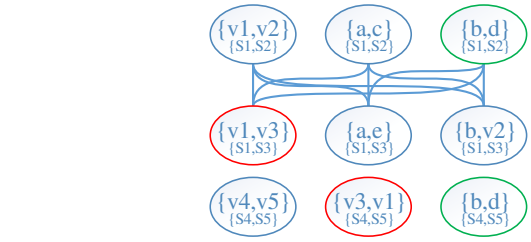
5. Two VPs are equivalent (denoted by $\equiv$) if they contain the same elements (definition 4.4).

2. And then eliminating all conflicts (edges) in $AG(c)$, by iteratively discarding a node with highest degree, that is a VP node which has the highest number of conflicts, until all conflicts are eliminated yielding $R(c)$.

Example 4 *To illustrate this step, let's consider the ongoing example of fig. 4.1.*

```
S1:   v1 = a + b;
S2:   v2 = c + d;
S3:   v3 = e + v2;
S4:   v4 = v3 - b;
S5:   v5 = v1 - d;
```

(a) Recall of example of fig. 4.1.



(b) Recall VPCG at first iteration.

Figure 4.5 – $AG(\{S4, S5\})$ at the first selection iteration of the example of fig. 4.1.

At the first selection iteration:

- $G = \emptyset$ *and* $C = \{c1 = \{S1, S2\}, c2 = \{S1, S3\}, c3 = \{S4, S5\}\}$
- *VPCG is shown in fig. 4.4b.*
- *In order to compute* $VPR(c3)$*:*
  - *Extract* $AG(c3)$ *from VPCG.* $AG(c3)$ *is depicted in fig. 4.5.*
  - *Eliminate conflicts(edges) from* $AG(c3)$*: iteratively remove one node with highest degree. Both nodes in* $AG(c3)$ *have degree of one, so removing any of them will do. Therefore, the number of remaining nodes* $ES = |AG(c3).\mathcal{V}| = 1$.
  - $V(G') = V(\{c3\}) = \{\{v4, v5\}_{c3}, \{v3, v1\}_{c3}, \{b, d\}_{c3}\}$. *Thus,* $|V(G')| = 3$.
  - $P(G') = P(\{c3\}) = \{\{v4, v5\}, \{v3, v1\}, \{b, d\}\}$. *Thus,* $|P(G')| = 3$.
  - *Hence,* $VPR(c3) = \dfrac{1 + 3 - 3}{3} = 1/3$.
- *Similarly,* $VPR(c1) = 1/3$ *and* $VPR(c2) = 1/3$. *The edges of SGG are weighted by the corresponding* VPR *as shown in fig. 4.4c.*
- *Since all candidates have same weight, one of them is selected randomly. Let's assume it's* $c3$ *that got selected.*
- *The updated VPCG and SGG are represented in fig. 4.6.*

At the second selection iteration:

- $G = \{c3\}$ *and* $C = \{c1, c2\}$
- *In order to compute* $VPR(c1)$*:*
  - $AG(c1)$ *is empty* $\Rightarrow ES = 0$.
  - $V(G') = V(\{c3, c1\}) = \{\{v4, v5\}_{c3}, \{v3, v1\}_{c3}, \{b, d\}_{c3}\, \{v1, v2\}_{c1}, \{a, c\}_{c1}, \{b, d\}_{c1}\} \Rightarrow |V(G')| = 6$.

(a) Updated VPCG. (b) Updated SGG.

Figure 4.6 – Updated VPCG (left) and SGG (right) after the selection of $\{S4, S5\}$ of the example of fig. 4.1.

- $P(G') = P(\{c3, c1\}) = \{\{v4, v5\}, \{v3, v1\}, \{b, d\}, \{v1, v2\}, \{a, c\}\} \Rightarrow |P(G')| = 5.$
- *Therefore* $VPR(c1) = \dfrac{0 + 6 - 5}{5} = 1/5.$

— *Similarly,* $VPR(c2) = 1/5$.

— *Again, both candidates have same weight, so one of them is selected randomly. Let's assume it's* $c2$.

— *SGG is then updated by removing* $c2$*'s nodes,* $S1$ *and* $S3$*, but also all nodes connected to them i.e* $S2$.

— *No more candidates are left, thus the selection procedure is done.*

Thus, the final set of selected groups is $\{c3, c1\}$.

Step 4: Groups Extension

This step aims at overcoming the size limitation, of only two statements per group, imposed by the first step. This allows the algorithm to efficiently target SIMD with wider data-path size.

Once SIMD group selection is done, if there is any group that can be extended, the basic block is updated by replacing each selected group's statements by a single new statement with new variables to represent its VPs (superwords), then the algorithm starts over at step 1 until no further grouping extension is possible.

Step 5: SIMD Groups Ordering

At this level, each selected SIMD group is a set of two or more, isomorphic independent, statements from the original basic block. The order of statements inside a group is not defined yet. So far, it has been assumed that, if two groups have an equivalent VP, it counts as a superword reuse regardless of the order of statements inside each group. In fact, this is not always true and a superword permutation/shuffling may be required in order for this reuse to happen. The cost overhead of these permutation operations may hinder the benefit of Simdization.
Hence, the aim of this step is to order statements inside each group in such a way that the number of necessary permutations is minimized. The ordered groups are then implemented as superword statements in the basic block.

Step 6: Speedup Estimation

At this level, the necessary SIMD instructions to implement the superword statements obtained from previous step, specially packing/unpacking and permutations, can now be accurately determined. A speedup estimation of the transformed basic block is performed and compared to that of the original scalar basic block. In case this indicates a performance degradation, the superword transformation is discarded and the basic block is left unchanged. The aim of this step is to make sure the generated superword solution will most likely perform better than the scalar one. However, this tightly depends on the cost model used for the target architecture.

Step 7: Data Layout Transformation

It is important to note that `holistic SLP` does not impose any constraints on packing array references. Consequently, the group selection step is likely to produce superwords with accesses to non-contiguous memory locations. Such superwords require costly packing operations which hinder the Simdization benefits, as illustrated in fig. 3.5.

To overcome this problem, `holistic SLP` optionally applies a data layout transformation to optimize superwords placement in memory, when possible, so that to minimize the overhead associated with accessing them.
The aim of this transformation is to minimize the cost of required packing/unpacking operations by trying to place the superword elements in contiguous (and aligned) memory locations, so they can be accessed using vector memory access operations without additional shuffling/insert/extract operations, as was shown in fig. 3.5.

This optimization targets two kind of superwords:

— Scalar superwords whose elements are all scalar variables.

— Array reference superwords whose elements are all array elements.

Scalar superwords are relatively simple to optimize. All is needed is to assign the superword elements consecutive memory locations. However, when two scalar superwords have elements in common and requires conflicting memory layouts, only one of them can be optimized.
The method employed in `holistic SLP` consists of optimizing the most reused scalar superword first and skipping all other conflicting superwords.

Optimizing array reference superwords, on the other hand, is more complex. In general, an element of such superword is a reference to a variable location in an array, usually determined by the surrounding loop iterators.

`Holistic SLP` leverages the polyhedral model and array replication to map elements of an array reference superword to contiguous memory locations in the transformed data layout. This transformation is only applicable to Static Control Parts (SCoPs) for superwords whose elements refer to the same, read-only, array.

4.3.2 Shortcomings

The `holistic SLP` algorithm, presented earlier in section 4.3.1 has several shortcomings. In this section we present and illustrate some of them. Later on, we propose some improvements to address them and further enhance the `holistic SLP` algorithm's efficiency.

4.3.2.1 Cyclic Dependencies

The group selection algorithm employed by `holistic SLP` as described in [78] does not consider cyclic dependency conflicts between SIMD group candidates. Thus, the obtained grouping solution is not guaranteed to be cyclic dependency free and might thus be illegal.
In fact, only common statement conflicts are considered when building the VPCG and SGG. That means none of them is carrying any information about cyclic dependency conflicts.
Actually, the described SGG model (see section 4.3.1) only carries information about common statement conflicts. A SGG node with degree higher than one represents a common statement between multiple (= degree of node) group candidates. Cyclic dependency conflicts, however, are not represented in this SGG model.
On the other hand, when SGG and VPCG are updated after the selection of a new SIMD group, only nodes representing the selected group and those connected to them are removed (see section 4.3.1). This only includes nodes representing candidates in common-statement-conflict with the selected candidate. In other words, nodes representing candidates in cyclic dependency conflict with the selected one remains in SGG and VPCG and may be selected in later iterations, thus, potentially introducing a cyclic dependency between the selected groups. Since the algorithm does not check for such cycles, the resulting grouping solution is not guaranteed to be legal and a valid schedule may not exist unless some selected groups are discarded!

Regardless, if we assume the lake of management of cyclic dependency conflicts in the algorithm they give is just for simplification (even though this was not stated). Then, given that no check for cycles is performed during group selection, the only way the final grouping solution will be guaranteed to be legal is by:

— Detecting all possible cyclic dependency conflicts before starting the selection process.

— And, represent all these conflicts in VPCG and SGG.

This solution is over conservative and prevent the selection of some candidates otherwise possible. For instance, in the aforementioned example of fig. 4.7a, if all cyclic dependency conflicts are considered, then the conflicts set is $\{\{c1, c2\}, \{c1, c3\}, \{c2, c3\}\}$. In this case only one candidate can be selected. However, if any two candidates are selected the solution would still be valid.

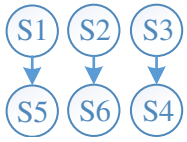
Example 5 *The example in fig. 4.7a illustrates this problem.*
At the first selection iteration:

— *The set of SIMD group candidates is* $\{c1 = \{S1, S4\},\ c2 = \{S2, S5\},\ c3 = \{S3, S6\}\}$.

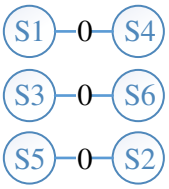
— *The set of common statement conflicts is empty.*

```
S1:   v1 = a + b;
S2:   v2 = c * d;
S3:   v3 = e / f;
S4:   v4 = v3 + g;
S5:   v5 = v1 * h;
S6:   v6 = v2 / i;
```

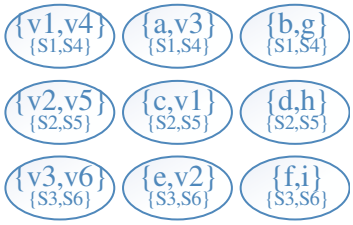
(a) Example C code.



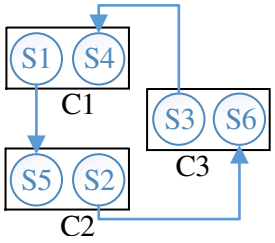
(b) Statements dependence graph.



(c) SGG at first iteration shows no conflicts.



(d) VPCG at first iteration shows no conflicts.



(e) Statements packing and dependencies show cyclic dependency between $c1$, $c2$ and $c3$.

Figure 4.7 – Example illustrating `holistic SLP` algorithm. It shows that VPCG and SGG do not represent cyclic dependency conflicts. The grouping solution is $\{c1, c2, c3\}$ which is not legal due to cyclic dependency between $c1$, $c2$ and $c3$.

— *The corresponding VPCG and SGG are depicted in fig. 4.7d and fig. 4.7c, respectively. They show no conflicts between candidates.*

— *All candidates have same weight, so either of them can be selected (randomly). Let's assume it's* $c1$ *that get selected.*

— *SGG is updated accordingly: nodes* $S1$ *and* $S4$ *(and the edge connecting them) are removed.*

— *VPCG is updated by removing all nodes belonging to* $c1$*, namely:* $\{v1, v4\}_{c1}$ *,* $\{a, v3\}_{c1}$ *and* $\{b, g\}_{c1}$*.*

At the second selection iteration:

— *The set of remaining SIMD group candidates is* $\{c2, c3\}$*.*

— *Still no conflicts represented!*

— *Either* $c2$ *or* $c3$ *get selected* $(VPR(c2) = VPR(c3) = 0)$*.*

— *VPCG and SGG are updated accordingly.*

At the third selection iteration:

— *Only one candidate is left and it get selected.*

The final grouping solution is, therefore, $\{c1, c2, c3\}$*. This solution is not legal due to the cyclic dependency between* $c1$*,* $c2$ *and* $c3$ *as can be seen in the data dependence graph in fig. 4.7e.*

4.3.2.2 Group Selection Process

`Holistic SLP` employs VP reuse estimation, detailed in section 4.3.1, to evaluate the benefit associated with the selection of a SIMD group candidate. This estimation is a key element since

it is the driver of the grouping decisions leading to the statements packing solution.
When this estimation is not accurate or precise enough, it can misleads the group selection process to make the "wrong" choice resulting in a less efficient solution.

The method used by `holistic SLP` suffers from two main shortcomings. It accounts for invalid VP reuse and it considers the same cost for all VP types.

Invalid VP Reuse

The `holistic SLP` algorithm considers a reuse between two VPs when they are equivalent [6] regardless of the type of dependency between them. In other words, if there exists an anti-dependency [7] or an output dependency [8] between two VPs, a reuse is accounted for. However, in such a case there is no actual reuse between them. This may result in a misleading estimation that prevents the algorithm from selecting the "best" group.

This problem can be easily fixed by eliminating anti- and output dependencies in the basic block prior to applying SLP extraction.

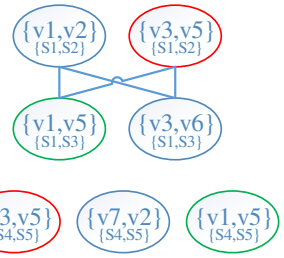
Example 6 *An example of such a case is illustrated in fig. 4.8. In this example the set of candidates is*

$$C = \{c1 = \{S1, S2\},\ c2 = \{S1, S3\},\ c3 = \{S4, S5\}\} \tag{4.5}$$

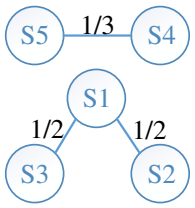
$c1$ conflicts with $c2$. The corresponding initial VPCG and SGG are represented in fig. 4.8b and

```
S1:   v1 = v3;
S2:   v2 = v5;
S3:   v5 = v6;
S4:   v3 = v7 + v1;
S5:   v5 = v2 + v5;
```

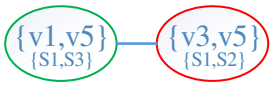
(a) example C code.



(b) VPCG at first iteration.



(c) SGG at first iteration.



(d) Auxiliary graph for candidate $\{S4, S5\}$ before conflicts elimination.

Figure 4.8 – Example showing how fake dependencies affect `holistic SLP` VP reuse estimation.

fig. 4.8c respectively. Following the VP reuse estimation method described by `holistic SLP`, a reuse is accounted for between $c1$ and $c3$ on the VP $\{v3, v5\}$ even though it is an anti-dependency. Consequently:

6. contain the same elements (see definition 4.4)
7. Write-after-Read dependency.
8. Write-after-Write dependency

— $VPR(c1) = VPR(c2) = 1/2$

— $VPR(c3) = 1/3$; *VP* $\{v3, v5\}$ *is reused once by* $c1$ *and* $\{v1, v5\}$ *is also reused once by* $c2$*, but they both conflict with each other as shown in* $AG(c3)$ *depicted in fig. 4.8d, thus only one reuse is considered.*

Since $c1$ *and* $c2$ *both have the maximum* VPR *value, one of them is selected randomly. If* $c1$ *is selected, the final grouping solution would be* $\{c1, c3\}$*. In this solution there is no actual VP reuse since* $\{v3, v5\}$ *used by* $c1$ *is not reused by* $c3$ *but written to. In other words, a packing is needed to form* $\{v3, v5\}$ *in order to be used by* $c1$ *and also an unpacking is needed to write the result of* $c3$ *to* $\{v3, v5\}$ *(unless* $\{v3, v5\}$ *is contiguous (and aligned) in memory).*

However, if the anti-dependency on $\{v3, v5\}$ *was not considered as a reuse,* $VPR(c1)$ *would be* 0 *and the final solution would become* $\{c2, c3\}$ *in which there is an actual reuse of* $\{v1, v5\}$ *between* $c2$ *and* $c3$ *(the packed value written to* $\{v1, v5\}$ *by* $c2$ *can be directly reused by* $c3$*).*

Same Cost for all VP Types

In `holistic SLP`, when forming SIMD group candidates, VPs are consequently obtained as the set of variables from the same position of the statements inside a group candidate. `Holistic SLP` does not impose any additional constraint on forming VPs, which can thus contain any variable whether it is a scalar or an array reference. In particular, VPs can contain references to different arrays and/or references to non-contiguous elements of the same array. Such VPs induce a high packing/unpacking cost.

During group selection, `holistic SLP` ignores the cost difference between different VP types when computing the VP reuse estimation associated with a candidate. In fact, only the number of VPs is used, thus assuming a unique cost regardless of the VP type. This may lead the group selection to select SIMD groups with many array reference superwords referring to different arrays or to non-contiguous array elements, completely ignoring the high packing/unpacking cost associated with implementing them.

To counter this problem, `holistic SLP` relies on data layout transformation to hopefully optimize the memory layout of such superwords and avoid the need for packing/unpacking operations. However, the data layout transformation, employed by `holistic SLP`, for array reference superwords suffers from many shortcomings:

— It is very restrictive; it can only be applied in the context of SCoPs for array reference superwords referring to the *same* array, that should be *read-only*.

— It may introduce significant additional memory requirements. Up to an array replication per array reference superword, in fact. When targeting embedded platforms, memory resources are generally very limited. In particular, the data cache or local scratchpad memory.

— It may also introduce execution time overhead to initialize the array replications with the transformed data layout from the original layout, which is generally performed at runtime. But also due to the increasing cache (or local memory) pressure.

All in all, completely ignoring packing/unpacking cost associated with array reference superwords

during group selection, and relying solely on data layout transformation to hopefully optimize superwords placement in memory and eliminate the packing/unpacking cost overhead is not a good strategy given the limitations and drawbacks of data layout transformations.

This problem can be solved by using a more accurate candidate benefit estimation method, taking into account the cost difference of various VP types. Alternatively, we propose to consider array access operations to contiguous elements of the same array as SIMD group candidates, rather than VPs, to make them part of the SLP solution space. We will discuss this solution later in section 4.4.2.

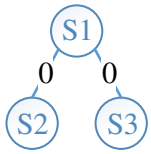
Example 7 *To illustrate this case, let's consider the example of fig. 4.9 assuming a maximum vector size of two.*

At the first iteration, only two candidates are available: $c1 = \{S1, S2\}$ *and* $c2 = \{S1, S3\}$. *Since they have no VP reuse,* $VPR(c1) = VPR(c2) = 0$. *Therefore, either* $c1$ *or* $c2$ *is selected randomly.*
If $c1$ *is selected then the* $\{a, b\}$ *and* $\{A[i], B[i]\}$ *should be packed into SIMD vectors and the result should be unpacked to store it in* $\{x, y\}$. $\{a, b\}$ *and* $\{x, y\}$ *are scalar superwords and can be optimized by data layout transformation. However,* $\{A[i], B[i]\}$ *cannot since it contains accesses to different arrays. Therefore a packing operation is required in this case.*
However, if $c2$ *is selected no packing operations are required to pack* $\{A[i], A[i+1]\}$ *(assuming alignment).*

In this case, $c2$ *is more likely to be better than* $c1$ *however* `holistic SLP` *cannot distinguish them and considers them as equally beneficial.*

```
S1:   x = a + A[i];
S2:   y = b + B[i];
S3:   z = y + A[i+1];
```

(a) C code snippet.



(b) SGG at first iteration.

Figure 4.9 – Example illustrating `holistic SLP` candidate benefit estimation.

Only consider the VPs of the candidate at hand

The candidate benefit estimation method used by `holistic SLP` only considers the amount of reuses of the VPs of the candidate at hand, say c, by the remaining candidates (in addition to already selected groups) – as shown in eq. (4.2). In other words, it does not consider the potential benefit of the remaining candidates, which have no reuse of c's VPs.

This has a great impact specially during the first group selection iteration, since no groups have been yet selected. In this case:

$$G' = \{c\} \tag{4.6}$$

and consequently, eq. (4.2) becomes:

$$VPR(c) = \frac{|V(\{c\})| - |P(\{c\})| + |R(c)|}{|P(\{c\})|} = \frac{|R(c)|}{|P(\{c\})|} \tag{4.7}$$

This means that the only contribution to the benefit evaluation of c in this case, is the estimation of the number of reuses of VPs of c (only) by the remaining candidates. This generally leads to situations where many candidates are estimated to have the same (maximum) benefit. In such cases the group selection process is unable to decide which candidate is better and will randomly select one of them, like in the examples 5 and 6 for instance.

In section 4.4.2, we show through experimental results, that this problem greatly impacts the performance of the SLP solution and we then propose a solution.

4.4 Proposed `Holistic SLP` Algorithm Enhancements

To recall the context of this work, the aim is to provide an efficient source-to-source Simdization transformation to target embedded processors in the context of the ALMA project, mainly XENTIUM and KAHRISMA. However, in order to better illustrate the advantage of our approach, we also considered two other embedded processors, namely, ST240 [4] and VEX [5]. All four target compilers cannot perform Simdization, therefore we implement a source-to-source transformation to convert a sequential C code into one with SIMD intrinsics of the target processor.

In order to do that, we first present several enhancements to the `holistic SLP` algorithm proposed by Liu et al, `holistic SLP` [78], thoroughly discussed in section 4.3. In this section, we address the issues of cyclic dependency conflicts and invalid VP reuse, described in sections 4.3.2.1 and 4.3.2.2. But more importantly, we:

— Define a more efficient and compact IR. We refer to it as Pack Flow and Conflict Graph (PFCG).

— Propose an alternative candidate benefit estimation and group selection method.

— Implement the proposed SLP extraction method and compare it against `holistic SLP` for a set of benchmarks on three different embedded processors.

The overview of the modified `holistic SLP` extraction algorithm is depicted in fig. 4.10. It is similar to the one used by `holistic SLP`, the main difference lies in the candidates identification and IR construction as well as the group selection process.
First we obtain the DFG of the input basic block, then we identify isomorphic operations as well as contiguous accesses to same arrays. Next we build the PFCG, that captures both, the candidate conflicts and the VP flow information. We present the PFCG in section 4.4.1. Like `holistic SLP`, we only consider SIMD group candidates of size two and we iteratively extends the selected ones when possible. However, the group selection process is based on a new candidate estimation method, that we later present in section 4.4.2. After selecting a new group,

we update the PFCG by eliminating all conflicting candidates and we eliminate any possible cyclic dependencies. Finally, the selected groups order is specified in such a way to minimize permutations. We can then, optionally, estimate the speedup of the obtained solution and prune it accordingly to make sure it performs better than the original scalar code, but in this work we only focus on the group selection process.

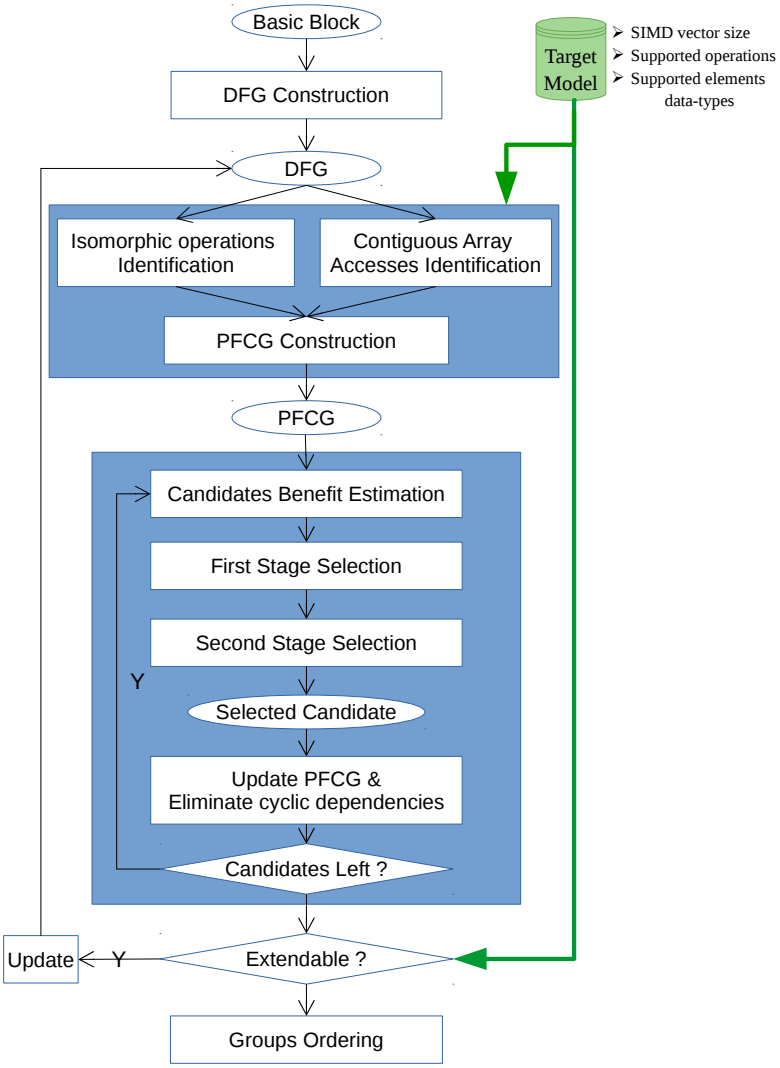


Figure 4.10 – SLP extraction framework overview.

4.4.1 Pack Flow and Conflict Graph (PFCG)

The IR used by `holistic SLP`, namely SGG and VPCG, holds necessary information for the SLP extraction process. SGG provides statement compatibility (group-ability) i.e. group candidates, as well as candidate common statement conflicts information. Whereas VPCG provides, VPs per candidate information, in addition to VP conflicts.

This information is partly redundant with SGG, since VPs of a candidate is an atomic property of the candidate, so that all VPs of a candidate c conflict with all VPs of another candidate in conflict with c. This redundancy is unnecessary. Note that VPCG does not explicitly hold information about VP reuse. Comparing VPCG nodes is necessary in order to identify equivalent VPs and account for reuse.
All in all, `holistic SLP` IR has several disadvantages:

— SGG cannot represent cyclic dependency conflicts (see section 4.3.2.1).

— Conflicts information is, unnecessarily, redundant between SGG and VPCG.

— VP reuse information is not captured explicitly in VPCG.

To address these disadvantages, we define a new IR, that we refer to as PFCG. It is a combination of a VP Flow Graph, similar to a DFG, and a Candidate Conflict Graph.

Définition 4.6 (PFCG) *PFCG is a mixed graph* $(P \cup C, X, F)$ *where:*

— $P \cup C$ *is the set of nodes.* C *is the set of SIMD group candidate nodes.* P *is the set of input variable pack nodes, which are not in* C*. Note that in this representation, a SIMD group candidate node also represents a VP; its output VP.*

— X *is a set of undirected edges representing conflicts between SIMD group candidate nodes. It includes all common statement conflicts as well as direct cyclic dependency conflicts*[9]*. Note that we only consider direct cyclic dependency conflicts when building the PFCG, indirect ones are dealt with during group selection step (see section 4.4.2).*

— F *is a set of directed edges representing VP flow between nodes.*

The PFCG is constructed from the basic-block's DFG as depicted by the procedure BuildPFCG in algorithm 2 (in section 4.4.4). We use DFG form of the basic block, to increase the potential of finding SLP at a finer level. We also eliminate anti- and output dependencies that might otherwise affect VP reuse estimation as illustrated in section 4.3.2.2.

