Custom Operator Identification for High-level Synthesis

Chenglong Xiao

N° d'ordre : 2012REN1E005 ANNÉE 2012

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

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Traitement du Signal et Télécommunications

Ecole doctorale MATISSE

présentée par

Chenglong XIAO

préparée à l'unité de recherche UMR6074 IRISA
Institut de recherche en informatique et systèmes aléatoires - CAIRN
École Nationale Supérieure des Sciences Appliquées et de Technologie

---

Custom Operator Identification for High-Level Synthesis

Thèse soutenue à LANNION
le 8 Novembre, 2012

devant le jury composé de :

Christophe JEGO
Professeur des Universités,
IPB/ENSEIRB-MATMECA / rapporteur

Philippe COUSSY
Maître de Conférences,
Université de Bretagne Sud / rapporteur

Daniel MÉNARD
Professeur des Universités,
INSA Rennes / examinateur

Thierry MICHEL
STMicroelectronics / examinateur

Olivier SENTIEYS
Professeur des Universités,
Université de Rennes 1 - ENSSAT / examinateur

Emmanuel CASSEAU
Professeur des Universités,
Université de Rennes 1 - ENSSAT / directeur de thèse

Acknowledgments

First of all, I would like to thank my advisor professor Emmanuel Casseau for his invaluable guidance. His rigorous attitude towards scientific research and his patience to students have always impressed me. Without his constructive comments and suggestions, this thesis would not have been possible.

I wish to thank the members of my thesis committee for spending their precious time to read and evaluate my dissertation: Christophe Jego (professor at IPB/ENSEIRB-MATMECA), Philippe Coussy (associate professor at Université de Bretagne Sud, Daniel Ménard(professor at INSA Rennes, Thierry Michel (engineer at STMicroelectronics Crolles), Olivier Sentieys (professor at Université de Rennes 1).

I also would like to thank all the members of the IRISA/CAIRN team. In particular, I would like to thank Dr. Kevin Martin and Dr. Antoine Floch for their help during my PhD. With their high-quality implementation work, my PhD work goes on faster and more smoothly.

Most importantly I would like to thank my mother Jinfeng Cai and my wife Shanshan Wang. Without their never ending support and encourage, this thesis would not be possible. This thesis is dedicated to my son Yao Xiao who is always the source of my happiness.

Custom Operator Identification for High-level Synthesis

Abstract: It is increasingly common to see custom operators appear in various fields of circuit design. Custom operators that can be implemented in special hardware units make it possible to reduce code size, improve performance and reduce area. In this thesis, we propose a custom operator based high-level synthesis design flow. The key issues involved in the design flow are: automatic enumeration and selection of custom operators from a given high-level application code and re-generation of the source code incorporating the selected custom operators.

However, automatic enumerating all the subgraphs is computationally difficult problem. In this thesis, we propose three enumeration algorithms for exact enumeration of subgraphs under various constraints. Compared to a previously proposed well-known algorithm, the proposed enumeration algorithms can achieve orders of magnitude speedup.

Selecting a most profitable subset from the enumerated subgraphs is also a time-consuming job. In this thesis, we present three different selection heuristics targeting different objectives. In addition, a branch-and-bound approach and a genetic algorithm are introduced to select the minimal number of matches that fully cover the data flow graph of a given application code. The greedy approaches are very efficient, but they may produce results that are sub-optimal. While the exact algorithms guarantee the optimum of solutions, but they fail to give solution in an affordable time in most situations. The proposed genetic algorithm makes trade-off between the two. The runtime and the quality of solutions can be controlled by some user-specified parameters.

During the code re-generation step, a graph isomorphism is required to group the structurally and functionally equivalent subgraphs that can be implemented with the same custom operator. An extended graph isomorphism algorithm that captures the characteristics of data-flow graph is proposed to determine the similarity between graphs after selection. Some specific problems existing in the graphs isomorphism check for data-flow graph are depicted in this thesis. Corresponding solutions for those specific problems are provided. To our knowledge, these were never mentioned in the previous literature.

