Titre: Title:	Techniques d'exploration architecturale de design à usage spécifique pour l'accélération de boucles
Auteur: Author:	Mame Maria Mbaye
Date:	2010
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Mbaye, M. M. (2010). Techniques d'exploration architecturale de design à usage spécifique pour l'accélération de boucles [Thèse de doctorat, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/402/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/402/
Directeurs de recherche: Advisors:	Yvon Savaria, & Samuel Pierre
Programme: Program:	génie électrique

UNIVERSITÉ DE MONTRÉAL

TECHNIQUES D'EXPLORATION ARCHITECTURALE DE DESIGN À USAGE SPÉCIFIQUE POUR L'ACCÉLÉRATION DE BOUCLES

MAME MARIA MBAYE

DÉPARTEMENT DE GÉNIE ÉLECTRIQUE

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION

DU DIPLÔME DE PHILOSOPHIÆ DOCTOR

(GÉNIE ÉLECTRIQUE)

AOÛT 2010

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

TECHNIQUES D'EXPLORATION ARCHITECTURALE DE DESIGN À USAGE SPÉCIFIQUE POUR L'ACCÉLÉRATION DE BOUCLES

présentée par : MBAYE, Mame Maria

en vue de l'obtention du diplôme de : Philosophiæ Doctor

a été dûment acceptée par le jury d'examen constitué de :

M. DAVID, Jean-Pierre, Ph.D., Président

M. SAVARIA, Yvon, Ph.D., membre et directeur de recherche

M. PIERRE, Samuel, Ph.D., membre et codirecteur de recherche

M. LANGLOIS, Pierre J.M., Ph.D., membre

M. ROY, Sébastien, Ph.D., membre externe

À ma grand-mère dont le souvenir sera à jamais gravé dans mon coeur.

REMERCIEMENTS

Je tiens à remercier M. Savaria, mon directeur de doctorat, pour sa disponibilité, ses conseils et son analyse qui m'ont permis de mener à terme mes travaux de recherche dans les meilleures conditions. Je tiens à souligner la contribution de M. Pierre, mon co-directeur de doctorat, pour son support intellectuel et humain. Je désire aussi souligner le support financier de Gennum Corp.

La contribution scientifique de M. Bélanger a été d'une très grande aide quant à la rédaction des différentes publications qui ont résulté de ces travaux. J'en profite pour souligner le support et la gentillesse des secrétaires Mme. Carrier et Mme. Laplante, sans oublier l'aide très appréciée de l'administrateur réseaux M. Lepage.

Je remercie mes amis pour leurs encouragements. Je dois souligner le soutien moral de mes frères et de ma sœur et bien sur celui de mes parents. Et, je terminerai par remercier mon mari qui a supporté les humeurs d'un doctorant, qui a partagé les hauts et les bas d'un doctorat et qui m'a soutenu et motivé par son amour.

RÉSUMÉ

De nos jours, les industriels privilégient les architectures flexibles afin de réduire le temps et les coûts de conception d'un système. Les processeurs à usage spécifique (ASIP) fournissent beaucoup de flexibilité, tout en atteignant des performances élevées. Une tendance qui a de plus en plus de succès dans le processus de conception d'un système sur puce consiste à spécifier le comportement du système en langage évolué tel que le C, SystemC, etc. La spécification est ensuite utilisée durant le partitionement pour déterminer les composantes logicielles et matérielles du système. Avec la maturité des générateurs automatiques de ASIP, les concepteurs peuvent rajouter dans leurs boîtes à outils un nouveau type d'architecture, à savoir les ASIP, en sachant que ces derniers sont conçus à partir d'une spécification décrite en langage évolué. D'un autre côté, dans le monde matériel, et cela depuis très longtemps, les chercheurs ont vu l'avantage de baser le processus de conception sur un langage évolué. Cette recherche a abouti à l'avénement de générateurs automatiques de matériel sur le marché qui sont des outils d'aide à la conception comme CapatultC, Forte's Cynthetizer, etc. Ainsi, avec tous ces outils basés sur le langage C, les concepteurs ont un choix de types de design élargi mais, d'un autre côté, les options de designs possibles explosent, ce qui peut allonger au lieu de réduire le temps de conception. C'est dans ce cadre que notre thèse doctorale s'inscrit, puisqu'elle présente des méthodologies d'exploration architecturale de design à usage spécifique pour l'accélération de boucles afin de réduire le temps de conception, entre autres.

Cette thèse a débuté par l'exploration de designs de ASIP. Les boucles de traitement sont de bonnes candidates à l'accélération, si elles comportent de bonnes possibilités de parallélisme et si ces dernières sont bien exploitées. Le matériel est très efficace à profiter des possibilités de parallélisme au niveau instruction, donc, une méthode de conception a été proposée. Cette dernière extrait le parallélisme d'une boucle afin d'exécuter plus d'opérations concurrentes dans des instructions spécialisées. Notre méthode se base aussi

sur l'optimisation des données dans l'architecture du processeur. Une première contribution de cette méthode est qu'elle préconise l'optimisation et la transformation de boucles dans les instructions spécialisées plutôt que dans le code source original. Une seconde contribution est la proposition d'une représentation à 5-patrons qui permet de concevoir des instructions spécialisées fortement couplées d'une boucle donnée. Un pipeline virtuel est créé durant l'exécution itérative des instructions spécialisées fortement couplées. Une autre contribution de cette méthode est l'application de la réutilisation et du regroupement de données dans des registres spécialisés, afin de diminuer le nombre d'accès mémoire. La méthode de conception de ASIP basée sur l'accélération des boucles a été appliquée sur trois algorithmes de traitement de signaux qui sont : un désentrelaceur ELA, une 2D-DCT et un filtre de Wiener. Les résultats expérimentaux ont montré que la méthode permet d'atteindre des performances importantes avec un facteur d'accélération allant jusqu'à x18, alors que d'un autre côté la complexité matérielle en termes de nombre de portes logiques ajoutées augmentait de 18% à 59%.

La première partie de la thèse doctorale montre le besoin d'une exploration architecturale en amont. En effet, avant de tenter d'extraire le parallélisme et de l'exploiter dans du matériel dédié, il faut d'abord évaluer les possibilités d'accélération et de parallélisme de la boucle traitée. C'est ce problème que nous avons visé dans la seconde partie de notre recherche, en proposant une méthodologie d'exploration architecturale qui contribue à trouver les aspects de la boucle qui limitent ou qui favorisent son accélération. Une des premières contributions de cette méthodologie est qu'elle permet de cibler plusieurs types de design tel qu'un design purement logiciel, un ASIP ou un ASIC. La contribution la plus importante de cette méthodologie est le processus d'analyse itératif basé sur des métriques orientées boucles. Ce processus permet d'explorer les possibilités d'accélération de boucle dans le but de trouver le type de design qui permettra d'atteindre la meilleure accélération. Les nouvelles métriques orientées boucles, entre autres, contribuent à déterminer quels aspects, parmi les

opérations arithmétiques et logiques et les accès de données, limitent l'exploitation des possibilités d'accélération ou quels aspects aideraient à accélérer la boucle. La méthodologie d'exploration architecturale a été appliquée sur des bancs d'essai issus d'applications telles que : JPEG, MPEG, XELA deinterlacer, Turbo Decoder et SAD. Le processus d'analyse a été déroulé pour analyser en profondeur l'accélération de l'algorithme de la SAD par un ASIP. De plus, les estimations d'accélération ont été comparées avec les accélérations de designs détaillés. Pour l'accélération en termes de nombre d'instructions exécutées, les taux d'erreur atteignent un maximum de 2.59%, alors que pour l'accélération en termes de cycles d'horloge, les taux d'erreur varient entre 2.61% et 13.41%.

Les métriques orientées boucles proposées ne tiennent compte que du parallélisme de la boucle au niveau instruction. Donc, une technique d'estimation du temps d'exécution d'un nid de boucles a été caractérisée. Une principale contribution de cette technique est qu'elle estime le temps d'exécution de boucles à plusieurs dimensions en tenant compte des contraintes architecturales du design, telles que le parallélisme et les entrées/sorties. La technique a été utilisée pour estimer le temps d'execution de nids de boucles tirés de l'algorithme du JPEG et de l'algorithme MVM (Matrix Vector Multiply). Les taux d'erreurs atteints varient entre 0% et 29% mais, d'un autre côté, les estimations sont obtenues en moins de 5 minutes alors que les temps d'exécution minimum exacts étaient obtenus après 10 minutes à même 24 heures dans le cas de la MVM (les temps d'exécution réels ont été obtenus en ordonnancant les opérations de nids de boucles entièrement déroulés).

ABSTRACT

Time to market is a very important concern in industry. That is why the industry always looks for new CAD tools that contribute to reducing design time. Application-specific instruction-set processors (ASIPs) provide flexibility and they allow reaching good performance if they are well designed. One trend that gains more and more success is C-based design that uses a high level language such as C, SystemC, etc. The C-based specification is used during the partitionning phase to determine the software and hardware components of the system. Since automatic processor generators are mature now, designers have a new type of tool they can rely on during architecture design. In the hardware world, high level synthesis was and is still a hot research topic. The advances in ESL lead to commercial high-level synthesis tools such as CapatultC, Forte's Cynthetizer, etc. The designers have more tools in their box but they have more solutions to explore, thus their use can have a reverse effect since the design time can increase instead of being reduced. Our doctoral research tackles this issue by proposing new methodologies for design space exploration of application specific architecture for loop acceleration in order to reduce the design time while reaching some targeted performances.

Our thesis starts with the exploration of ASIP design. We propose a method that targets loop acceleration with highly coupled specialized-instructions executing loop operations. Loops are good candidates for acceleration when the parallelism they offer is well exploited (if they have any parallelization opportunities). Hardware components such as specialized-instructions can leverage parallelization opportunities at low level. Thus, we propose to extract loop parallelization opportunities and to execute more concurrent operations in specialized-instructions. The main contribution of this method is a new approach to specialized-instruction (SI) design based on loop acceleration where loop optimization and transformation are done in SIs directly, instead of optimizing the

software code. Another contribution is the design of tightly-coupled specialized-instructions associated with loops based on a 5-pattern representation. Loop optimizations and transformations are performed directly in the 5-pattern representation to leverage loop's parallelism possibilities. The method was also used to show the benefit that can be derived from intermediate results stored in specialized registers, instead of sexternal memory that implies reloading them as needed. Thus, a virtual pipeline is created during specialized-instructions execution. The last contribution but not the least is the use of data grouping and reuse during the sequence-of-SI design to reduce the number of memory accesses. This approach has been applied to video processing algorithms such as the ELA deinterlacer, the 2D-DCT, and the Wiener filter. Experimental results show speedups up to 18x on the considered applications, while the hardware overhead in terms of additional logic gates was found to be between 18% and 59%.

The first part of the thesis shows the importance of design space exploration at high level. Loop-parallelization opportunity extraction and leveraging in dedicated hardware can lead to numerous solutions. The issue is to determine how to design the best solution that leverages acceleration opportunities. Thus, the first step of the architecture design space exploration should be the loop-parallelization opportunity analysis. We tackle this issue by proposing a design space methodology that helps to determine the loop aspects that limit performance or that allow leveraging loop acceleration opportunities. The main contribution of the methodology is an iterative analysis process flow that is based on new loop-oriented metrics to explore loop acceleration opportunities and that can help target the appropriate design style that will provide the best acceleration. The set of loop-oriented metrics help determine which aspect between data access and ALU/Control operations limit loop acceleration opportunities and which optimization technique might improve loop-processing time. These metrics also help evaluate whether the parallelization opportunities of some loop iterations are leveraged. The second contribution of the methodology is a framework in which many types of design can be specified such as pure software-, ASIP- or ASIC- oriented design. The new loop oriented

metrics help determine which aspects between data access and ALU/Control operations limit loop acceleration opportunities and which optimization techniques might improve loop-processing time. These metrics also help evaluate whether the parallelization opportunities of some loop iterations are leveraged. Some benchmarks from the signal and image processing field such as the JPEG, MPEG, XELA deinterlacer, SAD algorithms, were profiled. The iterative analysis process flow was fully unrolled to explore SAD acceleration by a ASIP-oriented design. Acceleration estimates were compared to the performances of detailed designs to gauge the precision of acceleration estimates. The comparisons showed that the error rates for the acceleration expressed as the number of executed instructions never exceeded 2.59%, while the errors on acceleration rates expressed in terms of the number of cycles varied between 2.61% and 13.41%.

Unfortunately, the metrics do not consider some high level parallelism opportunities. That is why a new execution-time estimation technique for nested loops is also presented in this thesis. The main contribution of the technique is that it allows estimating the execution time of multidimensional loops. The technique considers fine-grain parallelism allowed by the targeted design style, and coarse grain parallelism opportunities that depend on loop carried dependencies and I/O oriented dependencies. Thus, the estimated execution time is used to compare and to accurately gauge the efficiency of potential designs. Another significant contribution of the technique is that estimations are computed very fast while errors rate are reasonable. In fact, the technique was applied to estimate execution time of nested loops extracted from the JPEG and MVM (Matrix-Vector Multiplier) algorithms. The error rates vary between 0% to 29%, but the estimates were computed in less than 2 minutes, while accurate bounds on execution times were obtained after 10 minutes for the JPEG derived benchmark, and in 24 hours in the case of the MVM benchmark (the real execution times were obtained by scheduling the fully unrolled nest of loops.).

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

SIGLES ET ABREVIATIONS

ADL : Architectural Description Language

ASIC : Application-Specific Integrated Circuit

ASIP : Application-Specific Instruction-Set Processor

CS : Computation Step

CSTCSI : Clustered Sequence of Tightly-coupled Specialized-Instructions

DSP : Digital Signal Processing

FU : Functional Unit

HLS : High Level Synthesis

IS: Instructions Spécialisées

MAC : Multiplieur Accumulteur

MMD : Module Matériel Dédié

RISC : Reduced Instruction Set Computer

SARE : System of Affine Recurrent Equations

SI : Specialized-Instruction

SIMD : Single Instruction Multiple Data

SoC : System on Chip

VHDL : Very High Description Language

VLIW : Very Long Instruction Word

WCET : Worst Case Estimation Time

LISTE DES ANNEXES

CHAPITRE 1
INTRODUCTION

La loi de Moore prédit que le nombre de transistors par puce électronique pourra être doublé sans coût en terme de surface à tous les deux ans [124]. Malheureusement, la productivité des ingénieurs qui les conçoivent n'augmente pas à cette vitesse. C'est la raison pour laquelle l'industrie est toujours à la recherche de nouveaux outils et de méthodologies qui contribuent à combler l'écart entre l'évolution des technologies de fabrication et celle de la productivité des ressources humaines. Une solution qui a de plus en plus de succès est la conception basée sur le langage C (« C-based design »). Ce type de solution s'applique à des designs purement logiciels, purement matériels ou à ceux qui sont entre les deux. Pendant plusieurs décennies, les concepteurs de logiciel et de matériel ont œuvré dans des univers parallèles, bien que leurs travaux aient été fortement reliés. Cependant, avec l'utilisation accrue de processeurs dans les architectures à usage spécifique, chacun a avantage à comprendre ce que l'autre fait afin de tirer avantage des solutions offertes par l'autre communauté. La jonction des lignes de pensée est très significative quand il s'agit de design conçu avec des outils de synthèse à haut niveau. Ces outils de synthèse utilisent une description en langage évolué comme le C et ils génèrent le circuit matériel correspondant en langage RTL (*Register Transfer Level*) tel que VHDL (*VHSIC(Very High Speed Integrated Circuits) Hardware Description Language*) [38] ou sous forme de portes logiques [65].

La jonction des deux lignes de pensée a aussi lieu quand il s'agit de concevoir des processeurs à usage spécifique. L'objectif principal de ces différents types d'architectures est d'accélérer le temps de conception d'un design en se basant sur un prototype décrit dans un langage évolué. C'est dans ce contexte que notre travail de recherche doctorale s'inscrit. Nous ciblons l'exploration des possibilités d'accélération de boucles décrites en langage C par des architectures à usage spécifique de type ASIP (*Applicaton Specific Instruction-set Processor*) ou ASIC (*Application Specific Integrated Circuit*), par

exemple. Dans la suite de cette section, nous parlerons des éléments de la problématique. Nous formulerons par la suite nos objectifs de recherche et enfin nous présenterons un plan de la suite de ce document.

1.1 Éléments de problématique

Pendant très longtemps, les processeurs intégrés dans les SoC (*System-On-Chip*) ont joué le rôle de contrôleur dans l'architecture. Cependant, avec certains types de processeurs comme les processeurs de traitement de signal, communément appelés DSP (*Digital Signal Processor*), les processeurs peuvent contribuer considérablement à accélérer une application tout en augmentant la flexibilité de l'architecture, ce que les éléments matériels non programmables (qu'on appellera des modules matériels dédiés – MMD) ne permettent pas d'atteindre aussi aisément. À l'heure actuelle, on trouve plusieurs types de processeurs de systèmes embarqués, mais ceux qui attirent de plus en plus d'attention sont les processeurs à usage spécifique (ASIP). Ces processeurs constituent une alternative intéressante entre un processeur à usage général et un coprocesseur matériel dédié. Un ASIP est conçu en fonction des différentes opérations de l'application qu'il exécutera. Selon les besoins de l'application, le concepteur ajoutera des unités de calcul nécessaires à la bonne exécution de l'application. Certains ASIP, comme ceux de la société Tensilica [168], ont un jeu d'instructions de base que le concepteur peut étendre avec de nouvelles instructions. La conception du jeu d'instructions est une étape délicate. Vu le nombre élevé de combinaisons d'instructions possibles, il n'est pas évident pour le concepteur de trouver les meilleures combinaisons d'instructions spécialisées (IS) qui permettront d'accélérer l'application, raison pour laquelle on retrouve de plus en plus d'outils de génération automatique d'instructions spécialisées [140, 168]. Cependant, les résultats de ces outils sont quelque peu décevants, car généralement, ils ne tiennent compte que des possibilités de parallélisme au niveau instruction. Aussi, de nouvelles approches de génération de jeux d'instructions profitant des possibilités de parallélisme de l'application à haut niveau contribueraient à obtenir de meilleures accélérations.

La complexité et la taille des applications de systèmes embarqués évoluent à une vitesse exponentielle, ce qui pousse les concepteurs à démarrer le processus de conception de SoC à très haut niveau, afin de capturer toutes les fonctionnalités de l'application. Ces fonctionnalités sont spécifiées dans des langages de haut niveau tels que C, C++, SystemC, etc. De plus, pour que les systèmes soient flexibles et réutilisables, les concepteurs privilégient l'insertion de composantes logicielles dans les systèmes. Ainsi, de nombreux SoC sont en fait des systèmes matériel/logiciel (HW/SW). L'intégration des composantes logicielles complique la tâche des architectes et soulève le problème du partitionnement logiciel/matériel. Les architectes doivent avoir des connaissances poussées en conception de logiciel pour des systèmes embarqués, tout en étant des experts de design matériel. Dans ce contexte, les architectes doivent suivre des méthodologies de conception rigoureuses, moins basées sur l'expérience du concepteur, mais plutôt fondées sur une approche scientifique. Ils doivent aussi avoir des outils d'aide à la conception qui les supportent durant leur choix de partitionnement.

Durant le processus de partitionnement d'une architecture SoC, le concepteur définira quels segments critiques devraient être accélérés par un ou des éléments matériels non-logiciels (MMD). Nous avons volontairement évité de parler d'ASIC qui est un terme souvent employé, car de plus en plus d'architectures sont déployées sur des FPGA. La conception des MMD peut être très longue et coûteuse en termes de ressources humaines ou de temps de conception et de fabrication, par exemple. En fait, les concepteurs de MMD ne profitent pas entièrement des capacités technologiques disponibles; l'évolution de la courbe de productivité est très en retard par rapport à celle de l'évolution des capacités technologiques qui suit la loi de Moore [121]. Le temps que les concepteurs se familiarisent avec une nouvelle technologie, la prochaine technologie est déjà disponible. Donc, le problème qui se pose est de réduire le temps de conception des MMD et plus largement celui des SoC. Une des solutions qui s'offre actuellement est l'utilisation d'outils de synthèse à haut niveau. Ces outils permettent de décrire à haut niveau une spécification en langage C, par exemple, et ils se chargent de générer le code RTL

correspondant. Cependant, ces outils nécessitent une connaissance des principes de conception matérielle et le code donné en entrée doit être très bien écrit, sans compter qu'il faudrait fournir à l'outil un segment de l'application qui s'avère pertinent à accélérer.

Une des premières étapes du processus de partitionnement est le profilage d'une spécification de l'application, afin de noter les segments critiques de cette dernière. Les concepteurs ou les outils d'aide à la conception exploitent les informations colligées durant le profilage. Généralement, surtout lorsque le code est complexe, le volume de résultats de profilage peut être impossible à analyser par l'humain. Malheureusement, les outils de profilage n'ont pas beaucoup évolué avec le temps. Ils se limitent toujours à donner le temps d'exécution ou le nombre de cycles de l'application. Ils précisent aussi le nombre d'appels des différentes fonctions durant l'exécution de l'application. Ces informations sont utiles mais, vu les enjeux et les paramètres considérés durant le processus de conception d'architectures matérielles, le temps d'exécution ou le nombre d'appels des fonctions ne sont plus suffisants. Suresh et al. [163] ont proposé l'implémentation d'un profileur orienté boucle qui collecte le temps d'exécution des boucles d'une application, mais cela reste insuffisant. Le choix du type de design le plus approprié pour accélérer une boucle requiert des métriques plus précises afin de caractériser ses possibilités d'accélération et les aspects limitant ou favorisant son accélération.

Durant le processus d'exploration architecturale, le but est d'accélérer l'exécution d'un segment de code de la spécification C. Les boucles sont de bonnes candidates à accélérer par un design à usage spécifique. Cependant, le succès de leur accélération repose non seulement sur les dépendances de données implicites de la spécification, mais aussi sur les contraintes de design de l'architecture ciblée. Donc, le concepteur, avant de s'aventurer à essayer d'accélérer une boucle, devrait connaître les limites d'accélération ainsi que le temps d'exécution atteignable. En effet, dans certains cas, les concepteurs s'évertuent à vouloir accélérer une application qui n'est pas accélérable, donc une

estimation du temps d'exécution atteignable leur permettrait de cibler une accélération raisonnable et de mieux planifier le processus d'exploration architecturale.

1.2 Objectifs de recherche et contributions

L'un des principaux objectifs de cette recherche doctorale est de proposer une méthode de conception d'instructions spécialisées pour accélérer les boucles critiques d'une application. En effet, jusqu'à présent, les travaux de recherche sur la génération d'instructions spécialisées se sont focalisés sur l'exploitation des possibilités de parallélisme au niveau instruction mais, pour atteindre des accélérations plus élevées, il faut exploiter le parallélisme offert par les boucles de l'application. Cette méthode de conception de ASIP a fait l'objet d'un premier article [118] publié dans la revue « Journal of VLSI Signal Processing » en 2007. Les contributions de cette méthode de conception de ASIP sont:

- Une nouvelle approche de conception d'instructions spécialisées (IS) basée sur l'accélération de boucle en sachant que les optimisations de boucles sont effectuées directement sur les instructions au lieu d'être effectuées sur le code logiciel.

- La méthode innove en permettant de concevoir des instructions fortement couplées qui permettent de créer un pipeline virtuel.

- Cette méthode contribue à améliorer les performances, mais elle permet aussi de réduire le nombre d'accès en mémoire en appliquant du regroupement et de la réutilisation des données durant la conception de la séquence d'IS.

Le second objectif de recherche vise la proposition de métriques pour l'exploration des possibilités d'accélération d'une boucle avec une architecture telle qu'un processeur à usage spécifique ou un MMD généré par un outil de synthèse à haut niveau. En effet, les travaux de recherche se sont beaucoup focalisés sur des techniques de synthèse à haut niveau, mais il faudrait d'abord s'assurer que le type d'architecture visé est approprié à

l'accélération de la boucle. Les métriques proposées ont fait l'objet d'un article soumis à la revue « IEEE Transactions on VLSI » en 2010. Les contributions des travaux présentés dans cet article sont :

- Un cadre d'application dans lequel différents types d'architectures à usage spécifique peuvent être spécifiés. En effet, les contraintes de design considérées permettent de spécifier un processeur à usage général, un ASIP ou un ASIC.

- La proposition de nouvelles métriques orientées boucles qui donnent une vue sur le poids des calculs et des accès en mémoire dans le temps d'exécution d'une boucle donnée sur l'exploitation des possibilités d'accélération. Les métriques permettent aussi de trouver les facteurs qui contribuent à limiter ou à favoriser l'accélération d'une boucle au niveau instruction.

- Une méthodologie d'exploration architecturale basée sur les métriques orientées boucles qui aident le concepteur à trouver les transformations ou les optimisations pouvant améliorer l'accélération de la boucle et qui aussi aident à trouver le type d'architecture le plus approprié à l'accélération d'une boucle donnée.

Le troisième objectif de recherche vise la proposition d'une méthode d'estimation statique du temps d'exécution d'un nid de boucles accéléré par une architecture à usage spécifique. La technique a été présentée dans un article soumis à la revue « IEEE Transactions on CAD » en 2010. Les contributions apportées dans cet article sont :

- La prise en compte des opportunités de parallélisme au niveau instruction et à haut niveau durant le processus d'estimation du temps d'exécution d'un nid de boucles.

- L'utilisation d'un modèle mathématique pour représenter exactement les dépendances de données d'un nid de boucles. Ainsi, seuls les espaces d'itérations valides seront considérés durant l'estimation du temps d'exécution. En effet, les

méthodes existantes ont tendance à considérer des espaces d'itérations qui ne seront pas exécutés. Généralement, elles calculent les chemins pouvant être exécutés mais elles n'évalueront pas les itérations exécutables.

• Durant l'estimation des performances et en tenant compte des possibilités de parallélisme à haut niveau, les contraintes de design sont intégrées dans le modèle mathématique. En effet, les opportunités de parallélisme ne seront exploitées que si les contraintes de design le permettent. Il est nécessaire de tenir compte de ces dernières pour estimer le temps d'exécution, ce que les méthodes d'estimation du temps d'exécution d'une boucle ne font pas.

1.3 Plan de la thèse

Dans la suite de ce document, une revue de littérature qui expose les travaux significatifs reliés à notre recherche est présentée au chapitre 2. Dans un premier temps, une rétrospective des tendances de conception de processeurs spécialisés est présentée. Cette rétrospective est subdivisée en trois thèmes, à savoir la génération d'un processeur et de ses outils de support, la génération de la banque de registres et la génération du jeu d'instructions spécialisées. Puis, des travaux sur la synthèse à haut niveau sont présentés. Nous décrivons brièvement les différentes étapes du processus de synthèse à haut niveau. De nombreux travaux sur les nids de boucles sont basés sur une représentation mathématique nommée le modèle polyèdral. Ce modèle est une représentation mathématique de nid de boucles qui est utilisée, entre autres, pour détecter exactement les dépendances de données inhérentes d'un nid de boucles. Nous présentons le modèle polyèdral ainsi que différentes techniques de transformation et de parallélisation basées sur ce modèle. Étant donné qu'une partie de notre recherche porte aussi sur l'estimation du temps d'exécution d'un nid de boucles, nous présentons des travaux sur l'analyse statique du temps d'exécution et plus précisemment sur l'estimation du pire temps d'exécution (WCET).

Étant donné qu'il s'agit d'une thèse par articles, il est nécessaire d'exposer la démarche scientifique que nous avons adoptée durant cette recherche doctorale. Donc, au chapitre 3, nous parlons des événements qui nous ont amené à fixer nos objectifs de recherche et nous discutons de la pertinence des articles par rapport à nos objectifs. Nous discutons aussi du choix des revues dans lesquelles les articles ont été publiés ou soumis et nous faisons le point sur le statut des articles.

Chacun des articles a été incorporé dans le corps de la thèse. Le premier article, inséré au chapitre 4, présente une méthodologie de conception de processeurs spécialisés pour l'accélération de boucles. Au chapitre 5, un second article vise l'exploration architecturale de design à usage spécifique pour l'accélération de boucle. Au chapitre 6, le dernier article se focalise sur l'estimation du temps d'exécution d'un nid de boucles sur une architecture à usage spécifique.

Au chapitre 7, nous discutons en détails de nos travaux de recherche en faisant un retour sur les articles exposés. Nous parlerons des choix que nous avons eu à faire pour obtenir les résultats présentés. Ensuite, au chapitre 8, nous concluons en résumant brièvement le contenu de la thèse et en faisant des recommandations pour la suite de cette recherche.

CHAPITRE 2

REVUE DE LITTÉRATURE

Pendant très longtemps, deux importants domaines de recherche, à savoir celui des compilateurs logiciels et celui de la conception de matériel (processeurs, ASIC) ont évolué de façon indépendante. Des techniques d'optimisation et de transformation ont été proposées pour concevoir le meilleur code logiciel et, dans l'autre domaine, pour concevoir l'architecture matérielle qui atteindrait les meilleures performances au moindre coût. Finalement, de plus en plus, on voit que ces deux domaines commencent à partiellement fusionner pour certaines familles d'applications. Les codes logiciels conçus visent souvent un déploiement sur un processeur à usage général. Par contre, petit à petit, avec l'émergence des processeurs plus spécifiques tels que les DSP, les spécialistes se sont rendus compte que le compilateur devrait de plus en plus prendre en compte les caractéristiques de l'élément matériel sur lequel le logiciel serait déployé, afin de profiter au maximum de la puissance de calcul du matériel. Comme l'a bien exposé Leupers [98], pour profiter pleinement de la puissance de calcul du matériel, les développeurs de logiciels doivent souvent écrire leur code en assembleur au lieu d'utiliser des langages de haut niveau. Par exemple, les compilateurs de processeurs DSP ou ASIP ont beaucoup de mal à générer un code efficace. Leupers [98] a énuméré les techniques de génération de code selon trois étapes d'optimisation : la première étape concerne l'optimisation du code source, la seconde étape vise la mise en correspondance du code source avec le jeu d'instructions du processeur, et la troisième étape se rapporte à l'optimisation du code assembleur. Dans la littérature, de nombreux travaux ont porté sur les différentes techniques associées à ces phases d'optimisation. Il faut noter qu'une classe d'optimisation de code source très efficace est la transformation des boucles.

Les études ont montré que près de 90% du temps d'exécution est attribuable à 10% des lignes de code d'une application, en sachant que ces 10% de lignes décrivent

généralement des boucles [163]. Les techniques telles que la fusion de boucle, le déroulement de boucle, le pipelinage de boucle [74, 129], le report de boucle mailles (« *Loop Shifting* ») [43] sont très efficaces et elles permettent d'accélérer considérablement une application lorsque cette dernière s'y prête. En effet, selon le flux de données à traiter ou la taille des données manipulées par les boucles, certaines techniques d'optimisation de boucles pourraient coûter cher en termes d'espace et même d'énergie consommée. L'application des techniques de transformation doit se faire prudemment, comme l'ont mentionné Whitfield et al. [183]. Ces derniers attirent l'attention sur le fait que l'application de certaines transformations sur le code peut réduire les chances d'appliquer d'autres transformations qui permettraient d'atteindre de meilleurs gains. C'est la raison pour laquelle ils ont proposé une plateforme qui explore les possibilités de transformation d'un code et l'ordre dans lequel les transformations devraient être effectuées.

Une autre famille de techniques d'optimisation concerne l'optimisation des données. En effet, le nombre d'accès mémoire ou la taille des unités mémoire sont des facteurs très importants dont l'optimisation permet, en plus d'accélérer l'application, de réduire l'énergie consommée et les besoins en espace. Panda et al. [130] ont présenté une étude exhaustive des techniques d'optimisation de données. Par exemple, ils montrent comment, par des transformations de boucles améliorant la localité et la régularité d'accèes aux données, le nombre d'accès mémoire peut être réduit. Autre exemple, Kandemir et al. [82] ont proposé une technique d'amélioration de la localité des données d'un nid de boucles. En effet, une des causes d'échec lors d'accès à l'antémémoire est la faible localité des données accédées par les boucles, donc en améliorant la localité des données de nids de boucles, ils arrivent à réduire le taux d'échec de l'antémémoire.

En fait, la réduction des accès mémoire est devenue un enjeu autant pour la conception de code logiciel que pour la conception de matériel dédié. Par exemple, la technique de minimisation du trafic vers la mémoire de Kolson et al. [88] profite des redondances d'accès mémoire qu'on retrouve dans les boucles. Masselos et al. [112] ont

proposé une méthodologie assez efficace basée sur l'application de trois types de transformations : l'élimination de certains accès mémoire, la réduction de la taille des mémoires par l'implémentation d'une hiérarchie de mémoires, l'ordonnancement des accès mémoire par des techniques de réutilisation de données.

D'autres, comme Catthoor et al. [26], ont proposé des techniques pour limiter les besoins en termes d'espace mémoire d'une application. Ces techniques de réduction d'espace se basent sur l'analyse des dépendances de données entre les variables de l'application et le temps de vie de ces dernières. C'est ainsi que des chercheurs du groupe de recherche IMEC [76] ont suggéré une technique d'estimation des dépendances de données afin d'évaluer les besoins en mémoire d'une application [87].

D'un autre coté, les concepteurs de matériel doivent faire leur part en fournissant un matériel qui permettra de profiter des capacités de parallélisme des applications. Une approche qui prend de plus en plus d'ampleur pour la conception de design à usage spécifique est basée sur une spécification en langage C. Cette approche permet de faire le lien entre les mondes logiciel et matériel. À partir de la même spécification, une architecture logicielle ou matérielle peut être conçue selon les objectifs de conception. Trois principaux types de design peuvent être ciblés. Le premier est un design purement logiciel, le second est un design de processeur à usage spécifique (ASIP) et enfin, le troisième est un design purement matériel généré automatiquement par un outil de synthèse à haut niveau (HLS). Dans la suite de cette section, nous présentons les travaux que nous avons jugés pertinents dans la recherche de conception de ASIP. Dans un second temps, nous décrivons le processus de synthèse à haut niveau d'un design purement matériel. Ensuite, le modèle polyèdral, qui permet de représenter mathématiquement un nid de boucles, sera explicité, ainsi que les différentes représentations de dépendances de données d'un nid de boucles. Puis, différentes applications du modèle polyèdral seront montrées. Enfin, la section sera complétée par une rétrospective de différentes techniques d'analyse statique du temps d'exécution d'une application.

2.1 Processeurs à usage spécifique

Pendant très longtemps, la conception de processeurs spécialisés se faisait de manière *ad hoc*. En parallèle, la technologie des processeurs DSP s'est beaucoup développée. Ces processeurs pourraient être considérés comme des ancêtres des ASIP. En effet, les DSP sont conçus pour accélérer les applications de traitement des signaux, alors que les ASIP sont supposés être conçus pour une application spécifique, même si de plus en plus d'ASIP sont dédiés à une famille d'applications de type DSP. Par exemple, Tensilica a conçu une famille de processeurs appelée Diamond VDO [168] qui comportent plus de 400 instructions spécialisées visant la famille des applications de traitement vidéo.

Dans cette section, nous décrivons succintement des travaux qui ont porté sur la génération automatique d'un ASIP. Ensuite, nous discutons des différents défis que les chercheurs ont visés comme l'exploration d'architectures comportant des banques de registres et la génération automatique de jeux d'instructions spécialisées. Puis, nous mentionnons des recherches connexes à la conception d'ASIP qui portent sur l'estimation de l'énergie consommée ou les ASIP reconfigurables, par exemple.

2.1.1 Génération de processeur

Deux tendances existent dans le monde de la génération automatique de processeurs ASIP. Les deux tendances doivent permettre de générer automatiquement la suite d'outils logiciels de support tels qu'un compilateur, un simulateur, un profileur, etc. La première tendance repose sur la conception d'une architecture de base dont le jeu d'instructions pourra être étendu, comme c'est le cas des processeurs de Tensilica ou du NIOS de Altera [4]. Ces processeurs sont généralement appelés processeurs configurables, car ils comportent une architecure et un jeu d'instructions de base. Puis, selon les besoins du concepteur, des unités fonctionnelles pré-existantes pourront y être rajoutées ou du matériel dédié pourra y être inséré comme des instructions spécialisées, par exemple. Généralement, des interfaces prédéfinies permettent de relier l'architecture de base et les composantes rajoutées. Comme illustré par la Figure 1, l'architecture du processeur de

Tensilica comporte des blocs de base qui ne peuvent pas être modifés, tel que l'ALU de base. L'architecture comporte des blocs qui sont configurables, tels que la banque prédéfinie de registres, dont la taille peut être fixée selon les besoins du concepteur. Certains blocs comme les MAC (multiplieur-accumulateur) et les multiplieurs de 16 ou 32 bits sont optionnels. Ils peuvent donc être inclus ou retirés de l'architecture. Finalement, des blocs fonctionnels conçus exclusivement par le concepteur peuvent être rajoutés dans l'architecture du processeur; il s'agit des registres et des instructions spécialisés.

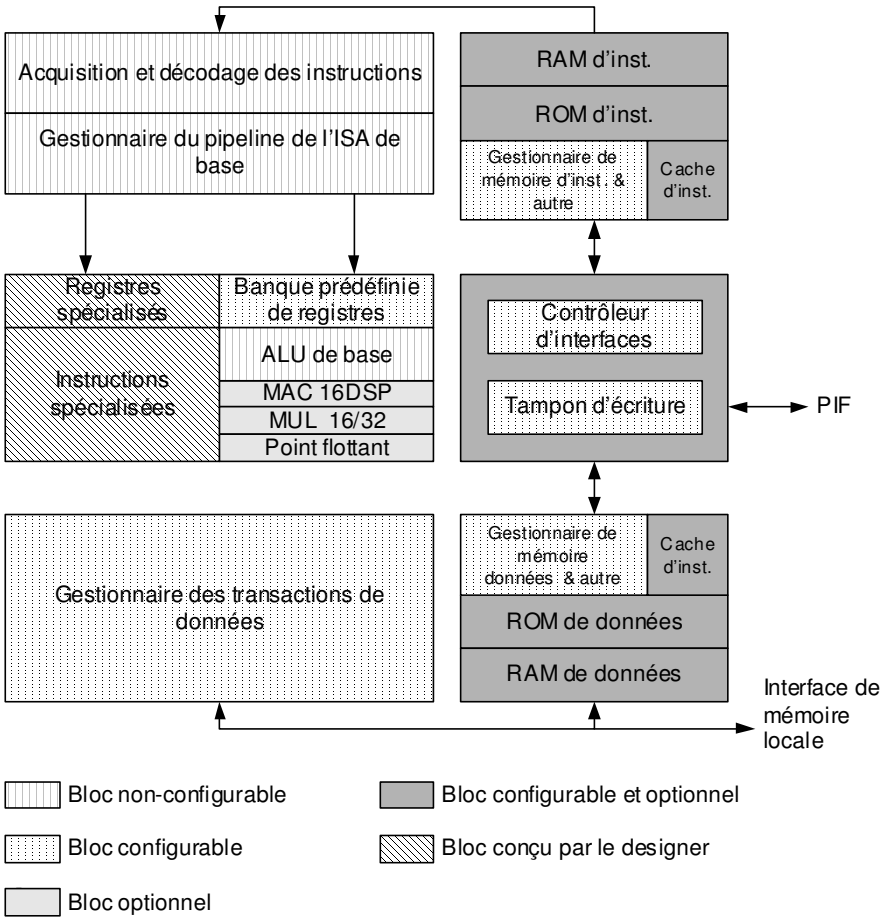


Figure 1. Vue simplifiée de l'architecture du processeur Xtensa 7.

La seconde tendance repose sur des langages de description architecturale (ADL) tels que LISA [21] et EXPRESSION [68]. Ces langages sont utilisés pour décrire

formellement toute l'architecture d'un processeur et ainsi permettre de générer automatiquement la suite d'outils accompagnant l'architecture. Ce type de langage contribue aussi à faciliter l'analyse formelle de l'architecture en effectuant, par exemple, des analyses de cohérence, d'ambiguïté et de performance sur celle-ci. Ces langages ADL pour la conception de ASIP permettent de décrire les différentes unités du processeur et son jeu d'instructions. Un des grands défis concerne la génération du compilateur et celle du simulateur, comme l'ont montré Braun et al. [21]. La facilité et la vitesse de génération du processeur ont permis de consacrer plus de temps à la recherche d'algorithmes de génération de jeu d'instructions spécifiques à une application donnée. Certains outils de conception de ASIP comme Lisatek de la compagnie Coware [42] ne permettent pas de sélectionner des unités fonctionnelles pré-fabiquées. Ces outils entrent dans la famille de processeurs décrits par un ADL. Un processeur conçu avec Lisatek requiert la conception du processeur à partir de zéro, donc il faudra définir son architecture, son jeu d'instructions en entier, les étages de son pipeline s'il y a lieu, etc.

2.1.2 Génération de la banque de registres et utilisation des mémoires

Jain et al. [81] ont montré l'importance de la taille de la banque de registres durant la conception d'un ASIP. Ils ont proposé une technique assez efficace de caractérisation de la taille de la banque de registres. Leur technique, par une analyse des résultats de profilage combinée à l'analyse des dépendances de données, allouera des registres tout en appliquant une stratégie de réutilisation de registres. Ils améliorent leur technique dans [80] en tenant compte du nombre d'échecs d'accès à l'antémémoire pour définir la taille de l'antémémoire ainsi que celle de la banque de registres. La caractérisation de l'antémémoire devient un enjeu de taille durant la conception d'un ASIP.

Plus récemment, Nalluri et al. [126] ont proposé une technique d'évaluation de la banque de registres pour la conception de processeurs à faible consommation de puissance. En effet, une observation montre que 80 à 90% des accès sur la banque de registres concernent 4 à 10 registres sur un total de 32 environ. Donc, la solution qu'ils

proposent vise à identifier la meilleure structure de banque qui permettra de diminuer la puissance dissipée induite par les accès à la banque de registres. Ils proposent la construction d'une architecture à deux banques de registres, Banque0 et Banque1, et les différents registres seraient répartis sur les 2 banques sans dédoublement, ce qui peut permettre de relâcher la pression qu'une seule banque supporterait.

2.1.3 Génération du jeu d'instructions spécialisées

Dans un premier temps, certains chercheurs [27-29] se sont surtout attardés à proposer des méthodologies de recherche des coprocesseurs ou des unités fonctionnelles qui accéléreraient une application spécifique. Ces études ont montré que l'ajout d'unités spécifiques de calcul comme un MAC ou un coprocesseur de calcul en virgule flottante, pouvait être très efficace pour accélérer une application. Il est cependant notable que ces premiers travaux ne prenaient pas en compte les contraintes d'espace. C'est dans ce cadre que Imai et al. [75] ont proposé une méthodologie de partitionnement logiciel-matériel qui vise la génération d'unités fonctionnelles (FU) et qui tient compte des coûts des FU. D'autres chercheurs, tels que Cheung et al. [33], ont combiné les approches de [75] et [67] et ont ajouté une passe d'analyse de surface de silicium durant le processus de conception du jeu d'instructions spécialisées. L'approche de Cheung et al. [33] commence par la sélection d'unités fonctionnelles prédéfinies, suivie de la sélection de possibles instructions spécialisées. Ensuite, ils estiment les performances et les coûts associés à chacun des éléments sélectionnés. Enfin, la sélection définitive est effectuée en tenant compte des contraintes d'espace spécifiées. La méthodologie de Cheung et al. [33] vise la conception de processeurs configurables dont le jeu d'instructions peut être étendu comme on peut le faire avec la technologie de Tensilica.