The advantages of using the PFCG are its ability to represent VP flow information, which eases the detection of VP reuse – for a given candidate, we only need to check its neighbors on flow edges to detect its VP reuses – and its ability to represent cyclic dependency conflicts while being more compact than VPCG and SGG.

Example 8 *Figure 4.11 shows the PFCG of the example of fig. 4.1 (recalled). Note the difference with VPCG and SGG, of the same example, depicted in figs. 4.4b and 4.4c, respectively. Using the PFCG we can easily detect, for instance, that the VP* $\{b, d\}$ *is used by both candidates* $\{S1, S2\}$ *and* $\{S4, S5\}$*. Also it is more compact than the VPCG and SGG.*

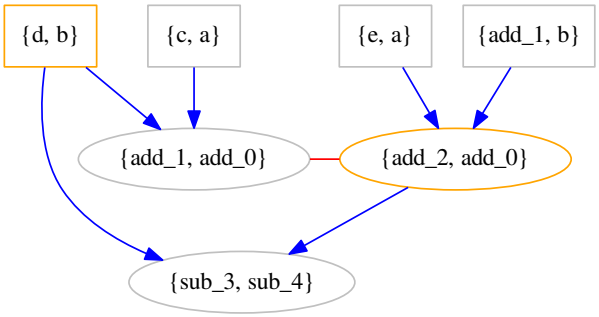
4.4.2 Group Selection Procedure

We propose an alternative group selection heuristic to the one employed by `holistic SLP` in order to address the shortcomings illustrated in section 4.3.2.2.

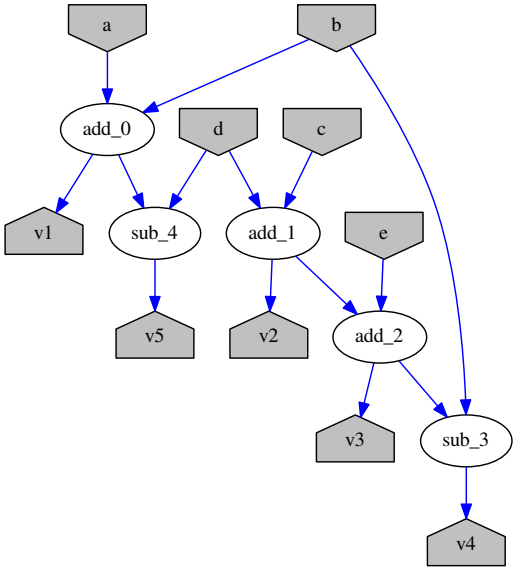
9. Direct cyclic dependency occurs between two candidates only. Indirect cycle occurs between more than two candidates.

```
S1:   v1 = a + b;
S2:   v2 = c + d;
S3:   v3 = e + v2;
S4:   v4 = v3 - b;
S5:   v5 = v1 - d;
```

(a) Recall example of fig. 4.1.



(b) Corresponding PFCG at the first iteration.



(c) Corresponding DFG.

Figure 4.11 – DFG and PFCG of the example in fig. 4.1. Rectangular nodes in the PFCG represent pack nodes, whereas elliptic shaped ones represent candidate nodes. Undirected edges represent conflicts between candidates and directed edges represent VP Flow. Highlighted nodes (with orange) indicate a reuse.

Unlike `holistic SLP`, we consider array access operations to contiguous elements of the same array as SIMD group candidates, rather than VPs, when building the PFCG. This will distinguish such VPs and make them part of the SLP solution space. This way the group selection procedure can decide to select such candidate based on its benefit estimation instead of being implicitly selected due to the selection of a candidate that uses/defines it.

Furthermore, we use a two-stages selection procedure based on a new candidate benefit estimation method that we present in section 4.4.2.1. In the first stage, we select the set of all candidates with the highest VP reuse-to-cost ratio. Among them, we select one with highest reuse-to-conflict ratio in the second stage.

The pseudo code of the group selection procedure is listed in algorithm 1. Each remaining candidate in the PFCG is evaluated using the candidate benefit estimation method. The one estimated as the most beneficial is selected, the PFCG is updated accordingly and the potential cyclic dependency conflicts are eliminated before starting a new iteration until all conflicts in the PFCG are resolved.

4.4.2.1 Candidate Benefit Estimation

The candidate benefit estimation method used by `holistic SLP` only considers the first neighbors of the candidate being evaluated, say c, in the PFCG. In fact, `holistic SLP` only considers

the reuses of c's VPs by the remaining candidates in addition to reuses of already selected groups, as shown in eq. (4.2). In other words, the potential benefit of the remaining candidates which have no reuse of c's VPs (i.e. which are not direct neighbors of c in the PFCG) is not considered.

This problem generally leads to situations where many candidates have the same maximum benefit. In such case the `holistic SLP` algorithm is not able to decide which candidate is better and will randomly select one of them. For instance, example 6 illustrates this case. This is specially frequent during the first iteration where no groups have been selected yet. In this case the only contribution to the benefit evaluation is the estimation of the number of reuses of VPs of c by the remaining candidates, as shown in eq. (4.7).

To illustrate the impact of this problem, we applied `holistic SLP` repeatedly (ten times) on each benchmark [10]. Every time multiple candidates can be selected (i.e. have the same maximum benefit), only one of them is selected randomly,as specified by `holistic SLP`.

For each benchmark, after applying Simdization using `holistic SLP`, we report the execution time improvement of the generated SIMD code over the original (sequential) code. We repeat this ten times, on three embedded processors, namely XENTIUM, KAHRISMA and ST240 [11]. The graphics of fig. 4.12 plot the execution time improvement obtained for each time the test is run. This results show a significant variation of the execution time improvement for most benchmarks, when using `holistic SLP`, across different runs. For instance, the generated SIMD code, obtained by applying `holistic SLP` on the infinite impulse response filter benchmark (**iir**), for ST240, can perform up to 23% better or as low as 2% worst than the original (sequential) code version, all depending on the SIMD group candidates being randomly selected each time multiple ones are judged equally beneficial by the candidate benefit estimation method of `holistic SLP`. This indicates that the `holistic SLP` candidate benefit estimation method is not precise enough and often lead to situations where the group selection process cannot decide which candidate is better and thus select randomly. This random selection often lead to a poor SLP solution as shown by the results in fig. 4.12.

In order to address this issue, we propose to consider more than the first neighbors of the candidate being evaluated, say c, in the PFCG. This allows to get a more global picture of the effect of selecting c. In addition, we also use a two-stage selection procedure to further reduce the cases where multiple candidates have same benefit estimation. First, we select the set of all candidates with the highest VP *reuse-to-cost* ratio, among which we then select one with highest *reuse-to-conflict* ratio in the second stage.

The pseudo code of the proposed candidate benefit estimation is listed in algorithm 5. Given a candidate c, we first extract a subgraph, $sub(c, N)$, from the PFCG containing c and its neighbors up to N levels. We then eliminate all conflicts in $sub(c, N)$ by iteratively removing a candidate with highest *conflict-degree-to-flow-degree* [12] ratio. The resulting subgraph not only represents

10. the benchmarks are described in section 4.5.3)
11. the target processors are described in section 4.5.2
12. Conflict degree of a candidate is the number of conflicting edges connected to it in $sub(c, N)$. Its Flow degree is the number of flow edges connected to it (it is an indication of the amount of VP reuse of the candidate).

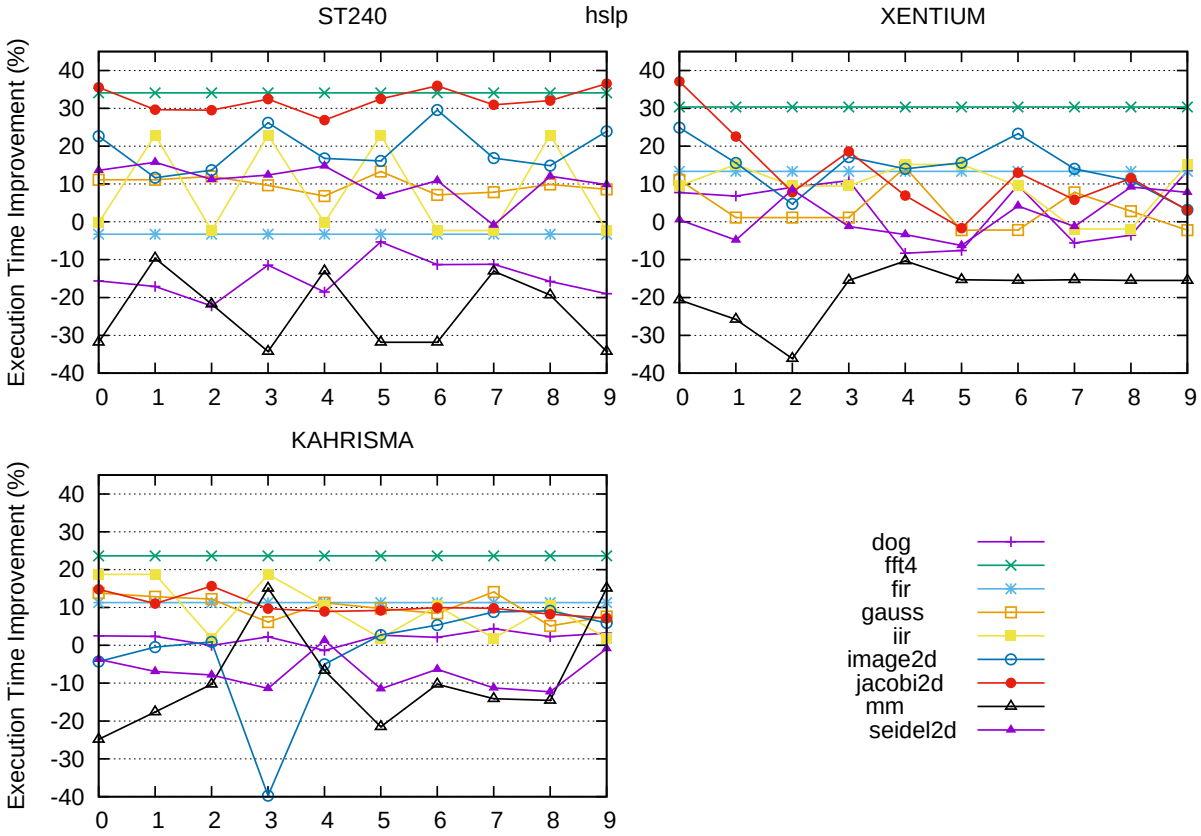


Figure 4.12 – Execution time improvement, over the sequential version, of various benchmarks obtained by applying `holistic SLP` for ten times.

the reuses of c's VPs but also those of some other non-conflicting candidates. Once the subgraph extracted, we compute the ratio of, the number of VP reuses of all candidates in the subgraph, to the estimated packing/unpacking cost.

To evaluate the proposed method, we run the same test as before but using our proposed method for SLP extraction (for $N = 3$) instead of `holistic SLP`. The results, reported in fig. 4.13, show much less variation in the execution time improvement for all benchmarks, suggesting that our method achieves a more precise candidate benefit estimation and is less prone to the random selection factor.

Example 9 *Back to example 7 (recalled in fig. 4.14).*

Using our proposed SLP extraction method, the set of candidates is:

$$\{c1 = \{add\_1, add\_2\}, c2 = \{add\_1, add\_3\}, c3 = \{A[i], A[i+1]\}\} \tag{4.8}$$

Note that $\{A[i], A[i+1]\}$ is considered as a candidate, unlike `holistic SLP`. The corresponding PFCG is depicted in fig. 4.15. The benefit estimation of the candidates is computed following the method listed in algorithm 3. For $c1$, the neighborhood graph only contains $c1$, therefore:

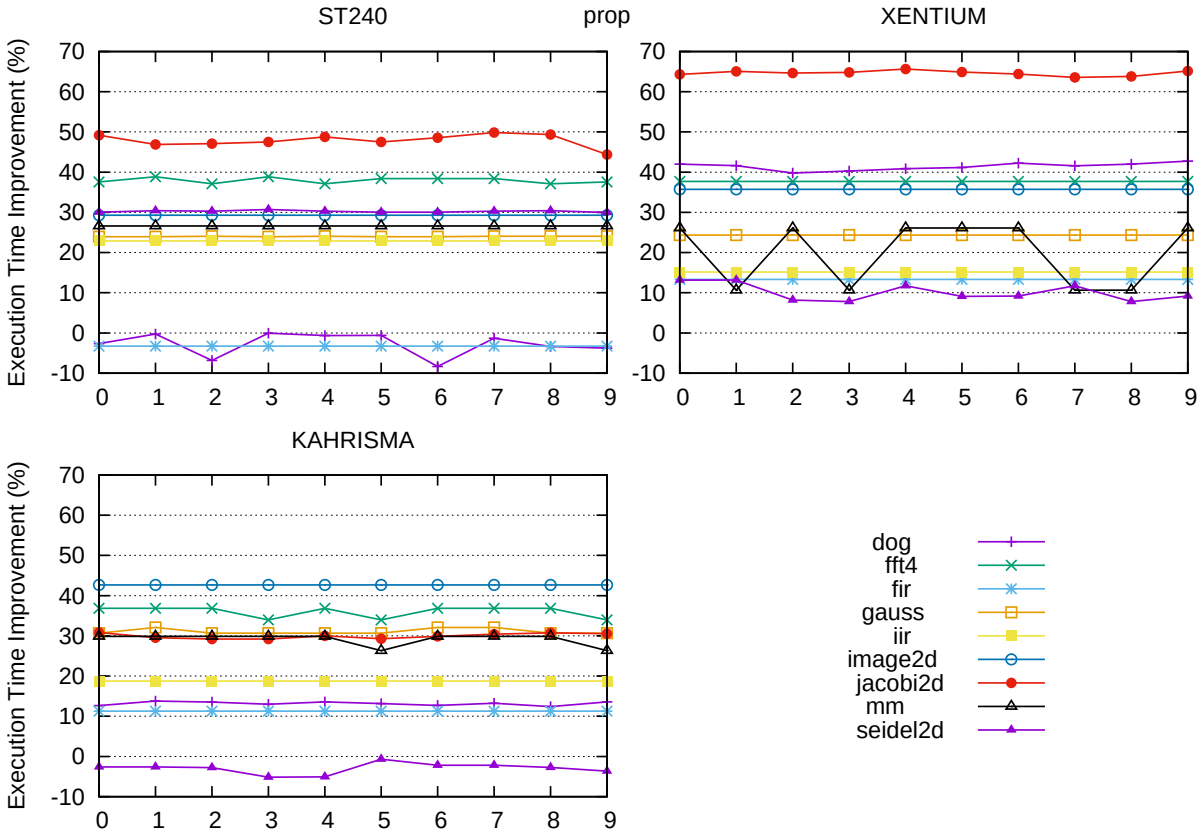


Figure 4.13 – Execution time improvement, over the sequential version, of various benchmarks after applying the proposed SLP extraction for ten times.

— $N_R = 1$. *There is no reuse.*

— $N_{pack} = 2$. *Estimated number of required packs is* 2; $\{b, a\}, \{B[i], A[i]\}$ *and* $\{add\_2, add\_1\}$.

— $N_{unpack} = 1$. *Estimated number of required unpacks is* 1 *since* $c1$ *has no successors.*

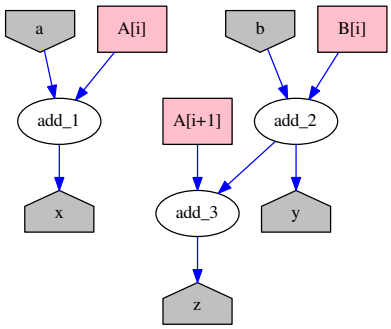
— *Thus the benefit of* $c1$ *is* $\dfrac{1 + N_R}{1 + N_{pack} + N_{unpack}} = 2/4$

For $c2$ *and* $c3$ *the neighborhood graph contains both* $c2$ *and* $c3$. *Thus, the benefit estimation for* $c2$ *is* $(1+2)/(1+1+1) = 3/3$ *and is the same for* $c3$. *In this case either* $c2$ *or* $c3$ *is selected at the first iteration, then the other will be selected in the next iteration. Hence, the final grouping solution is* $\{c2, c3\}$.

In contrast, when using `holistic SLP` $c3$ *is not considered as candidate and* $c1$ *and* $c2$ *are estimated to have the same benefit as illustrated in example 7. Therefore, the final grouping solution could be* $\{c1\}$ *or* $\{c2\}$ *(randomly).* $\{c2\}$ *– equivalent to the solution* $\{c2, c3\}$ *obtained by using our proposed method – would more likely perform better than* $\{c1\}$.

```
S1:   x = a + A[i];
S2:   y = b + B[i];
S3:   z = y + A[i+1];
```

(a) C code snippet.



(b) Corresponding DFG.

Figure 4.14 – Recall the example of fig. 4.9.

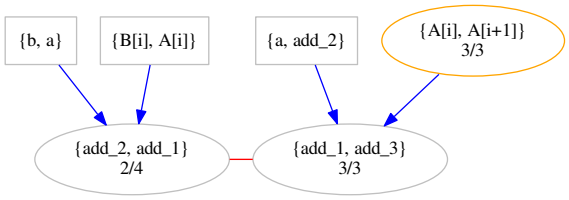


Figure 4.15 – PFCG of the example in fig. 4.14 at the first iteration using `prop-2`.

4.4.2.2 Update PFCG

After selecting of a new SIMD group, g, we update the PFCG following the procedure UpdatePFCG listed in algorithm 7. First, we move the node representing g in the PFCG from the candidates set to the selected groups set. Then, we eliminate all candidates in the PFCG that conflicts with g, that is all nodes connect to g by a conflict edge. When eliminating a candidate, c, we make sure to keep the PFCG consistent. So:

— If c has flow successors, we convert it into a VP node, i.e. it is not a group candidate anymore, and we remove all its incoming flow edges. If the source of the removed incoming flow edge is a VP node that is only connected to c then we remove it as well.

— Otherwise, we first handle the incoming flow edges of c similarly, before removing it.

4.4.2.3 Cyclic Dependency Conflicts Elimination

Since we do not consider all cyclic dependency conflicts, namely indirect ones, when building the PFCG, we eliminate such conflicts after updating the PFCG due to the selection of a new SIMD group. For each of the remaining candidates we check whether it may introduce a cyclic dependency, with previously selected SIMD groups, if it get selected. If so, we eliminate it from the PFCG in order to guarantee the final grouping solution to be cyclic dependency free. The pseudo-code of the cyclic dependency conflicts elimination procedure is listed in algorithm 8.

4.4.3 Algorithm complexity

Problem size

The IR construction time, whether for the VPCG used by `holistic SLP`, or the PFCG in our proposed algorithm, is roughly quadratic with respect to the number of group candidates. However, the IR is constructed only once before starting the group selection procedure.

Our proposed IR (PFCG) is more compact than the one used by `holistic SLP` (VPCG + SGG). If C is the set of SIMD group candidates then:

— The number of nodes in the PFCG is equal to the number of candidates ($|C|$) plus the number of input VPs which are not in C (cf. definition 4.6), i.e.:

$$N_{PFCG} = |C| + \sum_{c \in C} |\{v \in V_{in}(c) : v \notin C\}| \tag{4.9}$$

— The number of nodes in the VPCG equals the sum of VPs in each candidate:

$$N_{VPCG} = \sum_{c \in C} |V(\{c\})| = \sum_{c \in C} (1 + |V_{in}(c)|) = |C| + \sum_{c \in C} |V_{in}(c)| \tag{4.10}$$

Where $V_{in}(c)$ is a function that returns the set of *input* VPs for the candidate c. A candidate has only one output VP, therefore, the number of VPs for c is $|V(\{c\})| = 1 + |V_{in}(c)|$ [13].

$$\{v \in V_{in}(c) : v \notin C\} \subset V_{in}(c) \implies N_{VPCG} \geq N_{PFCG} \tag{4.11}$$

Candidate benefit estimation

In order to estimate the benefit of a candidate c, `holistic SLP` requires to build an *auxiliary graph* for c, $AG(c)$ (see section 4.3.1). The time for building $AG(c)$ from the VPCG (the method used by `holistic SLP`) is linear with respect to the number of nodes in the VPCG.

On the other hand, our candidate benefit estimation method requires to build a sub-PFCG ($sub(c)$) obtained by extracting the distance-N neighborhood of c on the Pack flow edges in the PFCG. The building time of $sub(c)$ is roughly constant with respect to the PFCG size, specially since very small values of N (≤ 3) are sufficient, as shown in the results of fig. 4.16. The test setup, benchmarks and target processors are presented later on in section 4.5.

Group selection

The group selection procedure (assuming no group size extension is performed) is roughly the same for our approach and that of `holistic SLP`. However, due to the difference in the complexity of candidate benefit estimation, the group selection of `holistic SLP` is:

$$O(|C| \times |V(C)|) \approx O(|C|^2) \tag{4.12}$$

13. the function V is defined in eq. (4.3)

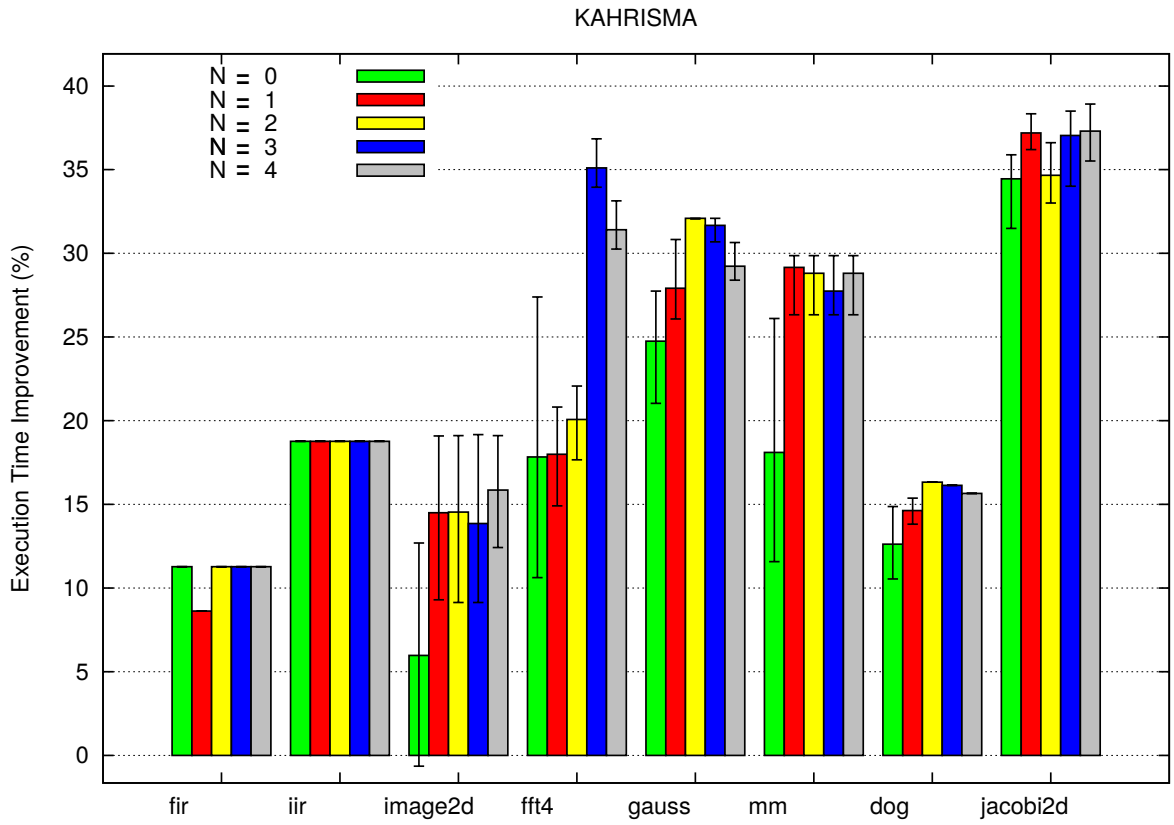


Figure 4.16 – Execution time improvement of the SIMD code obtained by applying our proposed SLP extraction method, compared to the original (sequential) code, for different values of N considered for constructing the distance-N neighborhood of a candidate (sub-PFCG) in the PFCG. The targeted processor is KAHRISMA.

Where C is the set of SIMD group candidates and $|V(C)|$ is the number of nodes in the VPCG. On the other hand, the complexity of our approach is roughly $O(|C|)$.

The group selection time of our approach varies, roughly, linearly with respect to the number of candidates, whereas that of `holistic SLP` varies quadratically. The experimental results presented in section 4.5.5.2 (figs. 4.21 and 4.22a) shows the same behavior.

4.4.3.1 Controlling PFCG Size

When building the PFCG, we consider each pair of isomorphic operations in the DFG for finding SIMD group candidates, as shown in algorithm 2. With larger basic blocks, the size of the PFCG (number of candidates) grows quadratically. In fact, for a set of K isomorphic operations, we can obtain up to

$$f(K) = (K-1) \times K/2 \tag{4.13}$$

candidates.

In order to reduce the PFCG size, and consequently cut down the groups selection time, we decompose each set of isomorphic operations into *batches* of limited size, and we only search for candidates inside each batch separately. In this case, for a set of K isomorphic operations and a batch size S, we can obtain up to:

$$(K/S) \times f(S) + f(K\%S) \approx \frac{K \times S}{2} \tag{4.14}$$

candidates. Hence, the number of candidates roughly increases linearly with the increase of the batch size.

The *batch size* allows to control the PFCG size and consequently the group selection time, at the cost of degrading the grouping solution performance. This is illustrated in the experimental results in section 4.5.5.2.

This concept is similar to basic-block splitting. But, in order to increase the potential of finding relevant candidates inside each batch, we first sort the set of isomorphic operations in topological order. This way independent operations are more likely to end up in the same batch.

4.4.4 Pseudo-code

In this section, we simply list the pseudo-code of PFCG construction and group selection procedures that we discussed earlier, for the sake of keeping them closer to each other.