We have developed and implemented a complete design flow for pattern based high-level synthesis. Unlike the previously proposed approaches, our design flow is quite adaptable and is independent of high-level synthesis tools (i.e., without modifying the scheduling and binding algorithms in high-level synthesis tools). Experimental results show that our approach achieves on average 19%, and up to 37% area reduction, compared to a traditional high-level synthesis. Meanwhile, the latency is reduced on average by 22%, and up to 59%. Furthermore, on average 74% and up to 81% code size reduction can be achieved.

Keywords: custom operator, subgraph enumeration algorithm, subgraph selection algorithm, code transformation, high-level synthesis

Identification d'opérateurs spécifiques pour la synthèse de haut niveau

La complexité croissante des applications à intégrer a conduit à concevoir les circuits à un haut niveau d'abstraction. A ce titre, par rapport à une synthèse classique de niveau dit " transfert de registres " (RTL), la synthèse de haut niveau (HLS) permet d'envisager une meilleure productivité. Par ailleurs, il est de plus en plus fréquent dans les différents domaines de la conception de circuits de faire appel à des opérateurs spécifiques, opérateurs qui permettent de réduire la taille du code, d'améliorer les performances ou de réduire la surface d'un circuit. Dans cette thèse, nous proposons un flot de conception basé sur l'identification d'opérateurs spécifiques pour la synthèse de haut niveau.

Synthèse de haut niveau

On constate depuis une vingtaine d'années que les applications à mettre en œuvre sont de plus en plus complexes. Elever le niveau d'abstraction permet de réduire considérablement le temps de conception. Aussi, la synthèse de haut niveau s'avère être de plus en plus utilisée dans les flots de conception de systèmes électroniques De nombreux outils commerciaux de synthèse de haut niveau ont été proposés par plusieurs fournisseurs. On trouve par exemple Autopilote, CatapultC, CTOS, Cynthesizer, SynphonyC et Cyber-Workbench. En outre, beaucoup d'outils de synthèse de haut niveau ont également été proposés par des universitaires (par exemple, Legup, GAUT, Trident et SPARK).

La synthèse de haut niveau, parfois appelée synthèse comportementale, est un processus de conception automatisée permettant de transformer des spécifications (par exemple, décrite en C, C + + ou SystemC) en des spécifications de bas niveau (transfert de registre) qui implémentent le comportement spécifié tout en satisfaisant les contraintes de conception. Les outils HLS acceptent des spécifications de haut niveau comme entrée. En général, la plupart des outils actuels de HLS commerciaux utilisent le langage C en entrée. Nous pouvons également trouver des outils HLS utilisant les langages tels BlueSpec, Esterel and MATLAB. En plus des spécifications de haut niveau, une bibliothèque de ressources contenant les informations détaillées sur les composants matériels ainsi que les contraintes de conception spécifiques sont fournies à l'outil HLS. A partir de ces points d'entrées, l'outil HLS effectue les tâches suivantes et produit en sortie une description de matérielle qui permet d'implémenter la spécification:

"Front End " de compilation: Les spécifications sont analysés et traduites en représentations intermédiaires. Plusieurs transformations préliminaires ou des optimisations de code, telles que l'élimination de " code mort ", l'élimination de fausses dépendances de données, l'équilibrage de branches, la propagation de constantes, des transformations de boucles, et l'élimination de sous-expressions communes sont réalisées au plus tôt de cette étape. Après ces optimisations, les spécifications sont transformées en une représentation intermédiaire appropriée. Afin de capturer à la fois les dépendances de données et les dépendances de contrôle entre les opérations de la spécification, les représentations intermédiaires (IR) qui conservent les informations présentées dans la spécification d'entrée sont utilisées. Le graphe de flot de contrôle et de données (CDFG) est considéré comme l'une des plus populaires IR. Le CDFG est un graphe orienté dans lequel un nœud peut être soit une opération soit un bloc de base. Les arcs dans un CDFG représentent le

transfert d'une valeur ou une commande d'un nœud à un autre. A l'intérieur de chaque nœud, un graphe flot de données est utilisé pour capturer les dépendances de données entre opérations. En outre, les dépendances de données entre les blocs de base peuvent être exprimées à l'aide des graphes de tâches hiérarchiques.