La recherche sur l'extraction d'un jeu d'instructions spécialisées a beaucoup évolué récemment. Plusieurs solutions ont été proposées [40, 63, 146] pour la sélection d'un jeu d'instructions spécialisées (IS). La plupart des travaux de recherche se basent sur l'analyse du graphe de flot de données de l'application à accélérer. En effet, dans la

majorité des cas, le graphe est analysé et des sous-graphes qui constituent des segments potentiellement intéressants à accélérer en sont extraits. Comme l'ont mentionné Galuzzi et al. [60], la recherche d'IS par analyse de graphe comporte trois problèmes : la complexité de l'exploration du graphe, la forme du graphe et l'imbrication (« overlap ») des patrons de sous-graphes. En effet, pour un nombre n de noeuds dans le graphe, il y aura 2^n sous-graphes possibles à analyser. C'est la raison pour laquelle on retrouve de nombreuses heuristiques qui réduisent l'espace d'exploration [140]. D'un autre coté, des algorithmes exacts sont aussi proposés, mais ils doivent parcourir tous les espaces possibles de recherche. Bien sûr, les heuristiques ne fournissent pas nécessairement des résultats optimaux, mais leurs résultats sont disponibles en moins de temps par rapport à ceux des algorithmes exacts. D'un autre côté, la forme du graphe étudié a un impact sur la qualité de la génération. En effet, selon qu'on traite un graphe acyclique ou cyclique, cela pourrait avoir un impact sur l'ordre d'exécution des noeuds et la détection du parallélisme. Le problème de l'imbrication apparaît lorsque l'algorithme de génération repose sur des patrons prédéfinis et que certains patrons, détectés durant l'analyse de graphe, se chevauchent.

Dans l'ensemble, les algorithmes de génération d'IS génèrent deux types d'instructions : les MISO (« multiple input and single output ») et les MIMO (« multiple input and multiple output »). Ces deux types se différencient par le nombre de sorties que l'instruction peut avoir; dans le cas des MISO [40, 61], l'instruction n'a qu'une seule sortie, alors que dans le cas des MIMO [9, 216], l'instruction pourra avoir plusieurs sorties. Bien sûr, cette différence a un impact considérable sur le nombre d'instructions trouvées et leur efficacité en terme de performance.

Bonzini et al. [19] ont très bien présenté les différentes approches employées pour l'extraction d'instructions spécialisées. Ils catégorisent les approches existantes en quatre types. Le premier type est itératif et il se compose de deux phases : une première phase trouve un sous-graphe qui offrirait des possibilités d'accélération et la seconde phase extrait les instructions spécialisées potentielles qui accéléreraient le sous-graphe résultant

de la première phase. Cette approche n'offre pas une couverture optimale du graphe de l'application, puisqu'elle vise les sous-graphes associés aux blocs de base de l'application. Un second type d'approche s'attaque aux inconvénients causés par la limite de deux entrées et une sortie des IS simples, en sachant que cette limite est souvent imposée par certains générateurs de processeurs spécialisés. Cette approche insère un ordonnanceur d'entrée/sortie qui réajuste les gains d'un sous-graphe selon la limite sur la bande passante de la banque de registres qui est, en fait, une contrainte de la technologie.

Le troisième type d'approche, inspirée de la solution de Clark et al. [35], se compose d'une première phase durant laquelle une liste des sous-graphes est extraite puis une analyse des sous-graphes isomorphes est effectuée, pour ensuite générer les IS. En effet, certaines recherches comme celles de Clark [35] et de Wolinski [187] ont employé la détection de sous-graphes isomorphes, afin d'exploiter les possibilités de réutilisation de matériel. Il ne faut pas oublier que l'ajout d'IS a un prix en termes de portes logiques, donc la réutilisation de matériel peut être un critère de sélection des IS. L'analyse individuelle des sous-graphes de l'application ne permet pas de profiter des similitudes qui pourraient exister entre ces derniers. Ce troisième type d'approche effectue une couverture optimale des branches de l'application mais elle ne prend pas en compte les contraintes technologiques.

Le dernier type d'approche proposée par [19] est inspirée du troisième type d'approche. Bonzoni et Pozzi suggèrent l'énumération des sous-graphes selon les contraintes architecturales de l'application (par exemple, l'élimination des sous-graphes trop longs ou dont la complexité de mise en œuvre matérielle serait trop élevée), ce qui permet d'éviter une recherche et une analyse complètes de tous les sous-graphes de l'application. Bonzoni et al. [19] proposent la détection de sous-graphes isomorphes, ce qui augmenterait les possibilités de réutilisation de matériel. Ensuite, la sélection et la génération des IS associées aux sous-graphes sont effectuées en fonction des contraintes technologiques. En effet, en fonction de la technologie visée FPGA ou ASIC, les IS sélectionnées seront sûrement différentes. Par exemple, la Figure 2 illustre un cas dans

lequel on retrouve trois blocs de base (BB) : BB1, BB2 et BB3. À chacun de ces BB est associé un sous-graphe.

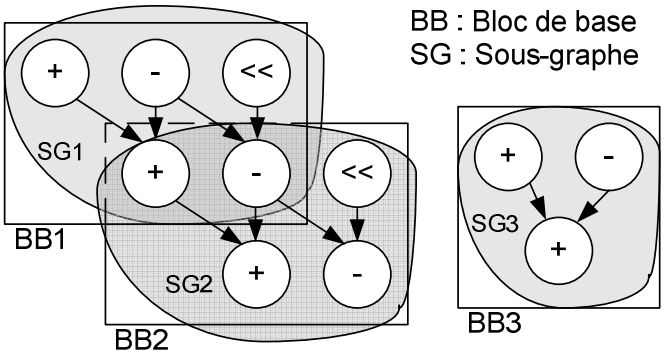


Figure 2. Exemple d'extraction de sous-graphes

Dans le cas d'une recherche exhaustive des sous-graphes, on aurait obtenu beaucoup plus de sous-graphes. On remarque aussi les deux sous-graphes isomorphes SG1 et SG2. Si une seule instruction spécialisée est conçue pour exécuter la fonctionnalité de SG1 et SG2, l'économie en termes de complexité d'une mise en œuvre matérielle est évidente.

Soulignons que plusieurs des algorithmes de génération d'instructions spécialisées reposent sur une fonction de coûts et une fonction d'objectifs. En effet, la conception d'un processeur spécialisé a un coût mais elle permet aussi d'atteindre un ou des objectifs en termes de performance. Ainsi, généralement, la génération pourrait être guidée par les performances à atteindre en respectant des coûts en termes de portes logiques ou d'énergie consommée.

2.1.4 Autres travaux significatifs

L'importance du nombre d'opérandes sur les performances des IS a été démontré [40, 194]. En fait, Yu et al. [192] ont montré qu'en plus du nombre d'opérandes, les performances des IS pouvaient dépendre des contraintes d'espace, du nombre d'IS et des blocs de contrôle (à savoir le nombre de blocs de base extraits de l'application). En effet, de nombreux compilateurs ne permettent pas d'extraire un sous-graphe chevauchant plusieurs BB.

Karuri et al. [84] ont proposé d'effectuer un profilage plus étoffé de l'application, ce qui permettrait de mieux choisir les segments de l'application à accélérer et dont les résultats pourraient être exploités par le compilateur d'instructions spécialisées. Par exemple, leur profileur collecte la fréquence d'exécution des branches de l'application ainsi que la longueur de leurs sauts. Ceci, en plus d'aider à la génération d'IS, permettrait de prendre de meilleures décisions en ce qui concerne les schémas de prédiction de branches, par exemple.

En plus d'accélérer l'application, un ASIP peut contribuer à diminuer l'énergie consommée par un système. Une technique très efficace de réduction d'énergie est basée sur l'optimisation de l'encodage des instructions [17, 89, 183]. Une autre technique très répandue est basée sur les architectures VLIW comme l'ont mentionné Jacome et al. [79]. En effet, encoder plusieurs opérations dans le même mot d'instruction réduit le nombre d'instructions à charger. La technique de Nalluri et Panda [127] citée précédemment contribue à diminuer les coûts de décodage d'adresses, par exemple. Nous tenons aussi à mentionner les travaux d'un groupe de recherche de l'IMEC [77] portant sur différents aspects de la conception d'un processeur à usage spécifique à faible consommation d'énergie. Leurs travaux concernent principalement l'optimisation de la localité des données, la conception d'architectures optimisant le chemin de données et la distribution des instructions dans une hiérarchie de mémoires, entre autres.

D'autres recherches sur l'énergie ont porté sur des techniques d'estimation de l'énergie consommée durant le processus de conception d'un ASIP [17, 54, 135]. En effet, comme l'ont mentionné Beucher et al. [17], la puissance dissipée augmente mais l'énergie consommée pourrait diminuer durant le processus d'accélération par un ASIP. Donc, il serait utile de l'estimer et de prendre les décisions appropriées pendant le processus de conception. Fei et al. [54] ont proposé une technique d'estimation d'énergie des processeurs spécialisés basée sur les opérations à exécuter, mais aussi sur les caractéristiques de l'architecture du processeur, telles que le pipeline ou la cache. Leur estimation prend en compte des traces d'une simulation et celles-ci sont utilisées pour

définir les unités du processeur qui entrent en jeu dans le calcul. D'un autre côté, Pitkänen et al. [135] ont proposé une technique qui tient compte de l'énergie statique et de l'énergie dynamique de chaque opération exécutée sur un processeur spécialisé. Leur estimation est effectuée selon le chemin critique de l'application considérée, donc ils auront tendance à surestimer l'énergie. La technique de Beucher et al. [17] repose sur la simulation du processeur sur une plateforme d'émulation déployée dans un FPGA. Durant la simulation, un rapport d'activité des commutations est collecté et utilisé pour estimer l'énergie. Malheureusement, ces derniers ne tiennent pas compte de certains facteurs tels que l'énergie latente, à savoir l'énergie consommée lorsqu'une porte ne commute pas, par exemple.

Une tendance importante qui est digne d'être mentionnée est celle qui porte sur la conception des processeurs spécialisés reconfigurables (rASIP). Un rASIP comporte un processeur de base qui est connecté à des composants reconfigurables (un FPGA, par exemple) à travers des interfaces connues. Les composants reconfigurables exécuteront des instructions spécialisées ou une fonctionnalité précise à accélérer. Comme l'ont mentionné Karuri et al. [85], la conception de rASIP fait face à de nombreux défis dont le grand nombre de designs possibles tant au niveau du processeur de base que des composants reconfigurables. En effet, il faudra définir l'architecture de base du processeur et déterminer les instructions ou les blocs d'instructions qui seront exécutés dans la partie reconfigurable. La recherche sur les rASIP a d'abord porté sur des approches de design d'architecture [69, 177]. D'un autre côté, certains chercheurs se sont focalisés sur des méthodologies de conception basées sur des outils de génération automatique de l'architecture. Par exemple, on peut citer la plateforme Dresc ("Dynamically reconfigurable embedded system compiler") qui est utilisée pour générer l'architecture Adres ("Architecture for dynamically reconfigurable embedded systems") [119]. L'architecture Adres comporte un processeur de base de type VLIW et des blocs fonctionnels répartis dans une matrice d'éléments de calcul CGRA (« Coarse Grain Reconfigurable Array ») qui sont exécutés dans la partie reconfigurable.

Malheureusement, dans cet outil, le processeur joue plutôt le rôle d'un contrôleur que d'un élément de calcul. Récemment, Karuri et al. [85] ont présenté une méthodologie de conception de processeur reconfigurable basée sur un langage de description architecturale, le LISA 3.0. En fait, ils ont étendu le LISA 2.0 afin de pouvoir spécifier la nature des instructions, à savoir si ces dernières seront exécutées dans le processeur ou dans la partie reconfigurable, par exemple. Ils proposent aussi de spécifier des CGRA. Ces CGRA seraient en fait des éléments de calculs (PE) interconnectés entre eux, mais qui pourraient aussi communiquer avec le processeur de base pour exécuter la fonctionnalité voulue. Il faut noter que les travaux de [85, 119] reposent sur le profilage de l'application pour le choix des instructions à accélérer par le matériel reconfigurable. Nous devons préciser que, lorsque les parties reconfigurables sont programmées dès l'implémentation de l'architecture dans le FPGA, on parle alors de processeur reconfigurable statique (srASIP). Lorsque les parties reconfigurables sont programmées dans le FPGA au fur et à mesure de l'exécution, alors on parle alors de processeur reconfigurable dynamique (drASIP). Bien entendu, les drASIP permettent potentiellement d'économiser de l'espace dans le FPGA lorsque l'application s'y prête, mais ils ont un coût associé à la re-programmation du FPGA. Mentionnons les travaux de Étienne Bergeron [16]; ce dernier a proposé une suite d'outils qui effectue la compilation, le placement et routage à la volée. M. Bergeron ne classe pas ses travaux dans la famille des rASIP, mais nous pensons que ces travaux s'appliquent à ce domaine de recherche. En effet, il propose un système nommé Dynamit; système dans lequel une application est exécutée en logiciel, mais certains de ses segments seront exécutés dans des blocs logiques du FPGA au lieu de les exécuter dans un processeur. La compilation, le placement et le routage des blocs logiques exécutant le segment à accélérer seront effectués à la volée.

2.2 Processus de synthèse à haut niveau

La synthèse à haut niveau est un processus de génération automatique d'un circuit matériel à partir d'une description comportementale [65]. Même si l'utilisation d'outils

de synthèse à haut niveau est de plus en plus populaire, la recherche dans ce domaine a débuté depuis plus d'une vingtaine d'années [57, 176]. La maturité des outils permet maintenant de générer des modules matériels ayant des performances très proches des modules matériels conçus par un humain expérimenté. L'évolution des outils de synthèse à haut niveau est due à l'efficacité des algorithmes d'ordonnancement qui ont été proposés par les chercheurs œuvrant dans ce domaine. La Figure 3 représente un flot simplifié du processus de synthèse à haut niveau. Le premier artéfact entrant dans le processus est la spécification décrite dans un langage à haut niveau. Actuellement, la plupart des outils considèrent une spécifcation décrite en langage C pur, mais des langages tels que SystemC, HandelC [27], TransmogrifierC [59], HardwareC [89] et d'autres, sont proposés pour spécifier des modules matériels. En effet, le langage C ne permet pas de transcrire certaines propriétés d'un MMD. Par exemple, les concurrences simples entre différents groupes d'opérations ne peuvent pas se faire à moins d'utiliser la librairie *thread* et cela alourdirait la spécification.

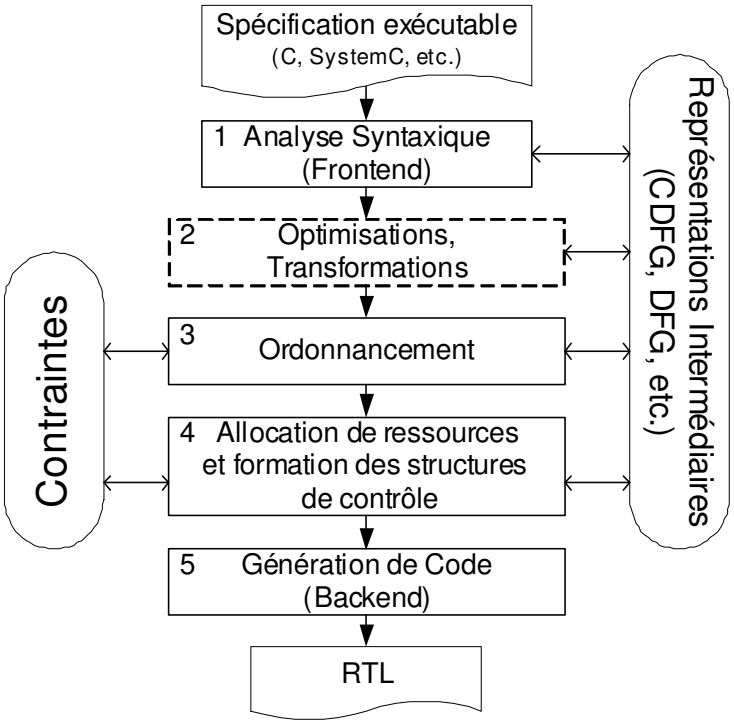


Figure 3. Cadre de travail simplifié des outils de synthèse à haut niveau.

La première tâche d'un outil de synthèse à haut-niveau est l'analyse syntaxique de la description comportementale. La spécification logicielle est manipulée par un analyseur syntaxique qui la transforme selon une ou des représentations intermédiaires telles qu'un diagramme de flot de données (DFG) ou de contrôle (CFG), ou un diagramme hiérarchique de tâches-HTG [66]. Le choix de la représentation intermédiaire dépend bien sûr des concepteurs de l'outil. Les représentations intermédiaires peuvent évoluer durant le processus de compilation.

Après avoir généré une première version de la représentation intermédiaire, cette dernière est manipulée en y effectuant des passes d'optimisation et de transformation. Ces passes vont généralement tenter de faire ressortir les possibilités de parallélisme offertes dans la spécification. Pendant très longtemps, il s'agissait d'effectuer de la construction de pipeline ou du déroulement de boucle. Cependant, le déroulement de boucles peut être très coûteux dans certains cas [66] [92]. Les algorithmes de construction de pipeline de boucles se basent généralement sur les algorithmes de construction de pipeline logiciel. Les algorithmes de construction de pipeline de boucle comme celui proposé par Cardoso et al. [24] sont très efficaces pour des boucles ayant un flot régulier de données. Cependant, lorsqu'il s'agit de boucles avec de nombreux énoncés complexes de contrôle, l'efficacité de ces algorithmes laisse à désirer. Les limites de ces techniques ont poussé les chercheurs à appliquer d'autres transformations telles que le report de mailles de boucle (« loop shifting ») ou la fusion de boucles, par exemple. Ces techniques permettent entre autres de réduire les besoins en mémoire de l'application. Le report de mailles de boucle consiste à déplacer des opérations d'une boucle entre différentes instances d'itérations de celle-ci, afin de modifier l'ordre des dépendances de données, ce qui a pour avantage d'apporter de nouvelles possibilités d'ordonnancement et de réduire le chemin critique de l'application [43]. Gupta et al. [66] ont ajouté le report de mailles de boucle comme une des passes d'optimisation dans leur outil SPARK. Ils ont obtenus des gains de performance intéressants comparés à ceux obtenus en appliquant du déroulement de boucle. L'optimisation de boucle par différentes techniques est un moyen

efficace d'exposer le parallélisme. Cependant, d'autres techniques qui s'inspirent des techniques de compilation de code logiciel peuvent être appliquées telles que l'élimination de code mort, l'élimination d'expressions redondantes, etc.

L'étape la plus importante du processus de synthèse à haut niveau est celle d'ordonnancement des opérations. Cette étape consiste à définir le nombre de cycles et leur durée, et à assigner les opérations qui seront exécutées à chaque étape de calcul [122]. Pendant très longtemps, les techniques d'ordonnancement les plus populaires visaient à minimiser le temps d'exécution du circuit généré. Les approches ASAP (ordonnancer aussi tôt que possible) et ALAP (ordonnancer aussi tard que possible) [58] ont été très utilisées. Malheuseusement, ces deux approches peuvent induire un circuit dont les opérations ne sont pas équitablement distribuées. Une technique qui combine ces deux approches est l'ordornancement forcé [134], qui vise à atteindre des contraintes de latence définies, tout en équilibrant la distribution des opérations, ce qui permet d'optimiser l'utilisation des ressources, tout en minimisant le temps d'exécution. L'algorithme de Paulin [134] se classe parmi les algorithmes d'ordonnancement basés sur la considération de contraintes de temps fixes, comme l'ont mentionné Gupta et Brewer [65]. L'inconvénient des algorithmes d'ordonnancement par contraintes de temps fixes est que ces derniers ne tiennent pas compte des contraintes de ressources. En effet, le but de ces algorithmes est de trouver le design ayant le plus petit temps d'exécution. D'un autre côté, la classe des algorithmes orientés ressources se focalise sur la génération d'un design qui respectera des contraintes de ressources dures et dont le temps d'exécution sera minimal, donc les contraintes dures de temps ne sont pas considérées durant l'ordonnancement. Avec le temps, de nouveaux algorithmes ont pu tenir compte des deux types de contraintes. Par exemple, une technique d'ordonnancement qui a fait son chemin est l'ordonnancement conduit en fonction du temps d'exécution (« time-driven scheduling ») [22, 90]. Cette technique basée sur l'estimation du temps d'exécution ordonnance les opérations afin d'atteindre le temps d'exécution visé. Une technique qui a aussi beaucoup de succès est l'ordonancement par programmation linéaire sur nombre

entiers (ILP – Integer Linear Programming). Cette technique permet de tenir compte de plusieurs types de contraintes et l'ordonanncement est effectué en résolvant un système d'équations linéaires [95, 185].

Un problème récurrent durant l'ordonnancement concerne les branchements qui sont dans le code. En effet, les branchements constituent généralement des barrières de traitement qui freinent l'exécution du flot de données. Ainsi, plusieurs algorithmes ont visé les applications orientées contrôle [105, 182]. Par exemple, Huang et al. [73] ont proposé un algorithme d'ordonnancement par arbre, afin d'ordonnancer chaque chemin de contrôle, ce qui contribuerait à appliquer du mouvement de code. En fait, des chercheurs ont concentré leur effort sur le mouvement de code par spéculation [146, 153, 155]. Plus récemment, Gupta et al. [66] ont utilisé de techniques de mouvement de code par spéculation durant l'étape d'optimisation et de transformation de leur outil SPARK [66]. Le mouvement de code (« code motion ») par spéculation consiste à déplacer un segment de code appartenant à un bloc de branchement hors de ce dernier. L'ordonnancement des opérations est généralement contraint pour atteindre des objectifs de performance, de coûts, etc. Dans de nombreux cas, la ILP est utilisée pour trouver le meilleur ordonnancement possible ou pour évaluer les bornes minimales ou maximales associées à un ordonnancement donné [31, 128].

L'étape suivante du processus de synthèse à haut niveau est l'allocation des ressources. Durant la phase d'ordonnancement, l'ordre d'exécution des opérations a été construit selon les contraintes fournies. Puis, durant l'étape d'allocation des ressources, le nombre et le type de ressources des unités fonctionnelles (FU) seront déterminés. Ensuite, durant l'allocation des ressources, à chaque opération, il faut associer une unité fonctionnelle. En effet, la réutilisation de FU est très utile pour atteindre les contraintes d'espace. De nombreux travaux ont portés sur l'optimisation du nombre de ressources utilisées. Par exemple, Potkonjak et al. [139] ont proposé d'appliquer des transformations telles que la resynchronisation (*retiming*) pour optimiser l'utilisation des ressources.

D'autres, comme Mondal et al. [123], ont proposé une technique de partage de ressources pour des architectures pipelinées, par exemple.

Étant donné que plusieurs opérations pourraient être exécutées par la même unité fonctionnelle, une des étapes du processus consiste à générer automatiquement l'unité de contrôle du circuit. Le contrôleur généré aura pour tâche d'aiguiller les bonnes données aux bons endroits. En effet, comme illustré à la Figure 4, on voit que le contrôleur active les différents multiplexeurs qui fournissent les différentes entrées des FUs à savoir l'additionneur, le multiplieur et le soustracteur. La complexité de ce contrôleur est fortement liée aux résultats de l'ordonnancement mais aussi aux optimisations effectuées. Le déroulement de boucle a un impact sur la complexité du contrôleur comme Kurra et al. [92] l'ont décrit. Ainsi, l'application de certaines techniques d'optimisation doit se faire prudemment; c'est ce qui a poussé Kurra et al. [92] à proposer une technique d'estimation du délai d'un contrôleur en fonction d'un facteur de déroulement donné.

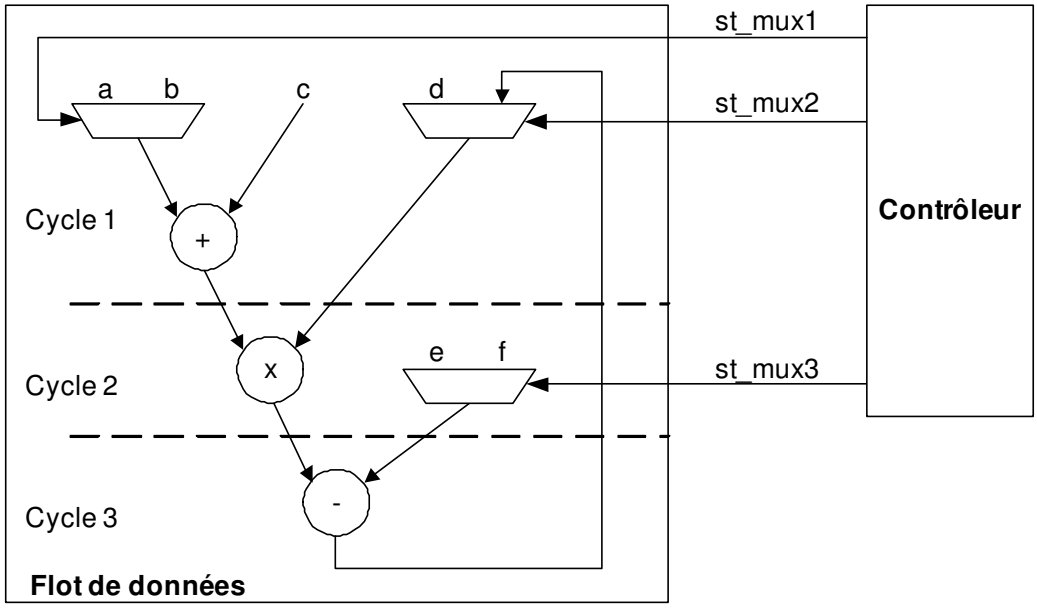


Figure 4. Exemple d'architecture matérielle générée par un HLS

Durant l'étape d'allocation de ressources, les différents éléments de mémoire sont définis et il faudra aussi générer, s'il y a lieu, des unités de gestion de mémoire, par exemple. En effet, le nombre et le type des éléments mémoire générés ont un rôle

important sur les performances et les coûts du circuit généré [9, 156]. Des chercheurs ont beaucoup étudié l'optimisation des unités de mémoire durant la synthèse. Certains ont proposé d'optimiser la taille des mémoires, par exemple [156]. D'autres ont étudié la hiérarchie des mémoires durant le processus de génération [28].

Enfin, la dernière étape du processus de synthèse à haut niveau est l'étape de génération de la description du circuit dans un langage de description matérielle RTL comme VHDL, Verilog ou autre.

Les premiers travaux sur la synthèse à haut niveau visaient des applications de type flot de données. Une technique très efficace qui a été appliquée à la synthèse à haut niveau visait la génération d'architecture pipelinée de flot de données [109, 133]. Arato et al. [5] ont énuméré les différents enjeux liés à la génération automatique d'architectures pipelinées.

Bien sûr, de nombreux travaux visent à améliorer une ou des parties de ces étapes. Par exemple, Molina et al. [122] ont proposé une méthode pour réduire l'espace induit par les unités fonctionnelles multi-cycles. En effet, certaines unités fonctionnelles, comme par exemple les multiplieurs, requièrent plusieurs cycles d'horloge d'exécution, ce qui a un impact sur la complexité du contrôleur. Raghunathan et al. [147] ont proposé d'utiliser des unités SIMD (Single Instruction Multiple Data) en lieu et place des unités fonctionnelles atomiques telles que les additionneurs. Ces unités SIMD exécutent simultanément plusieurs opérations. La technique qu'ils ont proposée a contribué à obtenir des gains de performance intéressants tout en réduisant l'énergie consommée.

D'autres travaux dignes d'être mentionnés sont ceux de Derrien et al. [152], qui portent sur la génération d'une description matérielle à partir d'un système d'équations affines récurrentes SARE (qui seront décrites dans la section 2.3). Leurs travaux ont été intégrés dans l'outil MMAlpha [144], qui permet de faire des transformations sur des nids de boucles et de générer entre autres des architectures systoliques. En terminant, nous

soulignons aussi la publication en 2008 d'un livre [41] qui décrit l'état de la recherche sur les HLS et les défis que les chercheurs devront surmonter comme, par exemple, la génération de circuit ayant plusieurs horloges ou la génération d'architecture orientée composantes.

Pour finir, les travaux de So et al. [159] devraient être mentionnés. Ces derniers ont défini une métrique de balance utilisée pour la conception de systèmes matériels qui ciblent la technologie FPGA. Cette notion de balance a été introduite par Carr et al. dans [25]. Cette métrique permet de déterminer le poids des opérations de chargement de données et des opérations arithmétiques et logiques sur les performances d'un design. La technique de conception de So et al. [159] basée sur la métrique de balance a pour but de concevoir un design purement matériel dont le poids des types d'opérations sera équilibré. Malheureusement, ils ne proposent pas d'autres métriques pour mieux cibler les aspects qui limitent ou qui contribueraient à atteindre cet équilibre. Sans compter le fait que contrairement à nos contributions, ils ciblent uniquement un design matériel de type FPGA. Il faut rappeler qu'une des contributions de cette thèse est la proposition d'un cadre de travail dans lequel plusieurs types de design peuvent être explorés et que ce cadre de travail cible l'accélération de boucles.

2.3 Modèle polyèdral et dépendances de données

Le modèle polyèdral a été très utilisé dans le domaine logiciel comme dans le domaine matériel. C'est un modèle qui permet de représenter le domaine de valeurs des indices d'itération d'un nid de boucles. À partir de ce domaine, les dépendances de données peuvent être déterminées plus précisément. Dans la présente section, le modèle polyèdral est décrit. Ensuite, les différents types de dépendances sont présentés. Puis, des représentations de dépendance de données telles que le graphe de dépendances réduites (RDG) et le système d'équations affines récurrentes que nous avons utilisé dans l'article inclus au chapitre 6 seront présentés. Enfin, le problème d'ordonnancement des dépendances en fonction des indices d'itération d'un nid de boucles sera abordé.

2.3.1 Définitions

Débutons par quelques définitions utiles :

Hyperplan : Soit un vecteur (1xN) non nul λ, *x* un vecteur de coordonnées de (Nx1) et une constante scalaire α, alors :

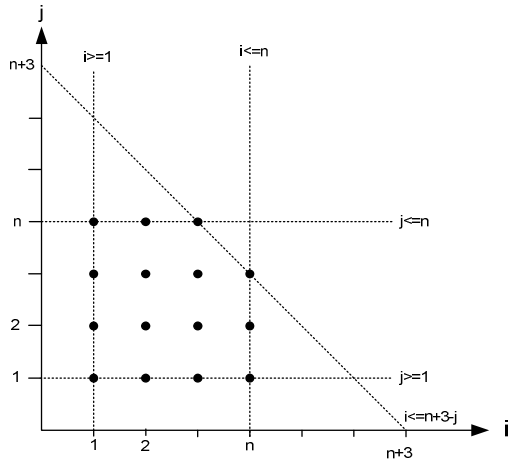
- Hyperplan : $\{x \mid \lambda x = \alpha\}$
- Demi-espace ouvert (« Open half space ») : $\{x \mid \lambda x > \alpha\}$
- Demi-espace fermé (« Closed half space ») : $\{x \mid \lambda x \geq \alpha\}$

Polyèdre : Un polyèdre est une intersection d'un nombre fini de demi-espaces fermés $\{x \mid \lambda x \geq \alpha\}$. Un polyèdre est formulé comme $P = \{x \mid Ax \geq b\}$, où ***A*** est une matrice de constantes et ***b*** est un vecteur de constantes, comme décrit par [104].

Polyèdre paramétré : C'est une intersection d'un nombre fini de demi-espaces fermés $\{x \mid \lambda x + \beta s \geq \alpha\}$, où ***β*** est un vecteur de constantes et ***s*** un vecteur de paramètres. Le polyèdre paramétré est représenté sous la forme $P = \{x \mid Ax + Bs \geq b\}$ où ***A*** et ***B*** sont des matrices de constantes, ***s*** est le vecteur de paramètres et ***b*** est vecteur de constantes dans lequel l'espace est limité, comme décrit par [110].

Par exemple, la Figure 5 illustre le modèle polyèdral d'un nid de boucles qui exécute l'énoncé *S1*. La Figure 5 présente la structure du nid en sachant que *S1* n'est exécuté que lorsque l'indice d'itération *i* est inférieur ou égal à l'expression (*n+2-j*).

```
for (i=1; i <= n; i++)
{
   for (i=1; i <= n; i++)
   {
      if (i <= n + 3 –j)
        S1
   }
}
```



(a) (b)

Figure 5. Exemple du modèle polyèdral d'un nid de boucles, a) structure du nid, b) représentation graphique du domaine d'itérations de S1

Le modèle correspondant est un polyèdre paramétré en sachant que *n* est le paramètre du modèle. Le polyèdre de *S1* $P = \{x \mid Ax + Bs \geq b\}$ est tel que :

$$Ax = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ - & -1 \\ -1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix}, \qquad Bs = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} (n), \qquad b = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ -3 \end{bmatrix}$$

Donc, P est représenté comme suit :

$$P = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ - & -1 \\ -1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} (n) \geq \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ -3 \end{bmatrix}$$

2.3.2 Dépendances de données

Banerjee, dans [11], a présenté les enjeux reliés à l'analyse de dépendance de données. Dans ce livre, il définit qu'une dépendance entre deux énoncés ***T*** et ***S*** d'un code se décrit comme suit : on dit que ***T*** dépend de ***S***, ***S < T***, si l'exécution de l'énoncé ***T*** requiert la valeur générée par l'énoncé ***S***. On dira que ***T*** dépend lexicographiquement de ***S*** si l'exécution de ***S*** doit toujours précéder celle de ***T***, dans l'espace de valeurs possibles des indices d'itération. En effet, dans certains cas, la dépendance pourrait ne pas être vraie selon l'espace des indices d'itération.

Banerjee [11] décrit quatre types de dépendances qui sont :

- Dépendance de flot : si l'énoncé ***S(i)*** écrit dans ***M*** et que ***T(j)*** lit la valeur de ***M***, ***M*** étant une case mémoire ou une variable temporaire.
- Anti-dépendance : si ***S(i)*** lit ***M*** et que ***T(j)*** écrit dans ***M***.
- Dépendance de sortie : si ***S(i)*** et ***T(j)*** écrivent tous les deux dans ***M***.
- Dépendance d'entrée : si ***S(i)*** et ***T(j)*** lisent dans M.

Une dépendance peut être transportée par la boucle (« loop-carried ») ou elle peut être indépendante de la boucle (« loop-independent »). Une dépendance transportée par la boucle est telle que, pour toute paire (***S(i)***, ***T(j)***), ***T(j)*** dépend de ***S(i)*** quel que soit l'espace des valeurs d'indices d'itération des indices ***i*** et ***j***. Cette dépendance stipule que l'exécution de ***T*** à l'itération ***j*** dépend de l'exécution de ***S*** à l'itération ***i***. Une dépendance indépendante de la boucle est une dépendance qui existe seulement dans une itération précise à savoir que ***T(i)*** dépend de ***S(i)*** *(i.e. j=i)*.

Plusieurs types de graphe ont été proposés pour représenter les dépendances d'un nid de boucles comme l'ont montré Darte et al. [45]. Un graphe très utilisé est le graphe étendu de dépendances (EDG), aussi appelé le graphe de dépendances au niveau instruction. Un EDG, ***G***, est un graphe tel que :

- Pour toute variable V_i, il y a un noeud v_i correspondant dans ***G***.

- Pour chaque variable $V_i(p)$ qui dépend de $V_k(p-d_k,i)$, il y aura une transition entre les noeuds v_i et v_k et la transition aura un poids $d_{k,i}$, en sachant que $d_{k,i}$ est le vecteur de distance de la dépendance ente les variables V_i et V_k (vecteur de distance : différence entre les fonctions d'accès initiales de V_i et celle de V_k).

Le EDG a d'abord été introduit pour représenter des systèmes d'équations uniformes récurrentes (SURE) proposés par Karp et al. [83]. Les SURE ont été proposés pour mieux comprendre la structure des calculs dans un contexte répétitif et régulier en vue d'une parallélisation. Étant donné que les SURE ne permettaient pas de définir des systèmes répétitifs et irréguliers, un autre type de système a été proposé. Il s'agit des systèmes d'équations affines récurrentes (SARE). Les SARE peuvent être générés à partir d'un EDG. Les SARE ont été proposés en premier pour la conception d'architectures systoliques [145, 148, 189]. Dans le chapitre 6, une des étapes du processus d'estimation du temps d'exécution d'un nid de boucles consiste à générer le SARE correspondant. Une équation d'un SARE se formule comme suit :

$$(\forall x \in Is : v[f(x)] = F_v(w[g[x],..)) \quad \textbf{(1)}$$

Une équation d'un SARE définit les dépendances d'une variable ou d'un énoncé d'un programme selon un domaine d'itérations Is. Ainsi, l'équation 1 stipule que pour x compris dans le polyèdre Is, la valeur de v à l'itération $f(x)$ dépend de la fonction Fv qui prend comme argument la valeur de w à l'itération $g(x)$ et ainsi de suite. $f(x)$ et $g(x)$ sont les fonctions d'accès des variables v et w.

Par exemple, dans le nid de boucles illustré par la Figure 6.a, le Tableau *A* est mis à jour dans deux branches parallèles S1 et S2. Ces branches dépendent de la valeur des indices d'itération *i* et *j*. Le SARE décrivant les dépendances de *A* comprend deux équations *ARE1* et *ARE2* données à la Figure 6.b.

```
do i = 1,n
do j = 1,n
 if (i<n-2-j)
  S1 : A[i,j] = B[i+1,j] + C[i,j]
 else
     S2 : A[i][j] = B[i][j]
```

(a)

ARE1 : {i>=1, -i+n>=0, j>=1, -j-n>=0, i<-j-n-2} :
A[i,j] = Fv(B[i+1,j], C[i,j])

ARE2 : {i>=1, -i+n>=0, j>=1, -j-n>=0, i>=-j-n-2} :
A[i,j] = Fv(B[i,j])

(b)

Figure 6. Exemple de SARE. a) Exemple de nid de boucles. b) ARE de S1 et S2.

Comme on peut le remarquer dans la première équation *ARE1*, la variable *A* dépend des variables *B* et *C* dont les fonctions d'accès sont égales à [i+1,j] et [i,j], respectivement.

Le EDG facilite la représentation des dépendances entre différents énoncés de code ou différentes variables d'un programme. Ensuite, le problème qui se pose est de trouver un ordonnancement qui définit le temps d'exécution des énoncés en fonction des indices d'itération, par exemple. L'ordonnancement permettra aussi de déterminer le degré de parallélisme intrinsèque des énoncés d'un nid de boucles [53]. Plusieurs algorithmes d'ordonnancement [44, 52, 53, 83, 148, 157] ont été proposés pour trouver les fonctions de temps associées à chaque énoncé. Le but de ces algorithmes est de résoudre le problème formulé par l'équation 2.

$$x \in D_R,\ y \in D_S, <x, y> \in R_e \Rightarrow \theta(S, y) \geq \theta(R, x) + 1 \qquad \textbf{(2)}$$

L'équation 2 stipule que le temps d'exécution $\theta(S, y)$ de l'énoncé *S* à l'itération *y* doit être supérieur au temps d'exécution de $\theta(R, x)$ de l'énoncé de *R* à l'itération *x. Cela signifie que S,y ne pourra s'exécuter qu'après l'exécution de R,x plus 1,* sachant que 1 est la durée d'exécution de l'énoncé *R*. Comme nous l'avons dit précédemment, les SARE ont été proposés pour la parallélisation dans des systèmes multiprocesseurs ou des architectures systoliques dans lesquels on considère qu'une opération s'exécutera en un cycle d'horloge. C'est la raison pour laquelle l'équation 2 considère que la durée d'exécution de ***R*** sera égale à 1. Cette durée peut évidemment être modifiée en fonction

des délais de l'architecture ciblée. Donc, le but est d'exprimer le temps associé à chaque énoncé du programme en fonction des indices d'itération, en sachant que les contraintes de précédance devront être respectées pour tout domaine de valeurs des indices.

Parmi les algorithmes d'ordonnancement, le plus populaire et le plus utilisé est celui conçu par Feautrier. Ce dernier a d'abord proposé un algorithme qui cible des systèmes uni- et bi- dimensionnels [53]. Ensuite, il a généralisé l'algorithme pour cibler des systèmes multidimensionels [52]. Le problème d'ordonnancement est finalement réduit à la résolution d'un système d'équations linéaires. Les algorithmes d'ordonnancement qui étaient proposés avant celui de Feautrier nécessitaient la transformation du système d'équations affines en un système d'équations uniformes afin de trouver les fonctions de temps. Malheureusement, dans certains cas, cette transformation n'est pas faisable. L'avantage de l'algorithme de Feautrier est qu'il ne nécessite pas l'uniformisation du SARE. Cependant, il faut noter que les architectures systoliques requièrent des systèmes d'équations uniformes, raison pour laquelle, dans certains cas, il est utile d'effectuer l'opération inverse à savoir la transformation d'un système irrégulier à un système régulier [108]. Dans l'annexe A, nous détaillons l'algorithme d'ordonnancement de Feautrier. Cet algorithme a été implémenté dans un outil nommé PIP [14] qui est très utilisé par les chercheurs utilisant le modèle polyèdral.

2.4 Applications du modèle polyèdral

Le modèle polyèdral a été appliqué à plusieurs techniques d'optimisation de boucles. Dans la suite de cette section, nous énumérons différents travaux qui ont attiré notre attention dans l'abondante litérature qui se rapporte au modèle polyèdral. Nous les avons triés selon plusieurs catégories, telles que les techniques de transformation, d'optimisation de localité et enfin de conception de systèmes systoliques et d'architectures multiprocesseurs.

2.4.1 Techniques de transformation

Il y a presque une dizaine d'années, Darte et al. [43] ont proposé une technique de report de mailles de boucle basée sur le modèle polyèdral. Ils proposent la resynchronisation des instructions inspirée du concept de Move-and-Schedule de la famille d'algorithmes de pipelinage « modulo scheduling » [151] [93].

Le modèle polyèdral a été très utilisé pour l'accélération de code logiciel. De nombreuses techniques, telles que la fusion de boucles, le pavage (« tiling »), le pipelinage de boucles et autres, ont été explorées à l'aide du modèle polyèdral. Lim et al. [101] ont présenté un algorithme qui trouve des partitionnements possibles afin de maximiser le degré de parallélisme et de diminuer la synchronisation lors de l'optimisation d'un nid de boucles, en sachant que leur technique cible la transformation des boucles extérieures parallèles. Girbal et al. [62] ont proposé un cadre d'applications ("*framework*") pour l'application semi-automatique de techniques de transformation de boucles. Les notions présentées ont abouti à la conception d'un outil nommé GRAPHITE [137] qui analyse la possibilité d'appliquer certaines techniques de transformation de boucles et les applique, si possible. L'outil transforme les boucles selon les résultats de l'analyse et génére automatiquement le code correspondant. Il faut préciser que GRAPHITE est un outil intégré au compilateur GCC, ce qui est une première concernant l'utilisation du modèle polyèdral dans un compilateur logiciel très répandu.