Algorithm 1 Group Selection Procedure

```
1:  // Input: Pack Flow and Conflict Graph pfcg
2:  // Output: Set of selected groups.
3:  procedure GroupSelection(pfcg)
4:      while pfcg.C ≠ ∅ do
5:          for each candidate c ∈ pfcg.C do
6:              BenefitEstimation(c, pfcg)
7:          end for
8:          best ← Select(pfcg).
9:          UpdatePFCG(best, pfcg) // mark best as selected and Update pfcg
10:         EliminateCycles(pfcg)
11:     end while
12:     return pfcg.S
13: end procedure
```

Algorithm 2 Building PFCG

1: // **Input**: Data Flow Graph dfg. It is a Directed Acyclic Graph (DAG)
2: // **Output**: Pack Flow and Conflict Graph $pfcg$
3: **procedure** BUILDPFCG(dfg)
4: $pfcg.C \leftarrow \emptyset$ // Set of SIMD group candidate nodes
5: $pfcg.P \leftarrow \emptyset$ // Set of input Pack nodes
6: $pfcg.F \leftarrow \emptyset$ // Set of pack flow edges
7: $pfcg.X \leftarrow \emptyset$ // Set of candidate conflict edges
8: $pfcg.S \leftarrow \emptyset$ // Set of Selected candidates
9: **for each** pair (o_1, o_2) of isomorphic operations or contiguous array accesses in dfg **do**
10: // Operations batching is omitted
11: **if** o_1 and o_2 are independent and have compatible data-types **then**
12: $c \leftarrow \{o_1, o_2\}$
13: $pfcg.C \leftarrow pfcg.C \cup \{c\}$
14: // connect c's flow successors
15: **if** $\exists\ p \in pfcg.P : p \equiv c$ **then**
16: $pfcg.P \leftarrow pfcg.P \setminus \{p\}$
17: substitute (p, x) by (c, x), $\forall (p, x) \in pfcg.F$
18: **end if**
19: // connect c's flow predecessors
20: // $dfg.preds(n)$ is the set of predecessors of node n in the dfg
21: **for each** $v1_i, v2_i : v1_i \in dfg.preds(o_1)$ and $v2_i \in dfg.preds(o_2)$ **do**
22: **if** $\exists\ v \in pfcg.C : v \equiv \{v1_i, v2_i\}$ **then**
23: $pfcg.F \leftarrow pfcg.F \cup \{(v, c)\}$
24: **else if** $\exists\ v \in pfcg.P : v \equiv \{v1_i, v2_i\}$ **then**
25: $pfcg.F \leftarrow pfcg.F \cup \{(v, c)\}$
26: **else**
27: $pfcg.P \leftarrow pfcg.P \cup \{\{v1_i, v2_i\}\}$
28: $pfcg.F \leftarrow pfcg.F \cup \{(\{v1_i, v2_i\}, c)\}$
29: **end if**
30: **end for**
31: // connect c's conflicts
32: $pfcg.X \leftarrow pfcg.X \cup \{\{c, c'\} : c' \in pfcg.C$ and $(c' \cap c \neq \emptyset$ or c', c introduce a cyclic dependency) $\}$
33: **end if**
34: **end for**
35: **return** pfcg
36: **end procedure**

Algorithm 3 Estimate the benefit associated with selection of a candidate

1: // **Inputs**: A candidate c and the pack flow and conflict graph $pfcg$
2: // **Output**: Set the benefit of candidate c
3: **procedure** BENEFITESTIMATION($c, pfcg$)
4: $sub \leftarrow$ EXTRACTSUBPFCG($c, pfcg$)
5: ELIMINATESUBCONFLICTS(c, sub) // sub is now conflict-free
6: COMPUTEBENEFIT(c, sub, $pfcg$)
7: **end procedure**

1: // N is a positive integer parameter
2: // **Output:** distance-N pack flow neighborhood of c in $pfcg$
3: **procedure** EXTRACTSUBPFCG($c, pfcg$)
4: $sub.\mathcal{V} \leftarrow \{x \in pfcg.C \cup pfcg.S : distance(x, c) <= N,\ \{x, c\} \notin pfcg.X\}$, where:
5: $distance(x, c)$ is the distance between nodes c and x in respect to Pack Flow edges.
6: $sub.X \leftarrow \{\{x, y\} \in pfcg.X : x, y \in sub.\mathcal{V}\}$ // Set of conflict edges
7: $sub.F \leftarrow \{(x, y) \in pfcg.F : x, y \in sub.\mathcal{V}\}$ // Set of pack flow edges
8: **return** sub
9: **end procedure**

Algorithm 4 Eliminate Conflicts in sub

1: **procedure** ELIMINATESUBCONFLICTS(c, sub)
2: // Iteratively eliminate the candidate with highest conflict-to-reuse ratio
3: **while** $sub.X \neq \emptyset$ **do**
4: $x \leftarrow$ node $\in sub.\mathcal{V} \setminus \{c\}$ with highest $\dfrac{conflict\ degree}{1 + flow\ degree}$
5: $sub.\mathcal{V} \leftarrow sub.\mathcal{V} \setminus \{x\}$
6: $sub.X \leftarrow sub.X \setminus \{\{x, y\} \in sub.X\}$
7: $sub.F \leftarrow sub.F \setminus \{(x, y), (z, x) \in sub.F\}$
8: **end while**
9: **end procedure**

Algorithm 5 Compute Candidate Benefit.

1: **procedure** COMPUTEBENEFIT($c, sub, pfcg$)
2: $N_R \leftarrow |sub.\mathcal{V}|$ // Estimated number of VP reuse in sub
3: // Estimated number of additionally required packings.
4: $N_{pack} \leftarrow |\{x \in pfcg.C \setminus sub.\mathcal{V} : \exists (x, y) \in pfcg.F,\ y \in sub.\mathcal{V} \setminus pfcg.S\}|$
5: // Estimated number of additionally required unpackings.
6: $N_{unpack} \leftarrow |\{x \in sub.\mathcal{V} \setminus pfcg.S,\ x$ is not a Store candidate : $\nexists y \in sub.V \cup pfcg.S : (x, y) \in sub.pfcg\}|$
7: $c.reuse \leftarrow N_R$
8: $c.benefit \leftarrow \dfrac{1 + N_R}{1 + N_{pack} + N_{unpack}}$
9: $c.conflict \leftarrow |\{\{x, y\} \in pfcg.X : (x \in sub.\mathcal{V}) \oplus (y \in sub.\mathcal{V})\}|$
10: **end procedure**

Algorithm 6 Select the 'best' candidate in the PFCG

procedure SELECT($pfcg$)
    // First Stage: Set of candidates with maximun benefit estimation
    $bestSet \leftarrow \{c \in pfcg.C : c.benefit = max\}$
    // Second Stage:
    $best \leftarrow c \in bestSet$ with maximum $\frac{c.reuse}{c.conflict}$
    return *best*
end procedure

Algorithm 7 Updating PFCG after selecting a new group

procedure UPDATEPFCG($c, pfcg$)
    // Mark c as selected
    $pfcg.S \leftarrow pfcg.S \cup \{c\}$
    $pfcg.C \leftarrow pfcg.C \setminus \{c\}$
    // Eliminate candidates conflicting with c
    for each $\{x, c\} \in pfcg.X$ **do** // x conflicts with c
        ELIMINATECANDIDATE($pfcg, x$)
    end for
end procedure

procedure ELIMINATECANDIDATE($pfcg, x$)
    for each $(p, x) \in pfcg.F$ **do** // p is a predecessor of x
        $pfcg.F \leftarrow pfcg.F \setminus \{(p, x)\}$ // disconnect p from x
        if $p \in pfcg.P$ and $\nexists(p, s) \in pfcg.F$ **then**
            $pfcg.P \leftarrow pfcg.P \setminus \{p\}$ // p is a pack node with no other successors
        end if
    end for
    if $\exists(x, y) \in pfcg.F$ **then** // if x has successors
        $pfcg.P \leftarrow pfcg.P \cup \{x\}$ // convert x into a pack node
    end if
    $pfcg.C \leftarrow pfcg.C \setminus \{x\}$
    $pfcg.X \leftarrow pfcg.X \setminus \{\{x, y\} \in pfcg.X\}$
end procedure

Algorithm 8 Eliminate cyclic dependency conflicts

procedure CYCLEELIMINATION($pfcg$)
    for each $c \in pfcg.C$ **do**
        if $pfcg.S \cup \{c\}$ have a dependency cycle **then**
            ELIMINATECANDIATE($pfcg, c$)
        end if
    end for
end procedure

4.5 Experimental Setup and Results

In this section, we present the implementation of the proposed SLP extraction framework in section 4.5.1. We then present the test setup, benchmarks and target architectures in section 4.5.4, section 4.5.3 and section 4.5.2 respectively. Finally we present the experimental results in section 4.5.5.

4.5.1 Implementation

We implemented the proposed PFCG model along with the enhanced `holistic SLP` extraction framework, presented in section 4.4, as a source-to-source transformations in the Generic Compiler Suite (GeCoS). The corresponding pseudo-code is listed in section 4.4.4.
We also implemented the original `holistic SLP` extraction framework as described in section 4.3.1 and based on its presentation [78] by Liu et al.

GeCoS [47] is an open source, model-driven, eclipse-based compiler framework developed in the CAIRN/INRIA research team. It is mainly targeted towards, but not limited to, source-to-source transformations for embedded platforms and hardware generation using High-Level Synthesis (HLS).

The proposed `holistic SLP` flow implementation is illustrated in fig. 4.17.
The GeCoS C front-end, based on the C/C++ CDT front-end, parses the input C code and generates the equivalent GeCoS IR (CDFG).
Multiple pre-optimizations can be performed at this level, such as constant propagation, array scalarization, loop unrolling as well as polyhedral loop analysis and transformations, such as register level tiling.

The basic blocks annotated for SLP extraction, either manually using *pragma* annotations or based on previous transformation, are then converted to DFG representation, on which we apply SLP extraction. Note that in this implementation, we only support basic arithmetic SIMD operations such as additions, multiplications and shifts in addition to vector memory accesses.
SLP extraction determine the set of decided SIMD groups, which are then used to vectorize the DFG to SIMD form reflecting the SLP grouping.
The vectorized DFG is then converted back to CDFG introducing SIMD instructions and data-types. At this level, the SIMD instructions and data-types are generic and target-independent. However, the main characteristics of the target architecture were considered during SLP extraction, such as the SIMD vector size, the supported SIMD operations and the vector elements data-types.
Post-optimizations can be performed at this level, such as superword promotion ([124]).

Finally, the SIMD C code generation converts the GeCoS IR into a C code with SIMD macros implementing the SIMD data-types and operations. The implementation of these generic macros is target-dependent and is generated as a *.h* file by GeCoS. It defines the corresponding SIMD types and operations using the target's compiler SIMD intrinsics when available. This makes the

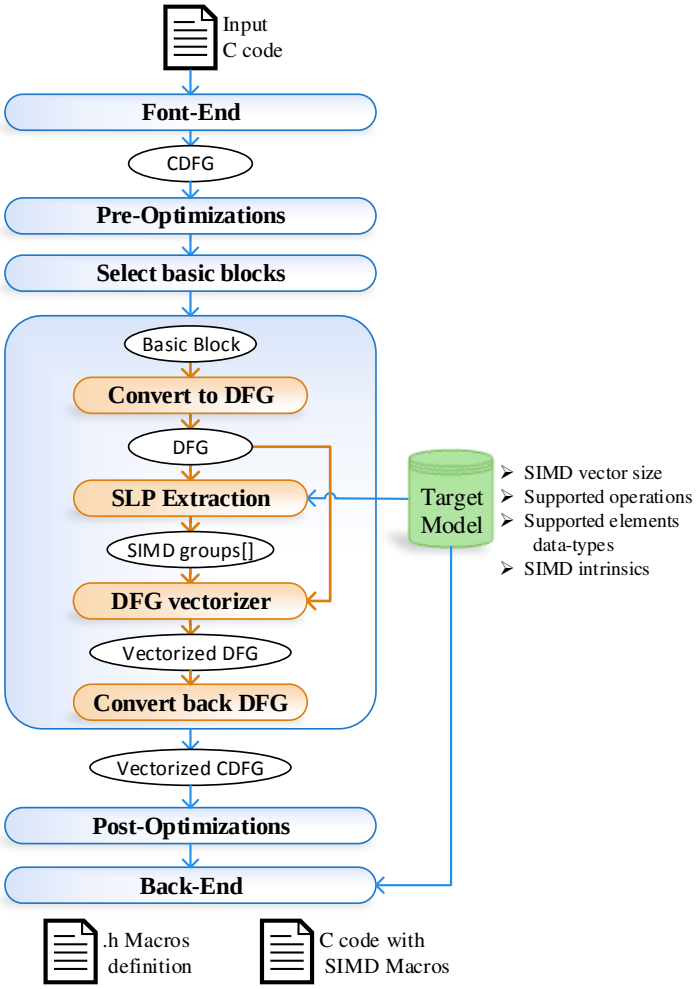


Figure 4.17 – SLP Extraction Framework implementation in GeCos.

framework easily expendable to support additional targets.

4.5.2 Target processors

We target three different embedded architectures, XENTIUM, ST240 and KAHRISMA.

4.5.2.1 XENTIUM

XENTIUM is a low energy consumption 32-bit (with 40-bit extension support) 12-issues wide Very Long Instruction Word (VLIW) Digital Signal Processor (DSP) core from RECORE SYSTEMS [104]. It has ten functional units, six of which can perform integer arithmetic and logic operations, two can perform multiply operations and two can perform load/store operations. XENTIUM also supports 2x16-bit SIMD operations. In this work, we only consider add, sub, mul, shifts and data manipulation SIMD instructions. It has four 8 32-bit and one 16 32-bit

register files, each of which has two read and two write ports.
An LLVM based compiler and a cycle-accurate simulator for the XENTIUM core are provided by RECORE SYSTEMS.

4.5.2.2 KAHRISMA

KAHRISMA (KArlsruhe's Hypermorphic Reconfigurable-Instruction-Set Multi-grained-Array) is a heterogeneous dynamically reconfigurable multi-core research architecture [58], developed by KIT.
In this work we use its 2-issue width single VLIW core configuration.
KAHRISMA core provides support for 4x8-bit and 2x16-bit SIMD integer arithmetic operations including addition, subtraction, multiplication and shifts as well as data manipulation operations.
An LLVM based compiler and a cycle-approximate simulator [127] for this core configuration are provided by KIT.

4.5.2.3 ST240

ST240 [4] is a 4-issues wide VLIW media processor from the ST200 family of STMicroelectronics. It has four integer units, 2 multiplication units, 1 branch unit and 1 load/store unit. It has a general purpose register file of 64 32-bit registers with 8 read and 4 write ports. ST240 also supports 2x16-bit SIMD operations.In this work, we only consider add, sub, mul, shifts and data manipulation SIMD instructions.
A compiler based on GNU GCC and Open64 as well as a cycle-accurate simulator for the ST240 core are provided STMicroelectronics.

4.5.2.4 VEX

VLIW EXample (VEX) is a configurable compilation-simulation system targeting a wide class of VLIW architectures [5]. It is developed by HAWLETT-PACKARD (HP). The interesting point about the VEX toolchain is that it is highly configurable. The configuration parameters includes: the number of clusters, the number of ways, functional units and registers per cluster as well as the latency associated with various operations in addition to the memory hierarchy parameters. But also the possibility to extend its instruction set to include new instructions.

We use this possibility to extend the default instruction set by implementing a 4x8-bit and 2x16-bit integer arithmetic SIMD instruction set. In this model, the SIMD registers are common with the scalar registers. In this work, we use the default configuration, with one cluster and 4 issues. We refer to it as VEX-4.

4.5.3 Benchmarks

We use several signal and image processing benchmarks for testing the proposed SLP extraction framework. They are listed in table 4.1 and the breakdown of the operations in the considered basic blocks is reported in table 4.2.

Each benchmark function is pre-optimized to expose SLP by applying either loop unrolling or register level tiling. And the kernel basic blocks are *pragma* annotated to be considered for SLP extraction.
The main function of each benchmark allocates and initializes the required data, it then resets the statistics counters before launching the benchmark function. Finally, it stops the statistics counters before generated the output.

Benchmark	**Info**	**Pre-optimizations**
fir	128-tap FIR Filter	partial 4x unrolled
iir	64 IIR Filer	partial 4x unrolled
fft4	size 4 Fast Fourier Transform	fully unrolled
image2d	2D 3x3 linear image filter	2x2 register level tiling
gauss	vertical followed by horizontal filter	2x2 register level tiling
mm	2D Matrix Multiplication	2x2x2 register level tiling
dog	Difference of Gaussian (2D stencil)	2x2x2 register tiling
jacobi2d	2D stencil from polybench-c-4.1	2x2x2 register tiling
seidel2d	2D stencil from polybench-c-4.1	2x2x2 register tiling

Table 4.1 – Test Benchmarks.

Benchmark	**add**	**sub**	**mul**	**Total**	**Load**	**Store**	**CAA**
fir	4	0	4	8	8	0	6
iir	0	4	8	12	16	0	12
image2d	0	9	0	9	18	0	12
mm	8	0	8	16	12	4	8
gauss	16	0	24	40	14	8	12
fft4	12	12	16	40	48	16	128
dog	32	0	48	80	37	16	44
jacobi2d	64	0	0	64	51	16	69
seidel2d	64	0	0	64	47	8	80

Table 4.2 – Number of operations and memory accesses in benchmarks. CAA represents the number of contiguous array access candidates.

4.5.4 Tests Setup

In order to compare the proposed SLP extraction method against `holistic SLP`, we employ the test setup illustrated in fig. 4.18.

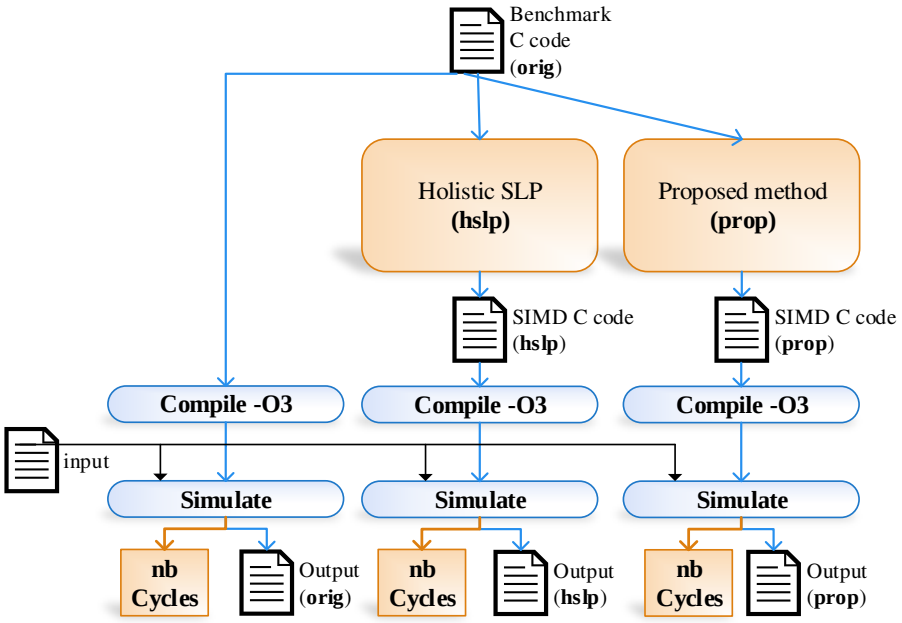


Figure 4.18 – Test procedure diagram.

For each benchmark, we generate three SIMD versions:

— `hslp`: obtained by applying `holistic SLP` extraction.

— `prop`: obtained by applying our proposed SLP extraction method.

We then compile all the SIMD versions, as well as the original (scalar) version – to which we refer by `orig` – using the target processor's with the optimization flag (-O3). We consider multiple embedded architectures presented in section 4.5.2.

Finally, we simulate each version on the target processor's simulator, using the same input data samples, and we extract the number of cycles required for executing the benchmark function.

4.5.5 Results

4.5.5.1 Proposed SLP extraction vs. `Holistic SLP`

In order to compare the proposed SLP extraction method (`prop`) against `holistic SLP` (`hslp`), we apply both SLP extraction flows ten times each (to capture the impact of the random selection factor in the group selection procedure discussed in section 4.4.2.1). And that is for all the benchmarks with 16-bit integer data-types. We report the average execution time improvement for `hslp`, and `prop` over `orig` (across the ten runs), in the histograms of figs. 4.19a, 4.19b, 4.20a and 4.20b for the targets XENTIUM, KAHRISMA, St240 and VEX-4 respectively. The minimum and maximum execution time improvement values over the ten runs are also reported

in the histograms.
The execution time improvement of each SIMD version, v, over the original version (`orig`) is computed as follows:

$$Improvement(v) = 100 * \frac{cycles(orig) - cycles(v)}{cycles(orig)} \tag{4.15}$$

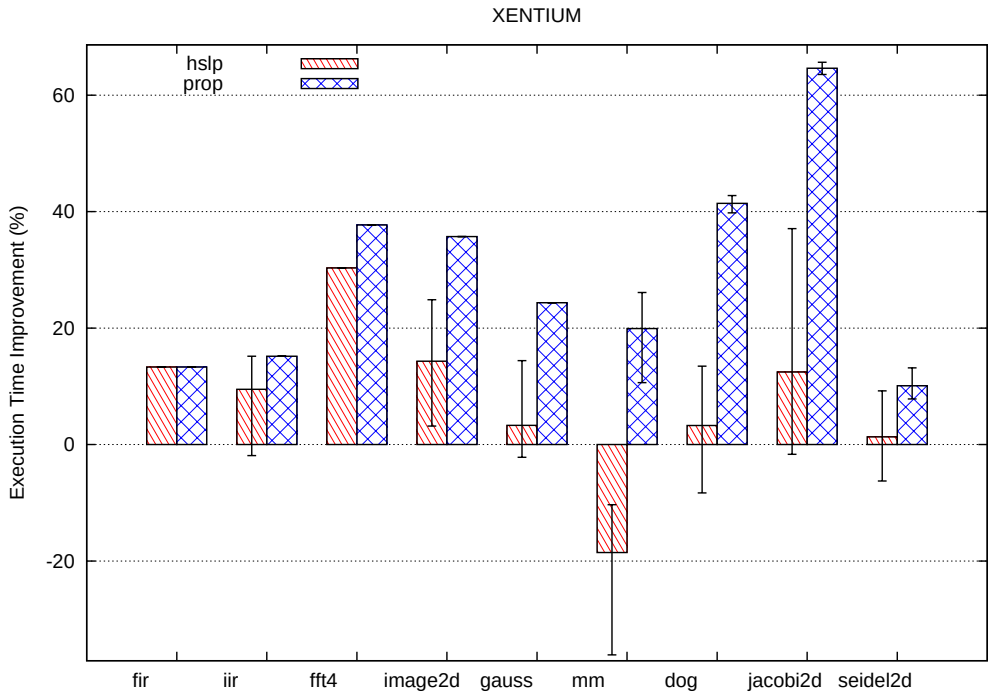
Note that for `prop`, we did not apply batching for PFCG construction (cf. section 4.4.3.1), and we used distance-3 neighborhood for candidate benefit estimation (cf. section 4.4.2.1).

Results on all targets show a consistent trend of `prop` performing better than `hslp` for all benchmarks.
For **fir**, both versions perform the same on all targets and are not affected by the random selection factor, simply because this benchmark is very simple and both methods always find the best possible grouping solution. Though on ST240 the scalar performs better than the both SIMD versions.
For **iir**, **image2d**, **gauss** and **jacobi2d**, `prop` consistently performs better on all targets with near zero variation across different runs. Whereas `hslp` performance varies significantly between runs, but on average it still performs better than the scalar version. This is because `hslp` candidate benefit estimation is unable to distinguish between candidates leading to a random group selection.
For **fft4**, both SIMD version performs good compared to the scalar version on all targets and across all runs. Though, `prop` still performs better. Note that `hslp` performance does not vary across runs. This is because of the operations scheme of **fft4**, which despite of `hslp` being sometimes unable of distinguishing between candidates, it still end up with the same grouping solution, mainly due to the high number of contiguous array accesses as shown in table 4.2.
For **mm**, `hslp` performs very poorly, it degrades performance compared to the scalar version on all targets (up to 20% worst) and varies very significantly between runs. On the other hand, `prop` consistently performs significantly better than the original version on all targets (up to 30% better).
For **dog** and **seidel2d**, the results vary across targets. `prop` results in significant performance improvement on XENTIUM for **dog** (40%), slight improvement on KAHRISMA (10%) and slight degradation on ST240 (3%). Whereas `hslp` often results in performance degradation.

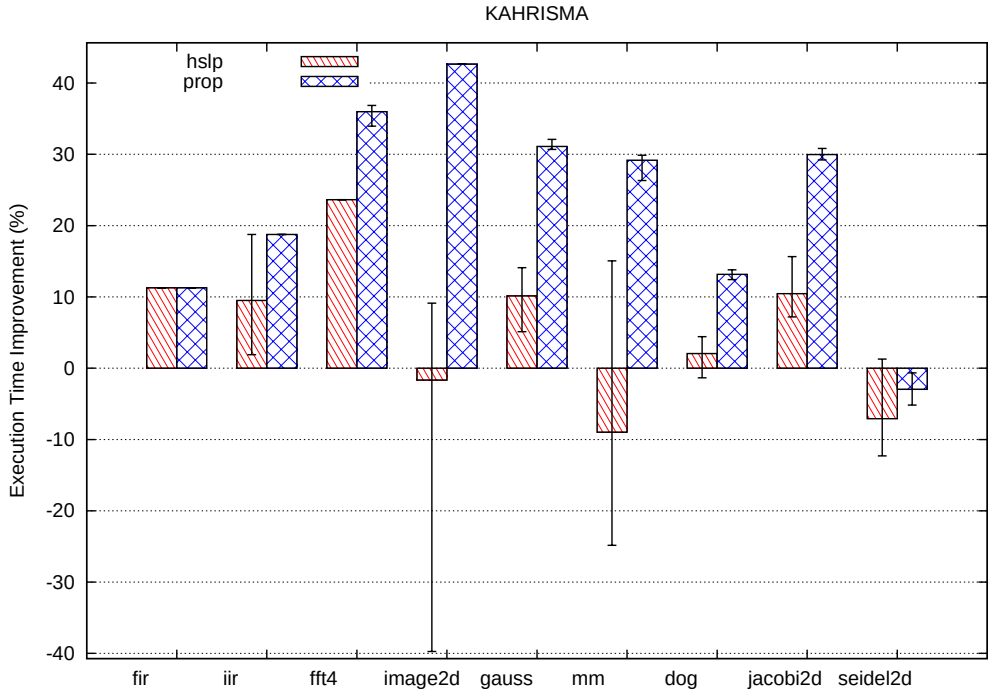
All in all, these results point towards two main conclusions:

— `prop` performs better on all benchmarks and almost always lead to performance improvement over the scalar version on all three targets. Also it has very slight variations across different runs. This indicates that the candidate estimation and group selection method is precise and accurate enough.

— On the other hand, `hslp` performance varies significantly across runs, on all targets, and often lead to performance degradation compared to the scalar version. This indicates that the `hslp` candidate benefit estimation and group selection method is not accurate/precise enough.

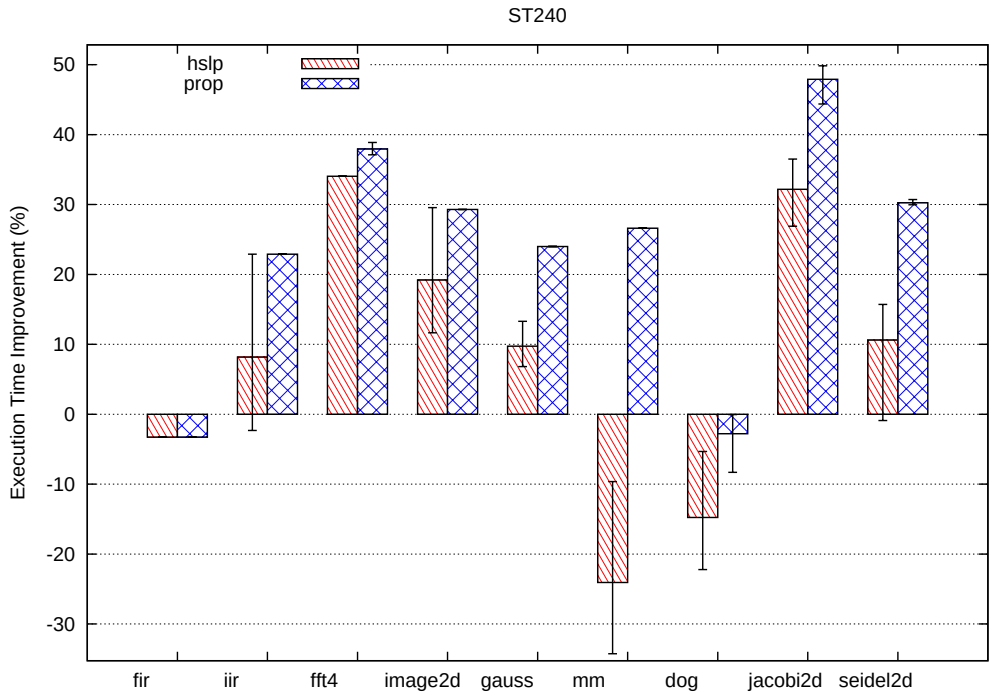


(a) Running on XENTIUM.