Allocation: Ce processus détermine les types de composants matériels nécessaires pour la mise en œuvre du matériel ainsi que leur quantité. Les composants matériels peuvent être des unités fonctionnelles pour les opérations, des registres pour stocker des valeurs, des bus et des multiplexeurs pour interconnections.

Ordonnancement: L'ordonnancement est le processus qui attribue des cycles d'horloge ou des pas de temps aux opérations afin que les contraintes temporelles soient satisfaites. Chaque opération est ordonnancée en fonction des dépendances de données et de contrôle entre opérations. Il existe plusieurs algorithmes d'ordonnancement: dès que possible (ASAP), le plus tard possible (ALAP), l'ordonnancement par liste, force dirigée et la programmation linéaire entière. L'ordonnancement et l'allocation des ressources sont généralement interdépendants. L'ordonnancement est effectué par rapport à des contraintes de ressources (celles qui sont affectées à l'étape d'allocation), tandis que l'allocation des ressources peut être améliorée si les opérations qui peuvent être exécutées en parallèle sont connus à l'avance (ces informations peuvent être obtenues à partir de l'ordonnancement).

Projection: Le processus de projection affecte les unités fonctionnelles aux opérations afin de pouvoir les exécuter, des registres pour stocker des valeurs qui circulent à travers les bus / multiplexeurs qui eux implémentent les transferts de données / contrôle. L'algorithme de projection décide de quelle unité fonctionnelle doit être utilisée pour effectuer une opération spécifique quand il y a plus d'une unité fonctionnelle capable d'exécuter l'opération au cycle en question. Les opérations et les données de durée de vie mutuellement exclusives peuvent partager le même composant matériel.

Génération de code: À la fin du processus de synthèse, la génération de code permet de produire une description de niveau RTL, qui inclus un chemin de données et une unité de commande. Le chemin de données se réfère aux unités fonctionnelles tels que des additionneurs, des multiplieurs, des unités arithmétiques et logiques (UAL) etc., des unités de mémorisation tels que des mémoires et des registres ainsi que les composants de communication/interconnexion tels que les bus et les multiplexeurs. L'unité de commande est une machine à états finis (FSM) qui génère des signaux de commande et contrôle du flot de données.

Opérateurs spécifiques

Il est fréquent de faire usage d'opérateurs spécifiques dans divers domaines de la conception de circuits. A titre d'exemple, l'usage d'opérateurs spécifiques permet de faire des compromis entre flexibilité et efficacité avec les processeurs extensibles. Un opérateur spécifique est composé d'un ensemble d'opérations de base (par exemple, des opérations arithmétiques de base telles que les additions, soustractions et multiplications). Une unité fonctionnelle spécifique implémente un opérateur spécifique.

Le premier avantage apporté par l'utilisation d'opérateurs spécifiques est la réduction importante du code source. En règle générale, l'opérateur spécifique encapsule plusieurs opérations de base et les assemble en un seul opérateur plus complexe. Ainsi, la taille du

code source peut être réduite, et le niveau de granularité du code source est augmenté. Les outils de synthèse de haut niveau peuvent alors produire des solutions plus rapidement avec le code réduit.

Avec les opérateurs spécifiques, des améliorations en terme de performances peuvent être obtenues. Les opérateurs spécifiques améliorent les performances de quatre façons possibles. Tout d'abord, les opérations de base à l'intérieur de l'opérateur spécifique sont chaînées selon les dépendances de données (chaînage d'opération). Deuxièmement, les opérations de base sans dépendance des données peuvent être parallélisées. Troisièmement, certaines techniques d'optimisation peuvent être appliquées pour réduire le chemin critique. Enfin, la mise en œuvre matérielle d'un opérateur spécifique est généralement plus rapide (chemin critique) que si on utilise des opérateurs de base (par exemple, le temps de latence d'une multiplication-accumulation (MAC) est inférieur à la somme de la latence d'un multiplieur et de la latence d'un additionneur).