Le processus d'optimisation d'un nid de boucles inclut une étape de génération de code. Ainsi, des chercheurs ont proposé des techniques de génération automatique de code. Par exemple, Kelly et al. [86] font partie des premiers à avoir proposé un générateur automatique de code, OMEGA [141]. Ce générateur a été proposé pour la transformation de code en vue d'optimiser l'ordonnancement des opérations d'un nid de boucles dans une architecture monoprocesseur. Quilleré et al. [143] ont exposé un algorithme de génération de code qui limite l'insertion de points de synchronisation dans le code généré. Leur algorithme a été implémenté dans le générateur LoopGen. Certaines

transformations peuvent induire une explosion de la taille du code. Bastoul et al. [13] ont défini un algorithme de génération de code qui permet d'éviter cette explosion. Cet algorithme a été implémenté dans un générateur de code nommé CLoopG [12] qui est très connu par la communauté d'utilisateurs du modèle polyèdral.

2.4.2 Optimisation de la localité des données

Une technique connue dans le domaine logiciel est la réutilisation de données. Le modèle polyèdral a été utilisé pour détecter les possibilités de réutilisation de données. Parmi les premiers à l'avoir appliqué, Wilde et al. [184] peuvent être cités. Ces derniers ont proposé une technique basée sur l'utilisation des fonctions affines de temps ainsi que la génération d'une fonction d'allocation des cases de la mémoire. Avant eux, Feautrier et al. [51] avaient proposé une technique d'analyse des données qui pourrait être utilisée pour la vérification d'un programme ou pour la parallélisation de ce dernier, par exemple.

Le modèle polyhedral a été largement utilisé pour l'estimation de l'espace mémoire par des chercheurs de l'IMEC [76]. Par exemple, ils ont proposé une technique d'estimation des dépendances de données en vue de l'optimisation de la taille de la mémoire [87]. Ils ont aussi proposé une technique d'estimation de la taille de mémoires hiérachiques en vue d'appliquer des transformations de boucle telles que la fusion et le report de mailles pour des applications manipulant un nombre important de données [72].

Fabroulet et al. [55] ont proposé une technique qui vise à diminuer le nombre d'accès mémoire en améliorant la localité des données. Leur technique est basée sur une transformation de l'alignement des données en mémoire qui, entre autres, effectue du retiming afin de minimiser la distance en mémoire des données utilisées.

Signalons finalement les travaux de Thies et al. [170] qui portent sur une plateforme mathématique pour l'analyse combinée des possibilités de parallélisme et d'allocation des ressources mémoires. Leur compilateur détermine la quantité appropriée de ressources en fonction de contraintes de temps spécifiées.

2.4.3 Conception d'architecture systoliques et de multiprocesseurs

Comme mentionné plus tôt, les SARE ont été proposés en premier lieu pour la conception d'architectures systoliques. Par exemple, Yacoby et al. [189] se sont intéressés aux architectures systoliques. Leur technique se base sur la détection de la faisabilité du calcul de la fonction affine de temps, afin d'assurer qu'une fonction de temps du système est calculable et donc qu'un ordre d'exécution existe. Ils établissent des conditions nécessaires à l'existence d'une fonction affine de temps. La technique de Quinton et al. [145] est basée sur le calcul de la fonction affine de temps du système. Si cette dernière existe, alors eux aussi proposent une technique de transformation du SARE en un système uniforme afin d'en établir une correspondance dans une architecture régulière. La génération d'une architecture systolique requiert une étape de pipelinage étant donné que, dans ce type d'architecture, les éléments de calcul PE (qui étaient des processeurs dans les premières architectures) ne peuvent communiquer qu'avec leurs voisins directs. Notons aussi l'existence du cadre d'applications MMAlpha [144] qui comporte le compilateur de code Alpha. Le langage Alpha est un langage de description de SARE. Dans le cadre d'applications MMAlpha, un code donné est ordonnancé pour calculer les fonctions affines de temps de toutes les variables du système, en vue de la génération d'une architecture systolique, entre autres. Le code VHDL de l'architecture systolique associée peut aussi être généré, par exemple.

Plus récemment, le modèle polyèdral a été appliqué pour la conception d'architectures multiprocesseurs. L'application de techniques de transformation de boucles (telles que le pavage et la fusion), pour profiter de la puissance de calcul d'architectures multiprocesseurs, a fait ses preuves. Le pavage de boucles est une technique éprouvée d'optimisation de la localité des données et de la parallélisation. Le pavage consiste à regrouper des données d'un espace d'itérations dans un petit bloc nommé « tuile ». Les données de ce nouveau bloc pourront ensuite être manipulées indépendamment et simultanément par un élément de calcul de l'architecture tout en favorisant la réutilisation de données. Pour la parallélisation et l'optimisation de la

localité des données, Uday et al. [173] présentent différentes transformations basées sur le pavage. Les transformations sont implémentées dans un compilateur PLuTo [174] qui génére un code parallèle OpenMP. Le code résultant peut ensuite être simulé sur une architecture multiprocesseurs.

Bouchebaba et al. [20] combinent le pavage et la fusion de boucles pour l'optimisation de la mémoire dans une architecture multiprocesseurs. Ils utilisent des tampons qui sont substitués aux tableaux temporaires qui étaient stockés dans une mémoire principale, ce qui permet de diminuer la taille de cette dernière.

2.5 Techniques d'estimation de performance

L'estimation du temps d'exécution d'un programme a toujours été une étape importante durant le processus d'optimisation et de conception d'un système logiciel ou matériel. Généralement, les performances sont calculées soit en effectuant une analyse statique du programme donné, soit en effectuant une simulation, soit en exécutant le code dans son environnement réel. Les deux dernières approches nécessitent une modélisation de l'architecture visée. L'approche par simulation est beaucoup plus précise que l'approche statique, mais elle nécessite généralement un temps beaucoup plus long pour obtenir les performances estimées. L'approche par analyse statique est moins précise, mais elle permet de calculer les performances en très peu de temps. C'est une approche qui a fait ses preuves et qui nécessite très peu d'effort et de ressources par rapport à l'approche par simulation. Les techniques d'analyse statique généralement visent à estimer le pire temps d'exécution (WCET) d'un code donné [107]. Les techniques d'estimation du WCET se subdivisent en deux branches. Certains chercheurs ont focalisé leurs efforts sur la détection des chemins parcourus (ce qui induit aussi la détection de chemins impossibles à exécuter) [97, 131, 169]. D'autres se sont plutôt concentrés sur la considération des contraintes architecturales telles que le pipeline ou la cache, par exemple [49, 78, 100, 190]. Il faut noter les travaux de Puschner [142] qui a proposé une technique de programmation orientée WCET qui aide à rencontrer les contraintes de

temps d'un système en temps réel en analysant les pires estimations durant le processus de programmation. Nos travaux sur l'estimation du temps d'exécution d'un nid de boucle s'inscrivent dans la lignée des estimateurs de WCET puisque nous estimons le temps d'exécution maximal d'un nid de boucles.

Nakanishi et al. [125] ont présenté une technique d'estimation du temps d'exécution d'une boucle basée sur une analyse statique. Ces derniers proposent une technique qui formule mathématiquement les dépendances de données de la boucle entièrement déroulée dans un graphe qu'ils nomment le graphe de tâche d'une boucle déroulée. Malheureusement, ces derniers n'effectuent aucune détection de chemin exécutable et, d'une façon plus importante, ils n'analysent pas le domaine d'itération des indices de la boucle ce qui fait qu'ils pourraient considérer des chemins ou des espaces d'itération qui ne devraient pas l'être. Il faut préciser que leur technique vise des boucles unidimensionnelles.

Récemment, Roychoudhury et al. [154] ont proposé une technique d'estimation du temps d'exécution d'une boucle basée sur la détection des chemins exécutables associés à chaque itération de la boucle. Leur solution est assez intéressante, puisqu'elle devrait permettre de ne pas considérer les chemins qui ne seront pas exécutés. Malheureusement, ils ne précisent pas comment ils gèrent le domaine d'itération des boucles ni si leur approche s'applique à un nid de boucles, puisque les résultats présentés concernent des boucles unidimensionnelles. Il faut aussi noter que leur approche ne tient pas compte des contraintes architecturales du système visé.

Clauss et al. [37] ont effectué des travaux intéressants en proposant une technique d'analyse paramétrique qui permet notamment d'évaluer le degré de parallélisme en plus de permettre d'estimer le temps d'exécution d'un nid de boucles. Malheureusement, leur technique ne tient pas compte du nombre et du type d'énoncés contenus dans le nid de boucles. Leur technique considère principalement le nombre de points contenus dans l'intersection des polyèdres constituant le système. Récemment, dans [36], ils ont proposé

une méthode d'analyse des accès en mémoire qui serait utile à l'estimation du nombre de données manquantes dans l'antémémoire (« cache miss »). Avant eux, Tagwi [167] avait tenté d'estimer le temps d'exécution d'un nid de boucles en calculant le nombre de points compris dans chacun des polytopes qui constituent le nid de boucles mais il ne générait pas une expression paramétrique du temps d'exécution. Un polyèdre est un polytope à trois dimensions. Récemment, Vivancos et al. [179] ont proposé une technique d'analyse statique de performance. Leur technique vise l'estimation des performances de boucles en temps réel. En fait, de nombreuses techniques requièrent la connaissance du nombre d'itérations des boucles considérées durant le processus d'estimation. Vivancos et al. [179] ont proposé de formuler le WCET d'une boucle en fonction des chemins et des paramètres de la boucle (qui sont entre autres ses limites d'itérations inférieures et supérieures si ces dernières ne sont pas connues). Ainsi, durant l'exécution du programme dans son environnement réel, lorsque les paramètres sont enfin connus, la formule est appliquée pour prendre des décisions dans le cas d'un ordonnancement dynamique de tâches, par exemple.

Hodstedt et al. [71] ont aussi formulé une technique d'estimation du temps d'exécution de tuiles résultant d'un nid de boucles. Cette estimation permet de déterminer la forme optimale des tuiles exécutées dans une architecture multiprocesseurs. Poplavko et al. [138] ont tenté d'estimer le temps d'exécution d'une boucle en vue de la parallélisation dans une architecture multiprocesseurs.

2.6 Sommaire

Dans ce chapitre, nous avons résumé les techniques qui ont rapport avec nos travaux de recherche. La recherche sur les processeurs à usage spécifique est très avancée, mais des améliorations sont encore nécessaires en ce qui concerne l'exploitation des possibilités d'accélération par des instructions spécialisées. La synthèse à haut niveau est un domaine connexe qui souffre des mêmes lacunes. En effet, les deux domaines peuvent prendre avantage d'un large éventail de techniques de transformation et d'optimisation en

apparence méconnues pour maximiser l'utilisation de modules matériels. Les boucles de traitement ont souvent été ciblées pour l'accélération d'une application tant dans le milieu logiciel que matériel. Cependant, le succès de l'accélération dépend de la technique d'optimisation appliquée mais aussi surtout de l'architecture matérielle dans laquelle la boucle sera exécutée. Comme nous l'avons vu, de nombreuses techniques de transformation et d'optimisation sont basées sur le modèle polyèdral. Ce modèle n'a pas été très utilisé dans le cadre d'un processus d'exploration architecturale. C'est ce que nous tentons de faire en proposant une technique d'estimation des performances d'un nid de boucles selon des contraintes architecturales données.

CHAPITRE 3

DÉMARCHE DE L'ENSEMBLE DU TRAVAIL DE RECHERCHE ET ORGANISATION GÉNÉRALE DE LA THÈSE

Un des buts de ce chapitre est d'exposer la démarche scientifique qui a été suivie et qui a abouti à l'obtention des résultats présentés dans les articles inclus dans cette thèse. Aussi, dans un premier temps, nous présentons l'organisation et nous discutons de la portée de la recherche. Ensuite, nous montrons le lien entre les différents articles présentés en soulignant le fil conducteur qui les relie. Puis, nous discutons du choix des revues visées pour la publication de nos résultats. Enfin, le statut de chaque article sera explicité.

3.1 Organisation et portée de la recherche

Pour montrer la démarche scientifique suivie, il est utile de revenir en arrière et de décrire le contexte dans lequel cette recherche doctorale a vu le jour. Cette recherche s'intègre dans un projet qui a débuté en collaboration avec une entreprise ontarienne : Gennum Corporation. Cette compagnie était très intéressée par les méthodologies de conception d'architecture à usage spécifique comportant des processeurs dans le but de réduire le temps de conception, tout en atteignant les performances et les coûts ciblés. La division de Gennum avec laquelle nous collaborions travaillait sur des applications de traitement des signaux et, plus précisément, de traitement d'images vidéo. Leur demande a coïncidé avec la disponibilité dans notre laboratoire de recherche d'un générateur automatique de processeurs spécialisés (ASIP) de la compagnie Tensilica. L'idée nous est donc venue de concevoir des processeurs spécialisés avec cette nouvelle technologie et d'évaluer les performances atteignables avec ces processeurs. En appliquant les méthodes existantes de conception de processeur spécialisé, nous nous sommes rendus compte que ces dernières ne visaient pas spécifiquement l'accélération de boucles. Les applications

de traitement d'images vidéo comportent très souvent des boucles de traitement. Comme l'ont remarqué certains chercheurs, 90% du temps d'exécution d'une application concernent 10% des lignes du code qui la décrit et, généralement, ces lignes sont comprises dans des boucles qui effectuent des tâches souvent relativement simples et répétitives [163]. La plupart des techniques de conception d'ASIP génèrent des instructions spécialisées faiblement couplées, ce qui entraine de nombreux mouvements de données dans l'architecture du processeur. De plus, ces instructions faiblement couplées ne permettent pas de profiter au maximum des possibilités de parallélisation à haut niveau.

C'est ainsi que la question s'est posée à savoir comment les architectures à usage spécifique pourraient aider à accélérer des boucles de traitement. Pour répondre à cette question, deux volets de recherche reliés ont été abordés. Nous avons d'abord exploré comment utiliser les processeurs spécialisés qui constituent une classe populaire d'architecture à usage spécifique, supportée par des boites à outils efficaces Ces travaux nous ont amenés à proposer une nouvelle méthode de conception systématique pour obtenir des processeurs spécialisés de haute performance. Dans le second volet, nous avons accentué et généralisé notre recherche d'exploration architecturale en vue d'accélérer les boucles de traitement. Étant donné la complexité des étapes d'une exploration architecturale, nous avons focalisé nos efforts sur le profilage de boucles de traitement. En effet, c'est une étape primordiale du processus d'exploration architecturale qui permet au concepteur ou aux outils de génération automatique de faire des choix d'architectures appropriées. Malheureusement, les outils de profilage qui étaient disponibles s'avéraient très peu utiles pour choisir le type d'architecture appropriée à l'accélération d'une boucle.

Donc, durant la première partie de cette recherche, nous avons consacré nos efforts à la réalisation des tâches associées au premier volet de recherche. Cela nous a menés à concevoir manuellement des processeurs spécialisés dédiés à des applications de traitement d'images vidéos. Ces travaux ont d'abord fait l'objet de deux articles de

conférence [113, 116] et ils ont abouti à la proposition d'une nouvelle méthodologie de conception de processeurs spécialisés basée sur l'accélération de boucles.

Durant la réalisation du second volet de recherche, nous avons conçu un outil d'aide à la conception construit sur la plateforme de compilation SUIF [161]. Nous avons conçu et implémenté un profileur statique qui analyse du code C, en extrait les boucles et donne des informations utiles à l'accélération d'une boucle. Nous avons proposé des métriques orientées boucles. Les résultats obtenus ont fait l'objet d'un article de conférence présenté à la conférence ASAP en 2008 [114]. Le succès de cet article nous a poussés à enrichir nos travaux par d'autres métriques et à proposer une méthodologie d'exploration architecturale itérative qui peut assister un concepteur à trouver l'architecture la plus appropriée pour l'accélération d'un nid de boucles donné. Un des buts des méthodologies proposées est de réduire le temps de conception. Pour y arriver, une information très utile au concepteur, durant le processus d'exploration architecturale, est de savoir quel est le gain en performance atteignable si la boucle est accélérée. Pour répondre à cette question, nous avons proposé une technique d'estimation du temps d'exécution d'un nid de boucles. Pour valider la pertinence de nos résultats, nous avons estimé le temps d'exécution de plusieurs nids de boucles extraits d'applications de traitement d'images telles que le JPEG2000 et le JPEG, par exemple.

3.2 Cohérence des articles

Les trois articles inclus dans cette thèse seront présentés ci-après en insistant sur le lien qui existent entre eux et comment ils nous ont permis de répondre aux objectifs fixés. Les contributions de chacun des articles sont aussi énumérées.

3.2.1 Article 1 – Chapitre 4 : « A Novel Application-Specific Instruction-Set Processor Design Approach for Video Processing Acceleration »

Nous avons proposé une nouvelle approche de conception de processeur à usage spécifique qui est basée sur l'accélération de boucle. C'est cette nouvelle approche qui a

fait l'objet de notre premier article de revue [118]. Notre approche tient compte des possibilités de parallélisme au niveau instruction et à haut niveau. Par exemple, nous proposons d'ordonnancer les opérations d'un nid de boucles dans des grappes séquentielles d'instructions. Puis, le nid de boucles est déroulé et les grappes d'instructions sont couplées en tenant compte, bien sûr, des dépendances de données. Notre approche a permis d'obtenir de très bonnes accélérations allant jusqu'à un facteur de 18. Cette approche est itérative et manuelle. Une des difficultés que nous avons rencontré se rapportait à l'absence d'un critère d'arret clair pour décider à quel moment le processus d'itération devrait finir.

3.2.2 Article 2 – Chapitre 5 : « Loop Acceleration Exploration for Application-Specific Instruction-Set Processor Architecture »

Durant la cueillette des résultats présentés dans le premier article, nous avons remarqué qu'après certaines optimisations, les performances avaient tendance à plafonner. Cela nous a poussés à approfondir nos recherches dans le domaine de l'exploration de design d'architectures à usage spécifique, car la question qui se posait était de déterminer pourquoi les performances stagnent au délà d'un certain niveau. Répondre à cette question permet d'anticiper et de déterminer qu'un plafond d'accélération est atteint. En poussant un peu plus notre recherche, nous nous sommes rendus compte que, lorsque le plafond était atteint, nos efforts d'optimisation étaient vains et que les accélérations obtenues étaient nulles ou non significatives, en dépit des efforts et des resources investies. Comme nous l'avons mentionné plus haut, notre approche est itérative et, durant le processus de conception, le concepteur peut ajouter des optimisations pour obtenir de meilleures performances. Malheureusement, notre approche ne tient pas compte des possibilités de parallélisme de la boucle traitée, donc le concepteur pourrait s'évertuer à optimiser une boucle qui ne peut pas ou ne peut plus l'être. Dans la littérature, nous retrouvons de nombreux travaux sur de nouvelles techniques d'exploration architecturale, mais ces dernières, généralement, ne visent pas les boucles et elles se basent sur des métriques élémentaires telles que le temps

d'exécution ou la fréquence d'exécution de segments de code. Ces métriques élémentaires ne donnent aucune indication sur les techniques d'optimisation à appliquer pour améliorer les performances d'un segment de code. Donc, nous avons identifié un besoin pour des métriques orientées boucles qui permettraient de définir les aspects qui limiteront ou favoriseront l'accélération d'une boucle. Ce sont ces métriques qui ont été présentées dans le second article [117]. Ces métriques pourront être utilisées par le concepteur durant le processus d'exploration architecturale. Ces métriques ont déjà un avantage, puisqu'elles permettent, en très peu de temps, de visualiser s'il y a un plafond ou pas. Aussi, et plus important, elles pourront aider le concepteur à cibler le type d'optimisation à effectuer pour que le plafond d'accélération soit élevé. Les métriques sont collectées grâce à un ordonnancement des opérations d'un nid de boucles. Cet ordonnancement est inspiré de celui qui a été proposé dans le premier article [118]. Contrairement au précédent article, l'ordonnancement s'effectue automatiquement et un graphe d'ordonnancement est généré afin de calculer les valeurs des métriques.

3.2.3 Article 3 – Chapitre 6 : « Loop Nest Performance Estimation for Application-Specific Architecture Design »

Le travail résumé dans le second article [117] comprend la première partie de notre contribution en ce qui a trait à l'exploration architecturale de plusieurs types d'architecture à usage spécifique. En effet, comme nous l'avons mentionné auparavant, un plafond est généralement atteint et, comme nous le montrons dans l'article, ce plafond est dû soit aux dépendances de données soit aux contraintes de design de l'architecture ciblée. Les questions auxquelles nous devions répondre sont : 1) quels sont les facteurs causant le plafond d'accélération d'une boucle et limitant ou favorisant cette accélération et 2) quel est le plafond d'accélération en tenant compte des possibilités tant au niveau instruction qu'à haut niveau. Le second article a répondu à ces questions en partie. Dans le premier article, l'approche proposée vise à accélérer une boucle en appliquant des transformations qui font ressortir le parallélisme à exploiter dans des instructions spécialisées. Cependant, cette approche se fait à l'aveuglette, puisqu'aucun indicateur

n'est utilisé pour savoir si une meilleure accélération pourrait être atteinte. Donc, les questions qui se posaient étaient les suivantes : à quel moment devrait-on arrêter les optimisations et a-t-on profité de toutes les possibilités de parallélisme? Aussi, dans la suite de cette recherche doctorale, nous avons tenté d'estimer le temps d'exécution d'un nid de boucles en tenant compte des possibilités de parallélisme, tant au niveau instruction qu'à haut niveau. Cette estimation permet de mesurer l'accélération atteignable selon les contraintes de design associées au type d'architecture ciblée. Durant le processus d'estimation du temps d'exécution, les résultats d'ordonnancement présentés dans le second article sont utilisés afin de calculer les performances en tenant compte des possibilités de parallélisme à bas niveau. Puis, dans une seconde passe d'ordonnancement, les possibilités de parallélisme à haut niveau sont prises en compte pour estimer les performances totales du nid de boucles.

3.3 Contribution personnelle et collaboration scientifique

Cette recherche doctorale englobe plusieurs domaines de recherche. En effet, dans la revue de littérature, le lecteur a pu noter la grande diversité de travaux concernant la conception de processeurs spécialisés et la synthèse à haut niveau, entre autres. Les travaux de recherche du premier volet ont été réutilisés par des étudiants dans le cadre de projets de cours et de projets de maîtrise. En effet, l'approche de conception de ASIP basée sur l'accélération de boucle est très facile à comprendre. Étant systématique, elle permet d'obtenir des accélérations substantielles dans un temps raisonnable.

Durant le second volet de cette recherche, nous avons voulu faire le lien entre le monde des processeurs et celui des circuits générés à haut niveau. En effet, la génération d'instructions spécialisées et celle de circuits générés à haut niveau reposent sur l'analyse de graphes et l'exploitation des possibilités de parallélisme à bas et à haut niveau. Donc, notre outil permet une exploration architecturale de différents types d'architecture dans le même cadre de travail, ce que l'on ne retrouve pas encore dans la litérature, à notre connaissance.

Dans le cadre de ce doctorat, nous avons collaboré avec de nombreux étudiants. Par exemple, en proposant des projets de cours (cours ELE6305A: circuits intégrés de très grand échelle II), cela nous a permis de raffiner la méthode de conception de processeurs spécialisés basée sur l'accélération de boucles. Dans le cadre de ces projets de cours, nous tenions un rôle de gestionnaire de projet étant donné que nous avons proposé les sujets de projet et que nous avons activement guidé les étudiants tout le long de l'exécution des projets. Les étudiants ont conçu des processeurs spécialisés avec notre aide. Ces processeurs ont été intégrés dans un environnement de simulation duquel certains résultats présentés dans le premier article proviennent. Soulignons que nous avons conduit et produit la majorité des travaux scientifiques présentés dans le premier article (Chapitre 4), même si l'aide des étudiants nous a fait gagner un temps considérable quant aux aspects pratiques relatifs à l'utilisation des outils de génération de Tensilica. Aucune collaboration directe n'a été initiée pour la production des résultats exposés dans les articles 2 et 3. Nous tenons cependant à souligner l'aide apportée par Normand Bélanger durant la rédaction de tous les articles.

Soulignons aussi la collaboration avec M. Tremblay qui a mené à la rédaction d'un article de conférence [172] qui portait sur l'amélioration de l'utilisation des ressources d'un processeur DSP multi-ALU. Nous avons, par notre expertise en ce qui a trait à l'exploration architecturale, contribué à l'établissement de métriques qui permettent d'évaluer l'utilisation des ALU multiples du processeur.

3.4 Choix des revues et statut des articles

Le premier article [118], intitulé « A Novel Application-Specific Instruction-Set Processor Design Approach for Video Processing Acceleration », a été publié dans la revue "VLSI journal for digital signal processing" de la maison Springer. Cette revue avait fait un appel spécial de papiers sur des architectures et des méthodes de conception pour des applications de traitement de signal. Étant donné que nos travaux ont portés sur l'accélération de boucles extraites d'applications de traitement d'images et de vidéo,

notre premier article répondait à tous les critères d'appel de la revue. Cet article est disponible sur le site web de la maison Springer.

Concernant le second article [117], intitulé « Loop Acceleration Exploration for Application-Specific Instruction-Set Processor Architecture», nous avons opté pour "IEEE Transactions on VLSI" car nous proposons une approche d'exploration architecturale basée sur de nouvelles métriques et l'article focalise sur le processus d'exploration architecturale plutôt que sur l'outil dévelopé.

Le troisième article [115], inititulé « Loop Nest Performance Estimation for Application-Specific Architecture Design», a été soumis à la revue "IEEE Transactions on CAD". Cette revue cible la diffusion de nouveaux algorithmes et d'outils d'aide à la conception. Étant donné que le troisième article présente une nouvelle technique d'estimation de temps d'exécution d'un nid de boucle, les concepts présentés contribuent directement à l'automatisation des tâches du processus de conception d'architecture. De plus, la solution proposée pourrait facilement être implémentée dans un outil d'aide à la conception.

CHAPITRE 4

A NOVEL APPLICATION-SPECIFIC INSTRUCTION-SET PROCESSOR DESIGN APPROACH FOR VIDEO PROCESSING ACCELERATION

Mame Maria Mbaye[1], Normand Bélanger[1], Yvon Savaria[1], Samuel Pierre[2]
[1]Dept. of Electrical Engineering, [2]Dept. of Computer Engineering
École Polytechnique de Montréal
P.O. Box 6079 Station Centre-Ville, Montréal, QC, H3C 3A7
mbaye@grm.polymtl.ca

Abstract — *Application-specific instruction-set processors (ASIPs) provide a good alternative for video processing acceleration, but the productivity gap implied by such a new technology may prevent leveraging it fully. Video processing SoCs need flexibility that is not available in pure hardware architectures, while pure software solutions do not meet video processing performance constraints. Thus, ASIP design could offer a good tradeoff between performance and flexibility. Video processing algorithms are often characterized by intrinsic parallelism that can be accelerated by ASIP specialized-instructions. In this paper, we propose a new approach for exploiting sequences of tightly-coupled specialized-instructions in ASIP design applicable to video processing. Our approach, which avoids costly data communications by applying data grouping and data reuse, consists of accelerating an algorithm's critical loops by transforming them according to a new intermediate representation. This representation is optimized and loop parallelism possibilities are also explored. This approach has been applied to video processing algorithms such as the ELA deinterlacer and the 2D-DCT. Experimental results show speedups up to 18x on the considered applications, while the hardware overhead in terms of additional logic gates was found to be between 18% and 59%.*

Keywords: application-specific instruction-set processor, design exploration, optimization, parallelism, data grouping and reuse.

4.1 Introduction

The purpose of introducing a new technology or methodology is to achieve faster and better designs regarding performance, area, and power or energy consumption. Market pressure and consumer demand have pushed the industry to consider flexibility as a key design parameter in order to shorten design turn-around time and to make designs reusable. Application Specific Instruction-set Processor (ASIP) technologies offer a good compromise in the flexibility/performance gap that exists between general-purpose processors and Application Specific Integrated Circuits (ASICs). The capability of ASIPs to be extended allows reaching higher performance while the resulting circuit remains programmable. ASIP design requires different skills that range from software programming to hardware design, which complicates the use of ASIP technologies. Efficient ASIP design requires experience and a sound design method.

4.1.1 Research Context

Typically, system designers cannot use all the possibilities offered by new technologies. For example, in 2003, it was claimed [121] that ASIC designers use less than half of the 100 million gates allowed by 90 nm CMOS technologies. This difficulty in leveraging the available technologies is due to the design productivity gap. Better methodologies and tools are needed in order to design systems that meet performance constraints and fully use available technologies. ASIP technologies face the same issue with the lack of efficient methodologies and tools. Thus, potential users have concerns about using ASIPs in their System on Chip (SoC) design flow. That is one reason why ASIP design exploration research receives a lot of attention in academia. ASIP design involves many subjects such as: processor architecture, specific instruction-set, memory hierarchy, etc. In this paper, a new approach to ASIP design is proposed; it relies on the creation of Clustered Sequences of Tightly Coupled Specialized-Instructions (CSTCSI).

Hardware implementation allows reaching a higher level of parallelism (thus, performance) than software, but it is more constraining. Software coding is very

sequential, so software parallelism extraction depends on the architecture of the processor on which the software is executed. For example, a processor pipeline allows exploiting instruction level parallelism. ASIP instruction-set design typically consists of taking software code segments and transforming them in specialized-instructions enriching the basic instruction set of a processor. The chosen code segments should have properties that allow efficient hardware execution. Thus, the parallelism available in software code is transferred to the processor core, where it allows for faster execution. Software transformation and optimization can be applied in order to increase the available parallelism. ASIP design requires many manipulations on the software in order to reach the best possible specific instructions-set under the given context [191]. Our work aims at finding the available parallelism within the application and then designing sequences of tightly-coupled specialized-instructions to increase the application throughput if possible.

The contributions of this paper are:

- A new approach to Specialized-instruction (SI) design based on loop acceleration where loop optimization and transformation are done in SIs directly, instead of optimizing the software code.

- A methodology for the design of tightly-coupled specialized-instructions associated with loops based on a representation that we call 5-pattern representation. Using this methodology, a loop body is mapped to a sequence of specialized-instructions that is designed iteratively according to the loop's parallelism possibilities.

- This method allows reaching high performance and reducing the number of loads and stores during the application execution by keeping intermediate results in registers built into the SIs, instead of storing them in memory as variables and reloading them as needed. This technique creates a virtual pipeline during specialized-instruction execution.

- We propose data grouping and reuse during the sequence-of-SI design. To our knowledge, optimizing data transfers during SI design has not been reported yet, in spite of the fact that it is very useful for loop parallelization, as the sequence of specialized-instructions needs to receive fewer data elements in order to perform a given number of operations in parallel.

This paper is organized as follows. Section 1.2 gives a summary of the existing literature on ASIP, from processor generation to instruction-set design, and loop acceleration. Section 1.3 presents the framework on which the proposed method is based. Section 2 describes the steps of our method, which are: loop transformation following the 5-pattern representation, user-defined register design, operation clustering in tightly-coupled specialized-instruction (TCSI), loop transformation for parallelism improvement, and data grouping to reduce the volume of memory transactions. Section 3 summarizes the results obtained by applying the proposed method to design CSTCSI's for video processing algorithms. Finally, Section 4 draws conclusions from this work.

4.1.2 Related Work

ASIP research has evolved over the past few years from ASIP automatic generation languages and tools to specific instruction-set compiler design [33]. Many challenges have been solved, such as demonstrating the ability to generate processors automatically using languages such as LISA, for example. On the other hand, application specific instruction-set design is still an open issue. At first, researchers [33, 67, 75] proposed methodologies based on pre-designed coprocessors/functional units. It was shown [67] that adding functional units or coprocessors such as multiplier accumulator (MAC), pipelined memory or floating point coprocessor could be very efficient for application acceleration. However, this work did not take into account the die area cost of the added units. Imai et al. [75] proposed an HW/SW partitioning algorithm that tackles specialized-instruction-set design. The approach proposed by Cheung et al. [33]combined solutions proposed by others [67, 75] while introducing area overhead analysis during

ASIP design exploration. Their approach starts with the selection of suitable cores, followed by specialized-instruction selection and, finally, by estimating specialized-instruction performance and estimating area relative to the selected cores. The clock period of the generated core was selected as the largest delay between the core clock period and the delay of the SI. Their approach was based on critical function acceleration with loosely- coupled SIs. It implies that the generated core will often have a longer clock period due to the added SIs.

In contrast to the work of Cheung and others, this paper systematically aims at reaching the core clock frequency for the created SIs to avoid degrading the execution time of code segments that do not use the SIs. This is usually possible through systematic pipelining.

Methods and tools for automatic generation of application-specific instruction sets were described [39, 140, 192]. They are based on the analysis of the data flow graph of the application's C code in order to identify MISO (Multiple-Input Single-Output) or MIMO (Multiple-Input Multiple-Output) patterns. They extract and select patterns of arithmetic operations from the C code data flow graph. Selected patterns are then replaced with application specific instructions. The method proposed by Cong et al. [40] tackles the issue of the number of inputs and outputs to the specialized-instructions. The number of inputs and outputs has a large impact on SI performance and cost effectiveness. Their SI-selection method is area-driven. Pozzi et al. [140] propose new algorithms and heuristics that can be applied to larger pieces of code. Goodwin et al. [63] worked on MIMO pattern identification. Yu et al. [192] showed that this research on MISO or MIMO pattern identification may impose severe constraints that limit the achievable performance.

Jain et al. [80] have shown that choosing the right number and size of registers contributes to reaching the desired acceleration. Their work stresses the issue of data loads and stores during processing. Cong et al. [39] also point out this issue and propose a

system of shadow registers. Research still remains to be done on data transfers, because they are still a significant issue with specialized-instruction design. Research has so far been focused on the acceleration of sequences of basic operations, while data-transfer reduction is more difficult. Shekhar et al. [158] have shown the positive impact of reducing the number of memory accesses with ASIPs. It is well known that memory accesses performed by software are very costly in terms of speed and energy consumption. So, designing good register files and keeping temporary data in those registers can significantly reduce the number of memory accesses, as will be shown in this paper.

Current research on SI design is mainly focused on loosely-coupled specialized-instructions that assemble arithmetic and logic operations. Clark et al. [34] have applied another approach to ASIP design by considering algorithm subgraph acceleration. Their work targeted the integration of customized accelerators into pre-designed processor cores; they also targeted the critical paths of the applications and intermediate-result store reduction. This paper goes further by proposing an effective approach that finds application bottlenecks, which are usually found in critical loops, and accelerate them with tightly-coupled specialized-instruction sequences.

Typically, 90% of an embedded application execution time is spent in 10% of the source code lines, and this 10% of code lines usually implements loops. Suresh et al. [163] and Park et al. [132] have worked on loop acceleration using hardware modules. They propose to perform loop acceleration with hardware modules such as reconfigurable coprocessor deployed in FPGAs. Their approach could be applied to SI design. Suresh et al. [163] also propose to use dynamic loop profiling during application execution. So far, loop optimization has not been fully exploited in the SI design method proposed by researchers such as Sun and al. [162]. In several cases, unrolling a software loop before SI design only increases the processing time required to identify and select SIs. Typically, software cannot leverage loop parallelism, while SIs could benefit from this parallelism. It is well known that hardware components can efficiently support parallel

operations. Park et al. [132] have also shown the importance of data reuse during loop acceleration. Our work uses loop parallelism in order to accelerate critical loops with sequences of tightly-coupled specialized-instructions.

Software pipelining algorithms have been very successfully used in the 1990s with algorithms such as UPRU [160], and Perfect pipelining [2]. Software pipelining tackles loop parallelism by transforming the loop body into a pipeline. This concept has been reused in many other techniques for DSP code generation or high-level synthesis. The present paper uses software-pipelining concepts for clustered sequence of tightly-coupled specialized-instructions (CSTCSI) unrolling.

By contrast with previous work, we believe it is preferable to ensure that the core clock frequency is not lowered by added SIs because, if applications other than the one being accelerated execute on this ASIP, their performance would be affected by any clock frequency reduction. This goal is achievable if SIs are designed to meet the core clock frequency. The preferred solution to meet that design objective is to split SIs that tend to produce critical paths longer than the core clock period and pipeline them to ensure the ASIP can run at full core clock frequency.

4.2 ASIP Design Framework

This work is based on a framework to accelerate application bottlenecks using an application-specific instruction-set processor. The framework is illustrated in Figure 7. At the beginning of the process, we propose to perform configuration and generation of a basic core. This will define the basic core clock frequency and area in terms of number of gates. In the rest of the process, the basic core properties will be used as a reference when assessing the performance gain and hardware overhead of the generated ASIP.

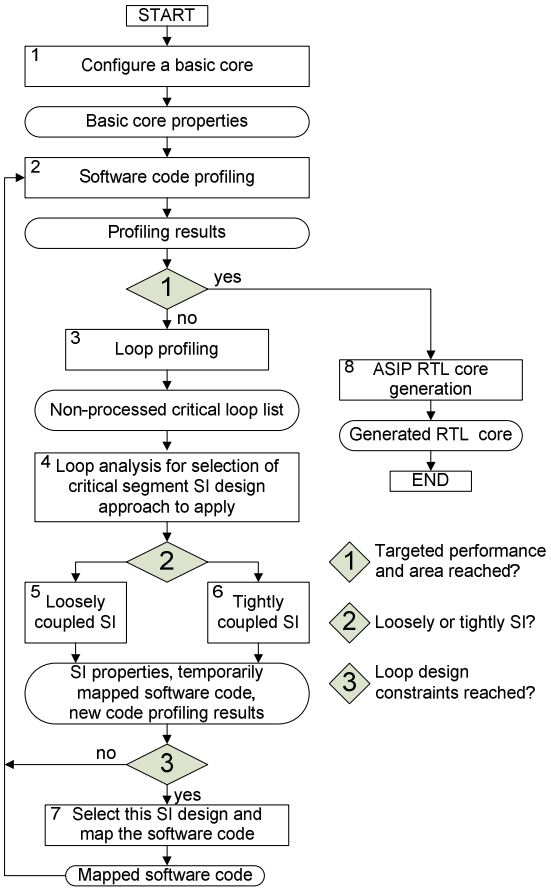


Figure 7. ASIP design exploration framework

In [116], we proposed an application-specific-processor design process that was used as a basis for the framework presented here. In the former framework, for instance, the basic core configuration step was not present. This step is very important since it will guide the rest of the SI design process. Our new framework also introduces the notions of loosely and tightly-coupled specialized-instructions.

In reference to Figure 7, the application is profiled on the basic core in step 2 and, if the target performance is not reached, loop-oriented profiling is done in step 3. Classic software code profiling can generate an unmanageable quantity of information that is not relevant due to the application's complexity [163]. Karuri et al. [84] proposed a coarse-grained source code profiling for ASIP design that helps reduce the design space. Their tool can be useful for loosely-coupled SI design. In our framework, we propose a specific

approach to such a loop analysis, which is very efficient, in order to track bottlenecks and to generate a reasonable quantity of information. Our framework tackles speeding up applications through critical loop acceleration with specialized-instructions. So, the loop-oriented profiling gathers the application's critical loop properties such as their execution time, the type of performed computations, the type of manipulated variables, etc. According to those properties, the critical loops are sorted. A critical loop is then selected and its acceleration technique with SI is defined (step 4). As we have seen previously, many SI design techniques are based on loosely-coupled specialized-instructions, but greater acceleration can be reached if the SIs are tightly-coupled. Thus, it is useful to define which SI technique should be applied to the selected loop according to its characteristics.

The loosely-coupled-SI design approach, applied in step 5, is useful when a loop segment is large and complex. In short, this approach is useful when loops do not have a dataflow structure, for example, when there are many jumps, or the code segment is too large in terms of number of lines, or the manipulated data structures are too complex, etc.

The tightly-coupled SI technique applied in step 6 is very efficient when the loop has a dataflow structure that comprises a sequence of basic operations with few branches, for example. In the rest of this paper, we will focus on the tightly-coupled-SI development technique that was found to produce good cost effective application acceleration when applicable. In short, this technique consists of creating a dataflow process with a sequence of tightly-coupled specialized-instructions. It minimizes data transfers by avoiding temporary-value loads and stores between the processor registers and memory. Hence, since most ASIPs are pipelined, we can create a virtual pipeline of CSTCSIs, where data is loaded in user-defined registers and then used during CSTCSI execution.

Once SIs are designed, their hardware overhead and performance are assessed and compared to the design constraints. Different criteria can be used in this context. For example, the ratio of speed-up to hardware overhead is interesting when die cost is a

concern or when cost effective solutions are desired. Another possible approach is to use only speed-up as a selection criterion with a hard limit on hardware overhead if there is a fixed transistor budget.

4.3 Sequence of Tightly-coupled Specialized-Instructions Design

The work presented here concerns the implementation of step 6 of the framework presented in introduction. This step (tightly-coupled SI design) deals with acceleration of critical loops based on analysis performed in steps 2, 3, and 4 of the framework. We propose a process flow summarized in Figure 8. In step 1, we begin by transforming the loop in an intermediate description called the 5-pattern representation (described in Section 4.3.1). This representation is used to differentiate the data transfer operations from the computation operations such as arithmetic, logical or multiplexing (selection) operations.

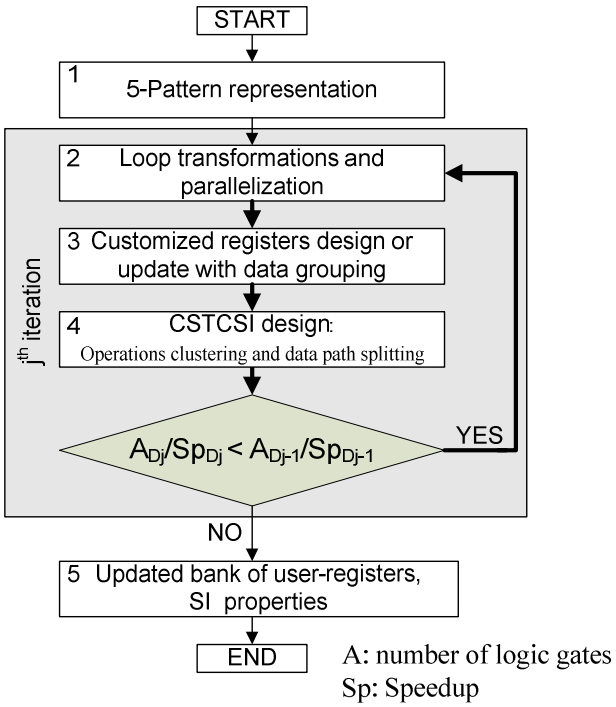


Figure 8. Clustered Sequence-of-Tightly-coupled-SI design process

Loop transformation and parallelization is applied in step 2 of the process (described

in Section 4.3.4). According to this representation, the input and output registers are also defined and designed in step 3 (explained in Section 4.3.2) and, in step 4 (described in Section 4.3.3), a CSTCSI is generated by following an iterative approach. We will see later in detail how a sequence of SI is designed, but, in summary, the sequence of SI is generated through clustering of code segments containing elementary operations.