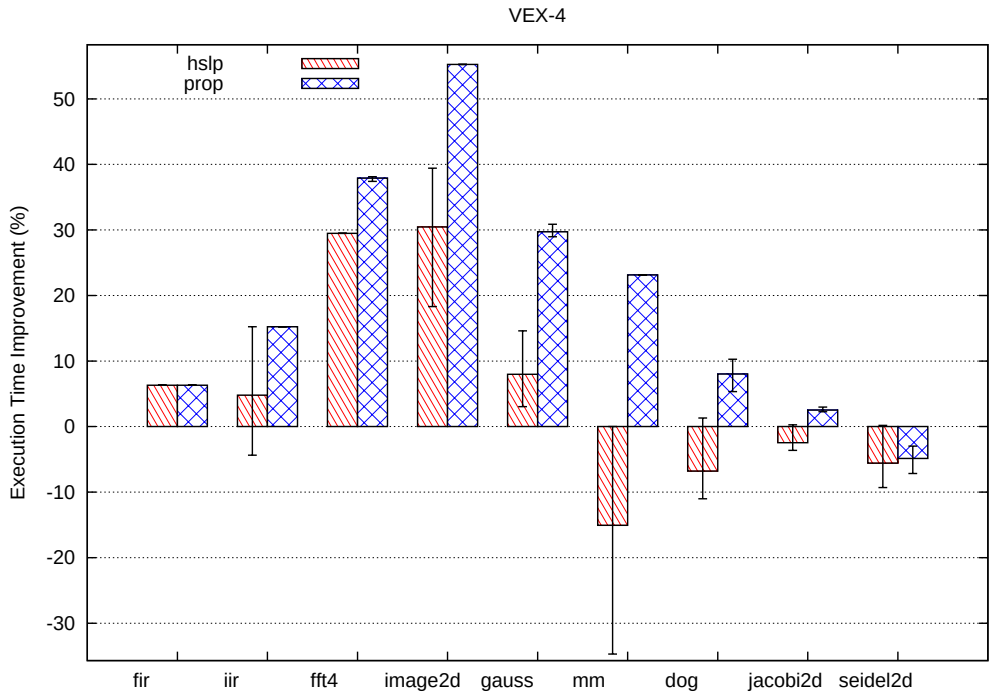


(b) Running on KAHRISMA.

Figure 4.19 – Execution time improvement of the SIMD code obtained by `prop` vs. `hslp`, over `orig`. The test is repeated 10 times for each benchmark. A bar represent the mean value and a line segment represents the minimum and maximum values of the execution time improvement.



(a) Running on ST240.



(b) Running on VEX-4.

Figure 4.20 – Execution time improvement of the SIMD code obtained by `prop` vs. `hslp`, over `orig`. The test is repeated 10 times for each benchmark. A bar represent the mean value and a line segment represents the minimum and maximum values of the execution time improvement.

4.5.5.2 Simdization Time vs. Performance

In order to evaluate the PFCG size control strategy presented earlier in section 4.4.3.1, we apply both `prop` and `hslp` on the **jacobi2d** benchmark for different batch size values and we measure the time it takes for group selection to finish. We refer to this as Simdization time. It includes PFCG construction and groups selection. We then generate, compile and simulate the generated SIMD code for each approach and report the execution time improvement (i.e. performance improvement) compared to the original (sequential) version, `orig`.

The Simdization time and performance improvement variations in respect to the batch size, for the benchmark **jacobi2d** running on KAHRISMA, are reported in figs. 4.21 and 4.22a for `prop` and `hslp`, respectively.

The Simdization time increases, roughly, linearly for `prop` and quadratically for `hslp`, with the increase of the batch size value and consequently with the number of candidates [14]. This results are in sync with the complexity analysis of `prop` and `hslp`, discussed earlier in section 4.4.3.

Besides, for `prop` the performance improvement increases logarithmically with the batch size, while Simdization time increases linearly. This means that selecting smaller batch sizes, down to a certain threshold (about 20 for **jacobi2d**), yields more Simdization time savings than performance penalties.

All in all, this results show that smaller operation batch sizes can effectively be used to shorten the Simdization time, which varies linearly. This does of course affect the performance but with a lesser degree, as can be clearly seen for **jacobi2d**. As a rule of thumb, the batch size value should be higher than half the size of the largest isomorphic operations set.

Note that the source-to-source compilation flow was implemented with no regard for the performance (Simdization time). It is implemented in Java and integrated to GeCoS which is an eclipse application that heavily relies on the Eclipse Modeling Framework (EMF). Therefore, the absolute values of the Simdization time are not very representative and can be reduced by using an optimized implementation, but the main point here is the trending of these values with respect to the batch size.

14. The number of candidates roughly increases linearly with the increase of the batch size, as shown in section 4.4.3.1.

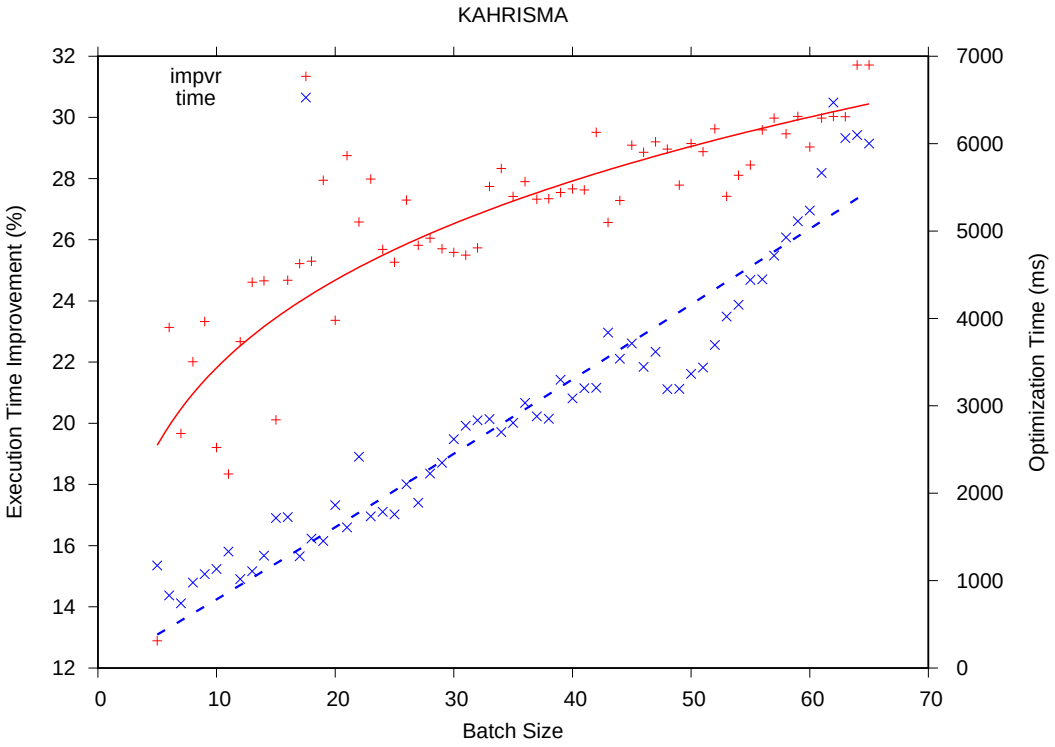
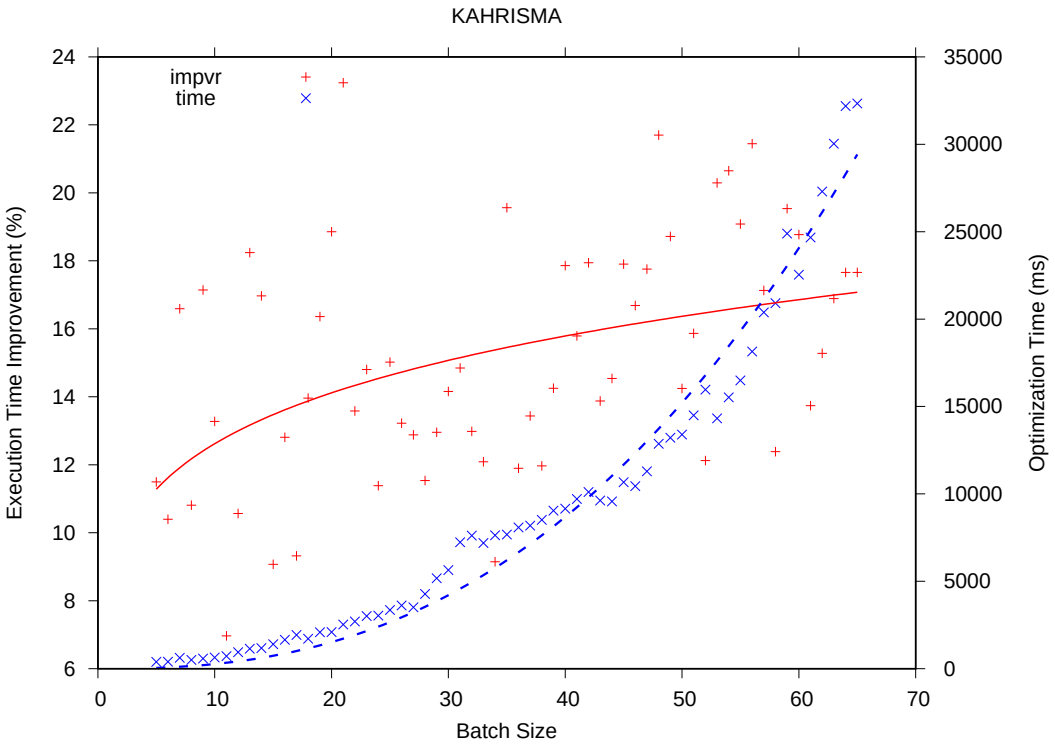


Figure 4.21 – Using `prop`.



(a) Using `hslp`

Figure 4.22 – Execution time improvement and Simdization time variation with respect to the batch size, for the **jacobi2d** benchmark running on KAHRISMA.

4.6 Conclusion

In this chapter, we presented the state-of-the-art of SLP, we throughly discussed `holistic SLP`, a SLP extraction algorithm proposed by Liu et al in 2012 [78], and shed light on some of its shortcomings. We then proposed several modifications to overcome them and improve its efficiency. More specifically:

— We proposed a new more compact and more efficient IR for SLP extraction, the PFCG.

— We proposed an alternative SIMD group selection and candidate benefit estimation method based on PFCG.

— We implemented and compared the propose solution against `holistic SLP` on several embedded processors using a set of signal and image processing benchmarks.

Experimental results showed a significant performance improvement of the obtained SLP solution, by using our proposed method compared to `holistic SLP`, without the need for data layout transformations. The proposed PFCG helps to speedup the candidate benefit estimation and consequently reduces the complexity of the group selection algorithm. In addition, the proposed operation batching method can effectively be used to reduce the size of the PFCG; the latter varies linearly with the value of the batch size. This helps reducing the group selection time, while still achieving good SLP solutions.

By this point, we have at our disposal a complete source-to-source flow capable of automatically generating a SIMD C code for different embedded processors. In the next chapter, we discuss the interaction between floating-point to fixed-point conversion and SLP extraction. We show that, existing approaches considering both transformations independently, yields less efficient solutions. We argue that considering SLP extraction in conjunction with Word Length Optimization (WLO) helps achieving better results. So, we propose a new SLP-aware WLO algorithm, which we integrate into the source-to-source compilation flow that we implemented.

Limitations

In this work we ignore the alignment constraint during group selection, we assume that all contiguous array reference superwords are aligned. When such superword is not actually aligned it induces a cost penalty, especially if the target does not support unaligned memory accesses. However, this cost penalty is generally not as high as in the case where the data are not contiguous, as previously illustrated in fig. 3.5. Indeed, a contiguous but unaligned data can be packed into a SIMD register by the mean of two vector loads followed by a permutation operation (assuming the target does not support unaligned accesses), whereas packing non-contiguous data may require as many loads as the number of elements in addition to several packing/permutation operations in order to pack all elements into one SIMD register (assuming the target does not support scatter/gather memory accesses).

The alignment problem can also be dealt with using post-optimization of the memory layout (such as array padding) and/or using loop transformation (and register shuffling like in [31]). In

anyway, this is worth considering in a future work.

In addition, in this work we only consider "simple" integer arithmetic SIMD operations (add, mul,...). We also use a very basic instruction selection procedure for SIMD code generation. However, many embedded processors (DSPs), such as XENTIUM and ST240, provide multiple "complex" SIMD operations, such as *muladd* and *addsubb*. Considering such SIMD operations during the Simdization would provide more alternatives to improve performance. This could be considered in the SLP extraction phase, during SIMD group candidates identification. Alternatively, a more complex SIMD instruction selection can be also used. This is also worth considering.

Besides, the performance of SLP solutions can be improved by considering inter basic-block (and inter-procedural) analysis to reduce packing/unpacking overhead by reusing superwords across basic blocks.

Finally, in this work we only targeted embedded processors with *subword* SIMD support. It would be interesting to see how the proposed SLP extraction method performs when targeting processors with wider SIMD data-path, such as the NEON extension for ARM processors.

Chapter 5

SLP-aware Word Length Optimization

Contents

5.1 Introduction 92
5.2 Motivations and Related Work 94
5.3 Joint WLO and SLP extraction 96
5.3.1 Overview and Intuition 97
5.3.2 Processor Model 98
5.3.3 SLP-aware WLO algorithm 101
5.3.4 Accuracy-aware SLP extraction algorithm 106
5.3.5 SLP-aware Scalings Optimization 109
5.4 Source-to-source Compilation Flow 112
5.4.1 Flow Overview 112
5.4.2 Integer Word Length determination 114
5.4.3 Accuracy evaluation 115
5.4.4 Select and sort Basic-blocks for SLP 115
5.4.5 Fixed-point/SIMD Code Generation 115
5.5 Experimental evaluation and Results 116
5.5.1 Experimental Setup 116
5.5.2 `WLO-then-SLP` source-to-source flow 117
5.5.3 Benchmarks 118
5.5.4 Target Processors 119
5.5.5 Results 119
5.5.6 Floating-point vs. Fixed-point 121
5.5.7 `WLO+SLP` vs. `WLO-then-SLP` 121
5.6 Conclusion 123

5.1 Introduction

Unlike general purpose processors, embedded processors must satisfy increasing need for computation power while complying to strict design constraints such as cost, performance and power consumption. Even though, many embedded processors, such as ARM cortex-A, provide hardware support for floating-point arithmetic, a good number of ultra low power embedded processors, such as ARM cortex-M0/1/3, TI TMS320C64x and Recore XENTIUM [104] do not, in order to reduce die area (cost) and/or minimize power consumption. This comes for the cost of restraining programmability to the use of fixed-point arithmetic, while application prototyping in many domains, such as signal processing and telecommunication applications, employs floating-point representation.
Although floating-point can be soft-emulated on such processors, it drastically degrades performance (up to 45x as reported in section 5.5.6). Instead, fixed-point implementations are preferably used. Hence, floating-point to fixed-point conversion is a crucial step for an efficient implementation when targeting such processors.

On the other hand, most embedded processors nowadays provide support for (subword) Single Instruction Multiple Data (SIMD) as a mean to improve performance for little cost overhead. In order to make efficient use of such processors, the software should also exploit SIMD opportunities.
As discussed previously, floating-point to fixed-point conversion (see chapter 2) and Simdization (see chapter 3) are tedious, error prone and time consuming transformations. Therefore, automated methodologies and tools are necessary in order to cut down applications time-to-market and consequently development cost.

Floating-point conversion consists of attributing a fixed-point format for each floating-point data and operation's intermediate result. The fixed-point format should specify the word-length in addition to the binary-point position.
Keeping full operations precision, in a fixed-point implementation, requires large word-lengths which, unless supported by the target processor, would also require costly software emulation, yielding a poor performance. Rather, quantizations are applied to limit word-lengths growth and consequently improving performance, for the cost of degrading the application's quality by introducing errors. Overflow errors, while significant, can be easily avoided by employing a conservative approach for Integer Word Length (IWL) determination. Quantization errors, however, propagate throughout the system, get amplified and can lead to significant error on the output. To avoid that, the conversion process must account for these errors and carefully select fixed-point formats that keep the computation's accuracy within an "acceptable" limit while maximizing performance.

This performance/accuracy trade-off has been identified and exploited when targeting customizable architectures, in the context of HLS [65] for instance, where the designer has more flexibility in customizing word-lengths supported by the architecture.
This is not the case when targeting processors with fixed data-path size. Employing smaller data

sizes – in the absence of SIMD – does not really benefit the application's performance. On one hand, it may require additional operations to perform data-type conversions, since all integer operations are performed on operands with the native word-size anyway (type promotion). But on the other hand, it may reduce the memory footprint which can improve performance. For example, in the C code snippet of fig. 5.1, the addition's operands are generally promoted, by the compiler, to `int` before performing the operation and the result is then converted back to `char`.
So all in all, it does not make much sense to use smaller word-lengths in this context.

```
char a,b,c;
...
c = a + b;
```

Figure 5.1 – C code snippet.

However, since most embedded processors nowadays provide support for SIMD, using smaller data word-lengths can be exploited by SIMD instructions to perform an operation on several packed data simultaneously, thus ultimately improving performance.

Intuitively, in this context using narrower word-lengths should normally translate to better performance on one side, due to the increased vectorization factor, but lower quality on the other side due to the reduced data precision. Previous work [85] followed this intuition when applying floating-to-fixed-point conversion. They aim at minimizing word-lengths without taking into account Superword Level Parallelism (SLP), which can be applied, independently, later on.

However, this intuition is unrealistically optimistic since, selecting narrower word-lengths during Word Length Optimization (WLO) does not necessarily result in performance improvement after applying SLP extraction, mainly because WLO is unaware of SLP grouping possibilities and the associated overhead.

In this chapter, we address these problems by jointly considering SLP extraction and WLO. More specifically:

— We propose a new SLP-aware WLO algorithm. To the best of our knowledge this is the first work to jointly consider both, WLO and SLP extraction.

— We implement it as a fully automated source-to-source compilation flow with a customizable and extensible fixed-point/SIMD back-end supporting multiple target architectures.

— We test our approach on several embedded processors against some signal processing applications.

The chapter is organized as follows:
First, we illustrate the aforementioned problems and discuss the related work in section 5.2. Then we present our proposed joint WLO and SLP extraction approach in section 5.3. In section 5.4, we present the fully-automated, floating-to-fixed-point conversion and SLP extraction source-to-source compilation flow. Finally, we present the experimental setup and results in section 5.5.

5.2 Motivations and Related Work

In many application domains, such as signal processing, floating-point arithmetic is used for rapid prototyping. When targeting embedded SIMD processors with no floating-point support, a floating-point to fixed-point conversion step is crucial to achieve good performance, by:

— Eliminating the need for software floating-point emulation which drastically impact performance.

— Reducing data sizes and consequently reducing the memory footprint, which may also improve performance.

— Enabling the exploitation of SIMD capabilities which can further improve performance.

The down side is mainly the complexity of fixed-point implementation.

In this context, floating-point conversion can be either applied manually or using automated tools as discussed in chapter 2.
Manual conversion is very tedious, error prone and time consuming, which limits the ability to explore the fixed-point design space, and generally pushes toward using uniform word-length fixed-point representations using the target's native word size. This often leaves no room for Simdization when targeting subword Multimedia extensions and limits SIMD opportunities when targeting superword Multimedia extensions (see section 3.2.2).

On the other hand, automatic floating-point to fixed-point conversion allows for better and faster exploration of the fixed-point design space, but existing methodologies and tools, at best, only exploit the fact that SIMD processors can operate on different data word-lengths and assumes that selecting smaller word-lengths yields better performance without considering the effective consequences of such choices on Simdization and ultimately on performance.

Cilio and Corporaal [26] presented an approach for automatic conversion of floating-point C code into fixed-point C code targeting embedded processors. The IWLs of some variables are specified by the user via `pragma` annotations. This information is propagated to determine the IWLs of the remaining variables. The proper scaling operations are then automatically inserted. Kum et al.[64] proposed a similar method except they used simulation to determine IWLs. Also, they proposed a scaling optimization algorithm that minimizes a scaling cost function, taking into account whether or not the target processor has a barrel shifter, using integer linear programming or simulated annealing.
Both of these approaches, only use the target's native word-length and no WLO is performed. Therefore, they are unable to take advantage of the SIMD capabilities of a processor.

In fact, during WLO the goal is to minimize a certain cost function – generally representing the performance (execution time) – as much as possible while keeping the accuracy within an acceptable limit. The quality of the cost evaluation or its ability to properly distinguish between two fixed-point solutions is crucial, since it is the main driver of the WLO decisions (along with the accuracy constraint).
A straightforward way to evaluate the cost associated with a fixed-point solution, while considering Simdization, could be obtained using simulation. The tested fixed-point solution needs

to be implemented, compiled (with applying Simdization) and run on the target processor, or simulated, with representative data samples in order to obtain a performance estimation. While accurate, this method is extremely slow, rendering it unpractical for design space exploration.
Alternatively, the tested fixed-point solution could be only compiled and then the execution time can be estimated using a cost model. Again, this method is very slow and unpractical, specially if optimizations such as Simdization are to be applied during compilation for each tested fixed-point solution.

Following the same logic, Menard et al proposed a simpler and less aggressive method [85]. It consists of considering all supported instructions, including SIMD, that can implement a given operation in the system. The goal of their proposed WLO is to select, for each operation, the instruction (associated to a word-length) which minimizes the overall execution time, subject to an accuracy constraint. The execution time is estimated using a simple model. The execution time associated with an instruction is provided by the processor's model. In case a SIMD instruction is selected to implement an operation, its execution time is simply divided by the maximum number, say N, of operations that can be executed in parallel by this instruction.
For example, when targeting a 32-bit processor with all instructions having an execution time of 1. If a 32-bit (scalar) instruction is selected to implement an operation, then the operation's execution time is 1. However, in case a 16-bit SIMD instruction is selected then its execution time is 1/2.
This implies two main assumptions:

— First, it assumes that when an SIMD instruction is selected to implement a (scalar) operation, N similar operations will ultimately be executed in parallel –if Simdization is later applied– by the same instruction, without actually knowing whether or not this is possible. Hence, the operation's execution time is estimated as $1/N$ th of the instruction's execution time provided by the model.

— Second, it assumes that no overhead is associated with a SIMD instruction. In another word, it completely ignores the overhead associated with the required data packing/unpacking operations.

These assumptions are very optimistic and unrealistic.

Example 10 *To illustrate this, let's consider the dummy example of fig. 5.2. For the sake of illustration, let's assume that WLO decides to attribute the following word-lengths:*

— *16-bit for* $A[]$*,* x*, operations* 1 *and* 3

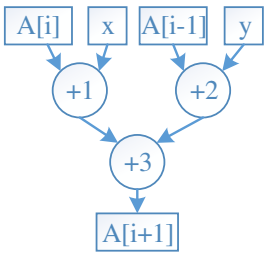
— *32-bit for* y *and operation* 2

Assuming a 32-bit target processor with 2x16-bit SIMD support.
If we consider applying loop vectorization after floating-point to fixed-point conversion. The for loop cannot be vectorized since it caries a dependency on A*.*

Also, if we consider applying SLP extraction on the loop's body, whose Data Flow Graph (DFG) is depicted in fig. 5.2. No SIMD group candidates are available because 1 *and* 2 *have different word-lengths, and* 1 *and* 3 *have a dependency.*

```
float A[N], x, y;
...
for(int i=1; i<N-1; i++)
   A[i+1] = (A[i] + x) + (A[i-1] + y);
```

(a) C code snippet.



(b) Loop's body DFG.

Figure 5.2 – Motivating example.

So, in both cases the assumptions made by WLO are wrong since Simdization of any of the operations is not possible.

However, if WLO would have selected operations 1 *and* 2 *to be 16-bit, SLP extraction would be able to pack them together into a SIMD group. But again, the overhead associated with packing/unpacking operation could be very high.*

The bottom line is that, unless WLO is aware of the Simdization opportunities and the associated cost, it has no way of predicting the impact of the decisions it takes on SIMD exploitation, and consequently on performance.
In this work, we propose to jointly consider Simdization and WLO as a solution to this problem. We focus on SLP extraction as Simdization method and we present a SLP-aware WLO algorithm.

5.3 Joint WLO and SLP extraction

Applying WLO without taking into account SLP extraction constraints will most likely yield inefficient solutions. This is because WLO decisions directly dictate the search space of SLP extraction. It may prevent, otherwise possible, SIMD grouping candidates by selecting, for instance, different word-lengths for operations that can be, otherwise, grouped together [1].

Besides, it is very important to note that WLO is performed under accuracy constraint. In other words, only a limited accuracy-degradation *budget* can be used to try to improve performance as much as possible. In this context, one of the main factors impacting performance is how well SIMD capabilities are being exploited. It is not wise for the WLO to spend the accuracy-degradation budget on optimizing operations that cannot be efficiently exploited by SLP to improve performance. If WLO is unaware of SIMD grouping possibilities and their associated overhead, it will blindly optimize word-lengths of operations that may not end-up in a SIMD group, either due to dependencies or conflicts with other groups. Even worse, it may optimize operations which lead to a SLP solution with a high packing/unpacking overhead.

Moreover, the order in which the operations are to be optimized is crucial. For instance, in the example 10, we know that only operations 1 and 2 can be grouped together. If we optimize first

1. Recall that all operations in a SIMD group must have the same word-length.

the word-length of 1 to make it "fit" in a SIMD instruction and then optimize 3, the accuracy-degradation budget may run out before getting the chance to optimize 2, so that it can be grouped with 1 in the same SIMD instruction. To avoid this, 2 should be preferably optimized before 3. This is not possible unless SLP opportunities (and their associated cost) are known to the WLO process.

Besides, the scaling operations, required to correctly implement a fixed-point specification using integer arithmetic operators, have a major impact on performance. Therefore scalings are also taken into account.

In order to address these problems, we propose an approach to jointly perform floating-point to fixed-point conversion and SLP extraction. We couple an accuracy-aware SLP extraction with a SLP-aware WLO algorithm. We also propose an SLP-aware scalings optimization algorithm. In the remainder of this section we present these algorithms.

5.3.1 Overview and Intuition

Since any decision of WLO can directly affect SLP, as shown previously, we use SLP extraction to guide it through. The aim is to let WLO focus on optimizing operations that belong to SIMD groups which are selected by the SLP extraction algorithm as the "best" SLP solution, taking into account the data packing/unpacking overhead.