En outre, les opérateurs spécifiques peuvent conduire à des gains en surface à travers les deux aspects suivants. Tout d'abord, en général, les flots de données internes d'un opérateur spécifique sont exempts de multiplexeurs. Deuxièmement, la mise en œuvre matérielle d'un opérateur spécifique conduit à une surface plus faible (par exemple, la surface d'un MAC est inférieure à la somme de la surface d'un multiplieur et la surface d'un additionneur). Il est à noter cependant que, en utilisant les opérateurs spécifiques, on peut parfois aboutir à un partage moins efficace des ressources: on trouve en effet plus d'instances d'opérations élémentaires dans un graphe d'origine que d'instances d'opérations spécifiques dans un graphe réduit.

Nous proposons un flot de conception tirant partie d'opérateurs spécifiques en amont de la synthèse de haut niveau et permettant de transformer le code source en lui incorporant des opérateurs spécifiques, afin d'améliorer les résultats des outils de synthèse de haut niveau. Contrairement aux approches précédentes de la littérature qui nécessitent de modifier les algorithmes d'ordonnancement et d'allocation des outils de synthèse de haut niveau, notre flot de conception est totalement indépendant de ces outils. Notre flot de conception peut ainsi être adapté à de nombreux outils de synthèse de haut niveau commerciaux.

Le flot de conception est composé de quatre étapes principales. Le point de départ est le code source de haut niveau (langage C par exemple). Dans la première étape, le code source est transformé en un graphe de flot de contrôle et de données (CDFG) en utilisant un compilateur open source GECOS. Ensuite, les sous-graphes possibles sont identifiés par un algorithme d'énumération à partir des graphes DFG (graphe flot de données) qui correspondent chacun à un bloc de base du CDFG. Puis un sous-ensemble des sous-graphes identifiés est sélectionné en fonction de différentes stratégies (nombre minimum d'opérateurs spécifiques, taille minimum de code, etc.). Enfin, le code source d'origine est transformé en un nouveau code source en intégrant les sous-graphes sélectionnés (les sous-graphes sélectionnés seront mis en œuvre en tant qu'opérateurs spécifiques). Le nouveau code source faisant appel aux opérateurs spécifiques est alors fourni en entrée de l'outil de synthèse de haut niveau ciblé.

Représentation intermédiaire: La spécification comportementale d'entrée de la synthèse de haut niveau est généralement composée d'une liste de déclarations séquentielles.

Les déclarations peuvent être des expressions de type opération, des structures conditionnelles et des boucles. Pour exprimer les dépendances de données dans la description d'entrée, le graphe flot de données est un bon candidat. Comme le graphe flot de données ne contient que des dépendances de données explicites, on peut faire appel à un graphe de flot de contrôle pour exprimer le contrôle d'une spécification d'entrée. Afin d'exprimer ces deux informations à la fois, les graphes de flot de données et de contrôle sont souvent utilisés et constituent une bonne représentation intermédiaire pour la synthèse de haut niveau.

Énumération de sous-graphes: L'efficacité de l'énumération de sous-graphes est meilleure si l'identification multiple d'un même sous-graphe peut être évitée. La plupart des travaux antérieurs conduisent à identifier plusieurs fois un même sous-graphe. Le temps d'exécution est alors inutilement augmenté. Dans cette thèse, nous présentons une approche efficace qui évite les identifications multiples de sous-graphes par une technique de suppression de nœuds lors de l'énumération des sous-graphes sous contrainte de taille. Dans cette thèse, nous présentons également un algorithme très flexible pour l'énumération exacte des sous-graphes sous contrainte d'entrées/sorties. L'algorithme est basé sur notre algorithme d'énumération sous contrainte de taille. L'algorithme peut être spécialisé pour générer tous les sous-graphes possibles ou seulement les sous-graphes connectés. Dans cette thèse, nous proposons également un nouvel algorithme pour l'énumération de sous-graphes sous contrainte d'entrées/sorties. Notre algorithme permet de résoudre le problème de manière efficace en profitant de la propriété topologique d'un graphe flot de données (DFG).