A progressive approach is applied during the CSTCSI generation. Thus, designs are generated one by one, and, at each iteration j, going from steps 2 to 4, the A/Sp ratio (A: number of logic gates, Sp: Speedup) of the last generated design D_j is compared to the ratio of the previously generated design D_{j-1}. If the new ratio is smaller, a new pass of transformation and parallelization is applied to the current loop and a new CSTCSI design is generated. If the ratio is higher, it means that we have reached the limit of acceleration with the explored options. In this case, design D_{j-1} will be selected and its properties and new user-defined registers will be kept in a bank of registers, in step 5.

In the rest of this section, the different steps of the STCSI design process are described.

4.3.1 5-pattern Representation

Previous work [39] has shown that the number of data transfers and the number of inputs and outputs of an SI have significant impacts on the acceleration that an SI provides to an application. In fact, a tradeoff between the number of input/output transfers on one hand, and the number of operations on the other hand must be reached in order to yield the best acceleration. If this issue is not considered, a costly SI may yield a disappointing acceleration. The method that we propose minimizes data transfers between the application's variables and the processor's user-defined registers.

As illustrated in Figure 9, our method focuses on loops that constitute critical code segments of an application. Loop statements usually process many operations, and code optimization techniques dealing with them can have a strong impact. A loop can be

viewed as a sequence of five patterns: initialization, input, core operation, output, and wrap-up patterns. This 5-pattern representation proposal is inspired by software pipelining techniques such as UPRU [160], and Perfect pipelining [2].

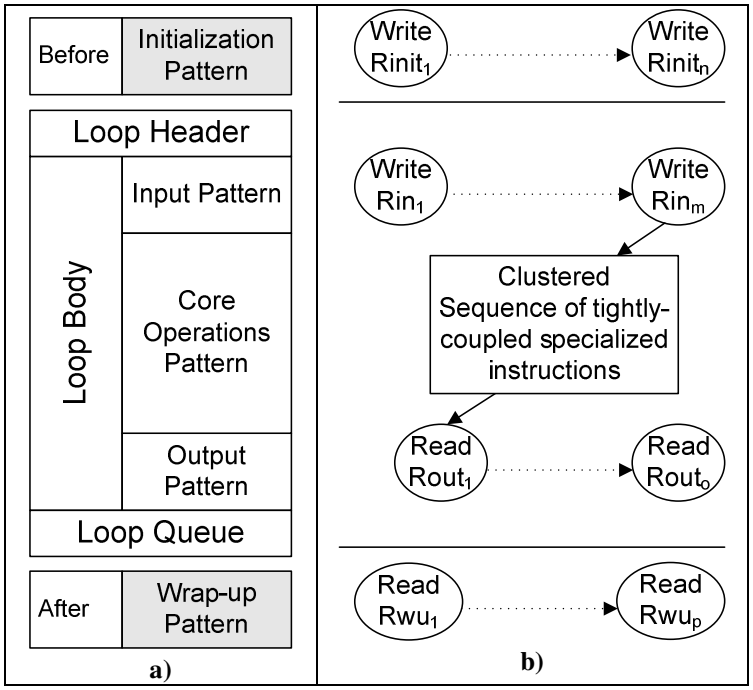


Figure 9. Pattern representation, a) intermediate representation, b) loop new code frame

The first pattern, the initialization pattern, deals with data transfers. This pattern gathers data that is required to update the Rinit registers. These registers must be initialized before loop execution. They can also be constant registers that need to be initialized, but whose values will not change during loop execution. The input pattern deals with data that will be loaded from the application variables and that will be required by the basic operation pattern of the loop iterations. This data will be stored in input registers: Rin, which will be initialized at the beginning of all iterations.

The core operation pattern gathers the loop's computation elementary operations and simple control statements composed of comparisons and multiplexing (selection) operations. The output pattern gathers memory operations that save loop iteration results stored in Rout registers, into application variables. Thus, at the end of the loop execution,

these registers should be read in order to update the application variables for the rest of the application execution. Some variables only need to be updated at the end of loop execution, which explains why the wrap-up pattern is needed. These variables will be mapped to wrap-up registers: Rwu. We should emphasize that a register can be reused to cumulate multiple functions such as input, initialization and output functions, for example.

Figure 10 illustrates a loop in its 5-pattern representation. In this example, we present the original code that has been transformed into the 5-pattern representation and then we give the code after the inner loop processing. In this example, coming from the noise reduction algorithm acceleration reported later, variable *noise* has been transformed into an initialization and wrap-up register: *Rinit-wu-noise*. This register is updated in the loop core operations. This loop has many inputs such as *t_im0*, *t_im1*, etc. Those inputs are used to calculate local mean and local variance stored in temporary variables: *t_mean* and *t_var*. At the end of each iteration, two arrays named mean and variance are updated with

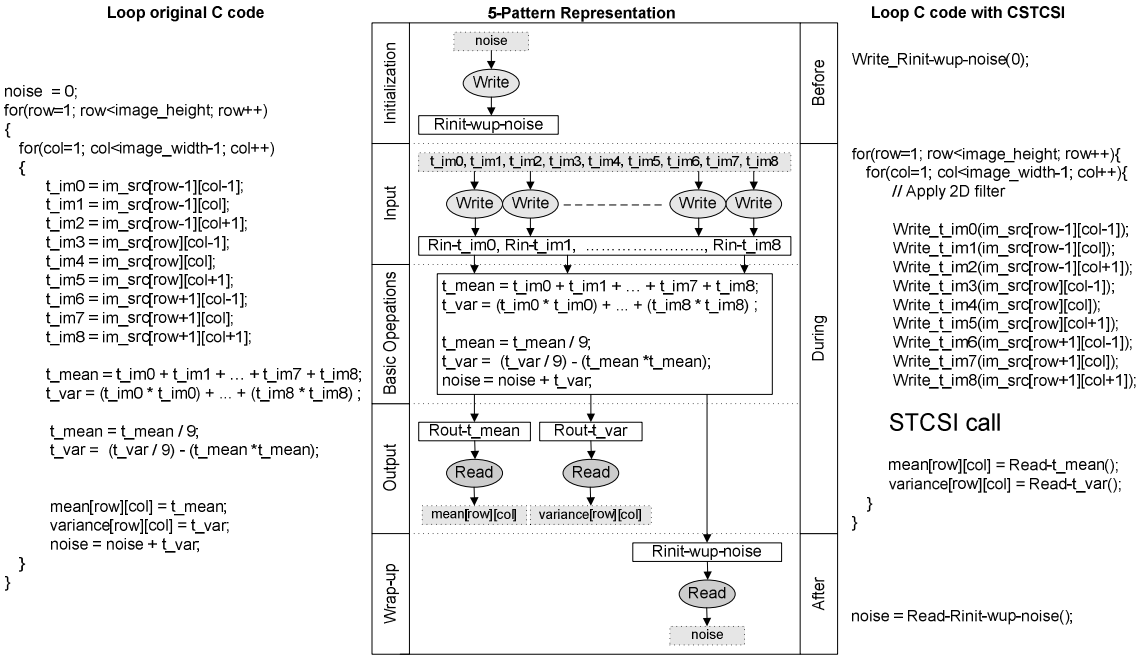


Figure 10. Example of a 5-pattern representation for a loop from the Wiener filter algorithm

user-defined registers *Rout-t_mean* and *Rout-t_var* values. Variable *noise* will be updated only after the execution of both loops. As can be seen in the final code, the CSTCSI is independent of data transfers, which have been separated from core operations.

It should be noted that, with this method, data transfers to and from user-defined registers and variables are avoided during CSTCSI execution. Grouping basic operations reduces dependencies to external data in order to allow more possibilities for parallelization and optimization during the rest of the CSTCSI design.

4.3.2 User-defined Register Design

The inputs and outputs defined in the 5-pattern representation are mapped to user-defined registers. The size of each register is the same as its associated variable. For example, if the associated variable is an integer, the register width is set to 32 bits. Other registers may also be added during the CSTCSI clustering and splitting steps. Those registers are called intermediate registers; they store temporary values computed by an SI and required by the rest of the sequence.

Obviously, the software code should be mapped according to the designed registers. Initialization and input registers should be updated at the appropriate time, and output and wrap-up registers should be read when appropriate. Otherwise, register values will be invalid or will be erased at the next execution of the CSTCSI.

4.3.3 CSTCSI Design: Operation Clustering and Data Path Splitting

The simple clustering technique that we use, illustrated in Figure 11, consists of first gathering the core operations blocks in a single SI that is then progressively split to match the critical path to that of the basic core clock frequency. Analysis and matching of critical paths to maximize the clock frequency is very important, as a key aspect of our method is to avoid reducing the basic core frequency. Thus, the potential CSTCSIs are compiled in order to assert the clock frequency they can sustain and the hardware overhead they introduce. Once the CSTCSI clock frequency is assessed, profiling can be

done in order to establish the speedup reached by the current design. Data path splitting creates a virtual pipeline. This pipeline is called virtual when the steps of a CSTCSI are executed in sequence with one feeding the other. This opens the possibility of repeating a step more often that the others or to make the execution of some step conditional. Note that in some cases a single instruction may capture all the processing needed in the core of a loop and repeated execution of the SI are fully pipelined. In this case the pipeline would not be virtual. If the configurable processor is pipelined, the sequence of TCSI will be executed through the processor pipeline, and so the CSTCSI will run faster.

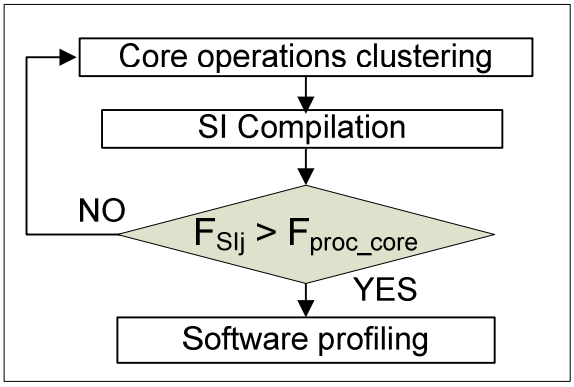
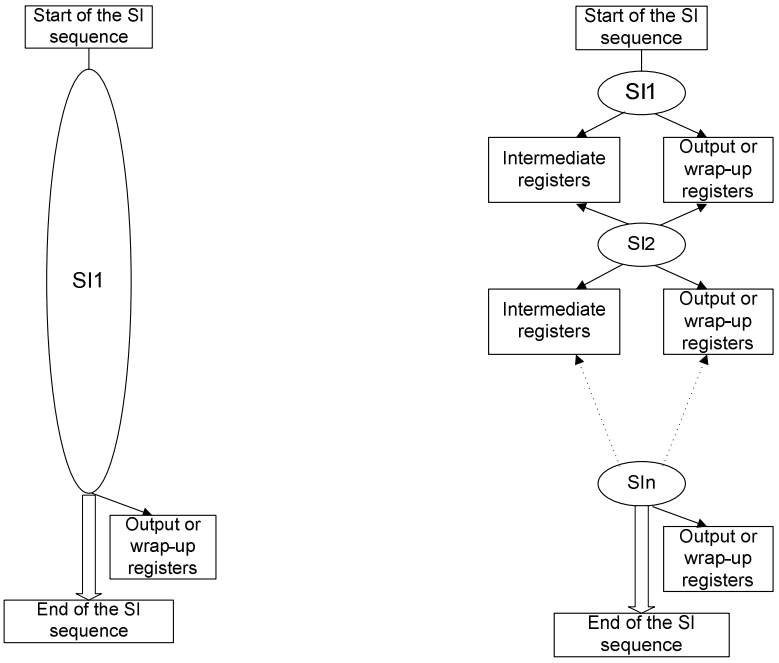


Figure 11. CSTCSI clustering process

During CSTCSI clustering, intermediate registers may be required in order to store temporary values, as shown in Figure 12. These registers are essential to the splitting operation. The intermediate results held by such registers are not accessible to the software, but the other SIs of the sequence require them. This is at the heart of the concept of tightly-coupled specialized-instructions; it allows direct forwarding of intermediate results from SI to SI through user-defined registers that are available only to the SIs. This allows maintaining a high clock rate while avoiding the creation of bottlenecks at the input and output of the SIs. Of course, implementing those registers adds latency, but most of the time, it was found to be quite cost effective. Temporary values are stored in registers rather than being written in temporary variables and then reloaded.



a) First specialized instructing b) Sequence of SI after N cutting operations

Figure 12. Sequence of specialized-instructions before and after clustering

Our goal is to minimize memory transactions, thus, during SI execution, the objective is to get a structure where no memory transfer is allowed as shown in Figure 12-b. SI intermediate results are stored in internal registers and the next SIs should use those registers' values. In fact, a dataflow structure is created where each step of the dataflow structure is an SI with a list of temporary values stored in intermediate registers. With this method, the application will run faster if the processor is pipelined. In this case, the sequence of SI execution complements the processor pipeline, which may create a virtual pipeline.

4.3.4 Transformation and Parallelization

The goal of this step is to increase the number of parallelization opportunities of the CSTCSI. It aims at increasing the number of computations performed in the CSTCSI. This is done by applying loop unrolling, data grouping, and data reuse optimization passes in order to reach our acceleration goals. The following discussion assumes that, when SIs that are part of a CSTCSI are in sequence, they are related by a true data

dependency. It is also assumed that such SIs have absorbed all possible elementary operations in an *as soon as possible* manner. The issue of spreading the hardware complexity over multiple SIs using a good scheduling that could reuse elementary functional units is not addressed in this paper.

CSTCSI unrolling

Since input data transfers are done before the beginning of CSTCSI execution, all data should be available for the first SI of the sequence, thus there is no data dependency during a CSTCSI execution. The potential for further acceleration with CTCSI after loop unrolling depends mainly on data dependencies between loop iterations. Such dependencies are also called inter-body dependencies. Figure 13-c shows the structure of a CSTCSI after loop unrolling, with an unrolling factor of *m* when there is no inter-body dependency while the initial CSTCSI is illustrated in Figure 13-a.

Loop unrolling can be applied only when the data dependency depth *ddd* is less than *N*, the number of tightly-coupled specialized-instruction (TCSI) in the sequence. If *ddd* is equal to or greater than *N*, the first SI of the next iteration should be launched after the current iteration execution, which precludes combining SIs from these consecutive iterations. If there is some data dependency between the loop iterations, *ddd* should be estimated. Then, according to *ddd* and *m*, where m is the unrolling factor, a new sequence is defined as shown in Figure 13-b, knowing that each SI regroups elementary SIs that were defined before loop unrolling as shown by the rectangles in the figure. Note that, in this figure, the first and second indices of the SIs give the SI identification number and the loop iteration number respectively. The two topmost elements in this figure are the first two SIs from the first loop iteration among the *ddd* SIs that must be executed before the second iteration can start. The other SIs are replaced with dots in the figure to make it more readable. The next two rectangles in Figure 13-b show the first two instructions from the group of *ddd* SIs that combine an SI from the first iteration and one from the second iteration. Again the rest of this group of *ddd* SIs is not shown for clarity. At some

point in the dataflow graph (depending on the relative values of *ddd* and N), a maximum in the number of SIs that are regrouped is reached as illustrated by the third pair of rectangles. Then, the number of SIs that are regrouped goes down progressively until only the last *ddd* SIs from the last iteration remain.

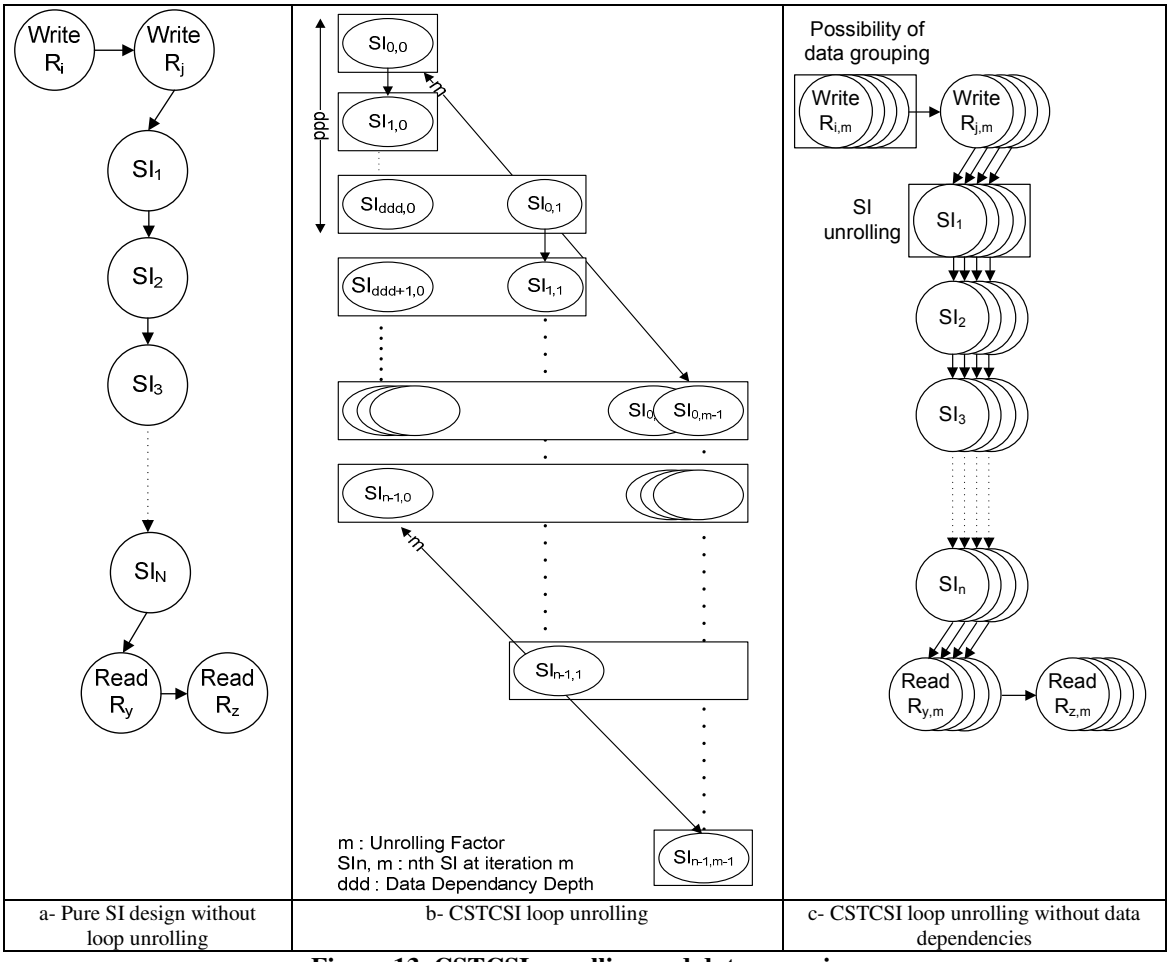


Figure 13. CSTCSI unrolling and data grouping

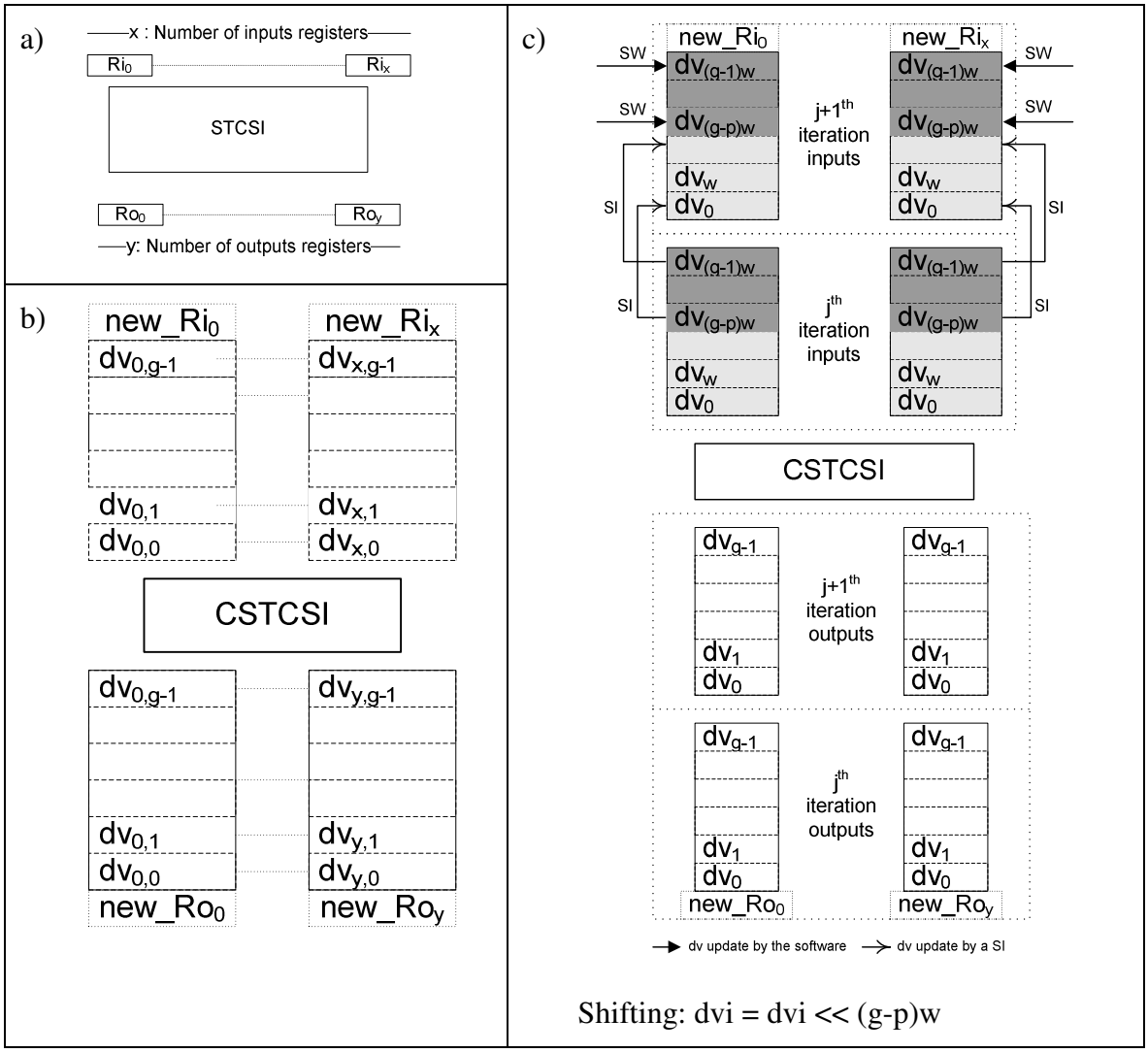
For example, when there is no data dependency in a CSTCSI, unrolling is simple, since, for each step of the sequence, the corresponding SI can be unrolled M times. In this case, the number of cycles required to execute the merged CSTCSI is the same as the initial sequence, regardless of the unrolling factor.

Unrolling is applied to the initial CSTCSI rather than unrolling the loop's software code and then generating the corresponding CSTCSI. The reason for this approach is that

unrolling the software code and extracting parallelism from it would, at best, yield the same result (i.e., lead to Figure 13-b) but it would require a lot more computation. This does not mean that all the parallelism in the application can be exploited by this technique. For example, loop fusion can make explicit additional available parallelism. Techniques of this kind can be used on the application before the proposed method and are therefore complementary.

Data grouping in user-defined registers

CSTCSI unrolling means that more data can be processed at the same time, thus more input operations are usually required before the beginning of each CSTCSI execution. In order to reduce the number of data transfers, we propose to send groups of data elements in a single block when possible. For example, in video processing algorithms, data is composed of pixels, thus a group of pixels can be sent in one block rather than sending pixels one by one. In practice, this means that, if a user-defined register was used to send one input data per iteration as shown in Figure 14-a, then g data elements or data values could be regrouped in one register, as shown in Figure 14-b, where g is the number of data values in a single register. In other words, Figure 14-b shows each register as a group of data values displayed on top of each other. This technique allows lowering the number of memory transactions, since reading simultaneously from memory 1, 2, or 4 pixels, for example, usually incurs the same latency. Of course, the system bus and the memory word length limit the extent of data grouping. The process of moving data values within registers (in order to reuse them) is illustrated in Figure 14-c. In this figure, the two indices are converted into one, in order to show how each register is shifted. The label in each sub word (i.e. for each data value) indicates the position of the first bit of the data value. It can be seen that the bits are shifted by a number of positions equal to $(g-p)w$, where g is the number of data values, p is the number of data values that are reused and w is the number of bits per data value.



x : Number of input registers

y: Number of output registers

dv : Data Value

j: Accelerated loop iteration

g: Number of grouped data value

p: number of reused data, with g > p

Figure 14. User-defined register design, a) initial design, b) with data grouping, c) with data grouping and reuse

The aim of data grouping is to feed the CSTCSI faster. This type of operation would usually not be efficient in a purely software code, as the processor typically does not have the required hardware to pack and unpack data efficiently. In a CSTCSI, bit-level data separation is natural, whereas software code would need costly (in terms of cycle count) bit manipulations.

Data reuse

In video processing algorithms (e.g. filtering, deinterlacing, compression), pixels are used several times to compute different output pixels. In most of these algorithms, computations are performed on a moving pixel window. Thus, some of the window pixels would be reloaded for the next iteration if a line buffer was not used. During the CSTCSI design, an analysis is done to detect which of the data already stored in registers is reused at the next iteration. By a simple register shifting operation, we avoid data reloading in user-defined registers. This shifting consists of moving data between input or initialization registers according to data redundancy patterns, thus removing data that is not needed anymore, as illustrated by Figure 14-c. It can be seen in this figure that, at each iteration ***j***, the registers' most significant bits depend on the ***g*** and ***p*** parameters, where ***g*** is the number of grouped data values and ***p*** is the number of reused data values. Parameter ***g*** must be strictly superior to ***p***, otherwise there is no possible data value to reuse. Thus, the register's new value, at the next iteration, will reuse part of its old value by moving bits from location ***j*** to ***j+g-p***. For example, register new_Ri_0 first data $dv_{0,0}$, new value at iteration ***j+1*** corresponds to data value $dv_{0,\mathbf{g-p}}$ at iteration ***j***. In fact, a register's least significant data values, 0 to ***p-1***, correspond to the register's most significant data values, ***p*** to ***g-1***, of its previous content. This data shifting is applied as part of the last SI of the sequence SI_n. So, during the next iteration, only the appropriate data (register bits) will be updated from the software.

Figure 15 gives an example where data reuse is applied. This example comes from the Deinterlacer algorithm [103]. In this case, data shifting was applied on two 64-bit registers ***Rlinebufup*** and ***Rlinebufdown*.** These registers contain eight 8-bit pixels that are used to compute 4 pixels stored in register Rpixels_out. At each next iteration of the algorithm, the old 32 least significant bits (Figure 15.a) of ***Rlinebufup*** and ***Rlinebufdown*** registers (bits 32 to 63) are reused. This is done by shifting them into the 32 most significant bits (Figure 15.b). As shown in Figure 15.c, bits 32 to 63 are shifted to the MSB positions of these registers for the next iteration.

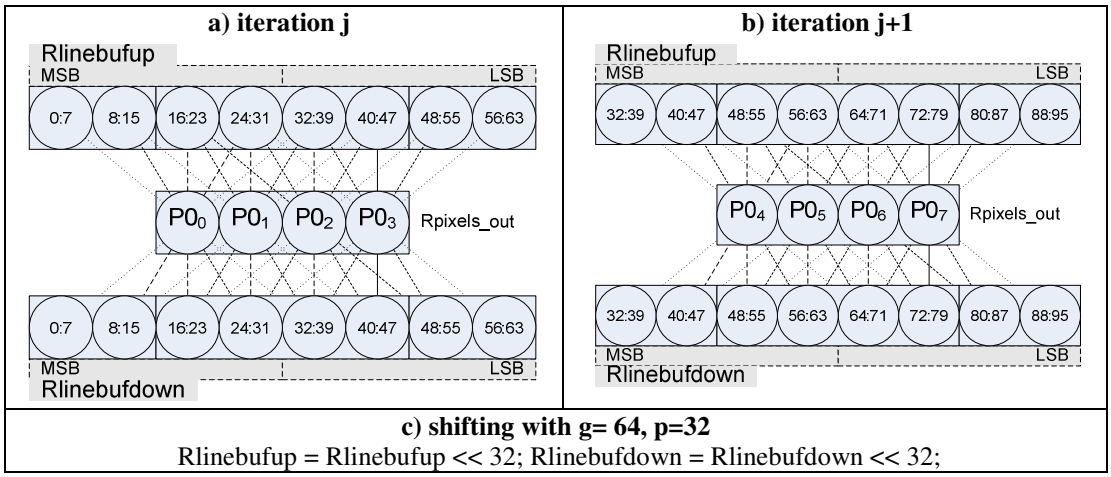


Figure 15. Data reuse example with user-defined registers

4.4 Experimental Results

4.4.1 Benchmark environment

The CSTCSI design methodology was applied manually on three video processing benchmarks: the Wiener Filter [102], the ELA Deinterlacer [103], and the 2D-discrete cosine transform (DCT) [1]. The first two algorithms are often used in digital television processing and the 2D-DCT is used in JPEG coding. The original code of these algorithms was first profiled as proposed by the framework (Figure 7), and their critical loops were analyzed.

Applying the proposed method to those benchmarks produced the following results. The Wiener Filter algorithm has two main critical nested loops, L1-W and L2-W, which have inner loops, as illustrated by Table 1.a. The ELA Deinterlacer algorithm has one main loop: L1-De, which includes an inner loop L1_1-De, as shown in Table 1.b. Finally, the 2D-DCT code has three main loops, but the first two loops were merged resulting in L12-Dct, as illustrated by Table 1.c. During the transformation and optimization step, the Wiener Filter's inner loops: L1_1_1-W and L1_1_1_1-W were entirely unrolled. We also unrolled by a factor of 4 the Wiener Filter and ELA Deinterlacer's inner loops of L1_1-

W, L2_1-W, L1_1-De, and the 2D-DCT main loops L_12-Dct and L_3-Dct. L2-Dct and L3-Dct core operations were identical; the difference between these two loops is their inputs and outputs, so the same CSTCSI are to be called during their execution.

Based on this analysis, the bodies of the inner loops of the Wiener filter were replaced with a CSTCSI design called D2-W. It then evolved to the D3-W design that was obtained by pipelining the critical path of D2-W.

Table 1. Benchmarks' loop structure

a) Wiener filter	b) Deinterlacer	c) 2D-DCT
L1-W	Field ..: L1-De	L12-Dct
L1_1-W	Field ..: L1_1-De	End L12-Dct
L1_1_1-W	End L1_1-De	
L1_1_1_1-W	End L1-De	L3-Dct
End L1_1_1_1-W		End L3-Dct
End L1_1_1-W		
End L1_1-W		
End L1-W		
Apply filter: L2-W		
L2_1-W		
END L2_1-W		
END L2-W		

The Deinterlacer's first design, D1-De, consists of one SI that processes 4 pixels at the same time. Then, considering the resulting execution time, speedup and hardware overhead, we split the SI data path, which resulted in design D2-De. Finally, D3-De was obtained by dividing the data path of D2-De into four SIs.

In the case of the 2D-DCT, the two loops, L1_Dct and L2-Dct, were merged. The 2-DCT was gradually refined into 4 designs, the first three, D1-Dct, D2-Dct, and D3-Dct, correspond to refinements of the CSTCSI obtained by gradually pipelining the SI to match the processor clock period. Thus, D3-Dct is the fastest design and it reaches the

target performance. These designs process 1 pixel per iteration. The last design, D4-Dct, processes 2 pixels simultaneously.

In order to assess the quality of the results, different reference processor cores were generated. Table 2 summarizes the benchmarks' core characteristics. These core properties have been used during speedup computation and analysis. For the DCT, we used a core with a floating-point (FP) coprocessor as some steps of the application operate on floating point numbers. The 2D-DCT CSTSI design requires floating point numbers conversion into fixed point, this conversion has been done with a tool from Cantin and al.[23].

Table 2. Benchmarks' basic core properties

	Wiener filter	Deinterlacer	2D-DCT
Processor family	T1050.0	T1050.4	T1050.4
Clock frequency (MHz)	100-215	135-255	135-230
Estimated gates	66860-80550	49760-59950	71480-86120
Cycles count (K)	47400	34000	12.011

Performance and hardware complexity were assessed using various tools. The designs were implemented using a modified version of the Bus Functional Model provided by Tensilica [168]. In those cases, the ASIP and memories communicate through an AMBA bus [6]s. This 32-bit bus is used to load 4-pixel data blocks read from an input memory and to store the computed 4-pixel output blocks to an output memory. This model allows profiling and computing improvements in a co-design environment. Also, the system was simulated using the Seamless environment (a Mentor Graphics tool) from which the profiling results were taken. SI synthesis was done with a 0.18 µm target technology. SI Synthesis was done with Design Compiler from Synopsys [165], in which emphasis was put on area optimization.

4.4.2 Results

In the rest of this section, the main results are presented. Performance is shown in Figure 16 and Figure 17 in terms of cycle count ratios and the corresponding speedup compared to the initial code. The 2D-DCT design D1-DCT yields a cycle count ratio of 12.51, but its execution time speedup is only 3.58, due to its longer critical path. That is the reason why the critical path was split resulting in design D3-W, which yields a cycle count ratio of 5.38 but a speedup of 6.26. Generally, a critical path split allows increasing the achieved clock frequency. As shown in Figure 19, clock frequency rose from 132 MHz for D1-DCT to 267 MHz for D3-Dct.

Hardware overhead in terms of number of gates in a design depends on the type of operations executed by the SI and the number and size of user-defined registers added to the processor. The Wiener Filter and 2D-DCT designs are costly according to results reported in Figure 18. For example, the hardware overhead of the Wiener Filter TCSI designs is between 9.27% and 59.11%. These applications require many multipliers that are relatively costly in terms of clock frequency and number of logic gates. The Deinterlacer does not require any multiplier and that is why the reported hardware overheads are so good in this case, with values less than 20%.

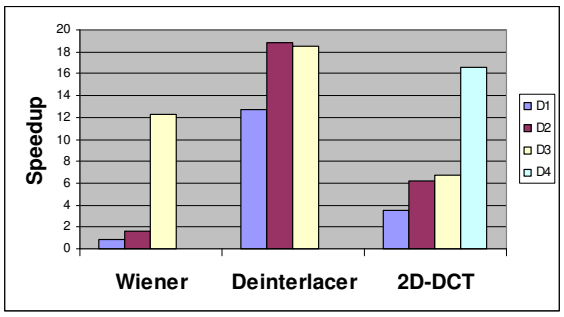


Figure 16. Speedup

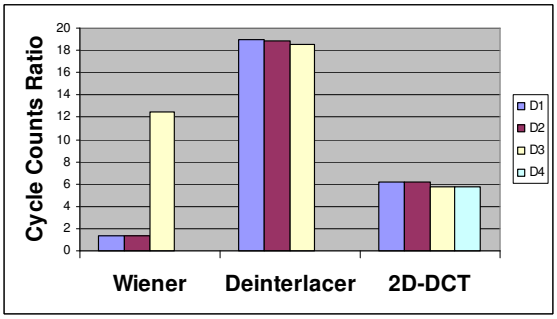


Figure 17. Cycle count ratio

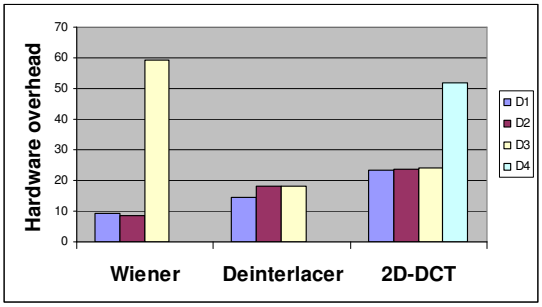


Figure 18. SI Hardware overhead

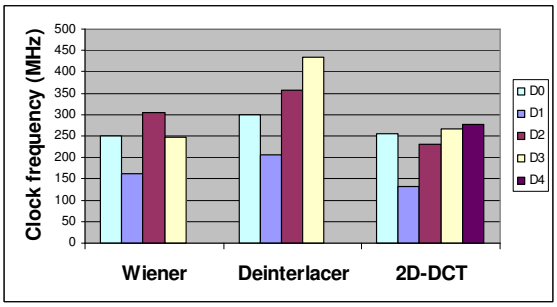


Figure 19. Clock frequency

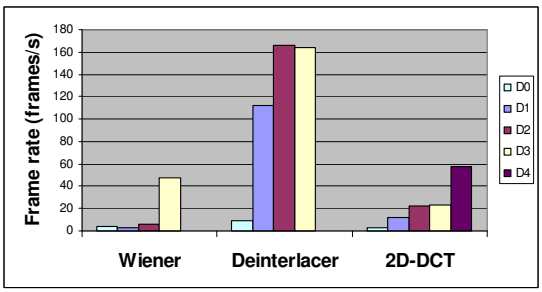


Figure 20. Frame rate

Since our approach targets video processing applications, it is interesting to assess if the developed designs reach a commonly required frame rate (30 frames/s) and resolution (288x352 8-bit pixels). Figure 20 reports the sustainable frame rates. As we can see none of these applications' original code could reach the 30 frames/s requirements. The Wiener Filter's original code frame rate is equal to 3.8 frames/s, while it is 8.8 for the Deinterlacer and 3.5 frames/s for the 2D-DCT. With our approach, the frame rate is increased above 30 frames/s with a maximum of 47 frames/s for the Wiener Filter, 165 frames/s for the Deinterlacer and 58 frames/s for the 2D-DCT. Based on the reported performances, a system designer can consider putting more processing on the same processor while preserving real time performance. Alternately, higher resolution image formats are increasingly popular. With resolutions and frame rate found in high definition television (HDTV), multiple processors would be needed for each application, in spite of the good accelerations. These accelerations could translate into a direct reduction of the number of processors needed to take care of each step of a complete application in a multi-processor system on chip (MPSOC) architecture.

For example, the Deinterlacer initial number of cycles associated to loads was 6160000, and it was lowered to 333000; a reduction by a factor of 18.49. The 2D-DCT benchmark initial number of cycles associated to stores was equal to 1691809; after the TCSI design, the number of clock cycles associated with stores is reduced by a factor of 4.54. In the D3-W and D3-De designs, every transformed loop processes and stores four pixels per iteration, while the D4-Dct loop processes and stores 2 pixels. These designs reach the highest load and store reduction with a factor of 12.59 in load reduction for D3-w and 11.65 for D3-De, for example. Thus, CSTCSI unrolling combined with data grouping leads to significant reductions in the number of load and store operations. In a pure software code, data is loaded, stored and reloaded later in the same sequence of operations. This shows that our methodology avoids costly temporary data loads and stores by storing temporary data in user-defined registers as described in Section 2.4.

Table 3. Number of cycles associated to loads and stores and reduction factor over the original code

		Loads	Load reduction factor over D0	Stores	Store reduction factor over D0
Wiener	**D0**	**11991458**	**-**	**3730037**	**-**
	D1, D2	**10781840**	**1.11**	**2358203**	**1.58**
	D3	**952293**	**12.59**	**443317**	**8.41**
Deinterlacer	**D0**	**6160000**	**-**	**2330**	**-**
	D1, D2	**333000**	**18.49**	**200**	**11.65**
	D3	**333000**	**18.49**	**200**	**11.65**
2D-DCT	**D0**	**5178269**	**-**	**1691809**	**-**
	D1,D2	**1679199**	**3.08**	**663779**	**2.54**
	D3	**1565152**	**3.3**	**85386**	**1.98**
	D4	**709792**	**7.29**	**372324**	**4.54**

4.5 Conclusion

ASIPs are a good solution for video processing algorithms acceleration. This family of applications has very often been accelerated with dedicated hardware modules. Typically, software implementations running on embedded processors are not able to reach the desired performance for an acceptable cost, size or dissipated power. ASIPs can reach this level of performance if they are well designed. We have proposed a new approach based on the development of CSTCSI (clustered sequence of tightly-coupled specialized-instructions) for ASIP design focused on loop acceleration. Each critical loop body is transformed and optimized in order to increase performance, thus, it is transformed into a sequence of tightly-coupled specialized-instructions. A 5-pattern loop representation decomposing loops into initialization, input, core, output, and wrap-up patterns was proposed to guide the process of creating SI sequences. Such sequences may be refined to reach higher processing parallelism when loops are unrolled and when user-defined registers are used to store intermediate results between specialized-instructions that compose the sequence. Our methodology was applied to the following video processing applications: ELA deinterlacing, noise reduction with Wiener filtering and DCT calculation. These applications contain repetitive set of operations and few branch statements. Our experimental results show that, when applied to video processing, our method provides significant speedups, with hardware overhead much smaller than the achieved speedups. The observed speedups are between 12-fold and 18-fold, while hardware overheads in terms of number of gates are between 18% and 59%. A significant contribution to the efficiency of the proposed method is the reduction in the number of load and store cycles associated with data reuse and grouping. Finally, it was shown that data communication is very costly in video processing algorithms as it increases execution time and power dissipation. On that matter, further work is needed to improve data reuse by applying data pattern analysis during the design. Another natural extension to this research would be the automation of the approach that would allow more design exploration to help identify and exploit available parallelism in the application.

CHAPITRE 5

LOOP ACCELERATION EXPLORATION FOR APPLICATION-SPECIFIC INSTRUCTION-SET PROCESSOR ARCHITECTURE DESIGN

M. Mbaye, N. Bélanger, *Member, IEEE*, Y. Savaria, *Fellow, IEEE*, S. Pierre, *Member, IEEE*

Abstract— Design space exploration is a delicate process whose success lay on the designers' shoulders. It is often based on a trial-and-error approach. Some basic metrics can be used to guide this process. In this paper, we explore accelerating loops from C-based specifications. We built a framework in which a design style, such as software-oriented or ASIP-oriented design, can be specified. We also propose an exploration process that allows targeting the main aspects that limit acceleration and the actions that can be made to improve it. The process is based on new loop-oriented metrics that provide insight in key design issue. They help to determine which aspects of the design between data accesses and ALU/Control operations limit or allow leveraging loop acceleration opportunities. We profile some benchmarks from the signal and image processing fields such as the Turbo Decoder and, the JPEG algorithms to illustrate how loop-oriented metrics help to point out aspects that limit or improve loop acceleration. The loop acceleration process was also used to explore design architectures that can leverage, as much as possible, the loop acceleration opportunities of the SAD algorithm.