The problem is that SLP extraction requires to know the operations data-types and word-lengths to construct the set of group candidates. Recall that all operations in a SIMD group candidate must have the same word-length, which have to be supported by the target SIMD Instruction Set Architecture (ISA), and that the overall size should not exceed the SIMD vector size. However, before WLO is performed the word-lengths of operations are not known yet! Thus, SLP extraction depends on WLO which requires SLP solution to guide it in order to find an efficient solution.

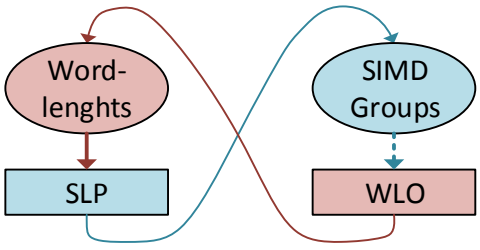


Figure 5.3 – WLO/SLP phase ordering problem.

To solve this phase ordering problem, we break the cycle by loosing the aforementioned word-length constraints on SIMD group candidates. This means we no longer impose that two operations must have the same word-length to form a group candidate. This way SLP extraction will have the "freedom" to select which candidate is most beneficial to become a SIMD group. Once SLP extraction makes it decision, the selected group is provided to WLO which sets the word-

length of all it's operations in such a way that the group becomes valid and implementable on the target processor (i.e. that the previously loosen constraints become enforced). The diagram of fig. 5.4 depicts the functional overview of the proposed approach.

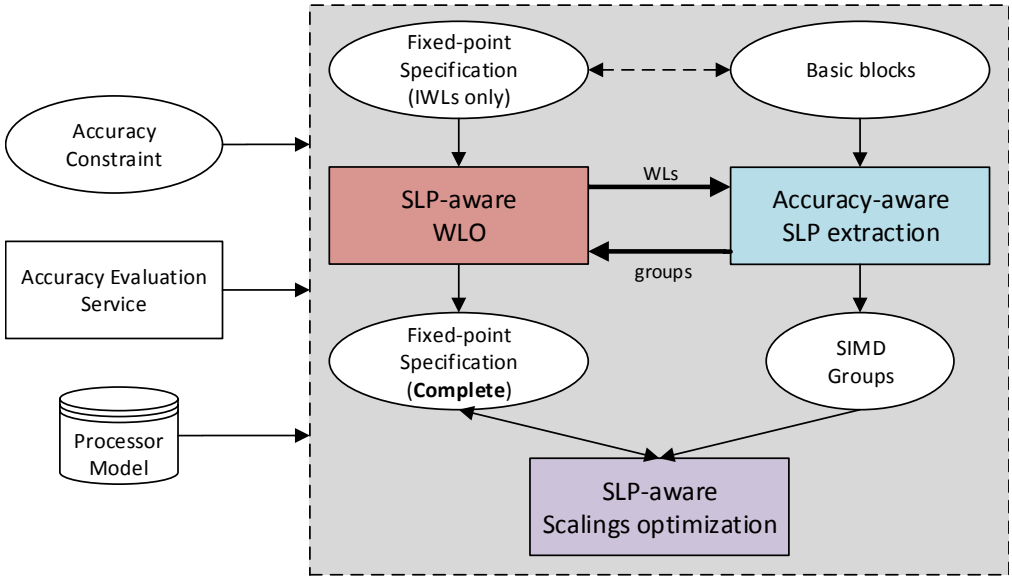


Figure 5.4 – Overview of the joint WLO and SLP extraction approach

Both, SLP extraction and WLO, require some knowledge about the target processor. In fact, SLP extraction is limited to the SIMD operations supported by the target and to the size of its SIMD data-path. For instance, if the target does not support SIMD division, then division operations will not be considered during group candidates extraction. Also, the size of a group cannot exceed the maximum size supported by the target, and its elements data-type/size must be supported by the target as well. For instance, only groups of two 16-bit elements are allowed when targeting a processor supporting only 2x16-bit SIMD. Similarly, we limit WLO to the word-lengths natively supported by the target processor, for two reasons. First, to avoid "expensive" software emulations and second, to limit the fixed-point solution space. The drawback is that it may not be always possible to find a valid solution which respects very tight accuracy constraints.

Therefore, both transformations are provided with a model of the target processor containing all necessary information. This model is presented in section 5.3.2.

5.3.2 Processor Model

A processor's model provides necessary information about the characteristics of the operations it can perform, which are needed for both SLP extraction and WLO. In this work, we model a processor by the following:

— O: the set of supported operators.

— V: the SIMD registers size.

Définition 5.1 (Operator) *An operator is specified by:*

- *T: the type of operation it can perform (add, sub, mul, shr, ...).*
- *WS: the word-lengths of all its input operands ($in_0, in_1, ...$) and output (out). For a binary operator:*
$$WS = (in_0.W,\ in_1.W,\ out.W) \tag{5.1}$$
- *N: the number of operations it can perform simultaneously (≥ 1).*
- *PM: the precision mode which determines whether the operator keeps full precision, Least Significant Bits (LSBs) only (case of typical integer operators) or Most Significant Bits (MSBs) only (case of fixed-point operators).*
- *QM: the quantification mode (truncation, rounding).*
- *OM: the overflow mode (wrap-around, saturation). This is not important in this work since we do no not allow overflows.*
- *S: the amount of pre/post-scaling for each input/output operand (0 if none, > 0 if right shift, < 0 if left shift).*
- *SIMD intrinsics for code generation.*

This model allows the representation of a wide variety of operators. It can represent scalar ($N = 1$) and SIMD ($N \geq 1$) operators. It can also represent generic integer as well as fixed-point specific operators thanks to PM, QM, OM and S parameters.

The models of the processors targeted in this work are presented in appendix A.

Multiplication Operators

For multiplication operators, the precision mode PM complements the information specified by the operand word-lengths to determine what operation is performed by the operator.

For a full-precision operator (op), the output's word-length ($out.W$) must be at least as wide a the sum of both input word-lengths:

$$op.out.W \geq op.in_0.W + op.in_1.W \tag{5.2}$$

In this case no quantization is applied by the operator and the full precision is preserved. In embedded processors, this usually corresponds to operators of size $(w, w, 2w)$, where the output is generally stored in a pair of registers; one containing the MSBs and the other containing the LSBs of the result. In the case of a SIMD operator, the results of the N operations performed in parallel might be organized in different ways in the output SIMD registers. For instance, the 2x16-bit multiplication instruction of XENTIUM returns the full-precision 32-bit result of each operation in a separate register. In case the output is used by another SIMD instruction, the LSBs (or MSBs) of both result registers must first be packed together into one register. An additional packing instruction is required in this case.

To avoid the word-length growth, integer multiplication operators generally only keep the LSBs of the result. It's up to the developer to deal with potential overflows. Assuming no

overflows, the LSBs of the result correspond to the exact result of the multiplication and the MSBs can be simply discarded. This is not the case when using such operator to implement a fixed-point multiplication. In fact, the result of the multiplication of two signed fixed-point numbers, a and b, with respective formats, $< W_1, I_1, F_1 >$ and $< W_2, I_2, F_2 >$, $c = a * b$ is of format $< W_1 + W_2, I_1 + I_2, F_1 + F_2 >$ (see section 2.3). Using a keep-LSBs (w, w, w) operator to implement such operation means that the MSBs which generally represent the integer part are discarded. The result in this case would be meaningless as illustrated in fig. 5.5. To avoid this,

Figure 5.5 – Fixed-point multiplication example.

the operand formats must satisfy the following constraints:

— I_3 must be wide enough to contain the values range of the multiplication's result (i.e. no overflow).

— $F_1 + F_2 \leq F_3$

The first condition is enforced during IWL determination (see section 5.4.2). The second condition can be enforced during WLO, by reducing the Fractional Word Lengths (FWLs) on the inputs so that their sum $(F_1 + F_2)$ "fits" in the available bits for the output's FWL, which can be determined based of the operators output word-length value (W_3) and the IWL (I_3), as such $F_3 = W3 - I3$.

In fact, when a keep-LSBs (w, w, w) operator is selected to implement this multiplication, the maximum number of bits available for the FWL of:

— a is $AF_1 = w - I_1$

— b is $AF_2 = w - I_2$

— c is $AF_3 = w - I_3$

In order to be able to correctly use this operator, the FWL of the output must not exceed AF_3 (i.e. $F_3 <= AF_3$). To get the maximum available precision we set $F_3 = AF_3$. This constraint must be satisfied, otherwise the result would be meaningless as shown above in the example of fig. 5.5. In this example actually, $w = 8, I_3 = 7$ so $AF_3 = 8 - 7 = 1$, however, $F_3 = F_1 + F_2 = 9$ thus $F_3 > AF_3$!

To enforce this constraint, the FWLs of the inputs (F_1 and F_2) must be reduced so that their sum become at least equal to F_3 (i.e. reduced by a total of $e = (AF_1 + AF_2) - AF_3$ bits. The

"extra" e bits can be entirely reduced from F_1 or F_2 or some from each. Thus, multiple solutions are possible here (they can be considered during WLO).

Reducing the FWL can be achieved by increasing the IWL by the same amount, since the FWL is implicitly determined from the values of the word-length and the IWL. For the example of fig. 5.5, $F_3 = AF_3 = 1$ and $e = (5+4)-1 = 8$. A possible solution is to remove 4 bits from the F_1 and 4 bits from F_2, as illustrated in fig. 5.6. In this case the result of the keep-LSB operator is correct.

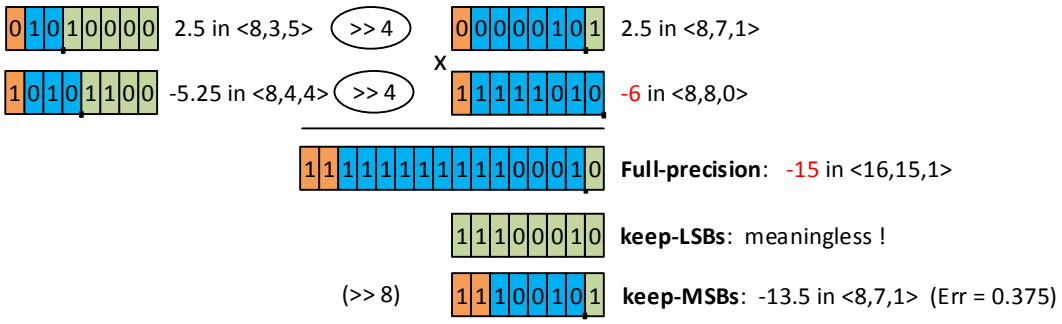


Figure 5.6 – Fixed-point multiplication example.

Alternatively, we can restrict the word-length of the input operands of a keep-LSBs (w, w, w) multiplication operator to $w/2$, thus emulating a full-precision $(w/2, w/2, w)$ operator. This solution is less flexible as it does not allow to assign more precision for the operands that needs it the most (e.g. $(w/3, 2w/3, w)$). However, it simplifies the exploration of the solution space because, unlike the first solution, it does not require to modify the IWLs during WLO. Therefore, IWLs, and consequently the scaling operations, can be determined before and remain fix during WLO.

Add/Sub Operators

As discussed previously in section 2.3, performing full-precision fixed-point add/sub operations requires the operands to be aligned to their maximum FWL. This may require increasing the operands word-length, and consequently the size of the operator, specified by WS. To avoid this, we limit the fixed-point formats of the operands in such a way to ensure that the underlying integer operation can be performed on the selected operator. We do so by aligning the operands to the *minimum* (instead of maximum) FWL; the operand with the highest FWL is right shifted by $|F1 - F2|$. In this case, up to $|F1 - F2|$ precision bits can be lost, but no overflow can be introduced, hence no word-lengths increase is required.

5.3.3 SLP-aware WLO algorithm

In this section we present the proposed SLP-aware WLO algorithm. We start by defining the solution space in section 5.3.3.1, and we present the algorithm in section 5.3.3.2. The latter relies

on an accuracy-aware SLP extraction algorithm, which we will present later on in section 5.3.4

5.3.3.1 Solution space definition

A fixed-point computation system is represented by a *fixed-point specification*. This specification is defined below:

Définition 5.2 (Fixed-point Specification) *The fixed-point specification of a system is a directed graph where a node represents either a data (definition 5.4) or an operation (definition 5.5). A fixed-point format is associated to each data node and operation's operand.*
A scaling amount (definition 5.6) is associated to each edge, representing a dependency between two nodes.
For instance, fig. 5.7 illustrates the representation of a fixed-point specification containing one operation node and three data nodes.

Définition 5.3 (Fixed-point Format) *A fixed-point format (see section 2.3) is specified by at least two of the following three parameters:*

- *Integer word-length: I (including sign bit for signed numbers)*
- *Fractional word-length: F*
- *Total word-length: $W = I + F$*

Définition 5.4 (Data Node) *A data node represents either a program variable, constant or an implicit operation's intermediate result. It may have multiple inputs and/or multiple outputs. All elements of an array variable are represented with the same data node. Only one fixed-point format is associated to a data node.*

Définition 5.5 (Operation Node) *An operation node represents an operation in the DFG. It has one data node as output and may have several data nodes as inputs. The output, and each input, have each an associated fixed-point format, which may be different from the format of the predecessor/successor data node as depicted in fig. 5.7.*

Définition 5.6 (Scaling) *To each edge is associated a scaling amount, computed based its predecessor (pred) and successor (succ) fixed-point formats:*

$$amount = pred.F - succ.F \tag{5.3}$$

If the scaling amount is zero, no scaling operation is required, otherwise, a right or left shift is necessary if the amount is respectively positive or negative.

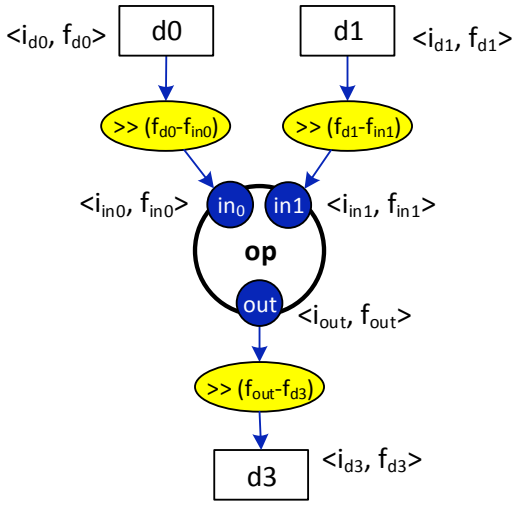


Figure 5.7 – Representation of an operation node (*op*) and its predecessors/successor data nodes (d_x) in a fixed-point specification. $< i_x, f_x >$ represents a fixed-point format with i_x representing the IWL and f_x the FWL. Elliptical nodes represent scaling operations; a negative amount corresponds to a left shift by the absolute value.

Each fixed-point format (definition 5.3) is specified by the values of its word-length (W) and IWL (I). We assume that the IWLs are pre-specified before starting WLO. The IWLs determination is presented later on in section 5.4.2.

The set of all possible word-length values, that each fixed-point format can take, is constrained by the target processor. In fact, an operation node in the fixed-point specification can be implemented using any operator, among the set of all supported operators of the target that can implement the correspondent operation type (add, mul, ...). Operators of the same type (T) generally corresponds to different word-lengths on which the operation can be performed on. For instance, when targeting ST240, providing 32-bit (scalar) and 2x16-bit SIMD add operators (see table A.2), an add operation node in the fixed-point specification, can be implemented using either operators. Therefore, the set of possible word-length values for the operation's operands is limited to $\{16, 32\}$, in this case. Even though this limitation can be lifted, by soft-emulating different precision operators for instance, we restrict word-lengths to those natively supported by the target processor. Not only to limit the solution space's size, but also to avoid the overhead associated with software emulations of the unsupported operators. In other words, we ensure that the required operators, needed to implement the fixed-point solution produced by WLO, are natively supported by the target processor.

The solution space of WLO, constructed based on the target processor's model, associates a set of possible operators for each operation in the fixed-point specification. Consequently, it associates a set of possible word-length values for each fixed-point format, in the fixed-point specification.

Algorithm 9 SLP-aware WLO algorithm pseudo-code

```
1: // Input: sorted list (by priority) of basic blocks for SLP extraction, BBs
2: // Input: fixed-point specification with IWLs pre-determined, SPEC
3: // Input: accuracy constraint, A
4: // Output: set of selected SIMD groups, G. determine word-lengths in SPEC
5: procedure SLP-WLO(BBs, SPEC, A)
6:     for each fixed-point format f in SPEC do
7:         f.W ← maximum possible word-length (in the solution space)
8:     end for
9:     for b in BBs do // visit in priority order (see section 5.4.4)
10:        G ← ∅
11:        while not done do
12:            pfcg ← BuildPFCG(b, SPEC, A) // see algorithm 11
13:            Selected ← GroupSelection(pfcg, spec, A) // see algorithm 12
14:            if Selected = ∅ then
15:                done
16:            end if
17:            Update b and prepare it for next iteration
18:            G ← G \ {e1, e2 : {e1, e2} ∈ pfcg.S}
19:            G ← G ∪ pfcg.S
20:        end while
21:        ScalOptim(G, SPEC, A)
22:    end for
23:    return G, SPEC
24: end procedure
```

5.3.3.2 Algorithm

The fixed-point specification representing the solution space along with the accuracy constraint are taken as inputs. The IWLs of all nodes in the fixed-point specification are assumed to be pre-determined. This step is detailed later on in section 5.4.2. The accuracy constraint, specified by the user, represents the maximum allowed noise power of the quantization error at the system's output.

The pseudo-code of the SLP-aware WLO algorithm is listed in algorithm 9.

Step 1: We start by selecting the highest precision solution available in the solution space. We do so by setting the word-length of each fixed-point format to the maximum value, in the associated set of possible word-lengths present in the solution space (lines 6-8). This solution generally corresponds to the case where minimum SLP is available, but on the other hand, it represents the highest precision fixed-point specification, that can be obtained using natively supported operators (word-lengths).

Step 2: Next, we process each basic block to be considered for SLP extraction, starting by the higher priority ones. Priority here depends on the contribution of the basic block to the overall execution time; a higher priority is attributed to the one that occupies more of the execution time. This is to ensure that the accuracy-degradation budget is wisely spent on optimizing most performance-impacting basic blocks first. The selection and sorting of basic blocks to be considered for SLP extraction is performed beforehand. We discuss this step in section 5.4.4.

For each basic block we iteratively apply the accuracy-aware SLP extraction (lines 11-20). At each iteration, we build the Pack Flow and Conflict Graph (PFCG) (line 12), representing the SLP solution space, and then we use the SLP extraction procedure to obtain the set of selected SIMD groups (line 13).

For SLP extraction we adapt the algorithm that we proposed earlier in chapter 4 to the accuracy constrained context of this work. This accuracy-aware SLP extraction algorithm is detailed in section 5.3.4. In a nutshell, when building the PFCG we drop the data-size constraints and we make sure that each SIMD group candidate can be selected without violating the accuracy constraint. Whenever a new SIMD group is selected, all the operation nodes (in the fixed-point specification) it contains are assigned the same operator (from the target model), which is capable of implementing the overall SIMD operation with the highest possible precision. Let:

- $spec = (nodes, edges)$ be a fixed-point specification.
- $g = \{e_1, e_2, ..., e_n\}$ be a selected SIMD group, e_i is an element in g, it represents an (scalar) operation node: $e_i \in spec.nodes$. All elements in g have the same operation type, t, by definition (see definition 5.7).
- $O = (o_1, o_2, ..., o_k)$ the list of all operators, of type t, available in the target processor model. The list is ordered by increasing precision, as such $precision(o_i) \leq precision(o_{i+1})$.

Initially (in step 1), all operation nodes in *spec* are assigned the highest precision operator of the corresponding type. For e_i nodes, this correspond to the operator o_k. Now when g is selected, the operator $o_j \in O$ is assigned for all nodes e_i ($i \in [1, n]$), such that:

$$\begin{cases} j \in [1, k] \\ o_j.N \geq n \\ j \geq x, \forall x \in [1, k] : o_x.N \geq n \end{cases} \tag{5.4}$$

In other words, o_j is the highest precision operator that can implement g. This corresponds to the procedure SetMaxWL, whose pseudo-code is listed in 10. In the remainder of this chapter, we will assume that this action is performed whenever we say a group is *selected*.

Example 11 *To illustrate this, let's consider the KAHRISMA processor's model (see table A.3) which supports 1x32-bit (o_3), 2x16-bit (o_2) and 4x8-bit (o_1) add operators. In this case, if a SIMD group of two add operations, $\{e_1, e_2\}$ is selected, then the operator o_2 is assigned for both e_1 and e_2.*

The elements of each selected group are then replaced by a new operation (representing the

Algorithm 10 Set the word-length of a SIMD group to the maximum possible

```
1: // Input: fixed-point specification, SPEC
2: // Input: a SIMD group, c
3: procedure SetMaxWL(c, SPEC)
4:     o ← highest precision operator that can implement c // as defined in eq. (5.4)
5:     for operation node e ∈ c (e ∈ SPEC.nodes) do
6:         assign Operator o to e in SPEC // this sets the word-lengths of the fixed-point
   formats of e's operands according to o.WS.
7:     end for
8: end procedure
```

group) in the basic block (line 17) in order to prepare for the next SLP extraction iteration, which allows the extension of the groups size when possible, otherwise the processing of the basic block is completed and we move to the next one (lines 14-16).

Step 3: The global set of selected groups (G) is updated after each iteration (lines 18-19). It is used to optimize scaling operations (line 21). The scaling optimization algorithm is presented in section 5.3.5.

As an output we obtain a complete fixed-point specification for the system, along with the set of selected SIMD groups. These information are finally used to generate fixed-point and SIMD C code, as presented in section 5.4.5.

5.3.4 Accuracy-aware SLP extraction algorithm

For SLP extraction we adapt the algorithm that we proposed earlier in chapter 4 to the accuracy constrained context of this work.

Building PFCG

First, we adapt the PFCG [2] construction procedure. The corresponding pseudo-code is listed in algorithm 11.

When constructing the PFCG of a given basic block, we identify the set of SIMD group candidates (C) as well as the conflicts (X) between them. Recall that, a group candidate (definition 4.3) is a pair of isomorphic and independent operations with same size yielding superwords that does not exceed the SIMD data-path size. And, that two groups are in conflict (definition 4.5) if they contain the same operation or if they cause a cyclic dependency.

However, in the context of this work, the word-lengths are not known at this level. Therefore, we loose all size constraints (line 6) when identifying group candidates. A candidate is thus defined as specified by Definition 5.7 .

2. PFCG is the Intermediate Representation (IR) we proposed in order to represent the SLP extraction solution space. See section 4.4.1.

Définition 5.7 (SIMD group candidate) *In this accuracy-constrained context, a SIMD group candidate is a pair of independent operations of the same type (add, sub, mul, ...). No constraints are imposed on the word-lengths.*

Algorithm 11 Adapt Building PFCG (algorithm 2)

```
1:  // Input: Data Flow Graph dfg. It is a Directed Acyclic Graph (DAG)
2:  // Output: Pack Flow and Conflict Graph pfcg
3:  // EvalAcc(SPEC): evaluates the accuracy of the current fixed-point specification
4:  procedure BuildPFCG(dfg, SPEC, A)
5:      for each pair (o1, o2) of isomorphic operations or contiguous accesses to same array in dfg do
6:          if o1 and o2 are independent then
7:              c ← {o1, o2}
8:              // make sure c is valid
9:              SetMaxWl(c, SPEC)
10:             if EvalAcc(SPEC) violates A then
11:                 skip c
12:             end if
13:             revert WL of c
14:             pfcg.C ← pfcg.C ∪ {c}
15:             // ... connecting c's flow edges is omitted; same as algorithm 2
16:             // connect c's conflicts
17:             pfcg.X ← pfcg.X ∪ {{c, c'} : c' ∈ pfcg.C and (c' ∩ c ≠ ∅ or c', c introduce a cyclic dependency) }
18:             SetMaxWl(c, SPEC); SetMaxWl(c', SPEC);
19:             if EvalAcc(SPEC) violates A then
20:                 pfcg.X ← pfcg.X ∪ {{c, c'}}
21:             end if
22:             revert WL of c, c'
23:         end if
24:     end for
25:     return pfcg
26: end procedure
```

Furthermore, in this accuracy constrained context, some candidates in C may not be valid. This is the case of a candidate c, that if *selected*[3] while all other candidates are not *selected*, the accuracy constraint is violated. In this case, c can never be implemented, using an available SIMD operator, without violating the accuracy constraint. Hence, we eliminate all invalid candidates in C (lines 8-13).

Besides, X does not represent all the conflicts in this context. In fact, two candidates are in conflict if: when both are *selected* while all other candidates are not *selected*, the accuracy

3. Recall that when a group is selected, the word-length of all the operation nodes it contains is set as specified earlier by eq. (5.4).

constraint is violated. In this case both candidates cannot coexist without violating the accuracy constraint, hence they are considered in conflict (lines 34-38). We refer to this type of conflicts as *accuracy conflict*.

SIMD Group Selection

After the PFCG is properly built, the group selection estimates the benefit associated with each remaining candidate and then iteratively selects the most beneficial one, g. This procedure is the same as presented earlier in section 4.4.2. the pseudo code is recalled in algorithm 12.

Algorithm 12 Recall Group Selection Procedure from algorithm 1

```
1: // Input: pfcg, is the PFCG.
2: // Input: SPEC, is the fixed-point specification.
3: // Output: Set of selected groups.
4: procedure GroupSelection(pfcg, SPEC)
5:     while pfcg.C ≠ ∅ do
6:         for each candidate c ∈ pfcg.C do
7:             BenefitEstimation(c, pfcg, SPEC)
8:         end for
9:         best ← Select(pfcg).
10:        UpdatePFCG(best, pfcg) // mark best as selected and Update pfcg
11:        EliminateCycles(pfcg)
12:    end while
13:    return pfcg.S
14: end procedure
15:
16: // Output: Set the benefit of candidate c
17: procedure BenefitEstimation(c, pfcg, SPEC)
18:    sub ← ExtractSubPfcg(c, pfcg) // see algorithm 3
19:    EliminateSubConflicts(c, sub, SPEC) // see algorithm 13
20:    ComputeBenefit(c, sub, pfcg)
21: end procedure
```

However, the candidate benefit estimation method is modified to adapt to the accuracy-constraint context of this work.

First, after extracting the sub PFCG graph of a give candidate c, $sub(c, N)$ (representing the distance-N pack flow neighborhood of c), we make sure the remaining candidates in $sub(c, N)$ can all coexist without violating the Accuracy constraint. The pseudo-code of this procedure is listed in algorithm 13.

Then, we also modify the computation of the benefit associated with the selection of c in order to take into account its impact of the accuracy. The pseudo-code of the modified candidate benefit estimation is listed in algorithm 14. We measure the deferential of accuracy, *deltaAcc*, before and after (temporarily) *selecting* all remaining candidates in $sub(c, N)$ (lines 10-17).