Sélection de sous-graphes: Diverses stratégies peuvent être utilisées pour guider la sélection sous-graphes. Dans cette thèse, nous nous concentrons sur trois stratégies différentes. Tout d'abord, la taille du code peut être un objectif d'optimisation important. En conception de systèmes embarqués, seule une petite quantité de mémoire est en général disponible pour stocker les instructions. Dans le contexte de la synthèse de haut niveau, la sélection du plus petit ensemble de sous-graphes conduit au code le plus compact. L'espace de conception est ainsi réduit, l'outil de synthèse de haut niveau peut alors produire des résultats en moins de temps. Ensuite, la fréquence d'occurrences d'un motif renseigne sur le partage possible des ressources. Une méthode de sélection basée sur la fréquence d'occurrences de motifs est présentée (basée sur le nombre d'instances de motifs). Enfin, la sélection de sous-graphes peut conduire à augmenter la longueur des chemins critiques. Ainsi, une sélection de sous-graphes qui prend en compte le surcoût en terme de latence est très important. Une méthode de sélection basée sur le chemin critique est également proposée dans cette thèse.

Les algorithmes de résolution exacte sont fortement consommateurs en temps d'exécution et échouent généralement à donner un résultat en raison d'un temps d'exécution trop long ou d'un débordement mémoire. Par exemple, un algorithme de type " branch-and-bound " requiert 20 secondes pour trouver le nombre minimum d'instances de motifs qui recouvrent entièrement le graphe d'un produit de vecteurs lorsque le graphe correspondant est un " petit graphe " qui contient seulement 8 nœuds. Lorsqu'on augmente la taille des vecteurs, cet algorithme exact ne parvient pas à produire de résultats en moins d'une heure lorsque le graphe contient 12 nœuds. Aussi, une approche heuristique efficace est nécessaire. Dans cette thèse, trois algorithmes de type heuristique visant des

objectifs différents sont représentés. Bien que l'algorithme exact est grand consommateur en temps comparé aux algorithmes heuristiques ou algorithmes gloutons, nous présentons cependant aussi un algorithme " branch-and-bound " pour connaitre le nombre minimum d'instances de motifs dans un but de comparaison.

Le problème de sélection de motifs peut être transformé en un problème de couverture. Le problème de couverture est un problème d'optimisation combinatoire classique, problème pour lequel les algorithmes génétiques sont bien adaptés, en particulier lorsque le problème est un problème NP-complet. Par conséquent, nous avons également essayé d'appliquer un algorithme génétique pour résoudre le problème de sélection de motifs dans cette thèse.

Transformation de code: Après l'obtention d'un ensemble de sous-graphes produits par l'étape d'énumération de sous-graphes, l'étape de sélection sous-graphes nous délivre un ensemble des sous-graphes sélectionnés. Il est nécessaire de déterminer si deux sous-graphes sélectionnés peuvent être exécutés par une même unité fonctionnelle spécifique (cela se fait en pratique avant la sélection). Cette tâche peut être considérée comme un problème d'isomorphisme de graphes. Nous avons développé un algorithme d'isomorphisme de graphes. Notre algorithme est en fait une extension de l'algorithme d'isomorphisme de graphes VF2. Nous avons étendu l'algorithme VF2 en analysant certaines caractéristiques du graphe flot de données. L'algorithme VF2 trouve la correspondance entre deux graphes en comparant graduellement des paires de nœuds. En général, les correspondances partielles sont étendues à des correspondances partielles plus grandes en ajoutant une paire de nœuds voisins compatibles. Une paire de nœud est dite compatible uniquement lorsque les deux nœuds satisfont un ensemble de règles de faisabilité. L'ensemble des règles de faisabilité permet de réduire efficacement l'espace de recherche. Nous avons amélioré l'algorithme VF2 en ajoutant une vérification de cardinalité sur les sommets, les arêtes et le nœud de départ afin de rejeter rapidement des graphes différents. De plus, l'algorithme propose résout le problème causé par les opérations non commutatives.