***Index Terms*—Loop-oriented metrics, Design space exploration, Application-specific architecture, Loop parallelism, Data accesses**

5.1 INTRODUCTION

For decades, software and hardware engineers successfully worked in separate worlds. Nowadays, under market pressure, these two engineering fields have to work closer to design products faster and more efficiently. One design methodology that can help engineers to work more closely is C-based design. This design approach is grounded on a

specification written in C or a C-oriented language. Under that design approach, the design process begins with a development team capturing a target functional specification in a software description expressed with that popular software description language. The C language is easily understandable by hardware engineers and it can bridge the gap between the hardware and software worlds. Many types of C-based design approaches have been proposed. For example, some application-specific instruction-set processor (ASIP) design methodologies and some high-level synthesis (HLS) tools take as input a C-based specification. When expressed at high level, they are much faster to develop than a register transfer level (RTL) description expressed in a hardware description language (HDL). In complex systems, these specifications are needed to validate that the functionality satisfies user requirements before going to any kind of implementation. High level descriptions are used as reference models for verification. If any embedded processor executing part or the totality of the C-based specification can meet the parametric application constraints in terms of processing rate, latency, power, and energy consumption, the design of that part of the system is complete. Generally, in demanding applications, one or more parametric specification cannot be met. This is usually due to demanding processing loops. This paper focuses on finding and figuring how to implement demanding processing loops that prevent plain embedded processors from meeting some target parametric specifications.

In both the software and hardware worlds, loop optimization has been an efficient solution that leverages acceleration opportunities in code segments. For instance, software pipelining [178] has been widely used in the software community. Thus, choosing which type of design will be used to implement a code segment does not depend only on its execution time anymore, but it also depends on the acceleration opportunities of that segment. In this paper, we propose a new approach that identifies and analyzes loop-acceleration opportunities during a C-based design process for ASIP. Our approach also aims at determining which aspects, among data access, operations parallelism, and branch statements, limits loop acceleration and need to be improved. Our approach is based on a profiler-scheduler that takes as input design constraints that allow

targeting a specific type of design such as pure software or ASIP-oriented design. Thus, the first contribution of our work is a framework for architecture exploration in which different design styles can be specified. The most important contribution of this work is an iterative exploration process based on new loop-oriented metrics to explore loop acceleration opportunities. This process can help target the design style that will provide the best acceleration. The set of loop-oriented metrics help determine which aspect between data access and ALU/Control operations limit loop acceleration opportunities and which optimization technique might improve loop-processing time. These metrics also help evaluate whether the parallelization opportunities of some loop iterations are leveraged.

The rest of the paper is organized as follows. In Section 2, the state of art of ASIP design, high-level synthesis techniques, and other related work is presented. In Section 3, we detail how the architecture is described using design constraints that allow targeting a specific design-style. This section also provides a framework to compute the values of loop-oriented metrics. In Section 4, the design scheduler is presented. The proposed loop-oriented metrics are described in Section 5 followed by some experimental results about the loop-oriented metrics, in Section 6. Then, the exploration process that uses the loop-oriented metrics is presented in Section 7. Section 8 shows a case study based on the SAD algorithm, where an exploration process lead to a design that leverages SAD loop acceleration opportunities. Finally, conclusions are drawn in Section 8.

5.2 Relevant Literature

Early work on ASIP design [18, 33, 67] is centered on adding functional units (FU) or coprocessors to some processor core. It shows that integrating tailored computing units can help to accelerate a specific application. For example, adding a MAC or a floating-point coprocessor can be very efficient to accelerate some application types. Some prior work, such as Gupta et al. [67], does not take resource constraints into account during FU generation. Some others, such as Binh et al. [18], introduce a new partitioning

methodology for FU generation that considers resource constraints in terms of design area. Cheung et al. [33] target configurable processor designs. They propose a technique that selects the appropriate FU depending on the processor configurable features and they also consider resource constraints during the selection.

Meanwhile, another research trend emphasizes specific solutions based on processor instruction-set extension. The approach gained credit as automatic processor-generation tools reached maturity. With technologies such as Tensilica's LX2 [168] and Altera's NIOS II [4], it is easier to test and to verify the instruction-set customization algorithm. Many such algorithms were recently proposed. Galuzzi et al. [60] surveyed the instruction-set extension problem and they distinguished two types of instruction that rely on techniques such as Multiple-Input, Multiple-Output (MIMO) [7, 188] or Multiple-Input, Single-Output (MISO) [40, 61, 140]. These two techniques consist of clustering operations in single instructions that can have multiple input operands. The main difference between these types of instruction is the number of output operand: MISO instruction supports one output while MIMO instruction can have more than one output. As mentioned by Atasu et al. [8], most of the algorithms proposed for instruction-set automatic generation [39, 40, 192] consider some hard constraints on the number of input and output operands for the customized instructions. These hard constraints were due to processor register file properties. On the other hand, in processors such as the Tensilica family, a customized instruction can have significantly more (implicit) input or output operands. Cong et al. [39] note that operand constraints can contribute to remove some acceleration opportunities. Thus, they propose a shadow register mechanism that mitigates the bandwidth limitation by implementing some registers in which variables with short lifetimes are stored. Their solution can be efficient if the processor instructions do not handle too many input and output operands. On the other hand, if the processor instructions handle a significant number of operands, it is not necessary to limit custom instruction-generation algorithms with operand constraints. That is why Atasu et al. [8] propose an efficient algorithm based on convex subgraphs enumeration. Their technique tackles maximal convex subgraphs of a given data flow graph, which is geared towards

scheduling more operations both in parallel and in sequence within a single instruction. In [118], we propose an ASIP design methodology that is also based on maximal convex subgraphs enumeration of critical loops. Our methodology also focuses on loop acceleration based on data reuse with the use of specific registers. Overall, the main goal of the custom instruction generation techniques is to leverage the application parallelization opportunities by executing as many operations as possible in the same instruction. Most of the proposed algorithms are distinguished by their approach to reach the best tradeoff between performance (in terms of acceleration) and cost (in terms of area or energy consumption). Unfortunately, little work tackles leveraging loop parallelization opportunities during instruction-set design. In [118], we propose a novel approach that is based on loop optimization during ASIP design. We show that loop optimizations (such as loop unrolling, or data-reuse that can be leveraged with the use of custom registers) contribute to obtain significant acceleration. Data optimization techniques are well documented in the software field [130], but they are not widely used during ASIP designs.

Loop optimization and transformation has been investigated by software and hardware researchers for decades. Many commercial and free software compilers, such as GNU GCC [56], apply loop transformations when possible. Panda et al. have enumerated, in [130], many loop transformation techniques for data and memory optimization. In fact, loop transformations do not just help to accelerate an application: they can also contribute to reduce area and energy consumption by optimizing data locality or data reuse, for example. In the software area, some of the most effective loop optimization or parallelization techniques are based on the polyhedral representation of nested loops [15]. For example, Wilde et al. [184] have proposed a memory reuse analysis technique based on that model. The polyhedral model has also been widely used by IMEC for their work on data transfer and storage exploration methodology (DTSE) [76] and, more specifically, on loop exploration for memory-size estimation [10]. Our profiler-scheduler, elaborated in this paper, analyzes variable's access function in order to evaluate data reuse opportunities during the acceleration-exploration process.

Code profiling has always been a strategic step for hardware or software architecture design exploration and for HW/SW partitioning, since most processes start with software-prototype profiling. Currently, few loop-oriented profilers are available, but we can mention FLAT [163] or LoopProf [171]. FLAT proposed by Suresh et al. provides loop execution time and the number of iterations performed during code execution. It also tracks loops that should be substituted by a hardware module (HM). More recently, Tipp et al. [171] presented a loop-oriented profiler LoopProf. Their profiler targets software-oriented design and gathers information for critical loop detection. The profiler also provides a call graph that can be analyzed to identify potential parallelizable loops. Most of the available loop-oriented profilers such as FLAT [163] or LoopProf [171] only track application bottlenecks and do not provide information about how software loops are amenable to acceleration with specialized-instructions (SI) or dedicated hardware modules. By contrast, the profiler presented in this paper implements metrics that will help make this kind of decision. In the ASIP field, profiling activities are based on old profiler metrics. One very popular profiler is GNU GPROF [64]. It provides some basic metrics such as the number of execution cycles associated with each called function or the number of times that each function was called. These metrics are useful to determine which functions need to be accelerated. On the other hand, narrowing the profiling to critical loops would be more informative. GPROF's metrics cannot be used to make some decisions for setting design properties. This led Karuri et al. [84] to propose new metrics that can be used during the ASIP architecture-design process. Their profiler tracks some metrics such as operator execution frequencies or conditional-branch execution frequencies, or average jump distance. Their metrics can help to determine which functional units should be included in a processor or to decide which branch-support hardware should be added in the architecture. Unfortunately, although their metrics tackle ASIP-oriented design, they do not propose loop-oriented metrics that would track critical loops and loop-acceleration opportunities before taking the decision to accelerate with an ASIP-oriented design, for example.

So et al. [159] define a balance metric for FPGA-based systems. This notion of

balance was first introduced by Carr et al. [25]. The metric is used to determine balance the weight of storage and computation operations on the design performance. Thus, they propose a design process that aims to obtain a balanced performance when designing storage and computation operations. Unfortunately, they do not propose further metrics that would help tackling more precisely the aspect to improve. Thus, our metrics and the exploration process that we propose can be used to enhance their design exploration for FPGA systems.

To the best of the authors' knowledge, loop-acceleration opportunity exploration has not been investigated in the past for ASIP design exploration. Many techniques have been proposed to accelerate loops (loop unrolling [180], loop shifting [43], loop interchange [150], loop parallelization [111], loop fusion [164], etc.) but their efficiency depends on a loop's potential to be accelerated. It is well known that executing loops with dedicated hardware design is an efficient solution to accelerate application. However, the success of this design option depends on the loop properties, such as its parallelization opportunities and the targeted design style architectural constraints. Thus, the main contribution of this paper is to provide a framework that allows targeting a specific design style such as software-oriented or ASIP-oriented design for loop acceleration. In this framework, each loop of an application is profiled according to metrics that help define loop acceleration opportunities and find which aspect, between data access and ALU/Control operations, limit loop acceleration. The metrics also help determine whether loop parallelization opportunities are well exploited by the targeted design style (knowing, as previously outlined, that design styles are often distinguished mainly by the number of operations that they allow executing in parallel).

5.3 Design Architecture

In a pure software design using embedded general-purpose processors (EGPPs) such as a RISC [70], a processor typically executes only one instruction (that contains only one atomic operation) at a time as illustrated in Figure 21(a). On the other hand, an ASIP

can execute an instruction that includes many atomic operations (see Figure 21 (b). Atomic operations can be executed in parallel or in sequence, as long as the instruction critical path meets the processor clock-period constraint. The control and steering operation overhead is about the same as for the EGPP, but the amount of processing per instruction is higher, thus the relative overhead is reduced.

As a first step in this work, we do not take into account the cost of control and steering operations. We focus on operation acceleration and their parallelization opportunities. Toward that goal, we define constraints that will help to select the best design.

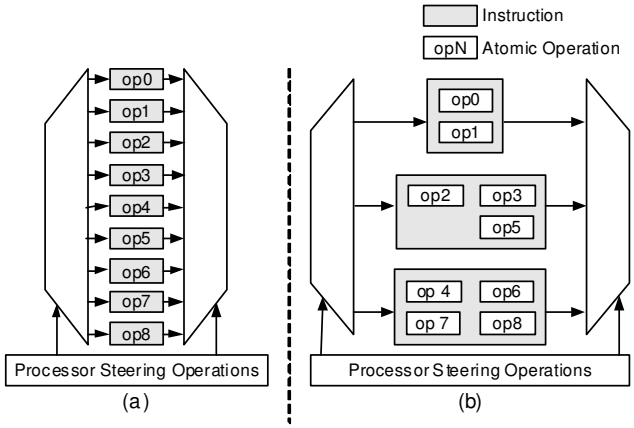


Figure 21. Operation-execution framework with different types of design: a) EGPP, b) ASIP-oriented

One important bottleneck that can limit performance is data accesses. Figure 22 shows a simplified view of loop-iteration execution. In general, independently of the design method used to implement the loop functionality, the execution time can be expressed as the sum of the load and store transaction latencies and the operation execution latency. It is clear that, if most of the non-parallelized/non-parallelizable portion of the code is found in load and store latencies, it would be more efficient to focus on data management improvements than on operation parallelism as shown by So al. [159].

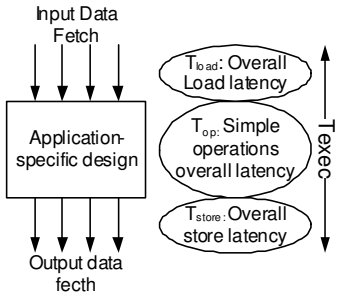


Figure 22. Iteration execution overview

The cost of data operations depends on the design memory hierarchy. Figure 23 illustrates the memory hierarchy that is usually implemented in each design style. Figure 23 (b), shows that user-defined registers (UDR) can be implemented to store specific data under the control of the application (i.e. the software compiler cannot generate code that accesses these registers directly). These UDRs are used to store data that cannot be reused efficiently with general purpose register-GPRs or caches. Caches are widely used in software-oriented designs to leverage data reuse opportunities and to avoid repetitive and costly memory accesses. On the other hand, depending on the cache size and the volume of data, cache miss overhead can have a significant (negative) impact on performance for some applications [47, 91]. The number of concurrent memory accesses is usually constrained. Most of the EGPP have a memory management unit that controls the I/O pins synchronization with external memories. In general, there is only one EGPP memory management unit, which typically means that only one memory transaction can be performed at a time. In ASIP designs, the number of memory management units can be extended, thus allowing more concurrent I/O transactions. In some EGPP systems, memory access latency can be lowered by adding an external memory controller that supports multiple outstanding requests [30].

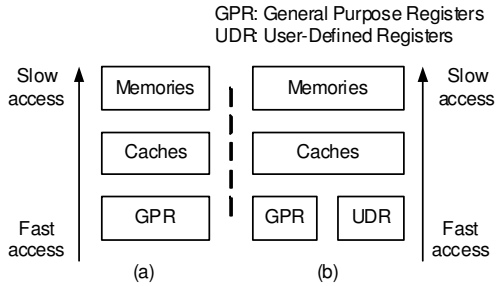


Figure 23. Simplified view of a memory hierarchy in different types of design: a) EGPP, b) ASIP-oriented

5.4 Design Scheduler: LProf

The aim of the design process is to explore the acceleration opportunities of a loop. To ease and to improve the loop acceleration exploration process, we use static estimates of the critical path because, at this phase of the exploration, the use of dynamic techniques to estimate the critical path delay would be time consuming. The design critical path delay is computed by a scheduler named LProf that uses the loop C code. LProf was briefly introduced in a previous paper [114]. The scheduler generates the computation graph of instructions under development. The computation graph is used to estimate coarsely the number of computation steps (CS) required to execute loop iterations. The number of CS can also be considered as the design critical path delay in terms of cycles.

A CS is an atomic instruction and the operations scheduled in a CS are the atomic operations assigned to one instruction, as illustrated in Figure 24. The operations (Op) are related by data dependencies.

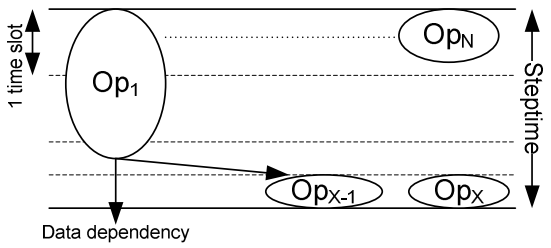


Figure 24. Representation of a computation step

Our scheduler takes as input the design constraints that are listed in Table 4. These design constraints will be used to describe the targeted design. A design is specified according to operator constraints and some global constraints. The operator constraints characterize the operators, their delay and their type (data-oriented, arithmetic- or logic-oriented, or branch-oriented). The global constraints control how acceleration and parallelization can be done in one computation step. A CS is characterized by the number of time slots, called *steptime*, that it contains. This parameter can be described as the maximum instruction execution delay. In general, the number of sequential operations will depend on *steptime*, on the operators delay, and on data dependencies. During scheduling, the sum of the delays of sequential operations in one CS must be less than or equal to *steptime*. Thus, stretching *steptime* allows executing more sequential operations in one CS (at the expense of a slower clock frequency). In a software design, the CS delay is the maximum delay of an atomic operation. The computation step delay can be stretched according to the requirements. Designs are also mainly distinguished from each other by the number of concurrently executed operations: *max_conc_op*. Increasing *max_conc_op* may allow multiple operations to be scheduled in the same CS. In software designs, *max_conc_op* is equal to 1.

Table 4. List of design constraints to the scheduler

	Name	**Description**
Constraints per operator	type	Operator type: data-oriented, arithmetic or logic (AL) oriented, or control-oriented.
	delay	Operator execution time.
Global contraints	I/O	I/O constraints such as the number of concurrent load, concurrent store, and concurrent load and store operations.
	steptime	Time allocated to each computation step.
	max_conc_op	Maximum number of concurrent operations in a CS.

Note that stretching *steptime* can introduce a slowdown factor compared to reference design *steptime*. But, as it was stated in [118], the designer shall target a slowdown factor (SDF) equal to 1, otherwise the efficiency of hardware resources invested to obtain the speedup factor is degraded by the SDF. An SDF of 1 is obtained if custom instructions in

an ASIP are designed to operate at the same clock frequency as the base processor. Thus, in the rest of the paper, the slowdown factor is assumed to be 1.

LProf implements an ASAP (As-Soon-As-Possible) scheduling of atomic operations. LProf takes as parameters the types of operations to ignore during the scheduling process. The scheduler's 3 parameters are:

- ID: ignore data-oriented operations;
- IAL: ignore arithmetic and logic -oriented operations;
- IC: ignore control-oriented operations.

For example, if ID is set, then LProf discards the data-oriented operator original delays and sets them to 0 during scheduling. Metrics proposed later are thus derived from versions of the same ASAP-scheduled graph, where classes of operation are temporarily assumed to take zero time to focus on the impact of classes of operations or features of the application.

LProf has been built using two compiler frameworks: SUIF [161] and Machsuif [106]. SUIF creates an intermediate representation (IR) that facilitates coarse-grained analysis and transformation of software code. Machsuif uses an IR allowing fine-grained analysis and transformation and it takes as input a SUIF IR. In this work, profiling passes were added to the SUIF framework and a scheduler was added in Machsuif.

5.5 Loop-oriented Metrics

The proposed metrics help identify which part of a loop iteration should be optimized according to the selected design style. The approach relies on the proportion of the iteration that is concerned with data accesses and the proportion that depends on ALU/Control operations. Table 5 lists the proposed loop-oriented metrics. In the ALU/Control group, six metrics define the maximum acceleration reachable when data-oriented operations are ignored during the scheduling phase. The first metric (A_{Op}) is the reachable acceleration when data operations are ignored. This metric reflects the true

performance when memory access can be removed, accelerated by a very large factor or masked by performing them in parallel with some other unavoidable tasks. When considered as a bound, it measures the weight of calculations and branch operations in the design performance.

Table 5. Loop-oriented metrics

Type	Name	Scheduler parameters	Definition
AL/Control Operations	A_{Op}	ID=1	Acceleration reached by operations when data-oriented
	A_{Arith}	ID=1, IC=1	Acceleration reached when data-oriented and control-
	A_{Branch}	ID=1, IAL=1	Acceleration reached when data-oriented and AL –
	DoP	ID=1	The overall DoP is the sum of partial DoPs of each CS divided by the number of CS. CS DoP is computed as follows: the number of scheduled operations it contains is divided by the maximum number of possible concurrent operations (the design constraint max_conc_op).
	op_mean	ID=1	Average number of operations executed per computation
	op_variance	ID=1	Variance of the number of operations per step. It indicates whether the operations are being spread out uniformly among the different CS.
Data	A_{Data}	IAL=1, IC=1	Acceleration bound due to data accesses when ALU and
	$A_{DataReuse}$	IAL=1, IC=1	Acceleration reachable with data reuse.
	A_{IO}	IAL=1, IC=1	Acceleration reachable with optimized number of I/O.

The ALU operations are distinguished from the branch operations and their impact in the design performance is gathered by two metrics A_{Arith} and A_{Branch}, respectively. In most cases, accelerating arithmetic and logic (AL) sequences of operations is easier than accelerating branch statements. Branch statements often reduce parallelization opportunities as they imply control dependencies. Thus, when A_{Arith} is higher than A_{Branch}, it means that the control statements overshadow some acceleration opportunities. The fourth metric (the degree of parallelism — DoP) measures how effective a system is at scheduling useful activities over a set of available operating units (ALU in a wide sense). As previously explained, a computation step is the span of time over which atomic operations are executed. The maximum value of the DoP efficiency measure is 1. When the DoP is low, the last two metrics, op_mean and op_variance, help analyze the operation distribution over the computation steps.

The second group of metrics, which concerns data operations, includes A_{Data} that

defines the maximum acceleration reachable when the ALU and branch oriented operations are ignored. If A_{Data} is lower than A_{Op}, it may be useful to look at means to optimize access to data. Toward that goal, we propose the next two metrics $A_{DataReuse}$ and A_{IO}. $A_{DataReuse}$ determines the acceleration reachable if data reuse opportunities are leveraged. A_{IO} is the reachable acceleration if the number of interfaces is increased, i.e. when the number of concurrent load, store, and load/store is left unconstrained. The acceleration is computed by dividing the number of CS of a specific design by the number of CS of a pure software design.

In order to demonstrate the use of these metrics, we scheduled the code segment illustrated in Figure 25.a. It has been extracted from a pipelined 2D DCT algorithm slightly modified to introduce data reuse opportunities. This segment was scheduled according to the following design constraints: the number of concurrent operations is unconstrained, the computation step delay, *steptime*, is set to 1, and the I/O constraints are set to 1. A reference design was also profiled. The reference design is a pure software design for which *max_conc_op* is equal to 1. The reference design has 63 CS. Figure 25.b draws the resulting initial scheduling graph when all types of operations are considered during the scheduling. The scheduling graph comprises 22 CS in which the nodes are described as follows:

- Arith OpX nodes represent the AL operations required for memory address generation belonging to block S1;
- LD OpX nodes concern the fetching of 8 pixels stored in array PIX_IN;
- Arith OpX' nodes belong to S2 block AL operations;
- STR Op node concerns the storage of the output pixel PIX_OUT.

Figure 25.c illustrates the scheduled graph when data operations are ignored (LProf set data operation delay to 0). In that case the number of CS reaches 11, leading to A_{Op} being equal to 5.72. From Figure 25.d, note that the number of CS, when AL and branch operations are ignored, is equal to 9 and the resulting A_{Data} is 7. Exploring the data reuse

opportunities allows neglecting seven of the 8 load operations. Figure 25.e presents the scheduled graph when data reuse opportunities are leveraged and AL and branch operations are ignored.

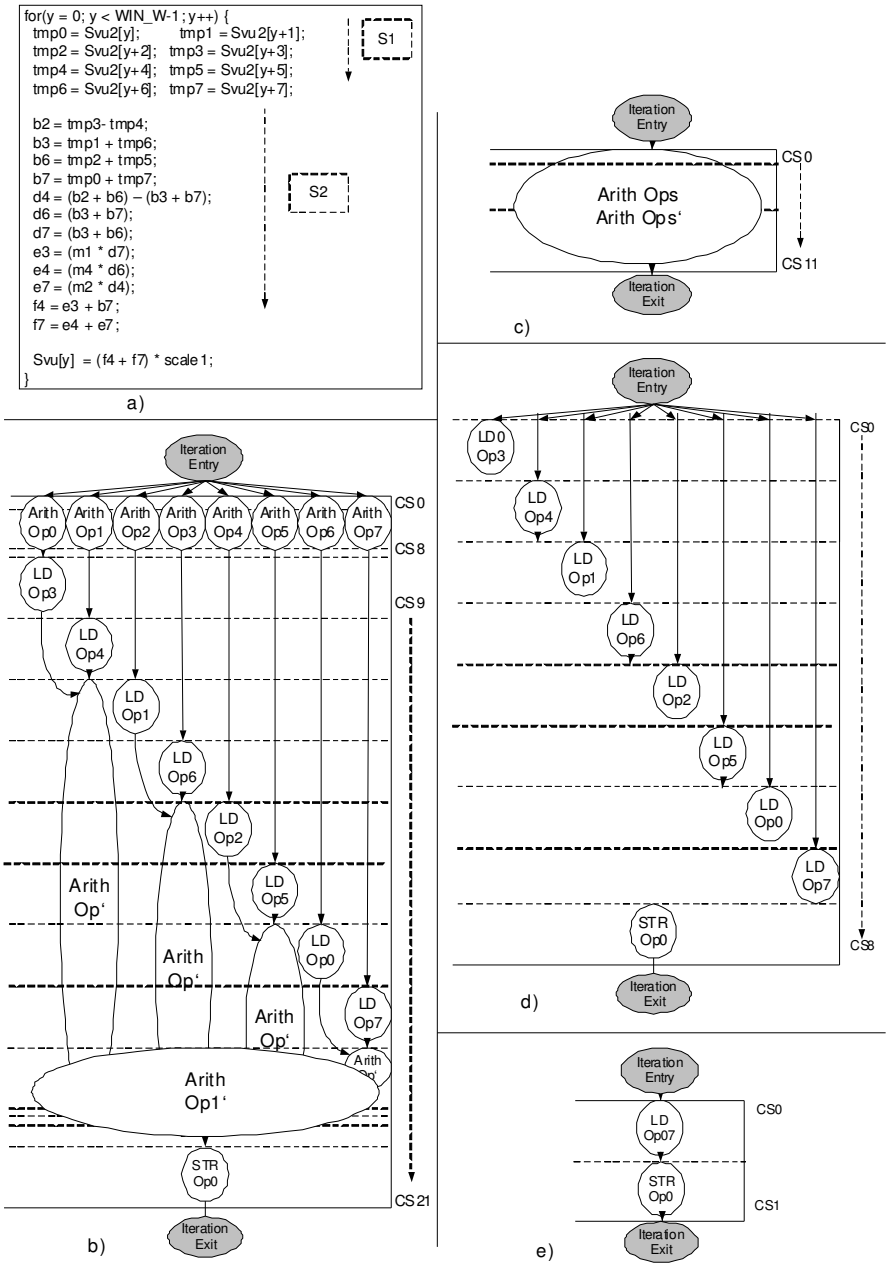


Figure 25. 2D DCT example. a) Loop body; b) Scheduled graph when ID=0, IAL=0, IC=0; c) Scheduled graph when ID=1, IAL=0, IC=0 d) Scheduled graph when ID=0, IAL=1, IC=1; e) Scheduled graph when ID=0, IAL=1, IC=1 and data reuse is applied

Notice that most of the “LD OPX” nodes have been removed, except the “LD OP7” node. A_{Reuse} is then equal to 31.5 for a new number of CS equals to 2. In this example, the AL operations seem to be the main bottleneck of the application with A_{OP} equal to 5.72 compared to an A_{Data} of 7. According to the reported metric values, the first aspect that should be targeted to accelerate the processing is the AL operations. Later in this paper, the metric exploration process is described along with a method to exploit these metrics in a design refinement process.

5.6 Experimental Results on Loop-oriented Metrics

The aim of the section is to show how the scheduler can be used to target a specific design architecture in order to explore different design aspects in terms of data, ALU or control operations. The section also tackles loop-oriented metrics to show how metrics can help to find promptly which design aspects limit or might improve the design acceleration. Some benchmarks were scheduled with LProf such as the JPEG image compression that comes from the CHStone project website [193]. We also profiled some home-written benchmarks, such as an ELA Deinterlacer [103], and a Turbo Decoder [181].

5.6.1 Design Constraints Impact

Leveraging design acceleration opportunities mainly depends on the design architecture in terms of processing capability. In the paper, the processing capabilities are specified using global design constraints such as the computation step delay, *steptime*, and the maximum number of concurrent operations, *max_conc_op*. Figure 26 illustrates how the global design constraints influence the acceleration of a loop extracted from a JPEG encoder. In the figure, s*teptime* and *max_conc_op* are both varied from 2 to 10. The global acceleration associated with each design style A_{Tot} is provided as a reference in the figure. As mentioned before, acceleration is a ratio of numbers of computation steps. This benchmark illustrates very well how increasing the number of concurrent operations

per CS (*max_conc_op*) can help to increase acceleration. Notice in Figure 26 that A_{Tot} for JPEG significantly improves as *max_conc_op* rises. For example, when *steptime* is set to 2, A_{Tot} varies from 3.71 to 13.

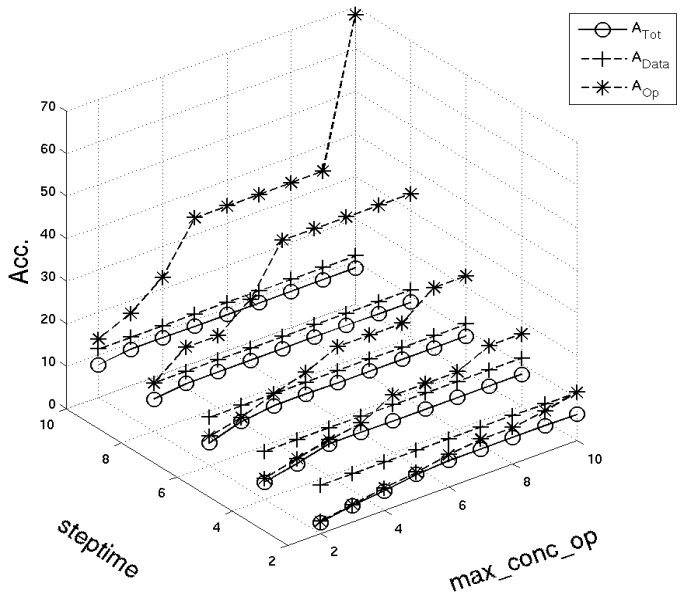


Figure 26. JPEG: Data access vs. ALU/Control-oriented metrics

Another way to reach better acceleration is to increase the computation step (CS) delay *steptime*. Notice in Figure 26 how A_{Tot} for JPEG significantly rises, even with low allowable parallelism. For example, when *max_conc_op* is equal to 2, A_{Tot} is equal to 1.19, 3.67, 4, 5.5, and 6.29 as *steptime* is raised from 2, 4, 6, 8, to 10. Overall, we notice that increasing *steptime* improves the number of scheduled operations per CS and, in the same way, it helps to leverage operation parallelization opportunities. Note that, in this benchmark, only one load and one store can be executed at a time, and thus A_{Data} stays constant even *max_conc_op* is increased.

5.6.2 ALU/Control-Oriented Metrics

LProf allows scheduling a design according to a specific design aspect. ALU/Control oriented metrics allow exploring the design in terms of AL and branch operations. That exploration aims to gauge the impact of those types of operations on loop acceleration opportunities. To illustrate ALU/Control-oriented metrics, a loop extracted from the

Turbo Decoder algorithm was scheduled with ID=1. Figure 27 shows the ALU/Control metrics of the scheduled designs. Notice that A_{Arith} does not significantly improve as much as A_{Branch} does when *max_conc_op* is increased or *steptime* is stretched. A_{Arith} reached its limit at 63.5 while the A_{Branch} maximum value is 158.5. Since A_{Op} combines these two metrics, A_{Op} follows the curve of the dominant aspect that is A_{Arith} in this loop. Thus, the metrics in Figure 27 reveal that the AL operations should be optimized first since A_{Arith} is always lower than A_{Branch}.

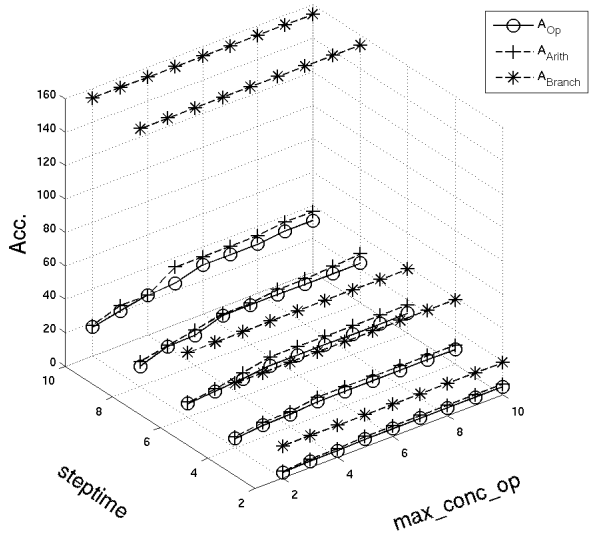


Figure 27. Turbo Decoder's ALU/Control-oriented metrics

5.6.3 Data-Oriented Metrics

The following illustration shows how the data-oriented metrics can be used to measure the impact of data operations on loop acceleration and whether data optimization techniques such as data reuse or I/O optimization can help to improve acceleration. One loop extracted from the ELA deinterlacer algorithm has been profiled. Some data reuse opportunities have been found in the XELA loop as illustrated in Figure 28. Notice a slight difference between A_{Data} and $A_{DataReuse}$. Also notice that better acceleration can be obtained when the I/O constraints are removed. In this case, A_{IO} improves as *max_conc_op* rises. Maximum A_{IO} is reached when the allowable parallelism *max_conc_op* is equal to 10. In fact, the XELA algorithm needs to load 10 inputs in

parallel. Adding I/O interfaces can be costly compared to data reuse which only implies the use of extra registers. Then, the designer has to take the appropriate decisions according to the targeted acceleration and design cost.

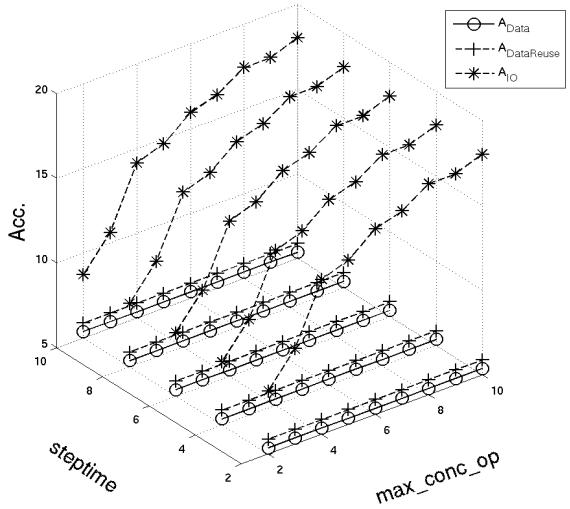


Figure 28. XELA: Data access oriented metrics

These illustrations showed how the loop-oriented metrics contribute to exploring design architecture acceleration opportunities. Each metric provided some insight on acceleration opportunities. They help identify a specific design aspect that might limit or improve the design acceleration. The metrics could be analyzed as part of a design space exploration process to obtain an architecture that can best leverage loop acceleration opportunities according to constraints.

5.7 Loop Acceleration Exploration

A design space exploration process is proposed in Figure 29. In this process, the first input is the operation properties described in Section 4. LProf uses these properties to compute the metric values. After profiling, the first metrics that should be analyzed are A_{Op} and A_{Data} to determine which aspect of the design should be improved first between data accesses and ALU/Control operations. The exploration process is iterative. For now, the exploration process is not automated as our main goal was to validate the metrics and

the loop acceleration exploration process. One of our future goals is to automate the whole exploration process.

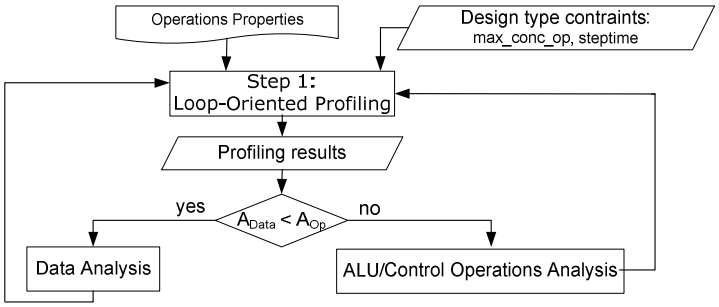


Figure 29. Overview of the loop-oriented metrics exploration flow

As a designer follows the steps of the process, the original code can be modified or design constraints may be changed. When such steps are taken, it is necessary to profile again and to re-iterate through the exploration process. If data memory accesses seem to limit acceleration ($A_{Data} < A_{Op}$), a detailed exploration should be done on data-oriented metrics. Figure 30 illustrates the data exploration process. The acceleration reachable with data reuse ($A_{DataReuse}$) should be compared to the acceleration reachable when I/Os are optimized (A_{IO}). When $A_{DataReuse}$ is higher than A_{IO}, it means that data reuse should be applied first in order to reach better acceleration. If $A_{DataReuse}$ is lower than A_{IO}, adding I/O interfaces should be considered. Applying I/O-interface optimizations consists in raising the number of concurrent loads or the number of concurrent stores or both. Data reuse can be done by replacing data-oriented operations with register-to-register move operations. The reusable data can be stored in specialized registers after their first load and later the register value is read as proposed in [118].

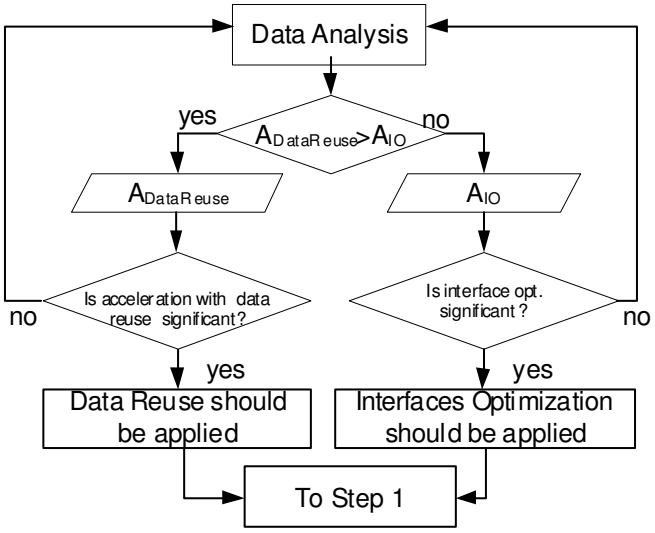


Figure 30. Data-oriented metrics exploration flow

The goal of the second branch of the metric exploration flow ($A_{Data} > A_{Op}$ in Figure 29) is to analyze whether the parallelism has been exploited effectively. When $A_{Data} = A_{Op}$, we arbitrarily start with an exploration of AL operations and control flow. Figure 31 shows the exploration process related to this second branch of the flow. First, it suggests a comparison between the acceleration associated with the branch statements (A_{Br}) and the one associated with the ALU (A_{Arith}). If A_{Branch} is lower than A_{Arith}, the branch statements should be analyzed first. Branch statements may prevent some parallelism opportunities by adding dependencies. Some transformations can be applied to the software code such as transforming control dependencies in data dependencies. We do not, at this point, automatically apply transformation techniques that can provide better acceleration.

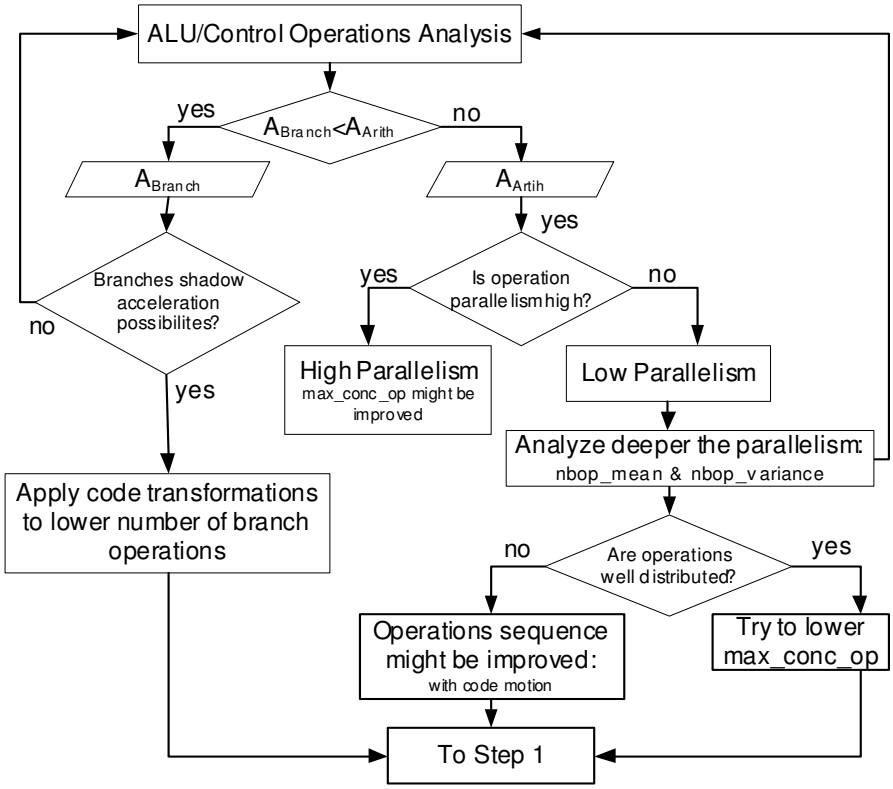


Figure 31. ALU/Control-oriented metrics exploration flow

When arithmetic and logic operations do not provide much acceleration opportunities ($A_{Arith} < A_{Branch}$ in Figure 29), their parallelism should be analyzed. Ideal parallelism is attained when overall DoP is equal to 1. If the DoP is equal to 1 or close to 1 (depending on the designer goal), the designer could consider increasing CS parallelization capability by raising the allowed maximum number of concurrent operations (*max_conc_op*). When the DoP is close to 1, the designer can also analyze the operation distribution to see if the parallelism is well exploited or not. Usually, when the DoP is not close to 1, raising *max_conc_op* may not increase acceleration much. A degree of parallelism (DoP) significantly less than 1 typically means that some operations or data dependencies limit acceleration. In that case, a more detailed exploration of the metrics may be needed. Indeed, the mean and variance of the number of operations per CS characterize how many operations are scheduled on average per CS and if they are well distributed in the

CS graph. High-level code optimization techniques such as loop unrolling and loop shifting may help spreading operation to improve DoP. These optimizations are normally available in compilers, thus they were not implemented in our framework.

5.8 Case Study: SAD Loop Acceleration Exploration

In order to show how the design space exploration process presented in the Loop Acceleration Exploration Section can lead to design an architecture that leverages loop acceleration opportunities, it was applied to the design of an ASIP dedicated to the SAD algorithm. The SAD algorithm (sum of absolute differences) computes vector motion for applications such as H.264. The sum of absolute difference (SAD) is defined for blocks of size N by N where a block located at position (x, y) in the current frame with the block displaced by (u, v) in the previous frame, is defined as:

$$SAD_{(x,\,y)}(u,v) = \sum_{i=0}^{N}\sum_{j=0}^{N}\left|F_t(x+i,y+j) - F_{t-1}(x+i+u,y+j+v)\right|$$

where F_t and F_{t-1} represent the pixel values in the current and previous frames respectively. The current block position is given by the position (x,y) of its top left pixel while (u,v) is the displacement vector of the previous block relative to position (x,y).In the rest of the section, the exploration phases are presented followed by resulting ASIP designs implementation.

5.8.1 Exploration Phases

As shown in Figure 29, Figure 30, and Figure 31, the loop exploration process is iterative. It is based on design constraints tuning and code transformations. The exploration was done in four phases. During each phase, some design options were explored. Table 6 details the designs properties and metrics. The reference design OD0 that is a pure software implementation was first profiled. The exploration phases are described as follows.