Finally, we modify the "best" candidate selection procedure. The corresponding pseudo-code

Algorithm 13 Eliminate Conflicts in *sub*

1: // **Input**: c, a SIMD group candidate.
2: // **Input**: *sub*, a subgraph of the PFCG; it is the distance-N pack flow neighborhood of c.
3: // **Input**: $SPEC$, the fixed-point specification.
4: **procedure** EliminateSubConflicts($c, sub, SPEC, A$)
5: apply conflict elimination as in algorithm 4
6: // Make sure the remaining candidates can all coexist without violating the Accuracy constraint.
7: **for** $n \in sub.\mathcal{V}$ **do**
8: $SPEC$.save(n)
9: SetMaxWl($n, SPEC$)
10: **end for**
11: **while** EvalAcc($SPEC$) violates A **do**
12: $x \leftarrow$ node $\in sub.\mathcal{V} \setminus \{c\}$ with highest $\frac{deltaACC}{1 + flow\ degree}$
13: $sub.\mathcal{V} \leftarrow sub.\mathcal{V} \setminus \{x\}$ // set of nodes
14: $sub.X \leftarrow sub.X \setminus \{\{x, y\} \in sub.X, \forall y\}$ // set of conflict edges
15: $sub.F \leftarrow sub.F \setminus \{(x, y), (z, x) \in sub.F, \forall y, z\}$ // set of pack flow edges
16: **end while**
17: **end procedure**

is listed in algorithm 15. In the first stage (line 3), we select the set of all candidates which have the maximum value of benefit (Pack reuse-to-cost ration), among them we select (line 5) the candidate inducing the minimum accuracy degradation (*deltaAcc*).

When a candidate is selected, the PFCG is updated normally, by eliminating all conflicting candidates (see section 4.4.2.2). Additionally, the word-length of all its elements is specified as explained earlier in eq. (5.4).

5.3.5 SLP-aware Scalings Optimization

Another major impact on the performance of a fixed-point implementation is the cost of scaling operations. Their impact depends on how well the target processor supports shifting operations. For example, a barrel shifter can generally perform a shift operation of any amount in constant time. Whereas shift registers require a variable time depending on the shifting amount.

In the context of SLP an additional critical factor is the fact that scaling operations may break some superword (vector) reuse chains and hence require the introduction of packing/unpacking operations. Thus, severely impacting the performance, not only due to the scalings cost but also to the additionally introduced packing/unpacking overhead. This is, in fact, because most embedded processors only support SIMD shifting instructions by the same amount of all the vector elements Therefore in case two elements of the same vector have to be shifted by different amounts, they need to be unpacked first, shifted independently and then repacked, before being

Algorithm 14 Compute Candidate Benefit.

procedure ComputeBenefit($c, sub, pfcg$)
  $N_R \leftarrow |sub.\mathcal{V}|$
  // Estimated number of additional required packings.
  $N_{pack} \leftarrow |\{x \in pfcg.C \setminus sub.\mathcal{V} : \exists (x,y) \in pfcg.F,\ y \in sub.\mathcal{V} \setminus pfcg.S\}|$
  // Estimated number of additional required unpackings.
  $N_{unpack} \leftarrow |\{x \in sub.\mathcal{V} \setminus pfcg.S,\ x$ is not a Store candidate : $\nexists y \in sub.V \cup pfcg.S : (x,y) \in sub.pfcg\}|$
  $c.reuse \leftarrow N_R$
  $c.conflict \leftarrow |\{\{x,y\} \in pfcg.X : x \in sub.\mathcal{V}\}|$
  $c.benefit \leftarrow \dfrac{1+N_R}{1+N_{pack}+N_{unpack}}$
  SPEC.save
  $acc_o \leftarrow EvalAcc(SPEC)$
  for $x \in sub.\mathcal{V}$ **do**
    SetMaxWl(x,SPEC)
  end for
  $acc_1 \leftarrow EvalAcc(SPEC)$
  SPEC.revert
  $c.deltaAcc \leftarrow acc_1 - acc_0$
end procedure

Algorithm 15 Select the 'best' candidate in the PFCG

procedure Select($pfcg$)
  // First Stage: Set of candidates with maximum benefit estimation
  $bestSet \leftarrow \{c \in pfcg.C : c.benefit = max\}$
  // Second Stage:
  $best \leftarrow c \in bestSet$ with minimum $c.deltaAcc$ value
  return $best$
end procedure

able to be used by an SIMD instruction [4].

Example 12 *For instance, in the example of figure 5.8, operations 1 and 2 are in a group. The scaling amounts associated to the output superword, used by group* $\{3,4\}$*, are* $f1-f3$ *and* $f2-f4$*, where* fx *is the FWL of operation* x*.*

4. In case masking operations are supported, packing/unpacking are not required but multiple SIMD instructions would still be needed to perform such operations. Though masking operations are rarely supported.

Algorithm 16 SLP-aware Scaling optimization pseudo-code

```
1: // Input: G, the set of selected SIMD groups.
2: // Input: SPEC, fixed-point specification.
3: // Input: A, the accuracy constraint.
4: procedure ScalOptim(G, SPEC, A)
5:     for each superword reuse (g1, g2): g1, g2 ∈ G do
6:         S ← g1.elements.FWL − g2.elements.FWL // list of required scaling amounts
7:         if all amounts in S are equal then
8:             skip
9:         else if all amounts are positive (i.e. right shifts are required) then
10:            m ← max(S)
11:            SPEC.save g1
12:            for e ∈ g1 do
13:                reduce FWL of e by (m − S[e])
14:            end for
15:            if EvalAcc(SPEC) violates A then
16:                SPEC.revert g1
17:            end if
18:        end if
19:    end for
20: end procedure
```

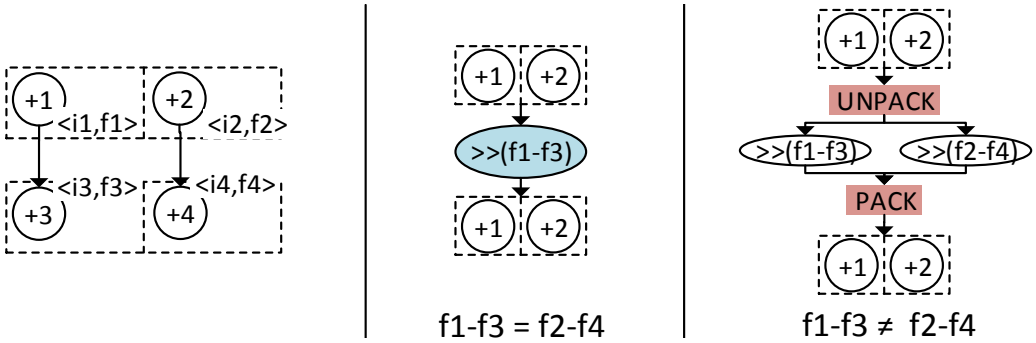


Figure 5.8 – Scaling example. f_x represents the FWL.

When the amount is zero it indicates that no scaling is required. If it is positive a right shift is required. In case $f1-f3 \neq f2-f4$, the scaling amounts are different, thus the shifting operations cannot be grouped and therefore packing/unpacking operations are required.

In this work, we address this problem by proposing an SLP-aware, accuracy-aware scaling optimization algorithm, which is listed in algorithm 16. The goal is to try to make all elements of an SIMD group scalable by the same amount, so that the scaling operations can be grouped together and implemented using an SIMD instruction without the need for packing/unpacking.

To avoid this case, we reduce the FWLs while keeping word-lengths intact (by increasing the corresponding IWLs) so that the scaling amounts of all elements in a group become equal. By

reducing the FWLs, the accuracy of the fixed-point specification may degrades, therefore this optimization is performed as long as the accuracy constraint is not violated.

Additionally, if the accuracy constraint allows it, we can try to eliminate scaling operations altogether, by reducing FWLs of all elements in a group so that their corresponding scaling amounts become zero.

5.4 Source-to-source Compilation Flow

We implemented the proposed, joint floating-point to fixed-point conversion and SLP extraction approach, as a fully automated source-to-source transformation in the compiler framework, GeCoS[47]. In section 5.4.1, we present an overview of this source-to-source flow. We then present the IWL determination, accuracy evaluation and basic-blocks selection procedures in sections 5.4.2 to 5.4.4. Finally we present the fixed-point and SIMD C code generator in section 5.4.5.

5.4.1 Flow Overview

The source-to-source flow diagram is depicted in figure 5.9. Starting from an annotated floating-point C code, we first construct the correspondent IR, which is then analyzed to determine the set of basic blocks to be considered for SLP extraction (see section 5.4.4), as well as to construct the fixed-point specification (see definition 5.2).

Then, the dynamic range of each node in the fixed-point specification is determined using IDFix [3], a floating-to-fixed-point conversion framework integrated to Generic Compiler Suite (GeCoS). This information is later used to specify the IWLs (see section 5.4.2) of all formats in the fixed-point specification.

Besides, we also use IDFix in order to evaluate the accuracy of a given fixed-point specification solution. To do so, the analytical expression of the system's output noise power is generated as a function of the fixed-point specification (see section 5.4.3). This is used during WLO as a metric to evaluate the fixed-point specification's accuracy and compare it against the user specified constraint.

The half-specified (IWLs only) fixed-point specification along with the model describing the target processor (see section 5.3.2) are used to determine the search space for WLO, as explained earlier in section 5.3.3.1. The proposed joint WLO and SLP extraction algorithm is used to obtain a fully specified fixed-point specification and the set of SIMD groups that yields the "best" performance while satisfying the accuracy constraint.

These information are finally used by the back-end to convert the original floating-point C code into fixed-point using native integer C data-types with explicit scaling operations in order to match the fixed-point specification. Furthermore, it implements the SIMD groups using an abstract C macros Application Programming Interface (API) and generates the corresponding

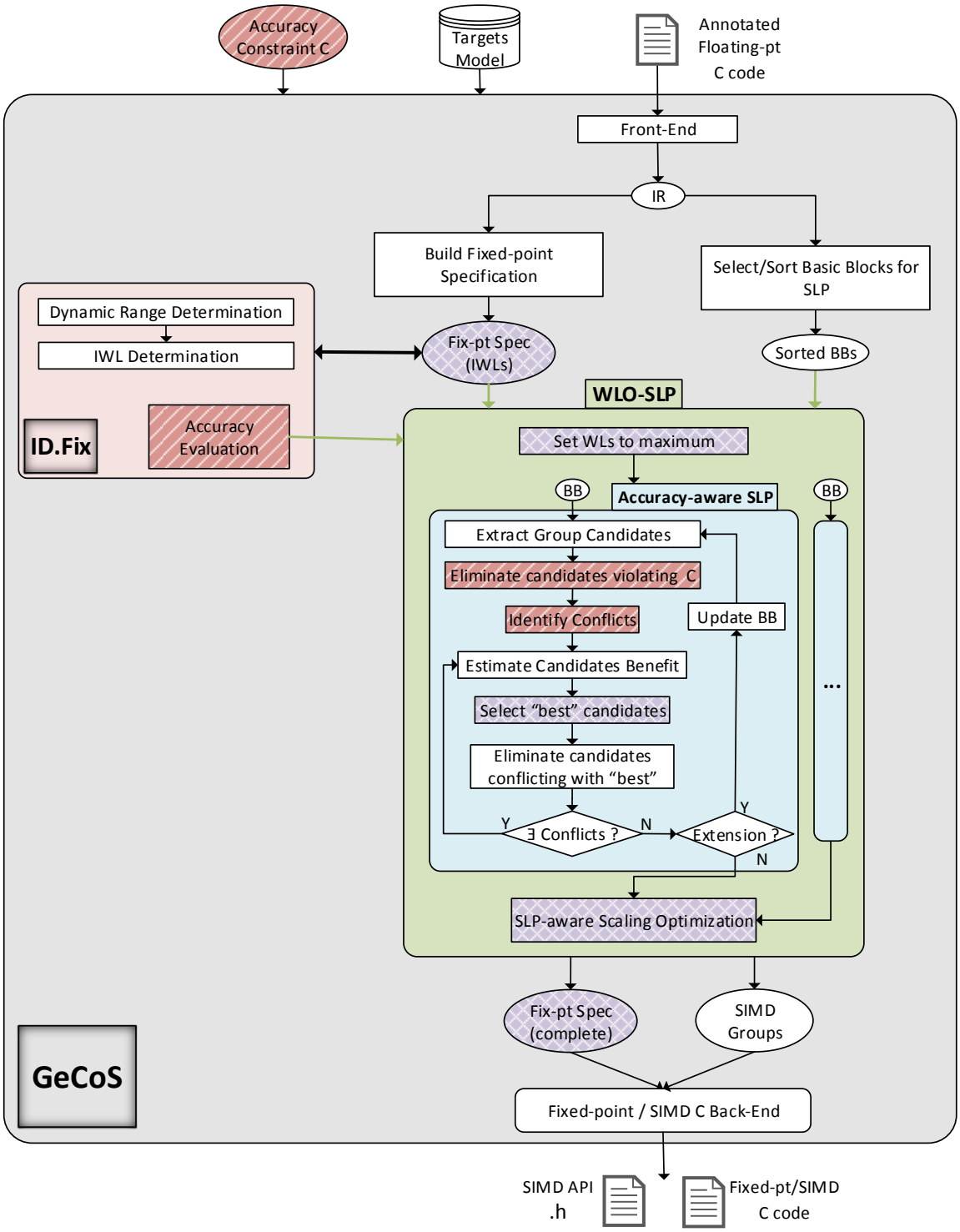


Figure 5.9 – SLP-aware WLO source-to-source compilation flow diagram.

API's implementation for the specified target processor, using its corresponding SIMD intrinsics. The implementation of the back-end is presented later in section 5.4.5.

5.4.2 Integer Word Length determination

The integer word-lengths of each data node d in the fixed-point specification is first determined based on its dynamic range, in such a way to avoid overflows:

$$d.DIWL = \max(nbBits(MIN),\ nbBits(MAX)) + Sign \tag{5.5}$$

Where:

— $[MIN, MAX]$ is the values range interval of data d.

— $nbBits(n) = \begin{cases} \lfloor \log_2(n) \rfloor + 1, (n > 0) \\ \lceil \log_2(|n|) \rceil, (n < 0) \\ 0, (n = 0) \end{cases}$

— $Sign = \begin{cases} 1, (MIN < 0) \\ 0, (MIN \geq 0) \end{cases}$

So, we set the IWL of each data to its DIWL and from now on, when modifying the IWL, we make sure that it never goes below DIWL:

$$d.I \geq d.DIWL \tag{5.6}$$

In other words, $d.DIWL$ is the minimum possible value for $d.I$.

The dynamic range intervals are obtained using interval arithmetic (see section 2.5.1) to propagate the user-specified ranges for all the inputs of the system. This method is already implemented in IDFix (see section 2.6.2). For doing so, each variable, in the input C code, representing an input of the system must be annotated by a `pragma` specifying the MIN and MAX values.

Then, we specify the IWLs of the inputs (in_0 and in_1) and output (out) fixed-point formats of each operation, based on its predecessors (d_0 and d_1) and successor (d_3) data nodes, as illustrated in fig. 5.7. For an add (or sub) operation:

— $m = \max(d_0.I, d_2.I, d_3.I)$

— $in_0.I = in_1.I = out.I = m$

For a mul operation:

— $in_0.I = p_0.I$

— $in_1.I = p_1.I$

— $in_2.I = in_0.I + in_1.I$

5.4.3 Accuracy evaluation

In order to evaluate the accuracy of a given fixed-point specification solution during WLO, we use the analytical accuracy evaluation method that is already implemented in IDFix (see section 2.6.2). Currently, this method only supports Linear Time-Invariant (LTI) and non-recursive non-LTI systems. Besides, the C code must satisfy various constraints, as mentioned in section 2.6.2, mainly. The code is later flattened by IDFix in order to construct the Signal Flow Graph (SFG). Thus, this method is currently not suitable for large applications.

Even though, accuracy evaluation is an essential part of this flow, it is however completely independent and orthogonal to the proposed SLP-aware WLO algorithm. Therefore, any alternative method can be seamlessly used instead. The same applies for dynamic range evaluation.

5.4.4 Select and sort Basic-blocks for SLP

In this accuracy constrained context, the order in which basic blocks are optimized is important. It is better to use the accuracy-degradation budget on fully optimizing a more performance-impacting basic block before moving to another one. This way we ensure that all potential superword reuses, in a basic block, are being exploited as much as the accuracy constraint allows it. Therefore, we assign a priority to each basic-block to be considered for SLP extraction. Priority here depends on the contribution of the basic block to the overall execution time; a higher priority is attributed to the one that occupies more of the execution time. This can be determined automatically based on profiling or static analysis, for instance. However, in this work we use `pragma` annotations to specify the set of basic blocks to be considered for SLP extraction and to determine their associated priority.

Since this step is completely separated from the proposed joint WLO and SLP extraction algorithm, an automated approach can easily replace the one currently used.

5.4.5 Fixed-point/SIMD Code Generation

First, we adapted the C++ code generator that was already implemented in IDFix (see section 2.6.2). It can generate Algorithmic C Datatypes or SystemC C++ code. This in fact already existed in IDFix, however it was only converting the data-type of floating-point variables into the corresponding fixed-point format specified by the fixed-point specification. The existing generator did not modify operations and relied on the C++ fixed-point library to perform scalings when needed. This can actually be a problem, since the implementation of the fixed-point operations in such libraries usually keep full precision by automatically increasing the word-lengths when needed (for example in Algorithmic C Datatypes, the operands of and add/sub are aligned to the maximum FWLs, and their word-length are extended), unless the result in explicitly casted or assigned to a variable. This means, that all operations whose results are not directly assigned to a variable, are implemented using full precision. Thus the generated code may not correspond exactly to the fixed-point specification. This is very important when

targeting a processor, since it may require simulation of full-precision operators, thus degrading the performance. Therefore we added support for proper operation scaling (casting) to reflect the exact specification.

On the other hand, we also implemented a configurable fixed-point C code generator. We did that for two reasons, first to avoid the unnecessary overhead inducing by using such libraries. Second, in order to target processors which do not have C++ compilers, such as VEX. The code generator can generate fixed-point C code using only native integer data-types and operations with proper scaling using shift and cast operations. Alternatively, it can generate code using a generic macros API, to represent fixed-point data-types and operations. This API can then be implemented for the target architecture. This helps make the code more readable. It currently only support truncation as rounding mode and does not support saturation.

For SIMD C code generation, we use the same code generator presented earlier in section 4.5.1.

5.5 Experimental evaluation and Results

In this section, we present the experimental evaluation process we used to test the validity of our proposed approach, performing joint SLP extraction and float-to-fixed-point conversion, compared to a classical approach, applying both transformations independently.

5.5.1 Experimental Setup

To represent our approach, we use the source-to-source compilation flow, presented earlier in section 5.4. We will refer to it as `WLO+SLP`.

The goal is to test how efficient `WLO+SLP` is in exploiting the accuracy/performance tradeoff, compared to a classical approach performing, first, float-to-fixed-point conversion, and then SLP extraction independently. To represent the latter approach, we implement a similar source-to-source compilation flow. This flow is presented below in section 5.5.2, we refer to it by `WLO-then-SLP`.

The experimental setup is illustrated in figure 5.10. Starting from the original floating-point (single-precision) version, called `float`, of a benchmark's C code, we apply both `WLO-then-SLP` and `WLO+SLP`, for a given accuracy constraint value. `WLO+SLP` generates one fixed-point SIMD C code. Whereas, `WLO-then-SLP` first generates a fixed-point C code (without SIMD), and then it generates the fixed-point SIMD C code (after applying SLP extraction). All four C code versions, are then compiled (with -O3) and simulated on the target processor's simulator, using the same floating-point input data. The floating-point input data is pre-converted to the corresponding fixed-point formats, for all versions except `float`. The number of cycles spent executing the benchmark function is finally reported for each version. The data allocation, initialization and conversion are not included.

We run this procedure for each benchmark, on different target processors and for different

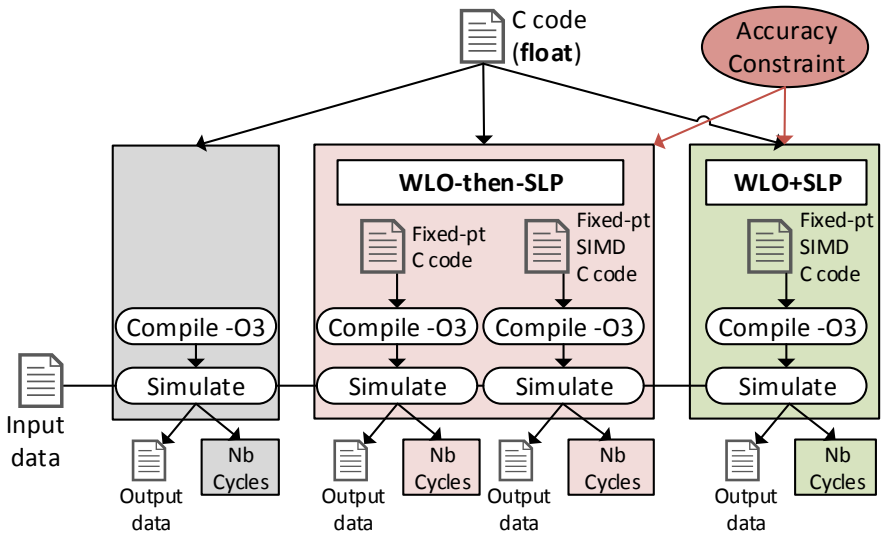


Figure 5.10 – Test setup

values of the accuracy constraint. The benchmarks we use are presented in section 5.5.3, and the targeted processors are presented in section 5.5.4.

5.5.2 `WLO-then-SLP` source-to-source flow

The source-to-source compilation flow implemented to represent `WLO-then-SLP` approach, is depicted in fig. 5.11. Unlike our approach, it first performs float-to-fixed-point conversion, and then SLP extraction independently.

We implement it in Gecos and we also use the framework, IDFix, to perform float-to-fixed-point conversion. For WLO we used the Tabu search algorithm presented by Nguyen et al [91]. To estimate the cost of a given fixed-point specification, a relative execution time is associated to each operation depending on its attributed operator's precision (word-length). This is very similar to the approach proposed by Menard et al [85]. For instance, when targeting ST240, which provides the possibility to implement an operation using either 32-bit or 2x16-bit operators, then the corresponding cost for the 32-bit operator is set as the double of that of 16-bit.

Once the word-lengths have been determined, the fixed-point specification is used to convert the IR from floating-point to fixed-point. This is used to perform SLP extraction using the algorithm we presented earlier in section 4.4.

Finally, we use the same back-end to generate both fixed-point and SIMD C code versions for the target processor.

Contrary to our approach, `WLO-then-SLP` performs WLO independently from SLP extraction i.e. without considering the impact of word-length selection on SLP opportunities and the associated overhead due to data packing/unpacking. Instead, it simply assumes that selecting narrower word-lengths is always better for SLP.

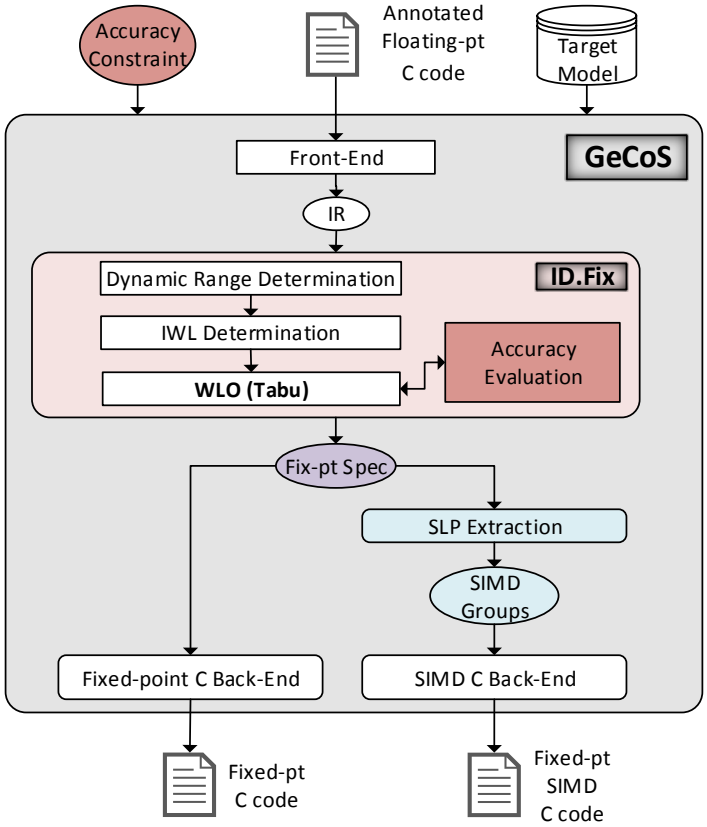


Figure 5.11 – `WLO-then-SLP` source-to-source compilation flow diagram.

5.5.3 Benchmarks

A 64-tap FIR, a 10th order IIR filters and a 3x3 image convolution (CONV) are used as benchmarks. The single-precision floating-point C code of each benchmark respects all the constraints of IDFix, discussed in section 2.6.2. It is annotated with `pragmas`, mainly in order to specify the dynamic range interval of the inputs and to determine the basic block to be considered for SLP extraction.

The benchmarks description is summarized in table 5.1. The innermost loop in FIR and IIR

App	Description
FIR	64-tap FIR filter, samples size 1x1024
IIR	10th order IIR filter, samples size 1x1024
CONV	3x3 image convolution, sample image size 32x32

Table 5.1 – Benchmarks

is partially unrolled by a factor of four to expose SLP, whereas the convolution kernel (3x3), CONV, is fully unrolled. The input samples, of size 1024 for FIR and IIR and 32x32 for CONV, are normalized to $[0, 1]$.

The main application used to invoke a benchmark function, allocates and initializes the necessary variable. Then it starts the execution cycles counter, depending on the target processor, just before invoking the benchmark function. It stops the counter right after the function returns and then it writes the output into a file.

5.5.4 Target Processors

In the context of the ALMA project, we target two embedded processors, XENTIUM and KAHRISMA. In addition, we also consider two other processors, ST240 and VEX, in order to test the validity of our approach. None of these target compilers provide automatic Simdization support.

XENTIUM [104] is a low energy consumption 32-bit 12-issue wide Very Long Instruction Word (VLIW) DSP core from Recore Systems. It has ten functional units, six of which can perform integer arithmetic and logic operations, two can perform multiply operations and two can perform load/store operations. XENTIUM supports 2x16-bit SIMD operations. It has four 8 32-bit and one 16 32-bit register files, each of which has two read and two write ports. The model we use to represent XENTIUM is provided in table A.1.

KAHRISMA [58] is a heterogeneous, dynamically reconfigurable, multi-core research architecture developed by the Karlsruhe Institute of Technology (KIT). In this work we use its 2-issue width single Very Long Instruction Word (VLIW) core configuration. It provides support for 4x8-bit and 2x16-bit SIMD integer arithmetic operations. The model we use to represent XENTIUM is provided in table A.3.

ST240 [4] is a 4-issue wide VLIW media processor from the ST200 family of STMicroelectronics. It has four integer units, 2 multiplication units, 1 branch unit and 1 load/store unit along with a general purpose register file of 64 32-bit registers with 8 read and 4 write ports. ST240 also supports 2x16-bit SIMD operations. The model we use to represent ST240 is provided in table A.2. Besides, ST240 provides hardware support for single-precision floating-point operations but without SIMD support.

VEX [5] is a parameterizable and extensible VLIW architecture model. We use it in two configurations; VEX-1 and VEX-4 with an issue width of 1 and 4 respectively. Since VEX does not provide support for SIMD, we implemented a 16-bit and 8-bit SIMD instruction set extension for supporting integer arithmetic, shift and data manipulation operations, using the provided extension mechanism. The model we use to represent VEX is provided in table A.4.

The main characteristics of these processors are summarized in table 5.2.