Phase A: Original code exploration (OD1, OD2, OD3, and OD4)

The exploration process starts with design OD1 that allows two concurrent operations per specialized-instruction. In OD1, the arithmetic and logic operations limit the acceleration ($A_{Op} < A_{Data}$, i.e. 1.5 < 4.5 and $A_{ALU} < A_{Branch}$, i.e. 2.25 < 4.5). Operations parallelism reveals that OD1's DoP has dropped to 0.56 and the mean number of operations is low with 1.13 and the variance is also low with 0.11. A low variance means that the operations are well distributed around the mean. To leverage the parallelism opportunities, *steptime* was increased from 1 to 2 which results in design OD2. The design has a better A_{Op} and A_{ALU} also rises. The parallelism is still medium in OD2. Some improvements were noticed but the variance is higher, meaning that operation distribution starts deteriorating. We tried to stretch *max_conc_op* and steptime in OD3 and OD4, respectively. According to our metrics, OD3 and OD4 provide no improvement.

Table 6. Designs Metrics

Phase	Designs	Constraints (steptime, max_conc_op, max_conc_load)	A_{Tot}	A_{Op}	A_{ALU}	A_{Branch}	A_{Data}	A_{IO}	A_{Reuse}	DoP	Mean	Variance
A	OD0[1]	(1,1,1)	1	1.28	1.8	4.5	4.5	4.5	4.5	1	1	0
	OD1[1]	(1,2,1)	1.12	1.5	2.25	4.5	4.5	9	4.5	0.56	1.12	0.11
	OD2[1]	(2,2,1)	1.38	2	3.6	4.5	4.5	9	4.5	0.5	1.38	0.69
	OD3[1]	(2,3,1)	1.38	2	3.6	4.5	4.5	9	4.5	0.33	1.38	0.69
	OD4[1]	(3,2,1)	1.38	2	3.6	4.5	4.5	9	4.5	0.43	1.38	0.69
B	AD5[1]	(2,2,1)	1.8	3	4.5	9	4.5	9	4.5	0.5	1.4	0.84
C	SD6[1]	(2,2,1)	2	3	4.5	9	4.5	9	4.5	0.56	1.56	0.91
	SD7[1]	(2,3,1)	2	3.6	6	9	4.5	9	4.5	0.37	1.56	0.91
D	UD8[1]	(2,2,1)	2.49	4.99	6.98	17.46	4.37	8.73	4.37	0.63	1.78	1.02
	UD9[1]	(2,2,2)	2.91	4.99	6.98	17.46	8.73	8.73	8.73	0.72	2.08	0.9
	UD10[1]	(2,3,2)	3.49	5.82	8.73	17.46	8.73	8.73	8.73	0.58	2.5	2.65
	UD11[1]	(3,3,2)	3.88	6.98	11.64	17.46	8.73	8.73	8.73	0.55	2.77	5.72
	UD12[2]	(3,3,2)	4.99	6.98	11.64	17.46	17.46	17.46	17.46	0.52	3	7.14

[1]: load operation delay is equal to 2 steptime. **[2]: load operation delay is equal to 1 steptime.**

Phase B: Branch operations enchancement (AD5)

At the end of Phase A, the AL operations are still limiting the acceleration. Thus, we devised a code transformation that assumes there is an absolute difference operator in the processor instruction set. In the implementation, the absolute difference computation that required a branch operation is executed by a specialized-instruction thus removing the

said branch operation. That is how AD5 was designed. A_{OP} has improved from 2 to 3. A_{Branch} rises from 4.5 to 9. A_{ALU} has slightly improved from 3.6 to 4.5. We can also notice that overall A_{Tot} has improved from 1.38 to 1.8. Unfortunately, the DoP is still low.

Phase C: Loop shifting (SD6, SD7)

Pass A and Pass B produced designs having low DoP, which implies that resources are not efficiently used. Thus, we decided to apply another transformation which is loop shifting. The overall acceleration A_{Tot} is better in designs with shifted operations, i.e. SD6 and SD7, but A_{OP} improvement is weak.

Phase D: Loop unrolling (UD8, UD9, UD10, UD11, UD12)

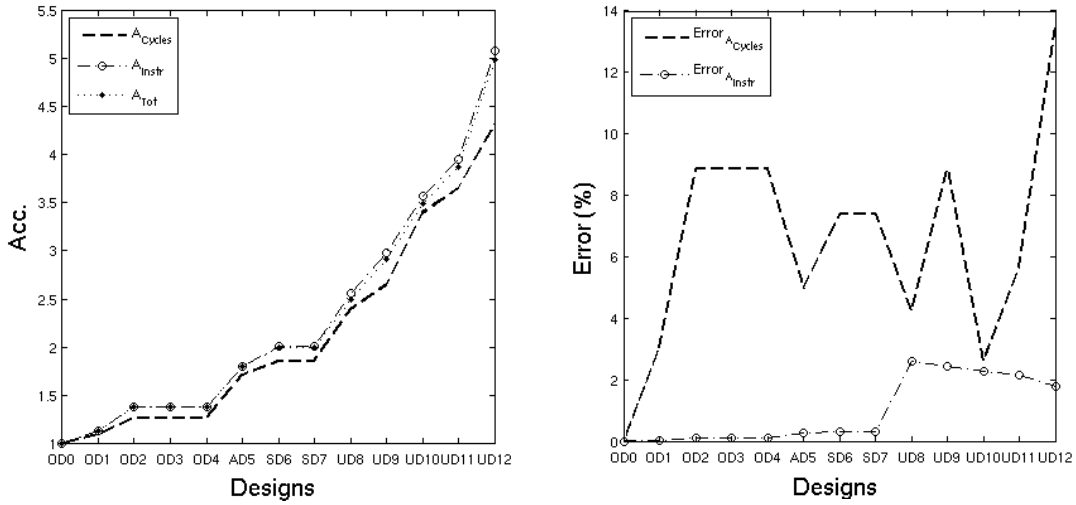
A last transformation, loop unrolling, was applied to try leveraging any loop parallelization opportunites. The first unrolled design UD8 reveals a shift in the limiting factor. Now, it is the data that limits acceleration, with A_{Data} equals to 4.37 and A_{Op} to 4.99. Thus, in UD9, we increased the number of concurrent loads to 2. UD9 obtains the best DoP with 0.72 and its variance is slightly better than for UD8. Almost all resources were used except in computation step containing branch operations. It is expected that the ideal DoP (=1) will never be reached in practice. Nevertheless, the designs with unrolled iterations have better performance. Their DoPs are higher and more operations are scheduled per CS. On the other hand, we can notice a high variance for UD12, which means that there is a significant discrepancy in operation distribution, with some CS almost empty and others fully occupied.

5.8.2 ASIP implementation

Loop-oriented metrics are computed from scheduling results issued by LProf. Thus, it is relevant to validate our results to gauge any discrepancies between the static analysis and the results of implemented designs. Consequently, specialized-instructions (SIs) corresponding to all the considered design options were specified in a hardware language.

Also, the software code that uses these SIs was simulated to collect the number of cycles related to the profiled application. We used the Tensilica [168] processor family to collect these simulation results. The computation steps that were scheduled in our tool were transformed in SIs written in the TIE language. This language is a proprietary Verilog-like language that must be used to describe specialized-instructions (SI) through their behavior and to declare their embedded user-defined registers.

The ASIP corresponding to the explored designs were simulated. Figure 32 summarizes the overall accelerations reached by each design. Figure 32.a presents the acceleration in terms of number of cycles (A_{Cycles}), in terms of executed instructions A_{Instr} and the estimated total acceleration A_{Tot} (in terms of execution time).



a) **b)**

Figure 32. Simulation Results: a) Accelerations, b) Errors between estimated and simulated accelerations

Notice that the A_{Instr} and A_{Tot} curves are very similar, but there are some significant discrepancies between A_{Cycles} and A_{Tot}. These discrepancies between the estimated and the simulated accelerations are expressed as relative errors in Figure 32.b. The error in terms of number of executed instructions $Error_{AInstr}$ is low with a maximum of 2.59% for UD8. The error is due to the addition of some instructions to initialize registers, before the beginning of the loop for example. The error between A_{Cycles} and A_{Tot}, $Error_{ACycles}$,

varies between 2.61% and 13.49% for designs OD1 to UD12. We should point out that the number of cycles provided by Tensilica instruction set simulator is the sum of cycles required by the executed instructions, plus the number of cycles related to the taken branches, added to the number of cycles related to pipeline interlocks. Our estimates do not consider delays due to taken branches and pipeline interlocks. After analysis of the simulations logs, we notice that the significant errors are due to the number of cycles related to these parameters. In order to obtain more accurate estimates, these parameters should be taken into account during the acceleration estimation process. However, this can only be done by combining static and dynamic analysis.

5.9 Conclusion

In this paper, we tackled loop acceleration exploration for application-specific designs. We presented a framework in which ASIP designs can be specified using design constraints such as the allowable parallelism and the instruction execution time. From a C-based specification, loop acceleration associated with a specific design can be explored using new metrics. The proposed metrics help explore loop acceleration from different standpoints: data operations, arithmetic and logic operations, or branch operations. We implemented a profiler-scheduler that allows computing the potential acceleration according to design constraints and the explored standpoints. Some loops extracted from benchmarks such as Turbo Decoder, JPEG, etc, were profiled to show how loop-oriented metrics gives an insight on aspects that limit or leverage acceleration opportunities. A loop acceleration exploration process was also presented. That process leads to obtain the best design that will leverage the loop acceleration opportunities. To validate the proposed design method, the loop acceleration exploration process was applied to the SAD algorithm. The exploration shows how loop acceleration can be significantly improved when the design capabilities are well managed through judiciously relaxed design constraints and code transformation application. Also, it was shown that, in this case (the SAD algorithm), the worst error between the estimated and the simulated

accelerations was about 13.5%. Several steps of the proposed exploration process could be automated as part of future work. More metrics could also be proposed to characterize more architectural optimization possibilities such as pipelining.

CHAPITRE 6

LOOP NEST PERFORMANCE ESTIMATION FOR APPLICATION-SPECIFIC ARCHITECTURE DESIGN

M. Mbaye, N. Bélanger, *Member, IEEE*, Y. Savaria, *Fellow, IEEE*, S. Pierre, *Member, IEEE*

Abstract— Good candidates for acceleration with an application-specific design are loop statements extracted from a C-based specification. During design space exploration, designers need to know the potential for acceleration of various loops in order to target reasonable performance. Thus, in this paper, nested-loops execution time is estimated according to fine- and coarse-grain parallelization opportunities. The estimated performance considers design styles such as application-specific instruction-set processors and ASICs. A design style is specified by adjusting design constraints such as the allowable parallelism at the instruction level, I/O constraints, etc. Our estimation approach is based on the polyhedral model of nested loops. It is applied on various benchmarks for which it is shown that I/O constraints may limit parallelization opportunites and reduce the acceleration potential. The main contribution of the paper is the proposed execution time estimation technique for multidimensional loops. The technique is at the core of a design space exploration process. Execution time of some loops extracted from the JPEG and MVM algorithms were estimated. The reported results show that performance estimates can be computed very quickly with a reasonnable accuracy.

***Index Terms*—Design space exploration, Application-specific architecture, Loop parallelism, Performance estimation, Polyhedral model.**

6.1 INTRODUCTION

Currently, the gap between designer productivity and market demands causes many projects to complete behind schedule in order to ensure that the quality requirements are met. That leads industry to constantly seek new tools or methodologies that will help

reduce the time to market. Design time strongly depends on the system specifications and on the methodologies applied during the design process flow. Recently, “C-based design” gained a lot of attention. These design methods rely on an executable specification written in C or a C-oriented language. The specification can target software, application-specific instruction-set processor (ASIP) [63], or high-level synthesized ASIC design [66]. The main difference between these last two types of design is their flexibility. ASIP design is partially software, therefore it offers more flexibility for future changes while ASIC design is 100% hardware, which typically means that it has no flexibility, unless the ASIC embeds one or more processor. ASIP and ASIC designs share the same goal that is to leverage the acceleration opportunities in the C specification. In C code, good candidates for acceleration are loop segments of code. Many loop acceleration techniques have been proposed in the literature [66, 118, 136] targeting ASIP or ASIC design, but there is always a lingering question, which is how good the obtained acceleration is. A reference is needed to gauge acceleration techniques efficiency. Currently, researchers [66, 118] compare their techniques to previous work in the same domain or to the plain specification. On the other hand, it would be more accurate to compare a technique efficiency to the loop maximum potential for acceleration. Having the loop maximum potential for acceleration would also help designers by providing them with a stopping criterion when they reach the target performance or when they are sufficiently close to the maximum feasible performance with the target architecture. Having such support would allow them to avoid wasting time trying to reach an unreasonable target. Thus, the main contribution of this paper is the estimation of loop-nest execution time that considers fine- and coarse-grain parallelism opportunities.

The work presented in this paper is part of a project that aims at loop-acceleration exploration with application-specific architectures. As a first step, we proposed a framework in which different kinds of application-specific architectures can be specified [114], mainly in terms of number of concurrent operations and in terms of I/O constraints. The framework includes a loop profiler/scheduler, LProf. This profiler

gathers newly proposed loop-oriented metrics that provide us with a good insight of the tradeoff between computations and memory accesses. The loop-oriented metrics are based on acceleration opportunities at the iteration level. The profiler only considers the instruction-level parallelism opportunities of loop iterations. The next step of the project, which is covered in this paper, is to estimate the loop execution time that considers coarse-grain parallelization opportunities.

The main contribution of this paper is the estimation of multidimensional loop execution time. The estimation considers fine-grain parallelism allowed by the targeted design style, and coarse grain parallelism opportunities that depend on loop carried dependencies and I/O oriented dependencies. Thus, the estimated execution time is used to compare and to accurately gauge the efficiency of potential designs. The estimation is performed in a framework, LProf, that allows design space exploration of different design styles [114]. Two phases of scheduling are performed to estimate the execution time of nested loops. The first one leverages the fine-grain parallelism opportunities, while the second phase considers the coarse-grain parallelism opportunities and both take the targeted design-style constraints into account.

The rest of the paper is organized as follows. In Section 2, a review of the relevant literature is presented. Some definitions about the polyhedral model and affine recurrent equations systems are given in Section 3. In Section 4, the profiler-scheduler tool LProf, which is used in the execution time estimation process, is briefly introduced. In Section 5, the loop execution-time estimation process is explained in details. Section 6 gathers experimental results. Finally, conclusions are drawn in Section 7.

6.2 Relevant Literature

Early work in ASIP design [33, 67, 75] was centered on adding functional units (FU) or coprocessors, such as multiplier accumulators MACs or floating-point coprocessors, to a processor core. These units can be very efficient in accelerating some application types. Meanwhile, another research trend emphasizes application specific solutions based on processor instruction-set extension. With technologies such as Tensilica's LX2 [168] and Altera's NIOS II [4] processors, it is easier to test and verify the instruction-set customization algorithm. Many such algorithms were proposed in recent years. Galuzzi et al. [60] surveyed the instruction-set extension problem and they divide instructions in two categories: Multiple-Input, Multiple-Output (MIMO) [8, 32] and Multiple-Input, Single-Output (MISO) [40, 61, 140]. These two techniques consist of clustering operations in a single instruction that can have multiple input operands. The main difference between these types of instruction is the number of output operands: a MISO instruction supports only one output while a MIMO instruction can have more than one output. As mentioned by Atasu et al. [8], most of the algorithms proposed for instruction-set automatic generation, such as in [39, 40, 192] consider some hard constraints on the number of input and output operands for the customized instructions. These hard constraints are due to processor register file properties. On the other hand, in processors such as the Tensilica family, a customized instruction can have significantly more (implicit) input or output operands. That is why Atasu et al. [8] proposed an efficient algorithm based on convex subgraph enumeration. Their technique tackles maximal convex subgraphs of a given data flow graph, which is geared towards scheduling more specifically arithmetic and logical operations both in parallel and in sequence within a single instruction. In [118], we proposed an ASIP design methodology that is also based on maximal convex subgraph enumeration of critical loops. Our methodology also focuses on loop acceleration based on data reuse with the use of application-specific registers. Overall, the main goal of the custom-instruction generation techniques is to leverage the application parallelization opportunities by executing as many operations as possible in

the same instruction. We propose a novel approach that is based on loop optimization during ASIP design. We show that loop optimizations (such as loop unrolling, or data-reuse that can be leveraged with the use of custom registers) contribute to obtain significant acceleration. Data optimization techniques are well documented in the software field [130] but they are not widely used during ASIP or high-level synthesis oriented designs (HLS).

Research on HLS started in the seventies. Early work involved scheduling heuristics for data-flow designs as described in [65]. Then, research focused on operation parallelization with techniques such as integer linear programming (ILP) [95, 185]. Pipelining was also widely applied to improve circuit performance [29, 133]. While datapath synthesis was gaining maturity, researchers noticed the impact of control flow on HLS quality. Control flow can prevent leveraging parallelization opportunities. This led Huang et al. [73] to propose a tree-based algorithm for control dominated circuits, for example. Radijojevic and Brewer [146] proposed a symbolic formulation of control-dependent resource-constrained scheduling that promotes speculative code motion through control path.

Li and Gupta [99] also proposed a technique that performs pre-synthesis transformation of the specification to improve operation parallelism. According to Gupta and Brewer [65], the first role of an HLS tool is to extract the parallelism inherent to the behavior described at high level. The goal of the SPARK tool [66] is to tackle fine-grain parallelization. The main strength of this tool is the application of code motion algorithms at the instruction level. The code motion algorithms leverage the parallelization opportunities. This tool also allows applying loop unrolling or loop shifting for code compaction but the user must specify which loop should be transformed and must set the unrolling factor.

Applying loop transformations, as proposed by Kurra et al. [92], Plesco et al. [136], or Derrien et al. [48], is an efficient way to leverage parallelization opportunities. It

should be mentioned, as stressed by Kurra et al. [92], that loop unrolling can lead to unexpected results, such as fast-growing latency and complexity of a finite state machine (FSM) control unit. Recently, Plesco et al. [136] proposed to use more powerful loop transformations with the WraptIt tool [15] prior to using SPARK. They show that more instruction-level parallelism opportunities can be unraveled with the application of loop transformation techniques. Plesco et al. use techniques that stem from the polyhedral representation of nested loops. Derrien et al. [48] also proposed to perform loop parallelization before HLS operations. Their work is implemented in the MMalpha compiler that is a powerful loop parallelizer, which is also based on the polyhedral model. We should also mention the availability of commercial HLS tools such as the Celoxica's DK Suite [27], and Mentor Graphics Corp.'s CatapultC [120]. Despite all the techniques and tools that have been proposed, HLS tools need more design space exploration features. For example, features such as enhancement of arithmetic operations, automation of memory allocation will lead to more optimized solutions [175].

Darte et al. have surveyed scheduling and automatic parallelization techniques in [45]. They mention some parallelization detection techniques such as the Allen-Kennedy technique [3] that is based on parallelization between loop levels. Lamport's method, also called the hyperplane method [94], is a known method that transforms nested loops to leverage parallelization opportunities. Lamport's method has been extended by Wolf and Lam [186] to produce permutable loops that are interchanged to maxime parallelization opportunities. Lamport's method and Wolf and Lam's algorithm tackle coarse-grain parallelization, On the other hand, Darte and Vivien have proposed, in [46], an algorithm that tackles fine-grain parallelization and their technique can be extended to extract coarse-grain parallelization. These techniques are based on an approximation of loop dependencies, i.e., with this technique, some depedencies might not be catched. Feautrier has proposed a parallelization technique based on exact dependeny analysis [52, 53]. His technique schedules loop statements according to their iteration domain, which is described as a polyhedron. Loop parallelization with a polytope model, also called

polyhedral model, was pioneered by Leugauer et al. [96] and Feautrier et al. [50]. As mentionned before, loop parallelization usually implies loop transformations such as loop tiling and loop interchange. These transformations usually improve the execution speed by executing in parallel groups of iteration computations and by reducing the communication costs.

Performance can be calculated in three ways: static analysis, simulation, and execution in the target environment. Execution in the target environment imposes that the exploration process be completed before proceeding. Estimation by simulation can be accurate, but it can require a lot of ressources and time. Static analysis is a good means to have an insight on the execution time at the lowest cost in terms of time and effort. The performance estimation technique presented in this paper is based on a static analysis process and it provides the worst case execution time – WCET [107]. A lot of research has tackled WCET estimation, but the proposed techniques usually fail to estimate the execution time of nested loops [107].

Some research focused on the detection of non-executable datapath to remove them from the estimation process [97, 131, 169]. Other research has tackled the design properties (such as the pipeline or the cache) overhead in the estimated execution time [78, 100, 190]. Unfortunately, the avalaible techniques do not consider loop parallelization opportunities, because many of them target single-instruction single-data processor architectures.

Few researchers targeted loop performance estimation, but we can mention the work of Nakanishi et al. [125]. They proposed a loop execution time estimation algorithm based on integer linear programming that avoids fully unrolling loops. Their algorithm is similar to our solution but they do not consider the loop data dependence iteration space. Thus, their estimations could consider some data dependencies that do not exist because data access iteration spaces are not in the same domain. On the other hand, their technique can consider iterations that do not have to be executed. Roychoudhury et al.

[154] also proposed a loop performance estimation technique based on detecting feasible and infeasible loop control paths, but their loop performance estimation techniques can be costly, as they do not take the iteration space and design constraints into account. It should be mentioned that these techniques estimate loop execution time for regular processor. Their techniques do not support loop execution time estimation for ASIP as does our technique.

In summary, many techniques have been proposed to accelerate loops but their efficiency depends on the acceleration potential of a loop. Unfortunately, many loop transformation and parallelizing techniques do not take the targeted architecture into account. It is well known that executing loops with a dedicated hardware design is an efficient solution to accelerate applications. However, the success of loop transformations or design options depends on the loop data dependencies that can unravel parallelization opportunities. It also depends on the targeted design-style architectural constraints. Very often, ASIP and ASIC design styles have the same goal, which is to leverage the fine and coarse grain parallelization opportunities of the applications. Design constraints, such as the allowable parallelism, distinguish those design styles.

6.3 SARE: System of Affine Recurrent Equations

This section presents the concept of system of affine recurrent equations (SARE) which is one of the basic concepts used in our scheduling approach. This section also introduces our scheduling approach, which allows finding the timing function associated with a SARE.

6.3.1 SARE

The concept of SARE relies on the polyhedral model [104, 110], which is a mathematical representation that describes nested loops statements. This representation allows capturing the statement values as a function of the nested-loops index domains Is.

A system of affine recurrent equations allows capturing the data dependencies of the program variables as a function of the nested loops indices by taking the iteration domains into account. A SARE model can be described as a set of equations [149], each of the form:

$$(\forall x \in I_S : v[f(x)] = F_v(w[g[x],\ldots)) \quad \textbf{(1)}$$

Each equation defines the index domain *Is* for which the dependence takes place. *v* and *w* are multidimensional array variables. This equation expresses that variable *v* at iteration *f(x)* is equal to a function F_v that has as argument the value of variable *w* at iteration *g(x)*. *f(x)* and *g(x)* are the access functions, also named index mapping functions of *v* and *w*. The access function is the linear function that computes the accessed memory location from the loop indices.

6.3.2 Scheduling

Feautrier [52, 53] was one of the first to propose a scheduling algorithm that allows expressing the execution time of each loop-statement. The execution time is expressed as a polynomial function according to the indices of the nested loops. The aim of Feautrier's scheduling algorithm is to find a timing function for each equation of the affine recurrent system. His algorithm uses a generalized dependence graph *GDG<V, E, D, R>*. This graph captures the dependencies between different statements of a program. A schedule for the *GDG* is a positive function such that, for every edge of *E* that has a source *R* and a destination *S*,

$$x \in D_R, y \in D_S, < x, y > \in \quad \Rightarrow \theta(S, y) \geq \theta(R, x) + 1$$

This expression means that statement *S* at iteration *y* cannot start before the end of statement *R* at iteration *x*. The schedule of *R* is a function of the time at which the

statement *R* at iteration *x* will start. Then, the scheduling function of *S* always outputs a value higher than the scheduling function of *R* at the iteration of the scheduling function plus 1, where 1 stands for the duration of any statement (S in the equation). By default, the duration is set to 1, because SAREs were used to target processors for which one statement is executed in one cycle.

The concept of GDG has been extended to define the dependencies between different array variables of a program [110]. In this paper, we propose to generate an iteration dependence graph, IDG (also called reduced dependence graph) that will be used to generate a SARE, which will be scheduled to find the execution time of a loop nest according to some design constraints.

6.4 Loop-oriented Profiler/Scheduler: LProf

The goal of this tool is to help determine loop-acceleration opportunities in an application. It is a profiler-scheduler that schedules loop-iteration operations according to design constraints. The scheduling outcome is profiled to compute loop-oriented metrics that are described in [117]. LProf supports 3 types of target architectures: embedded general-purpose processors (EGPP) [70], ASIPs, and ASICs. A typical EGPP executes only one atomic instruction at a time. On the other hand, ASIPs and ASICs execute an atomic instruction that includes many atomic operations. In the case of an ASIP, atomic operations can be executed in parallel or in sequence, as long as the atomic instructions critical path meets the processor clock frequency constraint [140].

LProf generates a computation graph that will be used to coarsely estimate the number of computation steps required to execute loop iterations. The number of computation steps (CS) is the number of instructions associated with each type of design. The operations scheduled in a CS are called atomic operations and are assigned to one instruction. The number of CS may also be described as the number of cycles required to execute an iteration of the considered loop.

Our scheduler takes as input the design constraints that are listed in Table 7. These design constraints are used to describe the targeted design style as we will see next.

Table 7. List of design constraints to the scheduler

Name	Description
Nb. of operator inputs	Number of input operands.
Nb. of operator outputs	Number of output operands.
Operator delay	Operator execution time.
I/O constraints	Number of concurrent load, concurrent store, and concurrent load and store operations.
steptime	Time allocated to each computation step.
max_conc_op	Maximum number of concurrent operations in a CS.

A CS is characterized by the number of time slots, called *steptime*, that it contains. This parameter can be described as the maximum instruction execution delay. In general, the number of sequential operations will depend on *steptime*, on the operator delays, and on data dependencies. During scheduling, the sum of the delays of the sequential operations in one CS must be less than or equal to *steptime*. Thus, stretching *steptime* allows executing more sequential operations in one CS (at the expense of a lower clock frequency).

LProf was built using two compiler frameworks: SUIF [161] and Machsuif [106]. SUIF creates an intermediate representation (IR) that facilitates coarse-grain analysis and transformation of software code. Machsuif uses an IR allowing fine-grain analysis and transformation. Machsuif takes as input a SUIF IR.

6.5 Loop Execution-Time Estimation

In this section, we describe the steps of the loop execution-time estimation process. As mentioned before, the estimation process is based on a mathematical representation of nested loops. Thus, after loops extraction (Step 1) from the C code, as illustrated in Figure 33, the polyhedral model of each loop nest is generated. Then, from the tagged CDFG (created during the scheduling of the operations belonging to the iteration) and from the polyhedral model, an iteration dependence graph (IDG) is generated (Step 3).

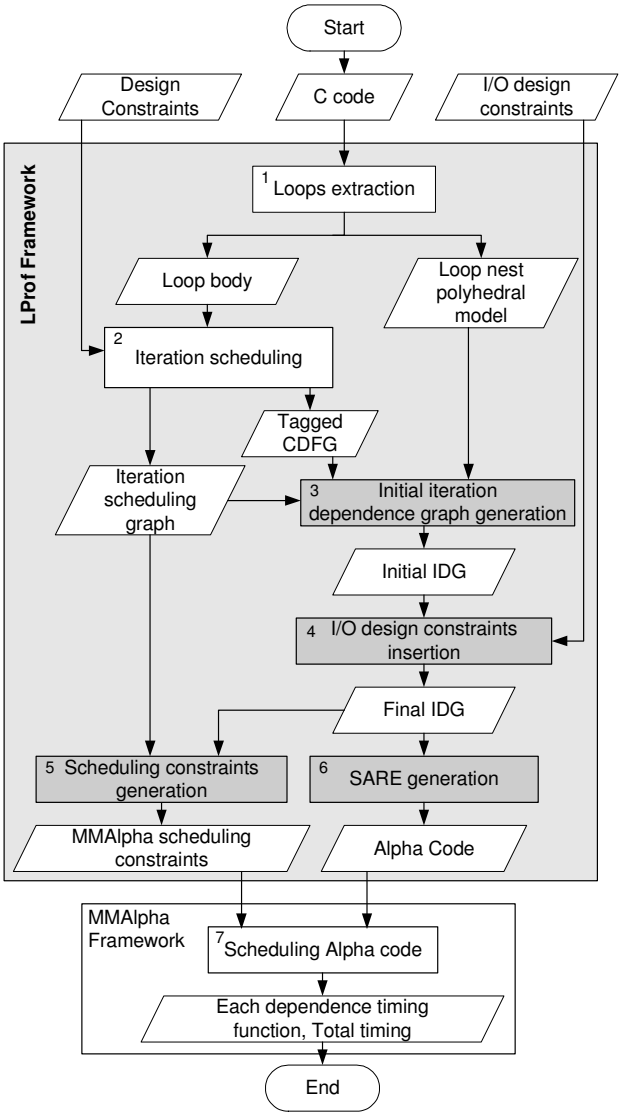


Figure 33. Loop execution time estimation process

This graph tracks I/O operations and data dependencies between iterations according to a data access expression. Later, some design constraints are inserted in the IDG (Step 4). This step contributes to include the I/O design constraints associated with the targeted design type. To implement the execution time estimation function, we chose the MMAlpha framework [144] in which a SARE can be scheduled. Thus, from the final IDG, the SARE is generated in the form of Alpha code (Step 6). The MMAlpha scheduling constraints are also generated (Step 5). These scheduling constraints are

delays associated with each memory access and they are computed using the scheduling results yielded by LProf in Step 2. Then, from the MMAlpha scheduling constraints and from the generated Alpha code, the execution time is computed in the MMAlpha framework (Step 7). The execution time function associated with each vertex of the IDG is provided by this framework and the total execution time of the loop is also estimated. In the rest of this section, the highlighted steps of the process in Figure 33 are described.

6.5.1 Initial Iteration Dependence Graph Generation (step 3)

This step consists on the automatic generation of a reduced dependence graph G = *(V,E,w,TSV,TSE,D)* that was slightly modified with regard to the original graph proposed in [55].

A vertex in *V(G)* represents a variable of the loop body or a local variable that is added during the I/O insertion (as explained below). The edges *E(G)* model the dependency between two variables (u, v), $u, v \in V$ such that $e = (u, v) \in E$. Each edge is weighted by a dependence distance vector *w*. A dependence distance vector $w(e) \in \mathrm{N}$ expresses the fact that the operation (*u,k*) (the instance of the memory access to variable *u* at the iteration *k)* must be completed before the execution of the operation (*v, k+w(e)*) (the instance of the memory access to variable *v* at the iteration k+w(e)). *TSV(u)* is defined as the time stamp at which *u* is available. *TSE(e,u)* is defined as the start time at which the variable *u* can be accessed for dependence *e*. Time stamp *TSV* is extracted from the scheduling of iteration operations. This time stamp is used during the insertion of I/O-oriented dependencies in the IDG that will be presented in the next section. *D(e,u)* is defined as the delay for the execution of the memory access of variable *u* in relation to edge *e*. These time stamps (*TSV, TSE, and D*) were added to the original IDG. These time stamps will be used to estimate the loop-nest execution time in step 7 of Figure 33. During the initial graph generation, the vertice time stamps *TSV(v)* are set to 0 and the edge time stamps *TSE(e)* are set according to the processed node time stamp from the scheduling results of LProf.

As an illustration, the simplified C code in Figure 34(a) matches the IDG drawn in Figure 34(c). The C code has been extracted from the Refpass algorithm of the JPEG2000 encoder from [166]. There is a three-dimension loop nest that has dependencies on indices *k, i,* and *j*. The iteration domain of the statements is presented in Figure 34(b). The loop nest manipulates three arrays: *flagspi*, *datai*, and *flagspo*. Notice that scalar variables *flag* and *datap* have been added to the graph since no direct path to a store node successor was found for statements S3, S4, and S5 in Figure 34(a). In Figure 34(c), one edge *e0* has been added between vertices *flagspi* and *flag*. This edge expresses the dependencies found in statements S3 and S4. The dependency stated in statement S7 is represented by edge *e1* that links *flagspi* to *flagspo*. An edge *e2* has also been added between vertices *datai* to *datap*. This edge shows the dependency of statement S5.

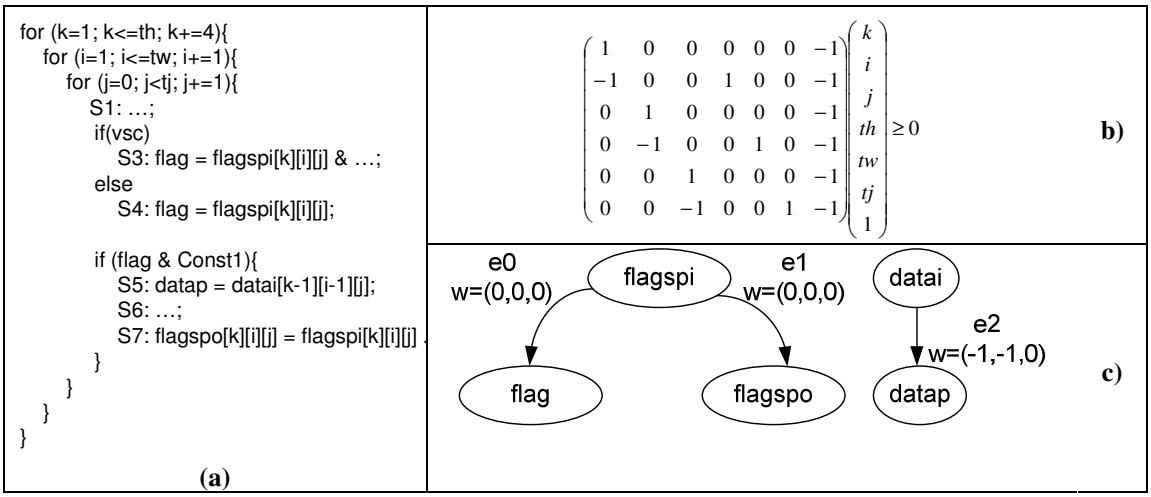


Figure 34. Example of IDG: a) Simplified C code; b) Domain of dependencies; c) Generated IDG

6.5.2 I/O Design Constraints Insertion (step 4)

The insertion of I/O constraints consists of introducing design-oriented dependencies in the initial IDG. As listed in the fourth row of

Table 7, three types of I/O constraints are considered: the maximum number of concurrent loads, *max_conc_load*, the maximum number of concurrent stores, *max_conc_store,* and the maximum number of concurrent loads and stores, *max_conc_loadstore*. Let us first explain how load operations are analyzed. Algorithm 1

explains the insertion of load design constraints. The goal of this algorithm is to add new vertices that will represent the load accesses performed in the loop iterations, and to create edges between the new vertices for which their dependence distance vector will be computed according to *max_conc_load*. The first step of the algorithm is to sort edges *E(G)* in terms of delay time and start time in list *El* (line 1 of Algorithm 1). The sorting prioritizes the load-oriented vertex processing in preparation for the distance vectors updating phase. Each edge *e(u,v)* is processed to track variable *u* that is being loaded. If a load-related edge is found, then a new vertex *v'* is added in *V(G)* to represent the load access in the IDG. The edge *e(u,v)* is then replaced by two edges *e(u,v')* and and *e(v', v)*. The weight of the new edge is also computed (line 12). The time stamp of the edges and their delay are extracted from the iteration scheduling graph F_{Tsched}.

To express design-oriented dependencies, edges are drawn between new vertices (lines 19 to 30 of Algorithm 1). Consider two new vertices *u'* and *w'* that state the loading of variable *u* and *w*, respectively. Because of the load constraints, *u* and *w* cannot be concurrently accessed, then two edges *e1=e(u',w')* and *e2=e(w',u')* are added and their weight is set according to *u'* and *w'* time stamps.

At this point, only one memory operation can be performed at a time. To allow concurrent load operations, the design-oriented edges must be updated according to the number of maximum concurrent load *max_conc_load*, which must be greater than or equal to 1. They can be considered as tokens that will be distributed to each load-oriented vertex. For each token, the vertex is allowed to release the dependence distance of its outgoing edges. Thus, larger distances mean that more accesses can be executed in parallel (this can be viewed as an increased flexibility). That is expressed in Algorithm 1 by decrementing (to increase its negative value) the relevant edges weight (lines 31 to 39).

The insertion of the other constraint types (i.e. *max_conc_store* and *max_conc_loadstore*) requires almost the same manipulations as Algorithm 1 does. For

the insertion of store design constraints, new vertices are added between vertices that are stored and their predecessors. For the load-store constraints type, vertices that are loaded or stored are manipulated to create intermediate vertices between the data dependencies.

Algorithm 1: add LOAD IO constraints in IDG (G, F_{TSched} , MAX_CONC_LOAD)

```
  /*insertion of load constraints dependency in G*/
1:    El←sort e(u,v) ∈ G(E) according to their delay d(e)
  /*insert design-oriented vertices and edges*/
2:    for all e(u,v) ∈ El do
3:       if u is an array then
4:          v'←Create a new vertex in G(V)
5:          Mark v' as a load vertex and a load-store vertex
6:          Mark u as a cloned vertex
7:          TS(v') = TSE(e,u) (from F_TSched)
8:          D(v') = D(u), D(u)=0 (from F_TSched)
9:          /*Transform e such as*/
10:         e_u→v'←Create edge (u,v')
11:         e_v'→v←Create edge (v', v)
12:         Compute w(e_u→v') and w(e_v'→v)
13:         Remove e from G(E)
14:         add v'' in A
15:      else
16:         v' = u
17:      end if
18:   end for
   /*Create design-oriented dependencies'*/
19:   for all v' in A do
20:      for all v'' in A do
21:         e←Create edge (v', v'')
22:         if (TS(v') <=TS(v'') then
23:            w(e) = 0
24:         else
25:            w(e) = -1
26:         end if
27:      end for
28:      e←Create edge (v', v')
29:      w(e) = -1
30:   end for
   /*Update distance vector according to max_conc_load*/
31:   counter = 1
32:   while counter < MAX_CONC_LOAD do
33:      for all v' in A do
34:         for each edge e_v',v'' do
35:            Decrement w(e_v',v'')← w(e_v',v'')-1
36:         end for
37:         Increment counter←counter+1
38:      end for
39:   end while
```

The initial IDG presented in Figure 34(c) has been modified to integrate load design constraints. Figure 35(a) shows an IDG for which *max_conc_load* is equal to 1 and Figure 35 (b) is the IDG when *max_conc_load* is set to 2. In the two IDGs, there are three new vertices *flagspiL1, flagspiL2,* and *dataiL*. For the first IDG (Figure 35 (a)), notice the self dependence edge *e5* that states that the load related to *flagspiL1* at iteration (k,i,j) has to wait for the load operated at iteration (k,i,j-1), that is why the edge weight is equal to (0,0,-1). In the second IDG (Figure 35 (b)), the number of loads is set to 2, thus we notice all dependence distance vectors related to *flagspiL1* have been decremented and that is why w(*e5*) is now equal to (0,0,-2) while it is equal to (0,0,-1) when *max_conc_load* is set to 1. Also note the dependencies between *flagspiL1* and *dataiL*: since the first load of *flagspi* is performed before the *datai* load in the C code, the dependence distance vector of *e3* is equal to (0,0,0) in Figure 35 (a). The edge *e4* that addresses the dependency from *dataiL* to *flagspiL1* has a distance vector equal to (0,0,-1) since the load in *datai* is performed after the first load of *flagspi*.

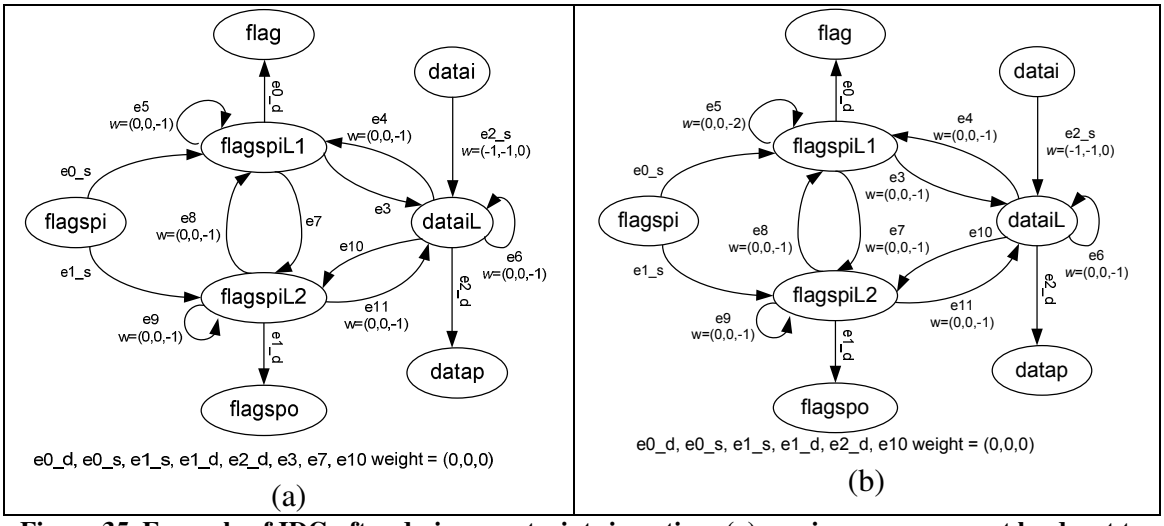


Figure 35. Example of IDG after design constraints insertion: (a) maximum concurrent loads set to 1; (b) maximum concurrent loads set to 2

6.5.3 Scheduling Constraints Generation (step 5)

This step consists on computing the duration associated with each vertex of the generated IDG. Vertex durations are used during the SARE scheduling in step 7 of Figure

33. The aim of this step is to consider I/O contraint effects on the vertex durations. At this point, each vertex has its time stamp *TSV(u)* set. A vertex to which is associated a design-oriented vertex should have its time stamp and its delay inititally set to 0. The duration of a vertex is the time elapsed between the completion of its predecessors and the end of its execution. Equation 2 states how the vertex duration is evaluated. This equation assumes the vertex is updated only once per iteration. The duration of vertex *v* is equal to its delay plus the offset between *v* and its nearest predecessor *p*. The offset is computed as follows. The difference between the start of the predecessor *p* and the start of *v* is computed (TSV_v - TSV_p) multiplied by the design *steptime*. Then, the delay of *p*, $Delay_p$, is subtracted from the difference to complete the computation. Recall that *steptime* is the number of time slots (TS) associated with a clock period. A time stamp corresponds to the clock cycle at which the operation is executed. The multiplication with *steptime* normalizes the offset in preparation to the addition with *v*'s and *p*'s delay that are expressed in terms of time slots. The nearest predecessor has the lowest offset with regard to *v*. The nearest predecessor also has the highest end-of-execution time stamp. When the vertex does not have any predecessor, its duration is set to its delay plus the delay associated to its time stamp (which is equal to its time stamp multiplied by the design *steptime*):

$$duration = \begin{cases} Delay_v + ((TSV_v - TSV_p) * steptime - Delay_p) & when \quad \Pr eds \neq \phi \\ Delay_v + TSV_v * steptime & when \quad \Pr eds = \phi \end{cases} \qquad \textbf{(2)}$$

Where *p* is *v*'s nearest predecessor.

Figure 36 illustrates an example of the duration computing. The design *steptime* is set to 2. The IDG has three vertices V1, V2, and V3 and their time stamp is equal to computation steps (CS) 0, 2 and 10 CS+1 slot of time, respectively. Their delays, given in parentheses in the vertex circle, are equal to 2, 3, and 1 TS, respectively. Since V1 and V2 do not have any predecessor, their duration will be equal to 2 and 7 units of time (using equation 2). The offset between V1 to V3, *offset1*, is equal to 19 while the offset between V2 to V3, *offset2*, is equal to 14 implying V2 is the nearest predecessor of V3.

V3's duration is equal to 15 time slots (Delay+offset=1+14). The duration of each vertex of the IDG is computed while knowing that the duration of the cloned vertex will be set to 0.

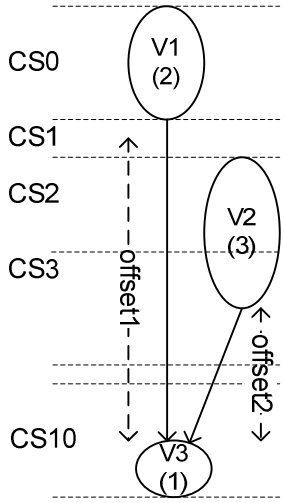


Figure 36. Scheduling example

6.5.4 SARE Generation (step 6)

The aim of this step is to generate the system of affine recurrence equations (SARE) from the manipulated IDG. The SARE is automatically generated (within LProf) in the form of Alpha code [144], which allows stating data expressions according to their index domains. Further in the process flow, this code is compiled and scheduled to yield the estimated execution time of the nested loops.

6.6 Results

The estimated execution times are used during the architecture exploration process in which loop acceleration possibilities are first analyzed. The goal of the design exploration proposed in [117] is to gauge acceleration opportunities of an application specific architecture compared to a pure software solution. The exploration process flow is based on the tweaking of design constaints and the application of loop optimization techniques that can leverage loop parllelization possibilities. In [117], the loop acceleration

opportunities were explored at instruciton level. Thus, the estimates provide a glimpse of the loop acceleration possibilities at iteration level according to I/O design constraints. Estimated execution times are drawn for benchmarks such as the YUV to RGB conversion from the JPEG decoder algorithm and the Matrix-Vector Multiply (MVM) used in many signal processing algorithms such as the discrete cosine transform (DCT) or the discrete Fourier transform (DFT).

6.6.1 Experimental results

The aim of this section is to show the impact of the design oriented constraints on leveraging the coarse-grain parallelism opportunities. Figure 37 illustrates the estimated execution time of a loop nest extracted from the JPEG decoder algorithm. The load and store operations have a delay of 4 time slots each. The loop nest performs the YUV to RGB conversion on a block of pixels. The x-axis represents the different designs that have been scheduled. The first digit in the design names stands for the maximum number of concurrent loads *max_conc_load* and the second digit stands for the maximum number of concurrent stores *max_conc_store*. The number of concurrent loads and stores was left unconstrained. One iteration execution time amounts to 284 time stamps (TS) on an EGPP. Without design constraints, there is no data dependency between iterations. Two to three loads and one store are performed per iteration.

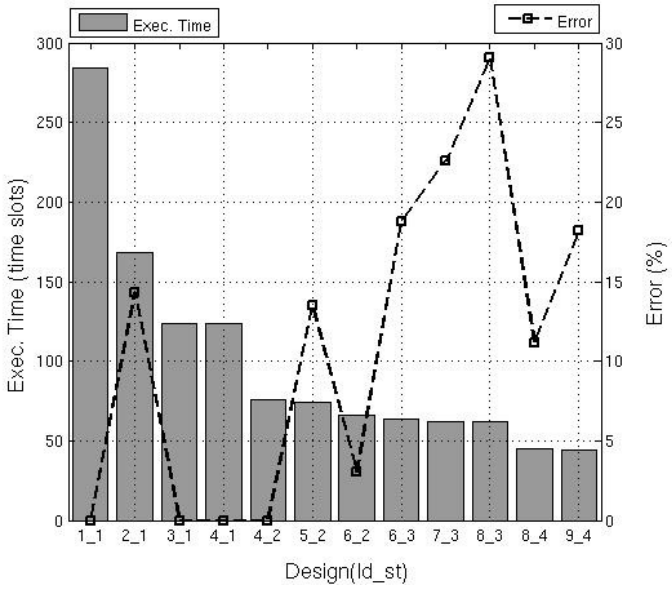


Figure 37. Estimated execution time of a loop nest from JPEG

In this benchmark, design constraints introduce some dependencies, which is why, in Figure 37, the estimated total execution time varies as the design constraints are loosened. In the first design, only one load and one store can be performed concurrently and the execution time hits a maximum of 284 TS. When the number of load is stretched in design 2_1 and 3_1 the execution time decreases to 168 and 124 TS as some parallelization opportunities are leveraged. From 3_1, the execution time is stable at 124 TS meaning that no more parallelization can be leveraged under the current data constraints. In this case, it is the store constraint, set to 1, that limits parallelization. Thus, the number of stores was increased to 2 in design 4_2, 5_2, and 6_2. The estimated execution time plunges from designs 4_1 to 4_2 from 124 to 76. In other words, the execution time was divided by almost 2. In the figure, note that there is a direct link between *max_conc_load* and the execution time. It seems that, for each increase of 2 or 3 loads and of 1 store, the total execution time is improved. For example, we can see that the total execution time is stable for design 2_1 and 3_1, designs 4_2 to 6_2, and it is also stable for designs 6_3 to 8_3 and between 8_4 and 9_4, etc. This is due to the 2 to 3 loads required to compute and store one output. These loads are performed for each iteration thus these thresholds have to be reached to break the design dependency between two

iterations. The loop was fully unrolled and the corresponding CDFG was scheduled with design constraints in order to compute the error between the estimates and the unrolled loop execution times.

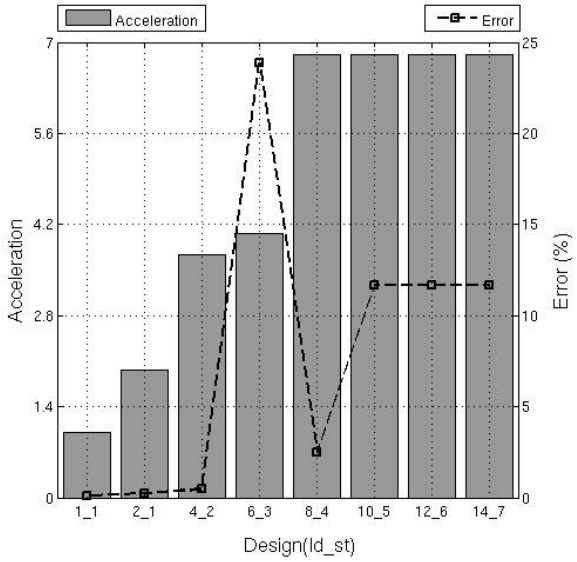
The accuracy of the execution time estimates obtained with the proposed method were evaluated as an error, as compared to the processing time estimates obtained when fully unrolling the loop operations before scheduling. In the loop of the JPEG algorithm converting YUV to RGB, the error varies from 0% to 29%. After analysis, we found that the error variations are due to the SARE scheduler. It should be recalled that the SARE scheduler is based on linear integer equations. As some simplications are done to find the best integer coefficients of the equations, the resulting coefficients become approximations of the theoretical best values.

The loop nest that performs a matrix-vector multiplication (MVM) has also been benchmarked. The MVM is composed of a 2-dimension loop nest as illustrated in Figure 38(a). Two loads are required per iteration and one store is performed to save result X1. Each iteration is parallelizable but there is a hard dependency between the iterations of the same line since X[i][j-1] is needed to compute X[i][j]. Figure 38 (b) provides the accelerations of different designs. The acceleration is identified by bars and is bounded to the left y-axis scale. It also gives the error rate related to the estimated acceleration.

```
for (i=0; i<10; i++){
   for (j=0; j<10; j++){
     if(j>0)
       X1[i][j]=X1[i][j-1]- A[i][j]*Y[j]
     else
       X1[i][j]= A[i][j]*Y[j]

     MVM[i]=X1[i][j]
   }
}
```

(a)



(b)

Figure 38. Estimated execution time of a nest of loops from MVM, (a) code, (b) Variation of the execution time

The error rates are drawn in a squared line, that relates to the right y-axis scale. The computed estimates assume that the load and store operations have a delay of 4 time slots each. The acceleration is computed according to the total number of time slot of the first design (i.e. 1_1). The first design represents a pure software design where only one load and one store can be performed at the same time. Thus, when the number of loads is increased to 2, the acceleration almost doubles to 1.95. In fact, the acceleration linearly increases with each increment of the number of loads by 2 and of the number of stores by 1. In the MVM, two loads (A[] and Y[]) are needed to compute and store one output (MVM[]). From design 8_4, the estimated acceleration reaches its limit of 6.8 for 8 allowed loads. At this point, it seems there is no more parallelism opportunities to leverage since the acceleration is stable. In reality, the acceleration should reach its limit with 10 allowed loads because there is a hard data dependency between each iteration of the "j" loop (X1[i][j]=X1[i][j-1]-A[i][j]*Y[j]). The estimates are not 100% accurate. In Figure 38, the error rate is reasonnable with less than 12% except for design 6_3 with 23.8%. The SARE scheduler generates a linear time function for each variable of the system. That linear function depends on the loop indices and their domain of values. In

the case of design 6_3, the function time was equal to “21+2i+18j”. We notice that even if 6 loads can be done at the same time, one output will be generated every “2i”. In fact, the scheduler found that 2 was the lowest coefficient that allows to resolve the SARE equations. Later, for design 8_4, the scheduler outputed a function time equal to “30+i+9j”. In that case, the outputs are scheduled back to back at each iteration of the “j” loop. As can be seen, the hard dependency is still respected since iteration [1,9] needs iteration [1,8] to be scheduled. Thus, to obtain better estimates, a non-linear and non-integer equation solver should be used, which imply a much higher computational load (thus longer estimation times would be required).

Notice that our performance estimation technique overestimates the loop execution time. In fact, the MMAlpha tool only generates a timing funtion that is within the boundary of the possible solutions according the loop data dependencies, and its indices iteration space.

Our technique is the first that considers application specific design contraints during the estimation of nested loops execution time. Thus, no existing technique produces results that can be compared directly to ours as previous works target monoprocessor architectures. Another reason why previously reported results cannot be compared to our work is that we look for parallelization opportunities at every possible level, which is not the case in somewhat comparable prior work.

One goal of our technique is to explore how the execution time varies according to parallelism opportunity exploitation and according to architectural design constraints. As mentionned before, the few estimation techniques that focus on nested loop execution time do not take concurrency between different operations or different iteratons into account. They usually assume that the loop iterations and their operations will be executed sequentially. Our technique can estimate the execution time as the existing techniques do, if the targeted design is configured as an embedded general processor. Our technique goes further by considering more types of design and more design constraints.

6.6.2 Estimation time

It should be recalled that the loop execution time estimator was devised to be used in a loop-acceleration design framework. Thus, the goal is to quickly evaluate the loop acceleration possibilities before the design and simulation of any model. The proposed loop execution time estimator executes very fast and allows to quickly obtain results. Table 8 presents the time required to perform time estimates versus the scheduling time of the fully unrolled loop execution time. It should be noticed that the estimator was run on an IBM ThinkPad R52 with a 1.86 GHz pentium M. There are 159 nodes in the CDFG of the JPEG loop iteration and 1290 nodes in the CDFG of the fully unrolled loop. With the proposed estimation technique, the results were computed in less that 2 minutes rather than 10.

Table 8. Estimation Time

	Nb. Iterations	Nb. nodes per iteration	Nb. of nodes of unrolled loop	Unrolled loop scheduling time	Estimation time
JPEG	30	159	1290	10 mn	2mn
MATVECT	100	76	6400	24 hours	<1mn

The scheduling time depends on the number of nodes in the CDFG and the number of dependencies. That is why the unrolled loop scheduling time for JPEG is still reasonable (approximately 10 min). On the other hand, for the MVM, the unrolled loop scheduling time was very large, which clearly illustrates the need for a fast loop estimation technique. It took 24 hours to generate and schedule the CDFG of the MVM unrolled loop. The number of nodes in the CDFG explodes to 6400 compared to 76 in the initial iteration. Thus, the number of nodes combined with the amount of data dependencies slowed considerably the scheduling process. By contrast, the results of our estimation technique were ready in less than 1 minute.

Notice that the estimated computation times depend more on the number of

dependencies drawn in the SARE rather than on the nest of loop boundaries. On the other hand, scheduling a fully unrolled nest of loop depends on the overall number of operations, and the number of data and control dependencies that might explode when the number of unrolled iterations is large as was illustrated with the MVM algorithm. Thus, in the context of design exploration, our nest of loop performance estimation allows quickly computing processing time estimates, while as previously shown, the error (between direct estimates and those obtained from fully unrolled loops) is reasonnable. Finally, last but not least, our technique guarantees that the estimates are always inside the boundaries of the solution space of a possible application-specific design according to the specified I/O constraints.

6.7 Conclusions

This paper tackled the estimation of nested-loop execution time during design-space exploration of application-specific designs. This work is part of a project that aims at exploring loop acceleration for different design types, such as application-specific instruction-set processors or ASICs, in a framework named LProf. We proposed an algorithm that integrates I/O-oriented constraints in the generated SARE and we proposed a method that computes the timing constraints associated to the SARE dependencies according to the scheduling results computed in LProf. Then, we showed how the estimated execution time is computed by scheduling the generated SARE and the timing constraints in the MMAlpha environment. The execution times of some benchmarks were estimated. The use of of our estimation technique shows how our technique allows quickly estimating nest of loops execution time, while the magnitude of the errors of those time estimates is reasonable. The estimates also contribute to inform the designer on how the I/O design constraints affects a loop nest performance. The estimated execution time considers that the loop iterations can be parallelized in different hardware units that are each constrained in resources such as the number of arithmetic and logic operators. In the future, we will include resource constraints in the SARE to

allow estimating the execution time when coarse-grain parallelization opportunities can be leveraged.

CHAPITRE 7

DISCUSSION GÉNÉRALE

Dans ce chapitre, nous faisons un retour sur certaines contributions présentées dans les articles présentés. En premier lieu, nous discutons du modèle à 5-patrons présenté au chapitre 4 en insistant sur les différences entre ce dernier et le modèle polyèdral. Nous poursuivons par un retour sur les métriques orientées boucle. Enfin, nous discutons de la pertinence du modèle d'estimation de temps d'exécution présenté.

7.1 Modèle à 5-patrons versus modèle polyèdral

Dans l'article présenté au chapitre 5, une des principales contributions est le modèle à 5-patrons. Ce modèle permet de structurer l'exécution d'une boucle en séparant les opérations de chargement et déchargement de données et les opérations arithmétiques et logiques exécutées dans la boucle. En sachant que les processeurs à usage spécifique permettent d'accélérer l'exécution des opérations arithmétiques et logiques grâce à leur exécution en parallèle et en séquence dans une ou plusieurs instructions spécialisées, nous avons proposé un modèle qui sépare les opérations arithmétiques et logiques des opérations de chargement. Le modèle préconise 5 patrons qui sont :

- Le patron d'initialisation dans lequel les chargements nécessaires et l'initialisation des registres spécialisés avant l'exécution des itérations sont effectués. Ce patron n'est exécuté qu'une seule fois.

- Le patron d'entrée qui est exécuté au début de chaque exécution d'une itération et qui regroupe les opérations de chargement nécessaires à l'execution de l'itération.

- Le patron des opérations arithmétiques et logiques dans lequel les opérations à effectuer à chaque exécution d'une itération sont spécifiées.

- Le patron de sortie qui précise les opérations de lecture ou d'écriture à opérer à la fin de l'exécution de chaque itération.

- Finalement, le patron de fin de boucle qui expose les opérations d'écriture à effectuer à la fin de l'exécution de toutes les itérations de la boucle.

Ce modèle à 5-patrons est utilisé pour effectuer des déroulements de boucle successifs selon les besoins du concepteur. Il est aussi utilisé pour profiter des possibilités de groupement et de réutilisation de données.

Au chapitre 6, nous avons présenté le modèle polyèdral. Ce modèle est utilisé pour représenter mathématiquement les domaines d'itérations d'une boucle. Ce modèle est aussi à la base de plusieurs techniques de transformations de boucles comme nous l'avons vu à la section 2.4. Une question pertinente est de savoir pourquoi ne pas utiliser le modèle polyèdral au lieu du modèle à 5-patrons. Le modèle polyèdral est en fait utilisé en combinaison avec une autre représentation. En effet, ce dernier permet de définir les domaines d'itérations, mais il ne permet pas de définir les dépendances de données. En combinant l'analyse d'un modèle polyèdral avec celle d'un graphe de dépendance de données, une architecture peut être dérivée. Notre modèle à 5-patrons, par contre, ne tient pas compte des domaines d'itérations, donc, les patrons pourraient considérer des opérations qui ne seront pas exécutées. On aurait pu utiliser le modèle polyèdral en combinaison avec le modèle à 5-patrons. Cependant, il aurait fallu automatiser le processus de conception, ce qui aurait requis un temps considérable. Cela aurait facilité le processus de déroulement de boucle et de regroupement de données, par exemple. Cependant, comme le but de la thèse était l'exploration d'architectures à usage spécifique, nous avons donc préféré aller de l'avant en proposant des métriques plutôt que de focaliser sur un outil de génération d'instructions spécialisées. Néanmoins, nous avons élaboré une preuve de concept qui montre l'efficacité de notre méthodologie pour l'accélération d'applications de traitement d'images et de vidéo par des processeurs à usage spécifique.

7.2 Retour sur les métriques orientées boucle

L'article présenté au chapitre 5 expose de nouvelles métriques orientées boucle ainsi qu'une méthodologie d'exploration architecturale qui les utilise pour mieux guider le concepteur en vue de l'optimisation d'un type de design ciblé ou en vue de trouver le type de design le plus approprié pour accélérer une boucle. Ces métriques sont calculées grâce aux résultats d'ordonnancement préalablement collectés. L'ordonnancement des opérations d'une boucle est effectué selon les contraintes de design précisées par le concepteur. L'ordonnancement est effectué avec la technique ASAP (ordonnancer aussi tôt que possible).

Même si les métriques permettent de trouver rapidement une tendance et de cibler les aspects qui limitent ou favorisent l'accélération d'une boucle par un type d'architecture donné, la question qui se pose est : quelle est la qualité des résultats d'ordonnancement générés? La technique d'ordonnancement ASAP a été utilisée car c'est une technique simple et rapide à implémenter, même si elle a des limites comme l'ont mentionné Paulin et al. [134]. La technique ASAP est une technique qui fournit de bons résultats lorsqu'il n'y a pas de contraintes de design. Mais, dans le cas contraire, il est évident que l'ordonnancement de certains chemins de données ne se fera pas de manière optimale. L'algorithme de Paulin et al. [134] aurait pu être appliqué. Nous pensons que l'application de cet algorithme nécessiterait de rajouter des contraintes de design plus spécifiques à chacun des types de design considéré, mais il serait conseillé d'analyser plus en détail l'applicabilité de leur algorithme durant notre processus d'ordonnancement. Même si nous aurions pu implémenter une technique d'ordonnacement plus efficace, la technique ASAP a permis de collecter des résultats satisfaisants qui ont permis de valider les métriques proposées.

Comme on l'a dit précédemment, un des buts des métriques proposées est de guider le concepteur durant le processus d'optimisation d'une boucle pour un type d'architecture donné. Donc, la question qui se pose est la suivante : a-t-on atteint ce but? Étant donné la

portée de la question, nous pouvons dire que nous avons répondu en majorité à cette question. En effet, nos métriques permettent de cibler les aspects qui limitent l'accélération d'une boucle. Ceci se fait en sachant que l'exécution des opérations en parallèle repose généralement soit sur les dépendances de données soit sur les contraintes architecturales qui font que les données ne seront pas disponibles au bon moment. C'est la raison pour laquelle nous avons établi deux groupes de métriques, à savoir les métriques orientées calcul et les métriques orientées données. En fait, certaines métriques permettent de trouver la classe d'optimisation à appliquer mais elles ne permettent pas de déterminer quel algorithme d'optimisation bien précis devrait être appliqué. Nous avions commencé à cibler cette lacune. Par exemple, nous avons implémenté dans notre cadre d'applications les algorithmes de report de mailles tels que ceux de Gupta et al. [66] et Darte et al. [43]. Malheureusement, le temps nous a manqué pour synthétiser et divulguer les résultats obtenus.

7.3 Pertinence de l'estimation du temps d'exécution minimum (WCET)

La technique d'estimation du temps d'exécution d'un nid de boucles en fonction des contraintes d'entrée/sortie (I/O) du design ciblé permet de tenir compte des possibilités de parallélisme à haut niveau. Comme nous l'avons vu, ces dernières pourront être exploitées seulement si les contraintes d'I/O le permettent. Nous avons considéré les contraintes d'I/O car nous ciblions principalement des applications de traitement de signaux. Ces applications se distinguent par un chemin de données élaboré et un volume de données à traiter. Les types de design considérés peuvent accélérer efficacement un chemin de données, mais le problème qui se pose toujours concerne le mouvement des données vers et hors du design. C'est la raison pour laquelle, pour l'instant, nous avons seulement considéré les contraintes d'I/O. Cependant, nous sommes conscients que certaines contraintes matérielles telles que le nombre d'opérations concurrentes auront un impact sur l'exploitation des possibilités d'accélération à haut niveau.

Un système d'équations affines récurrentes (SARE) est généré durant le processus d'estimation du temps d'exécution. Malheureusement, ce système émule les dépendances de données et de design mais il ne reproduit pas la fonctionnalité du nid de boucles. Nous avons hésité sur la pertinence de générer soit un SARE complet soit un SARE partiel. Finalement, nous avons opté pour le SARE partiel, car le premier problème qui se posait était celui de la syntaxe. En effet, notre cadre d'applications considère un code C en entrée. Donc, très rapidement, nous avons vu que certains énoncés en C ne pourraient pas être retranscrits en code Alpha ou que cette opération nécessiterait d'énormes transformations. En plus, le second point concerne l'ordonnancement des équations affines. En effet, ces dernières sont utilisées pour créer un système d'équations linéaires. Aussi, la taille du système d'équations linéraires a tendance à exploser selon le nombre de dépendances précisées dans le SARE, entre autres. Bien sur, plus le système est gros et plus le temps de résolution du système d'équations linéraires est élevé. Donc, pour éviter de rencontrer ces problèmes, nous avons opté pour la générération d'un SARE partiel qui est plus petit mais qui exprime les dépendances voulues. En fait, le SARE partiel tient compte du temps d'exécution des énoncés C qui ne peuvent pas être transcrits en langage ALPHA puisque les dépendances de données y sont tout de même représentées. Nous avons opté pour cette solution aussi parce que notre but n'était pas de générer le circuit final dans MMAlpha [144], mais juste de trouver la fonction de temps associée au circuit.

Un dernier point dont nous désirons discuter concerne les trois articles inclus dans cette thèse. Tous nos travaux ciblent l'accélération des boucles critiques d'une application. Nous avons sélectionné manuellement toutes les boucles présentées. Nous avons noté manuellement la complexité des calculs effectués dans les boucles pour juger qu'elles nécessiteraient une accélération. Il est vrai que les méthodologies et techniques proposées visent à explorer les possibilités d'accélération d'une boucle donnée. Cependant, il faudrait que le poids de la boucle sur le temps d'exécution total de l'application soit assez important pour que l'accélération en vaille la peine. C'est la raison pour laquelle nous avons implémenté une technique d'encapsulation des boucles.

Toutes les boucles d'une application sont encapsulées dans des fonctions ce qui permettra de collecter des résultats de profilage en utilisant un profileur comme gprof [64], par exemple.

CHAPITRE 8
CONCLUSION

Ce chapitre résume les travaux qui ont été exposés dans cette thèse. Ensuite, les limitations des travaux sont évoquées. Puis, des recommandations sont présentées en rapport avec des améliorations possibles aux travaux effectués et des pistes de recherche futures sont proposées.

8.1 Synthèse des travaux

Tout au long de cette recherche doctorale, nous avons eu à recenser des travaux de recherche plus ou moins reliés aux nôtres. Nous les avons triés en quatre grandes catégories de recherche qui sont : les processeurs spécialisés, la synthèse de circuits à haut niveau, le modèle polyèdral et ses applications et, finalement, les techniques d'analyse statique de temps d'exécution. La recherche sur les processeurs spécialisés est très variée. Elle s'étend de la détection d'unités de calcul à la génération automatique d'instructions spécialisées en passant par l'optimisation des banques de registres. Une importante remarque est la suivante : les travaux antérieurs ne s'attaquent pas particulièrement à l'accélération des boucles, alors que ces dernières sont clairement de bonnes candidates à l'accélération. Nous pensons que les techniques de design de processeurs spécialisés et plus spécifiquement celles qui servent à la conception d'instructions spécialisées devraient viser les segments de code qui pèsent le plus sur le temps d'exécution de l'application, sinon ces techniques s'avèrent quelque fois décevantes par rapport au potentiel d'accélération de l'application.

Il existe des travaux sur l'accélération des boucles. Par contre, le processus de génération automatique est tellement complexe que, finalement, les chercheurs ont tendance à se focaliser sur les étapes d'ordonnancement et d'allocation de ressources du

processus de génération. Par conséquent, on peut se demander si les outils allouent les ressources au segment de code approprié? Pour cela, il faudrait savoir quel est le poids du segment traité sur le temps d'exécution total de l'application considérée. De plus, comme nous l'avons vu, pour profiter des possibilités de parallélisme du segment à traiter, des techniques de transformation sont de plus en plus utilisées pour exposer le parallélisme à l'ordonnanceur. En effet, la plupart des ordonnanceurs sont seulement capables de trouver le parallélisme au niveau instruction, donc des techniques de transformation de code sont appliquées avant la phase d'ordonnancement. Le modèle polyèdral, rappelons-le, est une représentation mathématique des domaines d'itération d'un nid de boucles. Ce modèle a contribué à l'explosion d'un grand nombre de travaux sur des techniques de transformation et de parallélisation de boucles. Par exemple, la fusion de boucle peut être résolue en un clin d'œil en utilisant le modèle polyèdral. Ce modèle a largement été utilisé pour l'analyse et l'optimisation de données. Le modèle polyèdral est toujours utilisé en combinaison avec un graphe de dépendances de données pour effectuer certaines transformations. Ce graphe est analysé en fonction des domaines d'itération extraits grâce au modèle polyèdral. Ce modèle permet de détecter le parallélisme présent ou l'applicabilité de certaines transformations. Afin de déterminer l'accélération maximale atteignable par un segment de code, nous avons eu à recenser des travaux sur l'estimation statique du temps d'exécution. Nous avons constaté que la plupart des techniques existantes se rapportent à des architectures monoprocesseur. Il est donc rare que les techniques existantes s'appliquent directement à des architectures à usage spécifique. Une autre observation est qu'il y a peu de recherches sur l'estimation du temps d'exécution d'un nid de boucles. Les rares articles que nous avons trouvés ciblent l'estimation d'une boucle unidimensionnelle et se focalisent surtout sur les chemins d'exécution plutôt que sur les domaines d'itération de la boucle.

L'objectif principal de cette thèse était de proposer des méthodologies d'exploration architecturale de design à usage spécifique, méthodologies basées sur une spécification en C. Au début de notre projet de recherche, nous avions débuté par une large

exploration. Cependant, au fur et à mesure, nous en sommes arrivés à la conclusion que les boucles de traitement sont les candidates idéales en vue d'une accélération par des designs à usage spécifique, raison pour laquelle nous avons décidé de concentrer nos efforts sur l'accélération de boucles par des architectures à usage spécifique. Le premier type de design à usage spécifique que nous avons ciblé concerne les processeurs spécialisés. Aussi, dans un premier temps, nous avons eu à proposer une méthodologie de conception de processeurs spécialisés basée sur l'accélération de boucles. Cette méthodologie comporte plusieurs contributions dont une représentation à 5-patrons qui permet de séparer les calculs des chargements de données vers ou de la mémoire. En effet, nous avons considéré que l'exécution du corps d'une boucle peut être formalisée comme un chemin qui commence par le chargement des données à manipuler pour produire le résultat et qui se termine par l'écriture des résultats. Cette représentation à 5-patrons est ensuite utilisée pour générer une séquence d'instructions spécialisées fortement couplées. La seconde contribution est la transformation du modèle à 5-patrons afin d'exploiter le parallélisme à haut niveau d'une boucle dans la séquence d'instructions spécialisées. La troisième contribution de cette méthodologie est qu'elle propose le regroupement et la réutilisation de données afin de réduire le nombre d'accès mémoire. La méthode de conception a été appliquée sur des applications de traitement de signaux telles que : le désentrelaceur ELA, le filtre de Wiener et la 2D-DCT. Les designs conçus ont atteint de très bonnes performances avec un facteur d'accélération allant jusqu'à x18. De plus, la complexité matérielle en termes de portes logiques ajoutées a été raisonnable avec des taux d'augmentation variant entre 18% et 59%. Ces résultats nous ont conforté dans notre choix de cibler l'accélération de boucles par des architectures à usage spécifique.

Dans la première partie du projet de doctorat, les efforts ont été consacrés à la phase de conception mais, comme discuté plus tôt, deux aspects ont été abordés par la méthode proposée, à savoir l'exploitation des possibilités de parallélisme et l'optimisation des accès en mémoire. Donc, durant la seconde partie du projet, nous avons concentré nos

efforts sur le profilage de boucles. En effet, avant de commencer à concevoir un design, il est utile d'avoir des indications sur les possibilités d'accélération de la boucle à accélérer. Les deux derniers articles que nous avons présentés dans cette thèse ont pour but de répondre à cette question. Aussi, une méthodologie d'exploration architecturale basée sur des métriques orientées boucle a été proposée. Les métriques proposées permettent de trouver les aspects qui limitent ou qui favorisent l'accélération de boucle. Certaines métriques permettent de déterminer quels types d'optimisation contribueraient à accélérer une boucle. L'autre contribution présentée dans cet article est une méthodologie itérative qui contribue à trouver le design le plus approprié pour profiter des possibilités d'accélération de la boucle traitée. La méthodologie d'exploration architecturale a été appliquée pour déterminer les facteurs limitant l'accélération de boucles extraites d'algorithmes tels que : JPEG, MPEG, désentrelaceur d'ELA et décodeur turbo. Le processus d'analyse proposé a aussi été déroulé pour explorer en détail l'accélération de l'algorithme de la SAD sur un processeur à usage spécifique. La précision des estimations d'accélération calculées a aussi été évaluée pour déterminer le réalisme des estimations utilisées durant le processus d'analyse. Il s'avère que les taux d'erreurs sur les accélérations varient entre 2.59% et 13.49%, ce qui est raisonnable comme taux par rapport au temps d'exploration gagné s'il avait fallu concevoir et simuler les designs.

Malheureusement, le calcul des métriques ne tient compte que du parallélisme au niveau instruction de la boucle. Donc, dans le dernier article présenté, nous avons proposé une technique d'estimation du temps d'exécution d'un nid de boucles qui tient compte des possibilités de parallélisme à haut niveau. Cette technique d'estimation est la première qui considère des contraintes d'entrée/sortie d'un design durant le processus d'ordonnancement. Le processus d'ordonnancement est décomposé en deux étapes. La première étape d'ordonnancement effectue l'ordonnancement au niveau instruction et la seconde étape opère à haut niveau. L'innovation de cette technique est l'utilisation du modèle polyèdral et de systèmes d'équations affines récurrentes afin d'exprimer les contraintes de design de l'architecture à usage spécifique. La technique d'estimation a été

appliquée pour estimer le temps d'exécution de nids de boucles extraits des algorithmes de JPEG et de MVM (Matrix-Vector Multiply). Les taux d'erreurs entre les estimations calculées et le temps réel d'exécution de calcul varient entre 0% et 29%. Les estimations ont pu être calculées dans un temps très raisonnable ce qui permet d'explorer beaucoup plus de designs. Alors que les temps d'exécution possible en tenant compte de toutes les possibilité de parallélisation d'un boucle complètement déroulée ont quelque fois nécessité jusqu'à 24 heures.

8.2 Limitations des travaux

Certaines limitations ont été notées sur les travaux effectués. Par exemple, dans la méthodologie de conception de processeur à usage spécifique, la représentation à 5-patrons ne traite pas les énoncés de branchement. Les branchements sont reportés dans le code logiciel ce qui fait que cette représentation n'est pas très adaptée aux boucles comportant de nombreux branchements. Donc, pour certains branchements qui requièrent des sauts de code significatifs, il faudra concevoir plusieurs séquences d'instructions associées à chaque sous-branche. Dans ce cas, certaines initialisations de registres pourraient ne pas être nécessaires et les effectuer ralentirait l'exécution de la boucle. D'un autre côté, la représentation à 5-patrons ne gère pas directement les opérations de lecture et d'écriture de données. Ces opérations sont effectuées dans le code logiciel, alors qu'elles pourraient être effectuées dans les instructions spécialisées. Le calcul des adresses en mémoire et l'envoi de la requête de transaction mémoire pourraient aussi être effectués dans les instructions spécialisées ce qui contribuerait à exécuter plus de code en matériel et donc à accélérer la boucle. L'autre limitation de la méthodologie est qu'elle ne tient pas compte des coûts de conception, alors que la taille des instructions créées dans le processus d'optimisation en vue d'atteindre une accélération élevée a tendance à exploser. Cette taille est mesurée en termes de quantité de matériel requis pour une mise en oeuvre. Pour l'instant, c'est le concepteur qui doit définir si la taille du processeur demeure raisonnable afin de gérer le processus de conception en conséquence.

Le processus d'exploration architecturale basé sur des métriques orientées boucle comporte aussi des limitations. Nous pouvons citer l'application de la technique d'ordonnancement ASAP (« *As Soon As Possible* »). Même si, pour l'instant, notre processus ne tient pas compte des coûts de conception, il serait utile dans le futur d'utiliser un algorithme d'ordonnancement plus efficace. Nous aurions pu appliquer des techniques d'ordonnancement selon le type de design ciblé, par exemple. Une autre limitation est due aux contraintes architecturales proposées pour spécifier le design. Ces contraintes sont générales mais il faudrait ajouter des contraintes plus spécifiques à chaque type de design ciblé afin d'obtenir des résultats plus précis sur les possibilités d'accélération. Une autre limitation concerne le processus d'exploration. Ce dernier propose certaines actions à exécuter pour améliorer des performances d'une boucle mais aucun algorithme d'optimisation précis n'est proposé. Une autre limitation concerne l'utilisation du chemin le plus long pour le calcul des métriques et l'estimation du temps d'exécution d'un nid de boucles. En effet, nous n'avons pas vérifié si le chemin considéré était contenu dans un code mort ou si c'est un chemin fréquemment exécuté, par exemple. D'un autre côté, l'estimation du temps d'exécution pourrait considérer tous les chemins les plus longs associés à chaque domaine d'itération, cela permettrait d'obtenir des résultats plus précis.

8.3 Recommandations

Notre première recommandation consiste à automatiser le processus de conception de processeur à usage spécifique ayant des instructions spécialisées fortement couplées. Il faudrait extraire automatiquement la représentation à 5-patrons. Ensuite, l'application de certaines transformations pourrait être dirigée par le concepteur ou être effectuée automatiquement selon des objectifs et des contraintes de design préalablement spécifiés. Le modèle polyèdral pourrait être utilisé durant le processus de conception de processeurs spécialisés afin de trouver des techniques de transformation appropriées pour dévoiler le parallélisme à haut niveau. Enfin, il faudrait un générateur automatique de code matériel

et logiciel. Pour ce faire, il faudrait générer la description matérielle des instructions spécialisées conçues et le code logiciel initial devrait être modifié afin d'y insérer l'appel des instructions spécialisées dans les différentes branches de code. L'automatisation contribuerait à accélérer le processus de conception et à explorer plus de solutions potentielles. Une autre recommandation consiste à considérer les coûts de design tels que le nombre de portes logiques et l'énergie consommée par le circuit produit tout au long du processus de design des instructions spécialisées fortement couplées.

Une autre recommandation concerne les métriques orientées boucle. Il faudrait qu'elles tiennent compte du temps d'exécution total du nid de boucles au lieu de seulement considèrer le temps d'exécution d'une seule itération. Ainsi, les métriques considéreraient tous les domaines d'itérations et pourraient être plus précises. Cela signifie qu'il faudrait utiliser la technique d'estimation d'un nid de boucle pour calculer la valeur des métriques. Finalement, la dernière recommandation proposée est l'implémentation d'un générateur automatique d'ASIC et d'instructions spécialisées. En effet, pour l'instant, notre processus d'exploration architecturale fournit des informations par rapport à la capacité du nid de boucles à être accéléré, mais il faudrait que ce processus aboutisse à la conception complète du design. Nous avons ajouté un module de génération automatique de processeur spécialisé. Ce dernier prend les résultats d'ordonnancement de notre profileur pour générer les instructions spécialisés ainsi que le code assembleur associé à une boucle donnée. Il serait donc intéressant, dans le futur, que le module puisse générer le code RTL associé à une boucle et même générer les designs associés à une application complète.

Le processus d'exploration architecturale que nous avons proposé dans cette thèse pourrait être utilisé dans un cadre plus général de partitionnement logiciel/matériel pour la conception d'architecture multiprocesseurs. Les boucles critiques de l'application seront exécutées soit par des séquences d'instructions spécialisées soit par un coprocesseur matériel généré automatiquement. Ainsi, avec la méthodologie d'exploration des possibilités d'accélération d'une boucle, le concepteur pourra très

rapidement déterminer les boucles à accélérer et les types de design qui seraient les plus appropriés à leur accélération.

En conclusion, cette thèse est une fondation pour l'exploration architecturale de designs à usage spécifique. En permettant de cibler plusieurs types d'architecture dans le même environnement, le processus de conception est facilité. Ainsi, un plus grand nombre de solutions pourraient être explorées, ce qui est difficile dans les environnements existants. De plus, avec les contraintes de design simples proposées, le concepteur peut avoir rapidement une idée du niveau de performance que son futur design pourrait atteindre et prendre des décisions architecturales sans pour autant entrer dans des détails de conception et d'implémentation. Cette thèse est donc un premier pas important dans le processus de génération automatique d'un système complet et elle ouvre la porte à de nombreux sujets de recherche futurs.

RÉFÉRENCES

[1] L. V. Agostini, I. S. Silva, and S. Bampi, “Pipelined fast 2D DCT architecture for JPEG image compression,” in Proceedings of the 14th symposium on Integrated circuits and systems design, Pirenopolis , Brazil, 2001, pp. 226-231.

[2] A. Aiken, and A. Nicolau, “Optimal loop parallelization,” *SIGPLAN Not.,* vol. 23, no. 7, pp. 308-317, 1988.

[3] A. Allen, and K. Kennedy, “Automatic translation of FORTRAN programs to vector form,” *ACM Trans. Program. Lang. Syst.,* vol. 9, no. 4, pp. 491-542, 1987.

[4] Altera Corp. "Altera's NIOS II," 2008; http://www.altera.com/.

[5] P. Arato, V. Tamas, and I. Jankovits, *High Level Synthesis of Pipelined Datapaths*, John Wiley & Sons, Inc., 2001.

[6] ARM Ltd. "Amba bus," 2009; www.arm.com.

[7] K. Atasu, G. Dundar, and C. Ozturan, “An integer linear programming approach for identifying instruction-set extensions,” in Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, Jersey City, NJ, USA, 2005.

[8] K. Atasu, O. Mencer, W. Luk, C. Ozturan, and G. Dundar, “Fast Custom Instruction Identification by Convex Subgraph Enumeration,” in ASAP, Leuven, Belgique, 2008.

[9] M. Balakrishnan, A. K. Majumdar, D. K. Banerji, J. G. Linders, and J. C. Majithia, “Allocation of multiport memories in data path synthesis,” *IEEE*

Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 7, no. 4, pp. 536-540, 1988.

[10] F. Balasa, P. G. Kjeldsberg, M. Palkovic, A. Vandecappelle, and F. Catthoor, “Loop transformation methodologies for array-oriented memory management,” in Application-specific Systems, Architectures and Processors (ASAP'06), Steamboat Springs, CO, United States, 2006, pp. 205-212.

[11] U. Banerjee, *Dependence Analysis*, Kluwer Academic Publishers, 1997.

[12] C. Bastoul. "The CLooG Code Generator in the Polyhedral Model's," 2009; http://www.cloog.org/.

[13] C. Bastoul, “Code generation in the polyhedral model is easier than you think,” in Parallel Architecture and Compilation Techniques, 2004. PACT 2004. Proceedings. 13th International Conference on, 2004, pp. 7-16.

[14] C. Bastoul. "PIP/PipLib-Parametric Integer Programming," 2009; http://www.piplib.org/.

[15] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam, “Putting Polyhedral Loop Transformations to Work,” in International Workshop on Languages and Compilers for Parallel Computing, College Station, TX, United States, 2003, pp. 209-225.

[16] E. Bergeron, “Compilation efficace pour FPGA reconfigurable dynamiquement,” Département d'informatique et de recherche opérationnelle, Faculté des arts et des sciences, Université de Montréal, Montréal, 2008.

[17] N. Beucher, N. Bélanger, Y. Savaria, and G. Bois, “A Methodology to Evaluate the Energy Efficiency of Application Specific Processors,” in 14th IEEE

International Conference on Electronics, Circuits and Systems (ICECS 2007), 2007, pp. 983-986.

[18] N. N. Binh, M. Imai, and A. Shiomi, "A new HW/SW partitioning algorithm for synthesizing the highest performance pipelined ASIPs with multiple identical FUs," in European Design Automation Conference with EURO-VHDL '96 and Exhibition, Geneva, Switzerland, 1996, pp. 126-31.

[19] P. Bonzini, and L. Pozzi, "A Retargetable Framework for Automated Discovery of Custom Instructions," in Proceedings of the 2007 International Conference Application-specific Systems, Architectures and Processors, Montréal, QC, Canada, 2007, pp. 334-341.

[20] Y. Bouchebaba, B. Girodias, G. Nicolescu, E. M. Aboulhamid, B. Lavigueur, and P. Paulin, "MPSoC memory optimization using program transformation," *ACM Trans. Des. Autom. Electron. Syst.,* vol. 12, no. 4, pp. 43, 2007.

[21] G. Braun, A. Nohl, W. Sheng, J. Ceng, M. Hohenauer, H. Scharwchter, R. Leupers, and H. Meyr, "A novel approach for flexible and consistent ADL-driven ASIP design," in Proceedings of the 41st annual conference on Design automation, San Diego, CA, United States, 2004.

[22] F. Brewer, and D. D. Gajski, "Chippe: a system for constraint driven behavioral synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 9, no. 7, pp. 681-695, 1990.

[23] M. A. Cantin, Y. Savaria, D. Prodanos, and P. Lavoie, "An automatic word length determination method," in Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS '01), Sydney, NSW, Australia, 2001, pp. 53-56.