5.5.5 Results

In this section we report the experimental results. First, we compare the execution time of fixed-point SIMD C code obtained by our source-to-source flow, against the original floating-point version. Then, we compare the performance of the solution obtained by our approach

Processor	single-float	1x32-bit int	2x16-bit int	4x8-bit int
XENTIUM	×	✓	✓	×
ST240	✓	✓	✓	×
VEX	×	✓	✓	✓
KAHRISMA	×	✓	✓	✓

Table 5.2 – Target processors supported operations.

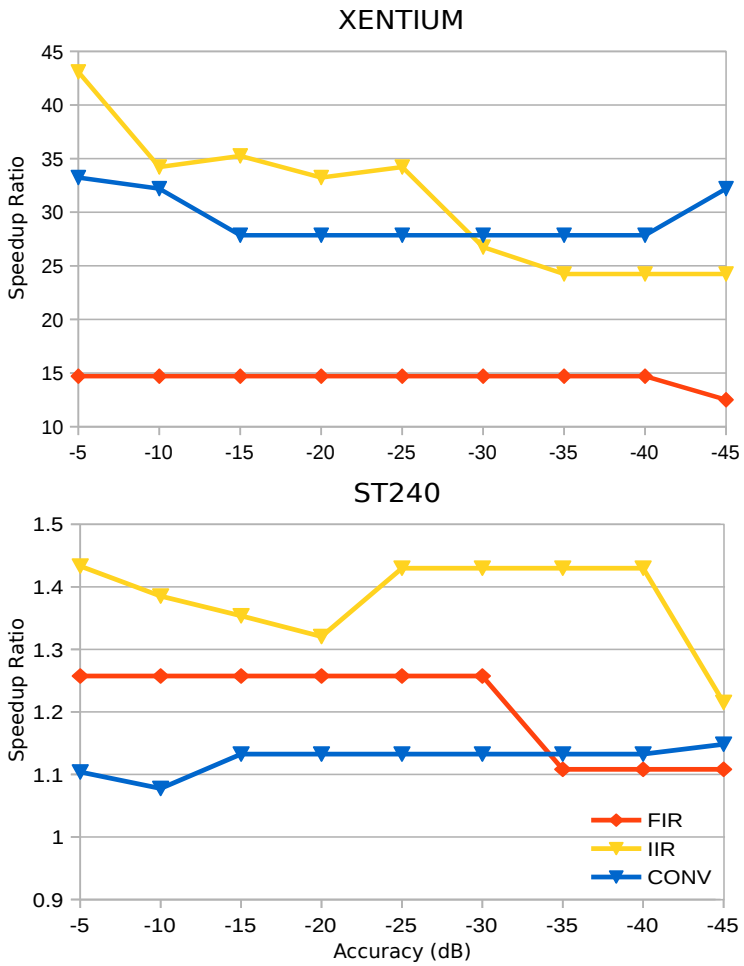


Figure 5.12 – Speedup of SIMD code version obtained by `WLO+SLP` flow, over the original `float` version, for different accuracy constraints expressed in dB (higher values (to the left) are less accurate)

(`WLO+SLP`) against a typical approach (`WLO-then-SLP`).

5.5.6 Floating-point vs. Fixed-point

In order to compare the performance of the fixed-point implementation (with SIMD) against floating-point, we run the fixed-point SIMD code obtained by `WLO+SLP` as well as the original single-precision floating-point code (`float`) on both XENTIUM and ST240, for different accuracy constraints. The execution time speedup of the fixed-point SIMD (`simd`) over `float` is computed as:

$$\texttt{speedup} = \texttt{nb Cycles(float)} \;/\; \texttt{nb Cycles(simd)}$$

The graphics of figure 5.12 plot the obtained speedup on both XENTIUM and ST240, for different accuracy constraints. The latter represents the maximum tolerable quantization noise power at the system output, specified in dB; smaller values represent tighter accuracy constraints (i.e. more accurate).

On XENTIUM, a speedup by a factor of 15 to 45 is achieved. This results are as expected, since this processor does not have a hardware support for floating-point arithmetic. Thus, floating-point support is provided by software emulation.

On ST240, even though it has hardware support for floating-point, a speedup up to 1.4x is obtained. It is mainly due to the exploitation of SIMD capabilities.

5.5.7 `WLO+SLP` vs. `WLO-then-SLP`

In order to compare the performance of our approach (`WLO+SLP`), performing SLP extraction and float-to-fixed-point jointly, against a typical approach (`WLO-then-SLP`), we run the test setup presented earlier, for all benchmarks, on each target processor and for different values of the accuracy constraint. For each test we compare the execution time speedup of the fixed-point SIMD version obtained by `WLO+SLP` against the one obtained by `WLO-then-SLP`. The speedup is computed against the fixed-point version (without SIMD) obtained by `WLO-then-SLP`, called `fixed`, as follows:

$$\texttt{speedup} = \texttt{nb Cycles(fixed)} \;/\; \texttt{nb Cycles(simd)}$$

The results are reported in the graphics of figure 5.13.

The overall results clearly show the advantage of our approach across all benchmarks on all processors.

For FIR, we can see our approach trading accuracy for performance improvement. It manages to efficiently exploit SIMD to achieve performance improvement (up to 1.6x on KAHRISMA and 1.3x on XENTIUM), whereas `WLO-then-SLP` mostly result in performance degradation after applying SLP extraction, illustrating the fact that WLO in this case is blindly optimizing without considering the impact of SLP on performance.

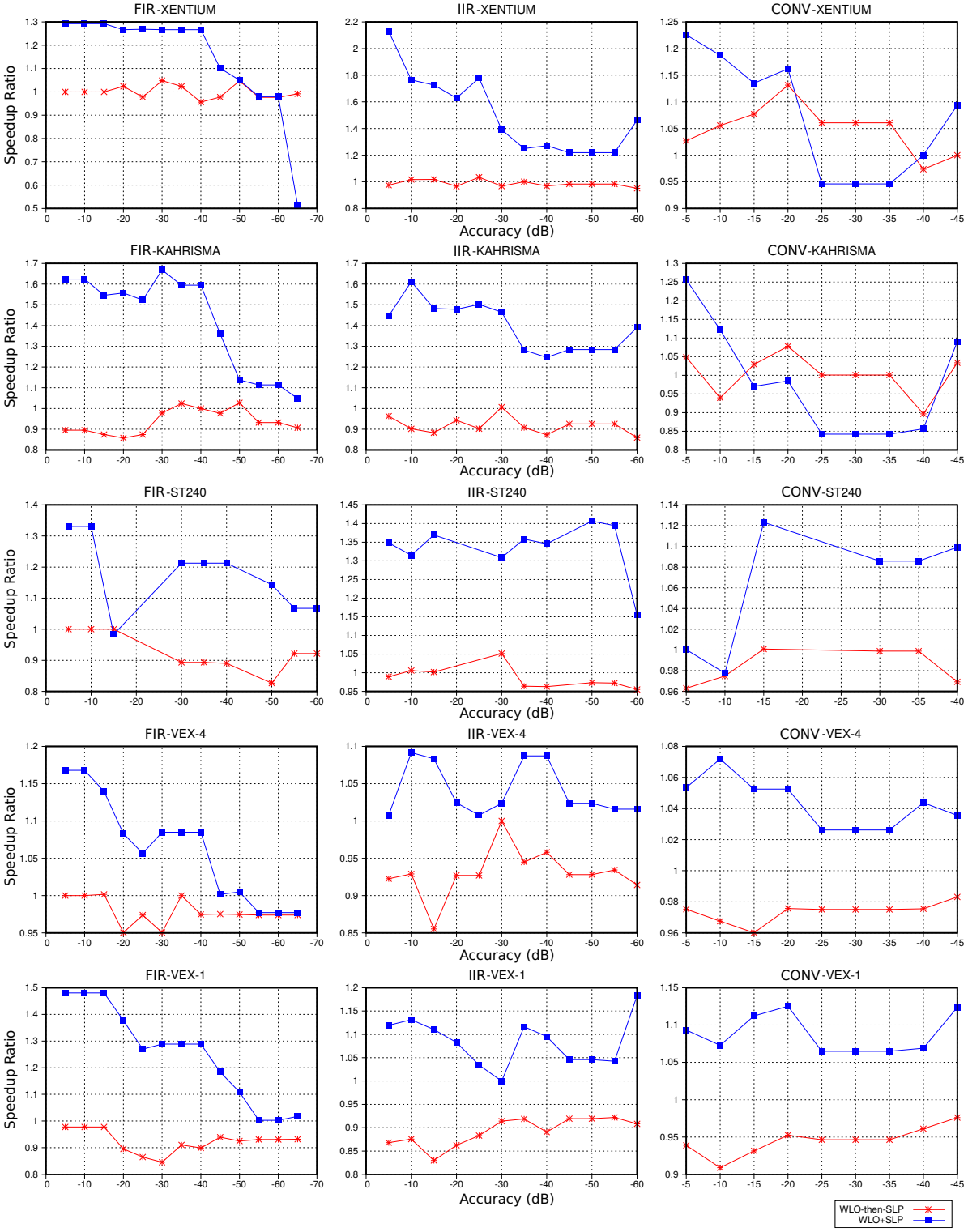


Figure 5.13 – Speedup (higher is better) comparison between SIMD versions of `WLO-then-SLP` and `WLO+SLP` vs. accuracy constraint expressed in dB (higher values(to the left) are less accurate). The baseline is the execution time of the (non SIMD fixed-point version of `WLO-then-SLP`.

The few points where our approach yields performance degradation, for FIR on XENTIUM at -65 db for instance, are due to a reduction in the execution time of the baseline fixed-point version, where the Tabu WLO algorithm manages to find a better fixed-point solution, since at this high accuracy constraint not many SLP opportunities are available.

Similarly for IIR, our approach yields consistently better performance improvement on all target processors, up to 2x on XENTIUM, 1.6x on KAHRISMA and 1.4x on ST240. `WLO-then-SLP` still mostly result in performance degradation after applying SLP extraction.

For CONV, on XENTIUM and KAHRISMA, a slight slowdown is observed between -25 dB and -40 dB. In this case, the selected SLP solution has a high packing/unpacking cost compared to the performance gain it achieves, thus inducing a performance degradation. However, a performance estimation of the SLP grouping solution can be used to discard such solutions, when a performance degradation is detected.

Besides, by comparing the results on VEX-1 and VEX-4 [5] we can notice the impact of instruction level parallelism (ILP). On VEX-4, with higher ILP capabilities, the speedup due to SIMD is less important than on VEX-1.

5.6 Conclusion

In this work we discussed the interaction between floating-point to fixed-point conversion and SLP extraction. We argued that considering both transformations jointly can yield more efficient SIMD solutions compared to a approach where the floating-point to fixed-point conversion is applied first, followed by SLP extraction So, we propose a new SLP-aware WLO algorithm capable of efficiently exploit the performance/accuracy tradeoff when targeting embedded processors.

We implemented the proposed, joint floating-point to fixed-point conversion and SLP extraction approach, as a source-to-source compilation flow in GeCoS. We also implement a typical approach in order to test the validity of our approach. For this, we tested both approaches on several embedded processors. The experimental results show the advantage of our approach in exploiting the performance/accuracy tradeoff compared to a typical approach. A speedup by a factor higher than 1.5 is achieved on XENTIUM and KAHRISMA for an IIR benchmark.

In summary, we concluded that:

— Fixed-point is more suitable when targeting embedded processors with no hard-float support: a speedup by a factor of up to 45 is observed on XENTIUM.

— Even on processors with floating-point support, using fixed-point in combination with SIMD can achieve better performance: a speedup by a factor of up to 1.4 is observed on ST240.

— Using SIMD does not always improve performance, especially if the WLO is not aware of the SIMD opportunities and their associated cost overhead.

5. VEX-1 and VEX-4 only differ in the issue-width; the former has only 1 and the latter has 4.

— Jointly performing WLO and SLP extraction helps achieving more efficient solutions, compared to the typical approach.

Limitations

The source-to-source compilation flow, we implemented in this work, is currently limited by the analytical accuracy evaluation procedure, which imposes many constraints on the input C code (as explained earlier in section 2.6.2). These constraints limit the coverage of the flow to "simple" applications. However, This limitation can be alleviated by using less constrained accuracy evaluation techniques such as simulation-based or simulation-analytical mixed approaches, for instance.

On the other hand, the proposed source-to-source compilation flow does not integrate common optimizations, such as constant propagation. In fact, the WLO and SLP extraction are performed on the original code without such pre-optimizations. Therefore, the generated C code, which uses SIMD intrinsics, seams to be preventing the target compiler from being as efficient in optimizing this code compared to the original C code with no SIMD intrinsics. This makes it very hard to fairly compare the performance between the generated SIMD code and the original one. This can be addressed either by integrating such optimization to the source-to-source flow and applying them prior to the WLO and SLP extraction.

Chapter 6

Conclusion

In this manuscript, we discussed the impact of two important code transformations when targeting low power embedded processors, namely, floating-point to fixed-point conversion and Superword Level Parallelism (SLP) extraction. This work was inducted in the context of the ALMA project, which aimed at providing a complete toolflow to target embedded multi-core architectures, starting from Scilab or Matlab down to parallel C code. Among other optimizations, floating-point to fixed-point conversion and Simdization are performed to improve performance of an application when targeting embedded processors. Especially, since ALMA target processors do not provide support for floating-point arithmetic.

Despite the fact that many embedded processors provide hardware support for floating-point arithmetic, a good number of ultra low power embedded processors, such as ARM cortex-M0/1/3, Texas Instruments (TI) TMS320 C64x and Recore Systems's XENTIUM, do not. This makes floating-point to fixed-point conversion a crucial optimization when targeting such processors. Specially when most application prototyping employs floating-point arithmetic. The results shown in chapter 5, reinforces this fact, since they show a speedup of up to 45x when converting floating-point to fixed-point in the case the target processor does not have hardware support for floating-point arithmetic. On the other hand, most of such embedded processors provide support for vector operations through subword Single Instruction Multiple Data (SIMD) instructions. These SIMD capabilities can be efficiently exploited using SLP extraction methodologies.

In chapter 4, we discussed the state-of-the-art of SLP extraction algorithms. We then proposed a new Intermediate Representation (IR), called Pack Flow and Conflict Graph (PFCG), for SLP extraction together with an improved SIMD group selection method, which we implemented as a source-to-source compilation flow. Experimental results showed a significant performance improvement (up to 50%) of the obtained SLP solution, by using our proposed method compared to a state-of-the-art SLP extraction algorithm (which we also implemented as a source-to-source compilation flow), without the need for data layout transformations. The proposed IR helps to speedup the SIMD group candidate benefit estimation and consequently reduces the complexity of the SIMD group selection algorithm. In addition, we proposed a way to control the size of

the PFCG by splitting the operations into batches. Experimental results showed that this can be used to reduce the group selection time, while still achieving good SLP solutions.

By this point, we have at our disposal a complete source-to-source flow capable of automatically generating a SIMD C code for different embedded processors. In the next chapter, we discuss the interaction between floating-point to fixed-point conversion and SLP extraction. We show that, existing approaches considering both transformations independently, yields less efficient solutions. We argue that considering SLP extraction in conjunction with Word Length Optimization (WLO) helps achieving better results. So, we propose a new SLP-aware WLO algorithm, which we integrate into the source-to-source compilation flow that we implemented.

However, exploiting SLP in a floating-point application, when targeting embedded processors with no support for floating-point, is not possible. In this case no SLP is available since the target does not provide a floating-point SIMD instruction set. In this case, in addition to its aforementioned positive impact on performance, floating-point to fixed-point conversion can additionally "unlock" SLP opportunities. In fact, during the WLO step in floating-point to fixed-point conversion, smaller data word-lengths can be selected for the cost of degrading the computation accuracy. But, selecting smaller word-lengths can increase the amount of potential SLP opportunities.

In chapter 5, we showed that existing methods use this observation to improve performance using SIMD. However, they do not consider Simdization during WLO. In other words, they perform floating-point to fixed-point conversion first, and later they can independently perform Simdization. In chapter 5, we show that such methodologies are not efficient in exploiting SLP opportunities. We argued that WLO and SLP extraction are inter-dependent, thus they should be applied jointly to achieve better results. So, we proposed a new methodology for joint WLO and SLP extraction and we implemented it in our source-to-source compilation flow. To demonstrate the validity of our approach we also implemented a compilation flow performing a typical approach, that converts floating-point to fixed-point first, and then apply SLP extraction independently. Experimental results comparing both approaches on a variety of embedded processors, showed that our methodology can achieve more efficient SIMD solutions.

Perspectives

In the work presented in chapter 4, we ignore the memory alignment constraints during SIMD group selection; we assume that all contiguous array reference superwords are aligned. When such superword is not actually aligned, it induces a cost penalty, especially if the target does not support unaligned memory accesses. This problem can be addressed by integrating a memory access alignment analysis to the compilation flow, based on which we can then modify the SIMD candidate benefit estimation to take the alignment cost overhead into account. Besides, optimization of the memory layout, such as array padding combined with some loop transformations, such as loop peeling, register level tiling with index set splitting, can also be used to reduce the amount of unaligned memory accesses. This is worth considering in a future work

to improve the performance of the proposed source-to-source flow. Also, the performance of SLP solutions can be improved by considering inter basic-block and inter-procedural analysis to reduce packing/unpacking overhead by capturing superwords reuse across basic blocks.

In addition, in this work we use a very basic SIMD instruction selection method and we only consider "simple" SIMD operations (add, sub, mul,...). However, many embedded processors (Digital Signal Processors (DSPs)), such as XENTIUM and ST240, provide multiple "complex" SIMD operations, such as *muladd*. Considering such SIMD operations can help achieving better performance. This could be considered during SLP extraction phase, by allowing the construction of "complex" SIMD group candidates to represent the aforementioned operations. Alternatively, this can be performed after SLP extraction by using a more complex SIMD instruction selection capable of detecting such operations patterns.

In this work, we mainly targeted embedded processors with *subword* SIMD support. We believe that such limited SIMD support would continue to be used in ultra low power embedded processors, since it comes for almost no extra cost, by simply partitioning the existing data-path. However, with the advances of computer architectures and manufacturing technologies, *superword* SIMD extensions, would be supported by more embedded processors. These extensions support wider SIMD vector sizes, which is expected to keep increasing. For instance, some ARM processors support 128-bit SIMD operations via the NEON extension. Therefore, it would be interesting to see how the proposed SLP extraction method performs when targeting such processors with wider SIMD data-path.

Besides, ARM processors, for instance, also support half-precision floating-point SIMD. In this context, a performance/accuracy exploration for different floating-point precisions can be achieved using a similar approach as the one we proposed in chapter 5. In fact, our approach can be easily adapted to perform joint SLP extraction and floating-point precision selection, under accuracy constraint. This would require a floating-point implementation accuracy evaluation; simulation-based methods can be used as a straightforward solution to provide this service.

Appendix A

Target Processor Models

In this appendix we present the models we use in this thesis work to represent the target processors. The model is defined in section 5.3.2.

A.1 XENTIUM

The model we use to represent the XENTIUM processor in summarized in table A.1.

T	**WS**	**N**	**PM**	**QM**	**OM**	**S**	
add	(32, 32, 32)	1	LSB	Trunc	Wrap	(0, 0, 0)	
add	(32, 32, 32)	1	LSB	Trunc	SAT	(0, 0, 0)	
add	(16, 16, 16)	2	LSB	Trunc	Wrap	(0, 0, 0)	
add	(16, 16, 16)	2	MSB	Trunc	Wrap	(0, 0, 1)	
add	(16, 16, 16)	2	MSB	Round	Wrap	(0, 0, 1)	
sub	(32, 32, 32)	1	LSB	Trunc	Wrap	(0, 0, 0)	
sub	(32, 32, 32)	1	LSB	Trunc	SAT	(0, 0, 0)	
sub	(16, 16, 16)	2	LSB	Trunc	Wrap	(0, 0, 0)	
sub	(16, 16, 16)	2	MSB	Trunc	Wrap	(0, 0, 1)	
mul	(32, 32, 64)	1	FULL	-	-	(0, 0, 0)	
mul	(16, 16, 32)	2	FULL	-	-	(0, 0, 0)	

Table A.1 – XENTIUM Model.

A.2 ST240

The model we use to represent the ST240 processor in summarized in table A.2.

T	WS	N	PM	QM	OM	S
add	(32,32,32)	1	LSB	Trunc	Wrap	(0,0,0)
add	(16,16,16)	2	LSB	Trunc	Wrap	(0,0,0)
sub	(32,32,32)	1	LSB	Trunc	Wrap	(0,0,0)
sub	(16,16,16)	2	LSB	Trunc	Wrap	(0,0,0)
mul	(32,32,32)	1	LSB	Trunc	Wrap	(0,0,0)
mul	(16,16,16)	2	LSB	Trunc	Wrap	(0,0,0)

Table A.2 – ST240 Model.

A.3 KAHRISMA

The model we use to represent the KAHRISMA processor in summarized in table A.3.

T	WS	N	PM	QM	OM	S
add	(32,32,32)	1	LSB	Trunc	Wrap	(0,0,0)
add	(16,16,16)	2	LSB	Trunc	Wrap	(0,0,0)
add	(8,8,8)	4	LSB	Trunc	Wrap	(0,0,0)
sub	(32,32,32)	1	LSB	Trunc	Wrap	(0,0,0)
sub	(16,16,16)	2	LSB	Trunc	Wrap	(0,0,0)
sub	(8,8,8)	4	LSB	Trunc	Wrap	(0,0,0)
mul	(32,32,32)	1	LSB	Trunc	Wrap	(0,0,0)
mul	(16,16,16)	2	LSB	Trunc	Wrap	(0,0,0)
mul	(8,8,8)	4	LSB	Trunc	Wrap	(0,0,0)

Table A.3 – KAHRISMA Model.

A.4 VEX

The model we use to represent the VEX processor in summarized in table A.4. We implement the SIMD instruction set as an extension to the Instruction Set Architecture (ISA).

T	**WS**	**N**	**PM**	**QM**	**OM**	**S**	
add	(32,32,32)	1	LSB	Trunc	Wrap	(0,0,0)	
add	(16,16,16)	2	LSB	Trunc	Wrap	(0,0,0)	
add	(8,8,8)	4	LSB	Trunc	Wrap	(0,0,0)	
sub	(32,32,32)	1	LSB	Trunc	Wrap	(0,0,0)	
sub	(16,16,16)	2	LSB	Trunc	Wrap	(0,0,0)	
sub	(8,8,8)	4	LSB	Trunc	Wrap	(0,0,0)	
mul	(32,32,32)	1	LSB	Trunc	Wrap	(0,0,0)	
mul	(16,16,16)	2	LSB	Trunc	Wrap	(0,0,0)	
mul	(8,8,8)	4	LSB	Trunc	Wrap	(0,0,0)	

Table A.4 – VEX Model.

Publications

[EMD17] A. H. El Moussawi et S. Derrien, "Superword level parallelism aware word length optimization", *in 2017 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2017.

[EDEMD16] N. Estibals, G. Deest, A. H. El Moussawi et S. Derrien, "System level synthesis for virtual memory enabled hardware threads", *in 2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, p. 738–743, March 2016.

[EMD16] A. H. El Moussawi et S. Derrien, "Demo: Slp-aware word length optimization". DASIP, 2016.

[FYEM+13] A. Floc'h, T. Yuki, A. H. EL Moussawi, A. Morvan, K. Martin, M. Naullet, M. Alle, L. L'Hours, N. Simon, S. Derrien *et coll.*, "Gecos: A framework for prototyping custom hardware design flows", *in Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, p. 100–105, Sept 2013.

[GVA+13] G. Goulas, C. Valouxis, P. Alefragis, N. S. Voros, C. Gogos, O. Oey, T. Stripf, T. Bruckschloegl, J. Becker, A. H. El Moussawi, M. Naullet et T. Yuki, "Coarse-grain optimization and code generation for embedded multicore systems", *in Digital System Design (DSD), 2013 Euromicro Conference on*, p. 379–386, Sept 2013.

Bibliography

[1] Algorithmic C Datattypes. https://www.mentor.com/hls-lp/downloads/ac-datatypes, 2016.

[2] ALMA project. http://alma-project.eu/, 2016.

[3] IDFix. http://idfix.gforge.inria.fr, 2016.

[4] ST240 core and instruction set architecture. www.st.com/resource/en/reference_manual/cd18059133.pdf, 2016.

[5] VEX. http://www.hpl.hp.com/downloads/vex/, 2016.

[6] A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. In *Proceedings of Seventh Annual ACM Symposium on Theory of Computing*, STOC '75, pages 207–217, New York, NY, USA, 1975. ACM.

[7] A. V. Aho, S. C. Johnson, and J. D. Ullman. Code generation for expressions with common subexpressions. *J. ACM*, 24(1):146–160, January 1977.

[8] Alfred V Aho, Mahadevan Ganapathi, and Steven WK Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):491–516, 1989.

[9] John R Allen and Ken Kennedy. *PFC: A Program to Convert Fortran to Parallel Form*. Rice University. Department of Mathematical Sciences, 1982.

[10] Randy Allen and Ken Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(4):491–542, 1987.

[11] Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. Efficient selection of vector instructions using dynamic programming. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 201–212. IEEE, 2010.

[12] Mick Bass, Patrick Knebel, David W Quint, and William L Walker. The PA 7100LC microprocessor: A Case Study of IC Design Decisions in a Competitive Environment. *HEWLETT PACKARD JOURNAL*, 46:12–22, 1995.

[13] P. Belanovic and M. Rupp. Fixify: A Toolset for Automated Floating-point to Fixed-point Conversion. In *International Conference on Computer, Communication, and Control Technologies CCCT'04*, August 2004.

[14] P. Belanovic and M. Rupp. Automated floating-point to fixed-point conversion with the fixify environment. *Rapid System Prototyping, 2005. (RSP 2005). The 16th IEEE International Workshop on*, pages 172–178, June 2005.

[15] Ravi Bhargava, Lizy K John, Brian L Evans, and Ramesh Radhakrishnan. Evaluating MMX Technology Using DSP and Multimedia Applications. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 31, pages 37–46, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[16] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic intra-register vectorization for the intel architecture. *International Journal of Parallel Programming*, 30(2):65–98, 2002.

[17] Hans-Juergen Karl Hermann Boehm and Robert Cartwright. *Exact real arithmetic: Formulating real numbers as functions*. Rice University, Department of Computer Science, 1993.

[18] David Boland and George A. Constantinides. A scalable approach for automated precision analysis. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 185–194, New York, NY, USA, 2012. ACM.

[19] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Notices*, 43(6):101–113, 2008.

[20] John Bruno and Ravi Sethi. Code generation for a one-register machine. *J. ACM*, 23(3):502–510, July 1976.

[21] G Caffarena, A Fernandez, C Carreras, and O Nieto-Taladriz. Fixed-point refinement of ofdm-based adaptive equalizers: An heuristic approach. In *EUSIPCO. Conference*, 2004.

[22] Gabriel Caffarena, Carlos Carreras, Juan A. López, and Ángel Fernández. Sqnr estimation of fixed-point dsp algorithms. *EURASIP J. Adv. Signal Process*, 2010:21:1–21:12, February 2010.

[23] M.-A. Cantin, Y. Savaria, and P. Lavoie. A comparison of automatic word length optimization procedures. In *Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium on*, volume 2, pages II–612–II–615 vol.2, 2002.