[24] J. M. P. Cardoso, "Dynamic loop pipelining in data-driven architectures," in Proceedings of the 2nd conference on Computing frontiers, Ischia, Italy, 2005.

[25] S. Carr, and K. Kennedy, “Improving the ratio of memory operations to floating-point operations in loops,” *ACM Trans. Program. Lang. Syst.,* vol. 16, no. 6, pp. 1768-1810, 1994.

[26] F. Catthoor, K. Danckaert, K. Kulkarni, E. Brockmeyer, P. Kjeldsberg, T. van Achteren, and T. Omnes, *Data Access and Storage Management for Embedded Programmable Processors*, Kluwer Academic, 2002.

[27] Celoxica. "Celoxica DK Design Suite," 2008; www.celoxica.com.

[28] H. Chao, S. Ravi, A. Raghunathan, and N. K. Jha, “Generation of Heterogeneous Distributed Architectures for Memory-Intensive Applications Through High-Level Synthesis,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 15, no. 11, pp. 1191-1204, 2007.

[29] L. Chao, A. S. LaPaugh, and E. H. M. Sha, “Rotation scheduling: a loop pipelining algorithm,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on,* vol. 16, no. 3, pp. 229-239, 1997.

[30] M. Chaudhuri, and M. Heinrich, “Integrated Memory Controllers with Parallel Coherence Streams,” *IEEE Transactions on Parallel and Distributed Systems,* vol. 18, no. 8, pp. 1159-1173, 2007.

[31] S. Chaudhuri, and R. A. Walker, “ILP-based scheduling with time and resource constraints in high level synthesis,” in Proceedings of the 7th International Conference on VLSI Design, 1994, pp. 17-20.

[32] X. Chen, and M. D. L., “Supporting multiple-input, multiple-output custom functions in configurable processors,” *J. Syst. Archit.,* vol. 53, no. 5-6, pp. 263-271, 2007.

[33] N. Cheung, J. Henkel, and S. Parameswaran, "Rapid Configuration and Instruction Selection for an ASIP: A Case Study," in Proceedings of the conference on Design, Automation and Test in Europe, 2003.

[34] N. Clark, J. Blome, M. Chu, M. Scott, S. Biles, and K. Flautner, "An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors," in Proceedings of the 32nd annual international symposium on Computer Architecture, 2005, pp. 272-283.

[35] N. Clark, A. Hormati, S. Mahlke, and S. Yehia, "Scalable subgraph mapping for acyclic computation accelerators," in Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems, Seoul, Korea, 2006.

[36] P. Clauss, and B. Kenmei, "Polyhedral Modeling and Analysis of Memory Access Profiles," in Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors, 2006.

[37] P. Clauss, and V. Loechner, "Parametric Analysis of Polyhedral Iteration Spaces," *The Journal of VLSI Signal Processing,* vol. 19, no. 2, pp. 179-194, 1998.

[38] B. Cohen, *VHDL Coding Styles and Methodologies*, Springer, 1999.

[39] J. Cong, Y. Fan, G. Han, A. Jagannathan, G. Reinman, and Z. Zhang, "Instruction set extension with shadow registers for configurable processors," in Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays, Monterey, CA, United States, 2005.

[40] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," in ACM/SIGDA International Symposium on Field Programmable Gate Arrays - FPGA, Monterey, CA, United States, 2004, pp. 183-189.

[41] P. Coussy, and A. Morawiec, *High-Level Synthesis from Algorithm to Digital Circuit*, Springer Netherlands, 2008.

[42] Coware. "LisaTek," 2009; http://www.coware.com.

[43] A. Darte, and G. Huard, "Loop Shifting for Loop Compaction," *Languages and Compilers for Parallel Computing*, pp. 415-431, 2000.

[44] A. Darte, and Y. Robert, "Constructive methods for scheduling uniform loop nests," *IEEE Transactions on Parallel and Distributed Systems,* vol. 5, no. 8, pp. 814-822, 1994.

[45] A. Darte, Y. Robert, and F. Vivien, *Scheduling and Automatic Parallelization*, Birkhauser Boston, 2000.

[46] A. Darte, and F. Vivien, "Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs," *Int. J. Parallel Program.,* vol. 25, no. 6, pp. 447-496, 1997.

[47] S. K. Dash, and S. Thambipillai, "Rapid Estimation of Instruction Cache Hit Rates Using Loop Profiling," in ASAP, Leuven, Belgique, 2008.

[48] S. Derrien, S. Rajopadhye, P. Quinton, and T. Risset, "High-Level Synthesis of Loops Using the Polyhedral Model," *High-Level Synthesis*, pp. 215-230, Kluwer Academic Publishers, 2008.

[49] J. Engblom, and B. Jonsson, "Processor Pipelines and Their Properties for Static WCET Analysis," *Embedded Software*, pp. 334-348, 2002.

[50] P. Feautrier, "Automatic parallelization in the polytope model," *The Data Parallel Programming Model*, pp. 79-103, Springer-Verlag 1996.

[51] P. Feautrier, “Dataflow analysis of array and scalar references,” *International Journal of Parallel Programming,* vol. 20, no. 1, pp. 23-52, 1991.

[52] P. Feautrier, “Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time,” *International Journal of Parallel Programming,* vol. 21, no. 6, pp. 389-420, 1992.

[53] P. Feautrier, “Some efficient solutions to the affine scheduling problem: I. One-dimensional time,” *Int. J. Parallel Program.,* vol. 21, no. 5, pp. 313-348, 1992.

[54] Y. Fei, S. Ravi, A. Raghunathan, and N. K. Jha, “A hybrid energy-estimation technique for extensible processors,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 23, no. 5, pp. 652-664, 2004.

[55] A. Fraboulet, G. Huard, and A. Mignotte, “Loop Alignment for Memory Accesses Optimization,” in Proceedings of the 12th international symposium on System synthesis, 1999.

[56] Free Software Foundation. "GCC, the GNU Compiler Collection," 2008; http://gcc.gnu.org/.

[57] D. D. Gajski, and L. Ramachandran, “Introduction to High-Level Synthesis,” *IEEE Design & Test,* vol. 11, no. 4, pp. 44-54, 1994.

[58] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*, 1st ed., Prentice Hall, 1994.

[59] D. Galloway, “The Transmogrifier C hardware description language and compiler for FPGAs,” in Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines, Napa Valley, CA, United States, 1995.

[60] C. Galuzzi, and K. Bertels, “The Instruction-Set Extension Problem: A Survey,” in Proceedings of the 4th international workshop on Reconfigurable Computing: Architectures, Tools and Applications, London, UK, 2008.

[61] C. Galuzzi, K. Bertels, and S. Vassiliadis, "A Linear Complexity Algorithm for the Generation of Multiple Input Single Output Instructions of Variable Size," *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pp. 283-293, Springer Berlin / Heidelberg, 2007.

[62] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam, “Semi-automatic composition of loop transformations,” *Intl. J. of Parallel Programming,* vol. 34, no. 3, pp. 261–317, 2006.

[63] D. Goodwin, and D. Petkov, “Automatic generation of application specific processors,” in Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems, San Jose, CA, United States, 2003, pp. 137-147.

[64] S. L. Graham, P. B. Kessler, and M. K. Mckusick, “Gprof: A call graph execution profiler,” *SIGPLAN Not.,* vol. 17, no. 6, pp. 120-126, 1982.

[65] R. Gupta, and F. Brewer, "High-Level Synthesis: A Retrospective," *High-Level Synthesis*, pp. 13-28, 2008.

[66] S. Gupta, R. K. Gupta, N. D. Dutt, and A. Nicolau, *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*, 1st ed., Springer, 2004.

[67] T. V. K. Gupta, R. E. Ko, and R. Barua, “Compiler-directed customization of ASIP cores,” in Proceedings of the 10th International Symposium on Hardware/Software Codesign, Estes Park, CO, United States, 2002, pp. 97-102.

[68] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "EXPRESSION: A Language for Architecture Exploration Through Compiler/Simulator Retargetability," *Design, Automation, and Test in Europe*, pp. 31-45, 2008.

[69] J. R. Hauser, and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor," in IEEE Symposium on FPGAs for Custom Computing Machines, 1997, pp. 12-21.

[70] J. L. Hennessy, and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed., Morgan Kaufmann, 2006.

[71] K. Hogstedt, L. Carter, and J. Ferrante, "On the parallel execution time of tiled loops," *IEEE Transactions on Parallel and Distributed Systems,* vol. 14, no. 3, pp. 307-321, 2003.

[72] Q. Hu, A. Vandecappelle, M. Palkovic, P. G. Kjeldsberg, E. Brockmeyer, and F. Catthoor, "Hierarchical memory size estimation for loop fusion and loop shifting in data-dominated applications," in Proceedings of the 2006 conference on Asia South Pacific design automation, Yokohama, Japan, 2006.

[73] S. H. Huang, Y. L. Jeang, C. T. Hwang, Y. C. Hsu, and J. F. Wang, "A Tree-Based Scheduling Algorithm for Control-Dominated Circuits," in 30th Conference on Design Automation, Dallas, TX, United States 1993, pp. 578-582.

[74] X. Huang, S. Carr, and P. Sweany, "Loop Transformations for Architectures with Partitioned Register Banks," in Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems, Snow Bird, UA, United States, 2001.

[75] M. Imai, N. Binh, and A. Shiomi, "A new HW/SW partitioning algorithm for synthesizing the highest performance pipelined ASIPs with multiple identical

FUs," in Proceedings of the conference on European design automation, Geneva, Switzerland, 1996.

[76] IMEC. "Data Transfer and Storage Exploration methodology (DTSE)," 2008; http://www.imec.be/design/dtse.

[77] IMEC. "Ultra-low-power high-performance application-specific instruction-set processor (ULPHP-ASIP)," 2009; http://imec.be/wwwinter/mediacenter/en/SR2006/681512.html.

[78] N. Imlig, and A. Tsutsui, "Performance estimation of embedded software with pipeline and cache hazard modeling," *High Performance Computing*, pp. 131-142, 1997.

[79] M. F. Jacome, and G. de Veciana, "Design Challenges for New Application-Specific Processors," *IEEE Design & Test,* vol. 17, no. 2, pp. 40-50, 2000.

[80] M. K. Jain, M. Balakrishnan, and A. Kumar, "An efficient technique for exploring register file size in ASIP design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 23, no. 12, pp. 1693-9, 2004.

[81] M. K. Jain, M. Balakrishnan, and A. Kumar, "Exploring storage organization in ASIP synthesis," in Euromicro Symposium on Digital System Design, Belek-Antalya, Turkey, 2003, pp. 120-127.

[82] M. Kandemir, I. Kadayif, A. Choudhary, and J. A. Zambreno, "Optimizing inter-nest data locality," in Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems, Grenoble, France, 2002.

[83] R. M. Karp, R. E. Miller, and S. Winograd, "The Organization of Computations for Uniform Recurrence Equations," *Journal of the ACM,* vol. 14, no. 3, pp. 563-590, 1967.

[84] K. Karuri, M. A. Al Faruque, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr, "Fine-grained application source code profiling for ASIP design," in Proceedings of 42nd Design Automation Conference, Anaheim, CA, United States, 2005, pp. 329-34.

[85] K. Karuri, A. Chattopadhyay, C. Xiaolin, D. Kammler, H. Ling, R. Leupers, H. Meyr, and G. Ascheid, "A Design Flow for Architecture Exploration and Implementation of Partially Reconfigurable Processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 16, no. 10, pp. 1281-1294, 2008.

[86] W. Kelly, W. Pugh, and E. Rosser, "Code generation for multiple mappings," in Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation (Frontiers'95), 1995.

[87] P. G. Kjeldsberg, F. Catthoor, and E. J. Aas, "Data dependency size estimation for use in memory optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 22, no. 7, pp. 908-921, 2003.

[88] D. J. Kolson, A. Nicolau, and N. Dutt, "Minimization of memory traffic in high-level synthesis," in Proceedings of the 31st annual conference on Design automation, San Diego, CA, United States, 1994.

[89] D. Ku, and G. DeMicheli, *HardwareC -- A Language for Hardware Design (Version 2.0)*, Stanford University, 1990.

[90] A. Kuehlmann, and R. A. Bergamaschi, “Timing analysis in high-level synthesis,” in IEEE/ACM international conference proceedings on Computer-aided design, Santa Clara, CA, United States, 1992.

[91] C. Kulkarni, C. Ghez, M. Miranda, F. Catthoor, and H. De Man, “Cache conscious data layout organization for conflict miss reduction in embedded multimedia applications,” in IEEE Transactions on Computers, 2005, pp. 76-81.

[92] S. Kurra, N. K. Singh, and P. R. Panda, “The impact of loop unrolling on controller delay in high level synthesis,” in Design, Automation and Test in Europe, Nice Acropolis, France, 2007, pp. 391-396.

[93] M. Lam, “Software pipelining: an effective scheduling technique for VLIW machines,” *SIGPLAN Notices,* vol. 23, no. 7, pp. 318-328, 1988.

[94] L. Lamport, “The parallel execution of DO loops,” *Commun. ACM,* vol. 17, no. 2, pp. 83-93, 1974.

[95] B. Landwehr, P. Marwedel, and R. Doemer, “OSCAR: optimum simultaneous scheduling, allocation and resource binding based on integer programming,” in Proceedings of the european conference on design automation, Grenoble, France, 1994.

[96] C. Lengauer, “Loop parallelization in the polytope model,” in 4th International Conference on Concurrency Theory, CONCUR'93, Hildesheim, Germany, 1993, pp. 398-416.

[97] K. Lermer, C. J. Fidge, and I. J. Hayes, “Linear Approximation of Execution-Time Constraints,” *Formal Aspects of Computing,* vol. 15, no. 4, pp. 319-348, 2003.

[98] R. Leupers, “Code generation for embedded processors,” in Proceedings of the 13th international symposium on System synthesis, Madrid, Spain, 2000.

[99] J. Li, and R. K. Gupta, “Decomposition of timed decision tables and its use in presynthesis optimizations,” in Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design, San Jose, California, United States, 1997.

[100] Y.-T. S. Li, S. Malik, and A. Wolfe, “Performance estimation of embedded software with instruction cache modeling,” in Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design, San Jose, CA, United States, 1995.

[101] A. W. Lim, and M. S. Lam, “Maximizing Parallelism and Minimizing Synchronization with Affine Partitions,” *Parallel Computing,* vol. 24, no. 3-4, pp. 445-475, 1999.

[102] J. S. Lim, *Two-dimensional signal and image processing*, Prentice-Hall, 1990.

[103] S.-F. Lin, Y.-L. Chang, and L.-G. Chen, “Motion adaptive interpolation with horizontal motion detection for deinterlacing,” *IEEE Transactions on Consumer Electronics,* vol. 49, no. 4, pp. 1256-65, 2003.

[104] V. Loechner, and D. K. Wilde, “Parameterized Polyhedra and Their Vertices,” *International Journal of Parallel Programming,* vol. 25, no. 6, pp. 525-549, 1997.

[105] M. Munch, N. Wehn, and M. Glesner, “An efficient ILP-based scheduling algorithm for control-dominated VHDL descriptions,” *ACM Transactions on Design Automation of Electronic Systems (TODAES),* vol. 2, no. 4, pp. 344-364, 1997.

[106] MACHSUIF. "MACHSUIF," 2008; http://www.eecs.harvard.edu/hube/software/software.html.

[107] S. Malik, M. Martonosi, and Y.-T. S. Li, "Static Timing Analysis Of Embedded Software," in Proceedings of the 34th Design Automation Conference, 1997, pp. 147-152.

[108] M. Manjunathaiah, G. M. Megson, S. Rajopadhye, and T. Risset, "Uniformization of affine dependence programs for parallel embedded system design," in Conference on Parallel Processing, International, 2001, pp. 205-213.

[109] M.-C. V. Marinescu, and M. Rinard, "High-level automatic pipelining for sequential circuits," in Proceedings of the 14th international symposium on Systems synthesis, Montreal, QC, Canada, 2001.

[110] A. Marongiu, and P. Palazzari, "Automatic Mapping of System of N-Dimensional Affine Recurrence Equations (SARE) onto Distributed Memory Parallel Systems," *IEEE Transactions on Software Engineering,* vol. 26, no. 3, pp. 262-275, 2000.

[111] W. H. Mary, P. A. Saman, R. M. Brian, L. Shih-Wei, and S. L. Monica, "Interprocedural parallelization analysis in SUIF," *ACM Trans. Program. Lang. Syst.,* vol. 27, no. 4, pp. 662-731, 2005.

[112] K. Masselos, F. Catthoor, C. E. Goutis, and H. Deman, "A systematic methodology for the application of data transfer and storage optimizing code transformations for power consumption and execution time reduction in realizations of multimedia algorithms on programmable processors," *IEEE Trans. Very Large Scale Integr. Syst.,* vol. 10, no. 4, pp. 515-518, 2002.

[113] M. Mbaye, N. Bélanger, Y. Savaria, and S. Pierre, "Application specific instruction-set processor generation for video processing based on loop optimization," in IEEE International Symposium on Circuits and Systems (ISCAS'05), Kobe Japan, 2005, pp. 3515-3518 Vol. 4.

[114] M. Mbaye, N. Bélanger, Y. Savaria, and S. Pierre, “Loop-oriented metrics for exploring an application-specific architecture design-space,” in International Conference on Application-Specific Systems, Architectures and Processors, Leuven, Belgium, 2008, pp. 257-262.

[115] M. Mbaye, N. Bélanger, Y. Savaria, and S. Pierre, “Loop Nest Performance Estimation for Application-Specific Architecture Design Space,” *submitted to IEEE Transactions on CAD*, 2010.

[116] M. Mbaye, D. Lebel, N. Bélanger, Y. Savaria, and S. Pierre, “Design exploration with an application-specific instruction-set processor for ELA deinterlacing,” in ISCAS, 2006.

[117] M. M. Mbaye, N. Bélanger, Y. Savaria, and S. Pierre, “Loop-Oriented Metrics for Exploring and Application-Specific Architecture Design-Space,” *submitted to IEEE Transactions on VLSI*, 2010.

[118] M. M. Mbaye, N. Bélanger, Y. Savaria, and S. Pierre, “A Novel Application-specific Instruction-set Processor Design Approach for Video Processing Acceleration,” *The Journal of VLSI Signal Processing,* vol. 47, no. 3, pp. 297-315, 2007.

[119] B. Mei, A. Lambrechts, J. Y. Mignolet, D. Verkest, and R. Lauwereins, “Architecture exploration for a reconfigurable architecture template,” *IEEE Design & Test of Computers,* vol. 22, no. 2, pp. 90-101, 2005.

[120] Mentor Graphics Corp. "CatapultC Synthesis," 2008; www.mentor.com.

[121] H. Meyr, “System-on-chip for communications: the dawn of ASIPs and the dusk of ASICs,” in 2003 IEEE Workshop on Signal Processing Systems, Seoul, South Korea, 2003, pp. 4-5.

[122] M. C. Molina, R. Ruiz-Sautua, J. M. Mendias, and R. Hermida, “Area Optimization of Multi-Cycle Operators in High-Level Synthesis,” in Design, Automation & Test in Europe Conference & Exhibition (DATE '07), 2007, pp. 1-6.

[123] S. Mondal, and S. Memik, “Resource sharing in pipelined CDFG synthesis,” in Proceedings of the 2005 conference on Asia South Pacific design automation, Shanghai, China, 2005.

[124] G. Moore, E., "Cramming more components onto integrated circuits," *Readings in computer architecture*, pp. 56-59, Morgan Kaufmann Publishers Inc., 2000.

[125] T. Nakanishi, K. Joe, C. D. Polychronopoulos, K. Araki, and A. Fukuda, “Estimating parallel execution time of loops with loop-carried dependences,” in Proceedings of the International Conference on Parallel Processing 1996, pp. 61-69.

[126] R. Nalluri, R. Garg, and P. R. Panda, “Customization of Register File Banking Architecture for Low Power,” in 20th International Conference on VLSI Design, 2007, pp. 239-244.

[127] R. Nalluri, and P. R. Panda, “Energy Efficient Application Specific Banked Register Files,” in IEEE VLSI Design And Test Symposium, Goa, India, 2006.

[128] M. Narasimhan, and J. Ramanujam, “On lower bounds for scheduling problems in high-level synthesis,” in Proceedings of the 37th conference on Design automation, Los Angeles, CA, United States, 2000.

[129] J. Ng, D. Kulkarni, W. Li, R. Cox, and S. Bobholz, “Inter-Procedural Loop Fusion, Array Contraction and Rotation,” in Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques, 2003.

[130] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg, “Data and memory optimization techniques for embedded systems,” *ACM Transactions on Design Automation of Electronic Systems (TODAES),* vol. 6, no. 2, pp. 149-206, 2001.

[131] C. Y. Park, “Predicting program execution times by analyzing static and dynamic program paths,” *Real-Time Systems,* vol. 5, no. 1, pp. 31-62, 1993.

[132] J. Park, P. C. Diniz, and K. R. S. Shayee, “Performance and Area Modeling of Complete FPGA Designs in the Presence of Loop Transformations,” *IEEE Transactions on Computers,* vol. 53, no. 11, pp. 1420-1435, 2004.

[133] N. Park, and A. C. Parker, “Sehwa: a software package for synthesis of pipelines from behavioral specifications,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on,* vol. 7, no. 3, pp. 356-370, 1988.

[134] P. G. Paulin, and J. P. Knight, “Force-directed scheduling for the behavioral synthesis of ASICs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 8, no. 6, pp. 661-679, 1989.

[135] T. Pitkänen, T. Rantanen, A. Cilio, and J. Takala, "Hardware Cost Estimation for Application-Specific Processor Design," *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pp. 212-221, 2005.

[136] A. Plesco, and T. Risset, “Coupling Loop Transformations and High-Level Synthesis,” in Symposium en Architecture de machines, 2008.

[137] S. Pop, G.-A. Silber, A. Cohen, C. Bastoul, S. Girbal, and N. Vasilache, “GRAPHITE : Polyhedral Analyses and Optimizations for GCC,” in GNU Compilers Collection Developers Summit 2006, Ottawa, ON, Canada, June 2006.

[138] P. Poplavko, T. Basten, and J. van Meerbergen, “Execution-time Prediction for Dynamic Streaming Applications with Task-level Parallelism,” in 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD'2007), Lübeck, Germany, 2007, pp. 228-235.

[139] M. Potkonjak, and J. Rabaey, “Optimizing resource utilization using transformations,” in IEEE International Conference on Computer-Aided Design, ICCAD'91, Santa Clara, CA , United States, 1991, pp. 88-91.

[140] L. Pozzi, K. Atasu, and P. Ienne, “Exact and approximate algorithms for the extension of embedded processor instruction sets,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 25, no. 7, pp. 1209-29, 2006.

[141] W. Pugh. "The Omega Project: Frameworks and Algorithms for the Analysis and Transformation of Scientific Programs," 2009; http://www.cs.umd.edu/projects/omega/.

[142] P. Puschner, “Algorithms for dependable hard real-time systems,” in Proceedings of the 8th International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'03), 2003, pp. 26-31.

[143] F. Quilleré, S. Rajopadhye, and D. Wilde, “Generation of Efficient Nested Loops from Polyhedra,” *Journal of Parallel Programming,* vol. 28, no. 5, pp. 469-498, 2000.

[144] P. Quinton. "Alpha Homepage," january 28th 2009; http://www.irisa.fr/cosi/ALPHA.

[145] P. Quinton, and V. van Dongen, “The mapping of linear recurrence equations on regular arrays,” *The Journal of VLSI Signal Processing,* vol. 1, no. 2, pp. 95-113, 1989.

[146] I. Radivojevic, and F. Brewer, "A new symbolic technique for control-dependent scheduling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 15, no. 1, pp. 45-57, 1996.

[147] V. Raghunathan, A. Raghunathan, M. B. Srivastava, and M. D. Ercegovac, "High-level synthesis with SIMD units," in Asia and South Pacific Design Automation Conference, 2002, pp. 407-413.

[148] S. V. Rajopadhye, and R. M. Fujimoto, "Synthesizing systolic arrays from recurrence equations," *Parallel Computing,* vol. 14, no. 2, pp. 163-189, 1990.

[149] S. V. Rajopadhye, S. Purushothaman, and R. M. Fujimoto, "On synthesizing systolic arrays from recurrence equations with linear dependencies," in 6th conference on Foundations of software technology and theoretical computer science, New Delhi, India, 1986.

[150] A. Randy, and K. Ken, "Automatic loop interchange," *SIGPLAN Not.,* vol. 39, no. 4, pp. 75-90, 2004.

[151] B. R. Rau, and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," *SIGMICRO Newsletters,* vol. 12, no. 4, pp. 183-198, 1981.

[152] F. A.-K. Rehab, "Resource-constrained loop scheduling in high-level synthesis," in Proceedings of the 43rd annual Southeast regional conference, Kennesaw, Georgia, 2005.

[153] M. Rim, Y. Fann, and R. Jain, "Global scheduling with code-motions for high-level synthesis applications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 3, no. 3, pp. 379-392, 1995.

[154] A. Roychoudhury, T. Mitra, and H. Negi, "Analyzing Loop Paths for Execution Time Estimation," *Distributed Computing and Internet Technology*, pp. 458-469, 2005.

[155] L. C. V. D. Santos, M. J. M. Heijligers, C. A. J. V. Eijk, J. V. Eijnhoven, and J. A. G. Jess, "A code-motion pruning technique for global scheduling," *Transactions on Design Automation of Electronic Systems (TODAES),* vol. 5, no. 1, pp. 1-38, 2000.

[156] H. Schmit, and D. E. Thomas, "Synthesis of application-specific memory designs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 5, no. 1, pp. 101-111, 1997.

[157] W. Shang, and J. Fortes, "On the optimality of linear schedules," *The Journal of VLSI Signal Processing,* vol. 1, no. 3, pp. 209-220, 1989.

[158] C. Shekhar, S. Raj, A. S. Mandal, S. C. Bose, R. Saini, and P. Tanwar, "Application Specific Instruction Set Processors: redefining hardware-software boundary." pp. 915-918.

[159] B. So, M. Hall, W. , and P. Diniz, C. , "A compiler approach to fast hardware design space exploration in FPGA-based systems," in Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, Berlin, Germany, 2002.

[160] B. Su, S. Ding, and J. Xia, "URPR\—An extension of URCR for software pipelining," *SIGMICRO Newsl.,* vol. 17, no. 4, pp. 94-103, 1986.

[161] SUIF. "SUIF 2 Compiler System," 2008; http://suif.stanford.edu/suif/suif2/index.html.

[162] F. Sun, S. Ravi, A. Raghunathan, and N. Jha, K., "A Scalable Application-Specific Processor Synthesis Methodology," in Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design, San Jose, CA, United States, 2003.

[163] D. S. Suresh, W. A. Najjar, F. Vahid, J. R. Villareal, and G. Stitt, "Profiling tools for hardware/software partitioning of embedded applications," in 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03), San Diego, CA, United States, 2003, pp. 189-98.

[164] G. Sylvain, V. Nicolas, C, B. dric, C. Albert, P. David, S. Marc, and T. Olivier, "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies," *Int. J. Parallel Program.,* vol. 34, no. 3, pp. 261-317, 2006.

[165] Synopsys Inc. "Design Compiler," http://www.synopsys.com.

[166] D. Taubman, and M. Marcellin, *Image Compression Fundamentals, Standards and Practice*, Springer, 2002.

[167] N. Tawbi, "Estimation of nested loops execution time by integer arithmetic in convex polyhedra," in 8th International Symposium on Parallel Processing, Cancún, Mexico, 1994, pp. 217-221.

[168] Tensilica Inc. "Tensilica Configurable Processor," 2008; http://www.tensilica.com/.

[169] H. Theiling, C. Ferdinand, and R. Wilhelm, "Fast and Precise WCET Prediction by Separated Cache and Path Analyses," *Real-Time Systems,* vol. 18, no. 2, pp. 157-179, 2000.

[170] W. Thies, F. Vivien, and S. Amarasinghe, “A step towards unifying schedule and storage optimization,” *ACM Transactions on Programming Languages and Systems (TOPLAS),* vol. 29, no. 6, pp. 34, 2007.

[171] M. Tipp, A. C. Daniel, G. Dirk, and P. Ramesh, “Identifying potential parallelism via loop-centric profiling,” in Proceedings of the 4th international conference on Computing frontiers, Ischia, Italy, 2007.

[172] J. P. Tremblay, Y. Savaria, C. Thibeault, and M. Mbaye, “Improving resource utilization in an multiple asynchronous ALU DSP architecture,” in Microsystems and Nanoelectronics Research Conference (MNRC'2008), Ottawa, ON, Canada, 2008, pp. 25-28.

[173] B. Uday, H. Albert, J. Ramanujam, and P. Sadayappan, “A practical automatic polyhedral parallelizer and locality optimizer,” *SIGPLAN Not.,* vol. 43, no. 6, pp. 101-113, 2008.

[174] R. Uday. "PLUTO - An automatic parallelizer and locality optimizer for multicores," http://www.cse.ohio-state.edu/~bondhugu/pluto/.

[175] P. Urard, J. Yi, H. Kwon, and A. Gouraud, "User Needs," *High-Level Synthesis*, pp. 1-12, Springer, 2008.

[176] J. Vanhoof, K. Van Rompaey, I. Bolsens, G. Goossens, and H. De Man, *High-Level Synthesis for Real-Time Digital Signal Processing: The CATHEDRAL-II Silicon Compiler*, Springer, 1993.

[177] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, “The MOLEN polymorphic processor,” *IEEE Transactions on Computers,* vol. 53, no. 11, pp. 1363-1375, 2004.

[178] H. A. Vicki, B. J. Reese, M. L. Randall, and J. A. Stephen, “Software pipelining,” *ACM Computing Surveys (CSUR),* vol. 27, no. 3, pp. 367-432, 1995.

[179] E. Vivancos, C. Healy, F. Mueller, and D. Whalley, “Parametric Timing Analysis,” *SIGPLAN Notices,* vol. 36, no. 8, pp. 88-93, 2001.

[180] S. Vivek, “Optimized Unrolling of Nested Loops,” *International Journal of Parallel Programming,* vol. 29, no. 5, pp. 545-581, 2001.

[181] B. Vucetic, and Y. Jinhong, "Turbo Codes," *The Springer International Series in Engineering and Computer Science*, Springer, ed., 2000, p. 344.

[182] K. Wakabayashi, and T. Yoshimura, “A resource sharing and control synthesis method for conditional branches,” in IEEE International Conference on Computer-Aided Design (ICCAD'89), 1989, pp. 62-65.

[183] D. L. Whitfield, and M. L. Soffa, “An approach for exploring code improving transformations,” *ACM Trans. Program. Lang. Syst.,* vol. 19, no. 6, pp. 1053-1084, 1997.

[184] D. Wilde, and S. Rajopadhye, “Memory reuse analysis in the polyhedral model,” in 2nd International Conference on Parallel Processing, Lyon, France, 1996, pp. 389-97.

[185] T. C. Wilson, N. Mukherjee, M. K. Garg, and D. K. Banerji, “An ILP Solution for Optimum Scheduling, Module and Register Allocation, and Operation Binding in Datapath Synthesis,” *VLSI Design,* vol. 3, no. 1, pp. 21-36, 1995.

[186] M. E. Wolf, and M. S. Lam, “A Loop Transformation Theory and an Algorithm to Maximize Parallelism,” *IEEE Trans. Parallel Distrib. Syst.,* vol. 2, no. 4, pp. 452-471, 1991.

[187] C. Wolinski, and K. Kuchcinski, “Identification of Application Specific Instructions Based on Sub-Graph Isomorphism Constraints,” in IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'2007), Montreal, QC, Canada, 2007, pp. 328-333.

[188] C. Xiaoyong, and L. M. Douglas, “Supporting multiple-input, multiple-output custom functions in configurable processors,” *Journal of Systems Architecture,* vol. 53, no. 5-6, pp. 263-271, 2007.

[189] Y. Yaacoby, and P. R. Cappello, “Scheduling a system of affine recurrence equations onto a systolic array,” in Proceedings of the International Conference on Systolic Arrays, 1988, pp. 373-382.

[190] J. Yan, and W. Zhang, “WCET analysis of instruction caches with prefetching,” *SIGPLAN Notices,* vol. 42, no. 7, pp. 175-184, 2007.

[191] P. Yu, and T. Mitra, "Characterizing embedded applications for instruction-set extensible processors." pp. 723-728.

[192] P. Yu, and T. Mitra, “Scalable custom instructions identification for instruction-set extensible processors,” in International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'2004), Washington, DC, United States, 2004, pp. 69-78.

[193] H. Yuko, T. Hiroyuki, H. Shinya, T. Hiroaki, and I. Katsuya, “CHStone: A benchmark program suite for practical C-based high-level synthesis,” in IEEE International Symposium on Circuits and Systems, ISCAS'2008, Seattle, WA, United States, 2008, pp. 1192-1195.

ANNEXES

ANNEXE A

Ordonnancement de système d'équations affines récurrentes-SARE

L'algorithme de Feautrier consiste à résoudre un système d'équations affines récurrentes (SARE) afin de trouver les fonctions de temps affines de chaque variable du système. Chaque équation du SARE se formule comme suit :

$$(\forall x \in Is : v[f(x)] = F_v(w[g[x],...)) \quad \textbf{(1)}$$

Dans cette équation, Is correspond au domaine d'existence de la dépendance. Donc, l'équation stipule que, dans le domaine Is, v dépend de la fonction Fv qui prend un ou plusieurs arguments tels que w. f(x) et g(x) spécifient les fonctions d'accès de v et w, respectivement.

Nous expliquerons l'algorithme de Feautrier en illustrant un exemple tiré du livre de Darte et al. [1]. Prenons, le nid de boucles suivants:

```
for i=0 to N loop
        S1: s(i) =0;
        for j=0 to N loop
                S2: s(i)=s(i)+a(i,j)*x(j)
        end loop
end loop
```

Figure 1. Exemple 1.

Dans ce nid de boucles, les deux énoncés S1 et S2 sont définis dans les domaines suivants:

$DS1 = (i \mid i \geq 0, N - i \geq 0)$

$$DS2 = (i, j \mid i \geq 0, N - i \geq 0, j \geq 0, N - j \geq 0)$$

On voit que les domaines DS1 et DS2 spécifient les polyèdres dans lesquels les indices d'itérations *i* et *j* sont valides.

Les dépendances des énoncés S1 et S2 se formulent comme suit :

$$(i, j \mid i \geq 0, N - i \geq 0, j = 0) \quad S1(i) \rightarrow S2(i, j)$$
$$(i, j \mid i \geq 0, N - i \geq 0, 1 \leq j, N - j \geq 0) \quad S2(i, j-1) \rightarrow S2(i, j)$$

La première dépendance stipule que pour *i* compris entre 0 et N et pour *j* égal à 0, l'énoncé S2 à l'itération *(i,j)* ne peut s'exécuter que lorsque l'exécution de l'énoncé S1 à l'itération *(i,j)* sera complétée.

La seconde dépendance stipule que pour *i* compris entre 0 et N et pour *j* compris entre 1 et N, l'énoncé S2 à l'itération *(i,j)* ne peut s'exécuter que lorsque l'exécution de l'énoncé S2 à l'itération *(i,j-1)* sera complétée.

Le but de l'algorthime de Feautrier est de trouver la fonction affine de chaque variable du SARE en sachant que la fonction aura la forme suivante :

$$\Theta(S, j, N) = Xsj + YsN + \rho S$$

Tel que Xs et Ys sont des vecteurs et ρS est une constante. En sachant que *j* est le vecteur d'indice d'itérations du nid de boucles et N le vecteur des paramètres de la fonction.

Plus spécifiquement, l'algorithme de Feautrier propose de résoudre un système d'équations linéaires qui respecte la contrainte de dépendance (S→R) suivante :

$$y \in DI_S \Rightarrow \theta(S, y, N) \geq \theta(R, he(y, N), N) + 1$$

L'équation linéaire à résoudre spécifie que toutes les itérations de S dans le domaine DIs doivent s'exécuter un cycle suivant l'exécution de la dépendance R dans le domaine DIs. Donc, on peut reformuler l'équation comme suit :

$$X_{Rj} + Y_{RN} + \rho_R \geq X_S he(j) + Y_S N + \rho_S + 1$$

L'ensemble des contraintes associées à chaque variable et à chaque domaine constitue un systèmes d'équations linéraires à résoudre. Il faut préciser que le système d'équations linéaires peut ne pas être résolu à savoir qu'il n'existe pas de fonctions affines qui respectent les contraintes de dépendances.

Deux principales techniques peuvent être appliquées pour résoudre le système d'équations linéaires de contraintes. Il existe le lemme de Farkas ou la technique des vertex. Mais, en général, Feautrier préfère utiliser le lemme de Farkas pour résoudre le système d'équations.
Le lemme de Farkas définit une contrainte affine positive dans le domaine D et elle se définit comme suit :

$$\Phi(x) \equiv \lambda_0 + \sum_{k=1}^{p} \lambda_k (a_k x + b_k), \forall k \in [0, p] \lambda_k \geq 0$$

En sachant que les paramètres λ_k sont nommés les multiplieurs de Farkas.

L'ordonnancement du système d'équations se base sur l'hypothèse suivante. Il doit exister une fonction Θ(S,j,N) positive dans le polyèdre DIS. Aussi, il devrait exister des constantes $\mu_{S,0}$,…., $\mu_{S,pS}$ telles que :

$$\Theta(S, y, N) \equiv \mu_{s,0} + \sum_{i=1}^{p_s} \mu_{S,i} (A_{S,i} y + B_{S,i} N + c_{S,i})$$

De même pour la contrainte suivante :

$$y \in DI_S \Rightarrow \theta(R, y, N) - \theta(S, he(y, N), N) - 1 \geq 0$$

Ainsi, on peut reformuler le problème sous la forme suivante:

$$\Theta(R, y, N) - \Theta(S, he(y, N), N) - 1 \equiv \lambda_{e,0} + \sum_{i=1}^{p_e} \lambda_{e,i}(A_{e,i}y + B_{e,i}N + c_{e,i})$$

L'équation détaillée de la dépendance (S→R) à résoudre est donc la suivante :

$$\mu_{R,0} + \sum_{i=1}^{p_R} \mu_{T,i}(A_{R,i}y + B_{R,i}N + c_{R,i})$$
$$- \mu_{s,0} + \sum_{i=1}^{p_s} \mu_{S,i}(A_{S,i}h_e(y, N) + B_{S,i}N + c_{S,i}) - 1$$
$$\equiv \lambda_{e,0} + \sum_{i=1}^{p_e} \lambda_{e,i}(A_{e,i}y + B_{e,i}N + c_{e,i})$$

Prenons, l'exemple de la Figure 1. :

dd1(i,j) = i pour D1={i,j| j==0} ou D1={i,j| j>=0, -j>=0}
dd2(i,j)=<i, j-1> pour D2={i,j| j>=1}

dd1 spécifie la première dépendance de données entre S1 et S2. En effet, la première itération de S2(i,j) ne peut commencer que si l'itération S1(i) est complétée.
dd2 formule la dépendance entre S2(i,j) et S2(i,j-1). En effet, S2(i,j) a toujours besoin que l'itération précédente soit complétée pour qu'elle puisse s'exécuter.

Donc les fonctions affines de temps de S1 et S2 peuvent être formulées comme suit :

$$\theta_{S1}(i, j) = \mu_{S1,0} + \mu_{S1,1}(i) + \mu_{S1,2}(N - i)$$
$$\theta_{S2}(i, j) = \mu_{S2,0} + \mu_{S2,1}(i) + \mu_{S2,2}(N - i) + \mu_{S2,3}(j) + \mu_{S2,4}(N - j)$$

Pour notre exemple, l'équation X correspondra à :

$$\mu_{S2,0} + \mu_{S2,1}(i) + \mu_{S2,2}(N-i) + \mu_{S2,3}(j) + \mu_{S2,4}(N-j) -$$
$$(\mu_{S1,0} + \mu_{S1,1}(i) + \mu_{S1,2}(N-i) - 1)$$
$$\equiv \lambda_{dd1,0} + \lambda_{dd1,1}(i) + \lambda_{dd1,2}(N-i) + \lambda_{dd1,3}(j) + \lambda_{dd1,4}(N-j) - \lambda_{dd1,5}(j)$$

La simplification de l'équation donnera :

$$\mu_{S2,0} - \mu_{S1,0} - 1 = \lambda_{dd1,0}$$
$$\mu_{S2,1} - \mu_{S2,2} - \mu_{S1,1} + \mu_{S1,2} = \lambda_{dd1,1}(i) - \lambda_{dd1,2}$$
$$\mu_{S2,3} - \mu_{S2,4} = \lambda_{dd1,3} - \lambda_{dd1,4} - \lambda_{dd1,5}$$
$$\mu_{S2,2} + \mu_{S2,4} - \mu_{S1,2} = \lambda_{dd1,2} + \lambda_{dd1,4}$$

Ainsi, durant la résolution du système d'équation linéaire, il s'agira d'éliminer le maximum d'inconnus. Une solution possible serait :

$$\lambda_{dd1,0} = \mu_{S2,0} - \mu_{S1,0} - 1 \geq 0 \qquad (X1)$$
$$\lambda_{dd1,1}(i) = \mu_{S2,1} - \mu_{S2,4} - \mu_{S1,1} - \lambda_{dd1,4} \geq 0 \qquad (X2)$$
$$\lambda_{dd1,3} = \mu_{S2,3} - \mu_{S2,4} + \lambda_{dd1,4} + \lambda_{dd1,5} \geq 0 \qquad (X3)$$
$$\lambda_{dd1,2} = \mu_{S2,2} + \mu_{S2,4} - \mu_{S1,2} - \lambda_{dd1,4} \geq 0 \qquad (X4)$$

Une solution de ce système d'équations linéaires trouverait que :

$$\mu_{S1,0} = \mu_{S1,1} = \mu_{S1,2} = \mu_{S2,1} = \mu_{S2,4} = 0$$
$$\mu_{S2,0} = \mu_{S2,3} = 1$$

Ce qui fait que les fonctions affines de S1 et D2 correspondrait à :

$$\theta(S1,i) = 0$$
$$\theta(S1,i,j) = j + 1$$

Une autre solution serait

$$\theta'(S1,i) = i$$
$$\theta'(S1,i,j) = i + j + 1$$
$$si$$
$$\mu_{S2,1} = \mu_{S1,1} = 1$$

Les fonctions affines de l'équation X permettent de voir que les énoncés *S1(i)* peuvent s'exécuter en parallèle puisque la fonction affine ne dépend pas des indices d'itération *i* et *j*. D'un autre côté, on voit que les énoncés *S2(i,j)* ne dépendent que de l'indice *j*. Donc, les énoncés d'itérations d'une même ligne sont dépendants, mais ceux de lignes différentes (ayant un *i* différent) peuvent s'exécuter en parallèle. La figure 2 illustre le code parallèle correpondant à l'ordonnancement trouvé.

```
  doAll i=0 to N
        S1: s(i) =0;
end doAll

for j=0 to N loop
        doAll i=0 to N
              S2: s(i,j)=s(i,j)+a(i,j)*x(j)
        end doAll
end loop
```

Figure 2. Code résultant d'un ordonnancement