[24] Chia-Ming Chang, Chien-Ming Chen, and Chung-Ta King. Using integer linear programming for instruction scheduling and register allocation in multi-issue processors. *Computers & Mathematics with Applications*, 34(9):1–14, 1997.

[25] Gerald Cheong and Monica Lam. An Optimizer for Multimedia Instruction Sets. *Contract*, 30602(95-C):0098, 1997.

[26] Andrea GM Cilio and Henk Corporaal. Floating Point to Fixed Point Conversion of C Code. In *Compiler Construction*, pages 229–243. Springer, 1999.

[27] Fernando Cladera, Matthieu Gautier, and Olivier Sentieys. Energy-aware computing via adaptive precision under performance constraints in ofdm wireless receivers. In *2015 IEEE Computer Society Annual Symposium on VLSI*, pages 591–596. IEEE, 2015.

[28] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. *SIGPLAN Not.*, 30(6):279–290, June 1995.

[29] Martin Coors, Holger Keding, Olaf Lüthje, and Heinrich Meyr. Design and dsp implementation of fixed-point systems. *EURASIP J. Appl. Signal Process.*, 2002(1):908–925, January 2002.

[30] Gaël Deest, Tomofumi Yuki, Olivier Sentieys, and Steven Derrien. Toward scalable source level accuracy analysis for floating-point to fixed-point conversion. *Proceedings of the 2014 IEEE*, pages 726–733, 2014.

[31] Alexandre E Eichenberger, Peng Wu, and Kevin O'brien. Vectorization for SIMD Architectures with Alignment Constraints. In *ACM SIGPLAN Notices*, volume 39, pages 82–93. ACM, 2004.

[32] Paul Feautrier. Dataflow Analysis of Array and Scalar References. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[33] Paul Feautrier. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International journal of parallel programming*, 21(5):313–347, 1992.

[34] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International journal of parallel programming*, 21(6):389–420, 1992.

[35] Dustin Feld, Thomas Soddemann, Michael Jünger, and Sven Mallach. Facilitate SIMD-Code-Generation in the Polyhedral Model by Hardware-aware Automatic Code-Transformation. *IMPACT 2013*, page 45, 2013.

[36] Liza Fireman, Erez Petrank, and Ayal Zaks. New Algorithms for SIMD Alignment. In *Compiler Construction*, pages 1–15. Springer, 2007.

[37] Randall J Fisher and Henry G Dietz. Compiling For SIMD Within A Register. In *Languages and Compilers for Parallel Computing*, pages 290–305. Springer, 1999.

[38] A. Floc'h, T. Yuki, A. El-Moussawi, A. Morvan, K. Martin, M. Naullet, M. Alle, L. L'Hours, N. Simon, S. Derrien, F. Charot, C. Wolinski, and O. Sentieys. Gecos: A framework for prototyping custom hardware design flows. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 100–105, Sept 2013.

[39] Franz Franchetti, Stefan Kral, Juergen Lorenz, and Christoph W Ueberhuber. Efficient Utilization of SIMD Extensions. *Proceedings of the IEEE*, 93(2):409–425, feb 2005.

[40] Jose Fridman. Sub-word parallelism in digital signal processing. *IEEE signal processing magazine*, 17(2):27–35, 2000.

[41] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 317–324, New York, NY, USA, 1997. ACM.

[42] M. Willems H. Keding, F. Hürtgen and M. Coors. Transformation of floating-point into fixed-point algorithms by interpolation applying a statistical approach. In *Proc. Int. Conf. on Signal Processing Application and Technology (ICSPAT)*, Toronto, sep 1998.

[43] Kyungtae Han, Brian L Evans, and Earl E Swartzlander. Data wordlength reduction for low-power signal processing software. In *Signal Processing Systems, 2004. SIPS 2004. IEEE Workshop on*, pages 343–348. IEEE, 2004.

[44] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J Ramanujam, and P Sadayappan. Data Layout Transformation for Stencil Computations on Short-Vector SIMD Architectures. In *Compiler Construction*, pages 225–245. Springer, 2011.

[45] Manuel Hohenauer, Felix Engel, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. A SIMD Optimization Framework for Retargetable Compilers. *ACM Transactions on Architecture and Code Optimization (TACO)*, 6(1):2, 2009.

[46] Libo Huang, Li Shen, Sheng Ma, Nong Xiao, and Zhiying Wang. Dm-simd: a new simd predication mechanism for exploiting superword level parallelism. In *2009 IEEE 8th International Conference on ASIC*, pages 863–866. IEEE, 2009.

[47] IRISA/INRIA. Generic Compiler Suite (GeCoS). "http://gecos.gforge.inria.fr", 2016.

[48] Byunghyun Jang, Perhaad Mistry, Dana Schaa, Rodrigo Dominguez, and David Kaeli. Data transformations enabling loop vectorization on multithreaded data parallel architectures. In *ACM Sigplan Notices*, volume 45, pages 353–354. ACM, 2010.

[49] Marta Jiménez, José M Llabería, Agustin Fernández, and Enric Morancho. A General Algorithm for Tiling the Register Level. In *Proceedings of the 12th international conference on Supercomputing*, pages 133–140. ACM, 1998.

[50] Holger Keding, Martin Coors, Olaf Lüthje, and Heinrich Meyr. Fast bit-true simulation. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 708–713, New York, NY, USA, 2001. ACM.

[51] Holger Keding, Markus Willems, Martin Coors, and Heinrich Meyr. Fridge: a fixed-point design and simulation environment. In *Proceedings of the conference on Design, automation and test in Europe*, pages 429–435. IEEE Computer Society, 1998.

[52] Seehyun Kim, Ki-Il Kum, and Wonyong Sung. Fixed-point optimization utility for c and c++ based digital signal processing programs. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 45(11):1455–1464, Nov 1998.

[53] Seehyun Kim and Wonyong Sung. Fixed-point simulation utility for c and c++ based digital signal processing programs. In *Signals, Systems and Computers, 1994. 1994 Conference Record of the Twenty-Eighth Asilomar Conference on*, volume 1, pages 162–166 vol.1, Oct 1994.

[54] Seehyun Kim and Wonyong Sung. A floating-point to fixed-point assembly program translator for the tms 320c25. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 41(11):730–739, Nov 1994.

[55] Seonggun Kim and Hwansoo Han. Efficient SIMD Code Generation for Irregular Kernels. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 55–64, New York, NY, USA, 2012. ACM.

[56] Taemin Kim and Yatin Hoskote. Automatic generation of custom simd instructions for superword level parallelism. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 362. European Design and Automation Association, 2014.

[57] Patrick Knebel, Barry Arnold, Mick Bass, Wayne Kever, Joel D Lamb, Ruby B Lee, Paul L Perez, Stephen Undy, and Will Walker. HP's PA7100LC: A Low-Cost Superscalar PA-RISC Processor. In *Compcon Spring '93, Digest of Papers.*, pages 441–447, Feb 1993.

[58] R. Koenig, L. Bauer, T. Stripf, M. Shafique, W. Ahmed, J. Becker, and J. Henkel. KAHRISMA: A Novel Hypermorphic Reconfigurable-Instruction-Set Multi-grained-Array Architecture. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 819–824, March 2010.

[59] David Ryan Koes and Seth Copen Goldstein. Near-optimal instruction selection on dags. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 45–54. ACM, 2008.

[60] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P Sadayappan. When Polyhedral Transformations Meet SIMD Code Generation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 48 of *PLDI '13*, pages 127–138. ACM, 2013.

[61] Stefan Kraemer, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. SoftSIMD - Exploiting Subword Parallelism Using Source Code Transformations. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '07, pages 1349–1354, San Jose, CA, USA, 2007. EDA Consortium.

[62] Andreas Krall and Sylvain Lelait. Compilation Techniques for Multimedia Processors. *International Journal of Parallel Programming*, 28(4):347–361, 2000.

[63] Alexei Kudriavtsev and Peter Kogge. Generation of Permutations for SIMD Processors. *ACM SIGPLAN Notices*, 40(7):147–156, 2005.

[64] Ki-Il Kum, Jiyang Kang, and Wonyong Sung. Autoscaler for c: An optimizing floating-point to integer c program converter for fixed-point digital signal processors. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 47(9):840–848, 2000.

[65] Ki-Il Kum and Wonyong Sung. Word-length optimization for high-level synthesis of digital signal processing systems. In *Signal Processing Systems, 1998. SIPS 98. 1998 IEEE Workshop on*, pages 569–578, Oct 1998.

[66] Ashok Kumar. The HP PA-8000 RISC CPU. *Micro, IEEE*, 17(2):27–32, Mar 1997.

[67] Ichiro Kuroda and Takao Nishitani. Multimedia Processors. *Proceedings of the IEEE*, 86(6):1203–1221, 1998.

[68] Samuel Larsen. *Compilation techniques for short-vector instructions*. PhD thesis, MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 2006.

[69] 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.

[70] Samuel Larsen, Rodric Rabbah, and Amarasinghe. Exploiting vector parallelism in software pipelined loops. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 119–129. IEEE Computer Society, 2005.

[71] Samuel Larsen, Emmett Witchel, and Saman Amarasinghe. Increasing and Detecting Memory Address Congruence. In *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*, pages 18–29. IEEE, 2002.

[72] Ruby B. Lee. Subword Parallelism with MAX-2. *Micro, IEEE*, 16(4):51–59, Aug 1996.

[73] Seogoo Lee and Andreas Gerstlauer. Fine grain precision scaling for datapath approximations in digital signal processing systems. In *IFIP/IEEE International Conference on Very Large Scale Integration-System on a Chip*, pages 119–143. Springer, 2013.

[74] Rainer Leupers. Code Selection for Media Processors with SIMD Instructions. In *Proceedings of the conference on Design, Automation and Test in Europe*, pages 4–8, 2000.

[75] Rainer Leupers and Peter Marwedel. Instruction Selection for Embedded DSPs with Complex Instructions. In *Proceedings of the conference on European Design Automation*, pages 200–205. IEEE Computer Society Press, 1996.

[76] Chaofan Li, Wei Luo, Sachin S Sapatnekar, and Jiang Hu. Joint precision optimization and high level synthesis for approximate computing. In *Proceedings of the 52nd Annual Design Automation Conference*, page 104. ACM, 2015.

[77] Bede Liu. Effect of finite word length on the accuracy of digital filters–a review. *Circuit Theory, IEEE Transactions on*, 18(6):670–677, Nov 1971.

[78] Jun Liu, Yuanrui Zhang, Ohyoung Jang, Wei Ding, and Mahmut Kandemir. A Compiler Framework for Extracting Superword Level Parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 47 of *PLDI '12*, pages 347–358. ACM, 2012.

[79] Peng Liu, Rong-cai Zhao, Wei Gao, and Shuai Wei. A New Algorithm to Exploit Superword Level Parallelism. In *2013 IEEE 11th International Conference on Dependable, Autonomic and Secure Computing*, pages 521–527. IEEE, 2013.

[80] Yangxurui Liu, Liang Liu, Viktor Öwall, and Shuming Chen. Implementation of a dynamic wordlength simd multiplier. In *NORCHIP, 2014*, pages 1–4. IEEE, 2014.

[81] J.A. Lopez, G. Caffarena, C. Carreras, and O. Nieto-Taladriz. Fast and accurate computation of the roundoff noise of linear time-invariant systems. *Circuits, Devices Systems, IET*, 2(4):393–408, Aug 2008.

[82] D. Menard, H.N. Nguyen, F. Charot, S. Guyetant, J. Guillot, E. Raffin, and E. Casseau. Exploiting reconfigurable swp operators for multimedia applications. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 1717–1720, May 2011.

[83] D. Menard, R. Rocher, and O. Sentieys. Analytical fixed-point accuracy evaluation in linear time-invariant systems. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 55(10):3197–3208, Nov 2008.

[84] Daniel Menard, Daniel Chillet, François Charot, and Olivier Sentieys. Automatic floating-point to fixed-point conversion for dsp code generation. In *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 270–276. ACM, 2002.

[85] Daniel Menard, Daniel Chillet, and Olivier Sentieys. Floating-to-fixed-point conversion for digital signal processors. *EURASIP J. Appl. Signal Process.*, 2006:77–77, January 2006.

[86] Daniel Menard, David Novo, Romuald Rocher, Francky Catthoor, and Olivier Sentieys. Quantization mode opportunities in fixed-point system design. In *18th European Signal Processing Conference (EUSIPCO-2010)(2010)*, pages 542–546, 2010.

[87] Daniel Menard, Romuald Rocher, Pascal Scalart, and Olivier Sentieys. Automatic sqnr determination in non-linear and non-recursive fixed-point systems. In *Signal Processing Conference, 2004 12th European*, pages 1349–1352. IEEE, 2004.

[88] Daniel Menard and Olivier Sentieys. DSP Code Generation with Optimized Data Word-Length Selection. *Software and Compilers for Embedded Systems*, pages 214–228, 2004.

[89] Daniel Menard, Olivier Sentieys, and Irisa Inria. Automatic Evaluation of the Accuracy of Fixed-point Algorithms. 2002.

[90] Dorit Naishlos. Autovectorization in GCC. In *Proceedings of the 2004 GCC Developers Summit*, pages 105–118, 2004.

[91] H-N Nguyen, D Menard, and O Sentieys. Novel algorithms for word-length optimization. In *Signal Processing Conference, 2011 19th European*, pages 1944–1948. IEEE, 2011.

[92] Hiromu Nobayashi and Christopher Eoyang. A Comparison Study of Automatically Vectorizing Fortran Compilers. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, 1989. Supercomputing '89.*, pages 820–825, Nov 1989.

[93] Diego Novillo. Openmp and automatic parallelization in gcc. In *In the Proceedings of the GCC Developers*, 2006.

[94] David Novo, Sara El Alaoui, and Paolo Ienne. Accuracy vs speed tradeoffs in the estimation of fixed-point errors on linear time-invariant systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 15–20. EDA Consortium, 2013.

[95] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-Vectorization of Interleaved Data for SIMD. In *ACM SIGPLAN Notices*, volume 41, pages 132–143. ACM, 2006.

[96] David A Padua and Michael J Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, 1986.

[97] Karthick Parashar, Daniel Menard, Romuald Rocher, Olivier Sentieys, David Novo, and Francky Catthoor. Fast performance evaluation of fixed-point systems with un-smooth operators. In *2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 9–16. IEEE, 2010.

[98] Alex Peleg and Uri Weiser. MMX Technology Extension to the Intel Architecture. *Micro, IEEE*, 16(4):42–50, Aug 1996.

[99] Michael Philippsen. A survey of concurrent object-oriented languages. *Concurrency - Practice and Experience*, 12(10):917–980, 2000.

[100] Vasileios Porpodas and Timothy M Jones. Throttling Automatic Vectorization: When Less Is More. *PACT 2015*, 2015.

[101] Ivan Pryanishnikov, Andreas Krall, and Nigel Horspool. Pointer alignment analysis for processors with simd instructions. In *In Proceedings of the 5th Workshop on Media and Streaming Processors*, pages 50–57, 2003.

[102] Ivan Pryanishnikov, Andreas Krall, and Nigel Horspool. Compiler optimizations for processors with simd instructions. *Software: Practice and Experience*, 37(1):93–113, 2007.

[103] Georgia Psychou, Robert Fasthuber, Francky Catthoor, Jos Hulzink, and Jos Huisken. Sub-word Handling in Data-parallel Mapping. In *ARCS Workshops (ARCS), 2012*, pages 1–7. IEEE, 2012.

[104] Recore Systems. Xentium VLIW DSP IP core. http://www.recoresystems.com/fileadmin/downloads/Product_briefs/2012-2.0_Xentium_Product_Brief.pdf, 2012.

[105] Gang Ren, Peng Wu, and David Padua. A preliminary study on the vectorization of multimedia applications for multimedia extensions. In *Languages and Compilers for Parallel Computing*, pages 420–435. Springer, 2004.

[106] Gang Ren, Peng Wu, and David Padua. An Empirical Study On the Vectorization of Multimedia Applications for Multimedia Extensions. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 89b–89b. IEEE, 2005.

[107] Gang Ren, Peng Wu, and David Padua. Optimizing data permutations for simd devices. *SIGPLAN Not.*, 41(6):118–131, June 2006.

[108] Lakshminarayanan Renganarayana, Uday Bondhugula, Salem Derisavi, Alexandre E Eichenberger, and Kevin O'Brien. Compact Multi-Dimensional Kernel Extraction for Register Tiling. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 45. ACM, 2009.

[109] Romuald Rocher and Daniel Menard. Analytical approach for numerical accuracy estimation of fixed-point systems based on smooth operations. *Circuits and Systems I: ...*, 59(10):2326–2339, 2012.

[110] Romuald Rocher, Daniel Menard, Nicolas Herve, and Olivier Sentieys. Fixed-point configurable hardware components. *EURASIP J. Embedded Syst.*, 2006(1):20–20, January 2006.

[111] J. R. Rose and G. L. Steele. C*: an extended c language for data parallel programming. In *International Conference on Supercomputing*, 1987.

[112] Sanghamitra Roy and Prith Banerjee. An algorithm for converting floating-point computations to fixed-point in matlab based fpga design. In *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, pages 484–487, New York, NY, USA, 2004. ACM.

[113] Sanghamitra Roy and Prith Banerjee. An algorithm for trading off quantization error with hardware resources for matlab-based fpga design. *IEEE Transactions on Computers*, 54(7):886–896, 2005.

[114] Radu Rugina and Martin Rinard. Pointer Analysis for Multithreaded Programs. In *ACM SIGPLAN Notices*, volume 34, pages 77–90. ACM, 1999.

[115] Richard M Russell. The CRAY-1 Computer System. *Communications of the ACM*, 21(1):63–72, 1978.

[116] Robert R Schaller. MOORE'S LAW: past, present, and future. *IEEE Spectrum*, 34(6):52–59, June 1997.

[117] Kevin Scott and Jack Davidson. Exploring the limits of sub-word level parallelism. In *Parallel Architectures and Compilation Techniques, 2000. Proceedings. International Conference on*, pages 81–91. IEEE, 2000.

[118] Enrique Sedano, Daniel Menard, and A L Juan. Automated Data Flow Graph partitioning for a hierarchical approach to wordlength optimization. 2014.

[119] Ravi Sethi. Complete register allocation problems. *SIAM journal on Computing*, 4(3):226–248, 1975.

[120] Changchun Shi and Robert W. Brodersen. An automated floating-point to fixed-point conversion methodology. In *Proc. IEEE Int. Conf. on Acoust., Speech, and Signal Processing*, pages 529–532, 2003.

[121] Changchun Shi and R.W. Brodersen. A perturbation theory on statistical quantization effects in fixed-point dsp with non-stationary inputs. In *Circuits and Systems, 2004. ISCAS '04. Proceedings of the 2004 International Symposium on*, volume 3, pages III–373–6 Vol.3, May 2004.

[122] Jaewook Shin. *Compiler Optimizations for Architectures Supporting Superword-level Parallelism*. PhD thesis, Los Angeles, CA, USA, 2005. AAI3196891.

[123] Jaewook Shin. Introducing control flow into vectorized code. In *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, pages 280–291, Sept 2007.

[124] Jaewook Shin, Jacqueline Chame, and Mary W Hall. Exploiting Superword-Level Locality in Multimedia Extension Architectures. *J. Instr. Level Parallel*, 5:1–28, 2003.

[125] Jaewook Shin, Mary Hall, and Jacqueline Chame. Superword-Level Parallelism in the Presence of Control Flow. In *Proceedings of the international symposium on Code generation and optimization*, pages 165–175. IEEE Computer Society, 2005.

[126] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *Int. J. Parallel Program.*, 28(4):363–400, August 2000.

[127] T. Stripf, R. Koenig, and J. Becker. A cycle-approximate, mixed-ISA simulator for the KAHRISMA architecture. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 21–26, March 2012.

[128] Wonyong Sung and Ki-Il Kum. Simulation-based word-length optimization method for fixed-point digital signal processing systems. *Signal Processing, IEEE Transactions on*, 43(12):3087–3090, Dec 1995.

[129] Rishi Surendran, Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. Inter-iteration scalar replacement using array ssa form. In *International Conference on Compiler Construction*, pages 40–60. Springer, 2014.

[130] Sriraman Tallam and Rajiv Gupta. Bitwidth Aware Global Register Allocation. *SIGPLAN Not.*, 38(1):85–96, January 2003.

[131] Hiroaki Tanaka, Shinsuke Kobayashi, Yoshinori Takeuchi, Keishi Sakanushi, and Masaharu Imai. *A Code Selection Method for SIMD Processors with PACK Instructions*, pages 66–80. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[132] Christian Tenllado, Luis Piñuel, Manuel Prieto, and Francky Catthoor. Pack Transposition: Enhancing Superword Level Parallelism Exploitation. In *ParCo*, pages 573–580, 2005.

[133] Christian Tenllado, Luis Pinuel, Manuel Prieto, Francisco Tirado, and F Catthoor. Improving Superword Level Parallelism Support in Modern Compilers. In *Hardware/Software Codesign and System Synthesis, 2005. CODES+ ISSS'05. Third IEEE/ACM/IFIP International Conference on*, pages 303–308. IEEE, 2005.

[134] Xinmin Tian, Hideki Saito, Milind Girkar, Serguei V. Preis, Sergey S. Kozhukhov, Aleksei G. Cherkasov, Clark Nelson, Nikolay Panchenko, and Robert Geva. Compiling C/C++ SIMD Extensions for Function and Loop Vectorizaion on Multicore-SIMD Processors. *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 2349–2358, May 2012.

[135] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 327–337. IEEE, 2009.

[136] Yi-Shin Tung, Chia-Chiang Ho, and Ja-Ling Wu. MMX-based DCT and MC Algorithms for Real-Time Pure Software MPEG Decoding. In *Multimedia Computing and Systems, 1999. IEEE International Conference on*, volume 1, pages 357–362 vol.1, Jul 1999.

[137] Shervin Vakili, J M Pierre Langlois, and Guy Bois. FINITE-PRECISION ERROR MODELING USING AFFINE ARITHMETIC. pages 2591–2595, 2013.

[138] Nicolas Vasilache, Benoit Meister, Muthu Baskaran, and Richard Lethin. Joint scheduling and layout optimization to enable multi-level vectorization. *IMPACT, Paris, France*, 2012.

[139] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*, pages 299–302. Springer, 2010.

[140] Suhrid A Wadekar and Alice C Parker. Accuracy sensitive word-length selection for algorithm optimization. In *Computer Design: VLSI in Computers and Processors, 1998. ICCD'98. Proceedings. International Conference on*, pages 54–61. IEEE, 1998.

[141] Peng Wu, Alexandre E Eichenberger, and Amy Wang. Efficient SIMD Code Generation for Runtime Alignment and Length Conversion. In *International Symposium on Code Generation and Optimization, 2005. CGO 2005.*, pages 153–164. IEEE, 2005.

[142] Peng Wu, Alexandre E. Eichenberger, Amy Wang, and Peng Zhao. An Integrated Simdization Framework Using Virtual Vectors. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 169–178, New York, NY, USA, 2005. ACM.

Résumé

Afin de limiter leur coût et/ou leur consommation électrique, certains processeurs embarqués sacrifient le support matériel de l'arithmétique à virgule flottante. Pourtant, pour des raisons de simplicité, les applications sont généralement spécifiées en utilisant l'arithmétique à virgule flottante.

Porter ces applications sur des processeurs embarqués de ce genre nécessite une émulation logicielle de l'arithmétique à virgule flottante, qui peut sévèrement dégrader la performance. Pour éviter cela, l'application est convertie pour utiliser l'arithmétique à virgule fixe, qui a l'avantage d'être plus efficace à implémenter sur des unités de calcul entier. La conversion de virgule flottante en virgule fixe est une procédure délicate qui implique des compromis subtils entre performance et précision de calcul. Elle permet, entre autre, de réduire la taille des données pour le coût de dégrader la précision de calcul.

Par ailleurs, la plupart de ces processeurs fournissent un support pour le calcul vectoriel de type SIMD (*Single Instruction Multiple Data*) afin d'améliorer la performance. En effet, cela permet l'exécution d'une opération sur plusieurs données en parallèle, réduisant ainsi le temps d'exécution. Cependant, il est généralement nécessaire de transformer l'application pour exploiter les unités de calcul vectoriel. Cette transformation de vectorisation est sensible à la taille des données ; plus leurs tailles diminuent, plus le taux de vectorisation augmente. Il apparaît donc un compromis entre vectorisation et précision de calcul.

Plusieurs travaux ont proposé des méthodologies permettant, d'une part la conversion automatique de virgule flottante en virgule fixe, et d'autre part la vectorisation automatique. Dans l'état de l'art, ces deux transformations sont considérées indépendamment, pourtant elles sont fortement liées.

Dans ce contexte, nous étudions la relation entre ces deux transformations, dans le but d'exploiter efficacement le compromis entre performance et précision de calcul. Ainsi, nous proposons d'abord un algorithme amélioré pour l'extraction de parallélisme SLP (*Superword Level Parallelism* ; une technique de vectorisation). Puis, nous proposons une nouvelle méthodologie permettant l'application conjointe de la conversion de virgule flottante en virgule fixe et de l'exploitation du SLP. Enfin, nous implémentons cette approche sous forme d'un flot de compilation source-à-source complètement automatisé, afin de valider ces travaux. Les résultats montrent l'efficacité de cette approche, dans l'exploitation du compromis entre performance et précision, vis-à-vis d'une approche classique considérant ces deux transformations indépendamment.

Abstract

In order to cut-down their cost and/or their power consumption, many embedded processors do not provide hardware support for floating-point arithmetic. However, applications in many domains, such as signal processing, are generally specified using floating-point arithmetic for the sake of simplicity.

Porting these applications on such embedded processors requires a software emulation of floating-point arithmetic, which can greatly degrade performance. To avoid this, the application is converted to use fixed-point arithmetic instead. Floating-point to fixed-point conversion involves a subtle tradeoff between performance and precision ; it enables the use of narrower data word lengths at the cost of degrading the computation accuracy.

Besides, most embedded processors provide support for SIMD (Single Instruction Multiple Data) as a mean to improve performance. In fact, this allows the execution of one operation on multiple data in parallel, thus ultimately reducing the execution time. However, the application should usually be transformed in order to take advantage of the SIMD instruction set. This transformation, known as Simdization, is affected by the data word lengths ; narrower word-lengths enable a higher SIMD parallelism rate. Hence the tradeoff between precision and Simdization.

Many existing work aimed at providing/improving methodologies for automatic floating-point to fixed-point conversion on the one side, and Simdization on the other. In the state-of-the-art, both transformations are considered separately even though they are strongly related.

In this context, we study the interactions between these transformations in order to better exploit the performance/accuracy tradeoff. First, we propose an improved SLP (Superword Level Parallelism) extraction (an Simdization technique) algorithm. Then, we propose a new methodology to jointly perform floating-point to fixed-point conversion and SLP extraction. Finally, we implement this work as a fully automated source-to-source compiler flow. Experimental results, targeting four different embedded processors, show the validity of our approach in efficiently exploiting the performance/accuracy tradeoff compared to a typical approach, which considers both transformations independently.

VU :

Le Directeur de Thèse

VU :

Le Responsable de l'École Doctorale

VU pour autorisation de soutenance

Rennes, le

Le Président de l'Université de Rennes 1

David ALIS

VU après soutenance pour autorisation de publication :

Le Président de Jury,