Après avoir réalisé la projection des sous-graphes fonctionnellement équivalents sur des opérateurs spécifiques identiques, l'ensemble des nœuds à l'intérieur d'un sous-graphe est remplacé par un super nœud. Le super nœud correspond donc au sous-graphe sélectionné. Afin de ne pas perdre la sémantique du code d'origine, les super nœuds contiennent toutes les informations correspondant aux sous-graphes remplacés. phase de régénération de code a pour rôle de traduire correctement ce super nœud en code équivalent.

Une fois les sous-graphes remplacés par des super nœuds, un pragma spécifique peut être inclus dans le nouveau code source généré pour chaque opérateur spécifique. L'opérateur est alors présenté comme une fonction. Avec ce pragma spécifique, les outils de synthèse de haut niveau (par exemple, CatapultC (Mentor Graphics)) ordonnancera et projettera les opérateurs spécifiques exactement comme ils le font pour les opérateurs de base. Pour l'outil de synthèse de haut niveau toutes les fonctions non-inlinées sont considérées comme des opérateurs spécifiques par défaut (le pragma n'est donc pas nécessaire).

Contributions:

Dans cette thèse, nous développons un flot de synthèse de haut niveau basé sur l'utilisation d'opérateurs spécifiques. Les points clés de ce flot de conception sont les

suivants: énumération automatique et sélection des opérateurs spécifiques à partir d'une application spécifiée à haut niveau, et régénération d'un code source intégrant les opérateurs spécifiques sélectionnés. Cette thèse apporte les contributions suivantes:

Énumération de sous-graphes: Comme le nombre de sous-graphes dans un DFG est une fonction exponentielle du nombre de nœuds du DFG, l'énumération de sous-graphes est un problème difficile. Par exemple, dans un DFG provenant du benchmark de spécification GSM qui comporte 490 nœuds, le nombre de sous-graphes possibles vaut 341.641 lorsque les contraintes d'entrées et de sorties (E/S) sont 4 et 2 (au plus) respectivement. Lorsque les contraintes d'E/S sont 6/2, le nombre de sous-graphes passe à 1.454.539. Nous présentons trois algorithmes évolutifs d'énumération de sous-graphes qui permettent d'énumérer tous les sous-graphes de manière très efficace sous des contraintes de conception différentes. Pour énumérer les sous-graphes sous contrainte de taille, nous proposons un nouvel algorithme qui permet d'éviter les identifications multiples de sous-graphes. Sur la base de l'algorithme proposé sous contrainte de taille, un algorithme étendu qui vise à énumérer tous les sous-graphes possibles ou seulement sous-graphes connectés est ensuite présenté. En outre, nous présentons un algorithme au temps de calcul polynomial qui consiste en une amélioration d'un algorithme de référence pour énumérer tous les sous-graphes possibles, à savoir les sous-graphes disjoints et les sous-graphes connectés. Les expériences montrent que nos algorithmes permettent de gagner jusqu'à 2 décades en terme de temps de calcul par rapport à l'algorithme de référence.

Sélection de sous-graphes: La sélection d'un sous-ensemble de l'ensemble des sous-graphes générés par l'énumération de sous-graphes est également un problème coûteux en temps. Notre objectif est donc de proposer des algorithmes efficaces en temps. Trois approches heuristiques différentes de sélection de sous-graphes visant des objectifs différents sont présentées et une comparaison de ces algorithmes est faite dans la partie expériences. En outre, nous présentons un algorithme exact ainsi qu'un algorithme génétique pour la sélection d'un nombre minimum d'instances de motifs qui couvrent le graphe de l'application. L'algorithme génétique proposé est un compromis entre l'approche gloutonne et l'approche exacte. En d'autres termes, il réalise un compromis entre le temps d'exécution de la sélection et la qualité des résultats de la sélection.

Transformation de code: Nous avons étendu un algorithme existant d'isomorphisme de graphes afin de rejeter rapidement des graphes différents en tirant partie de certaines caractéristiques des graphes flots de données. Certains problèmes spécifiques que pose la vérification d'isomorphisme de graphes et de sous-graphes sont d'abord exposés. À notre connaissance, les problèmes mentionnés n'ont jamais été considérés dans la littérature. Les solutions correspondantes pour répondre aux problèmes soulevés sont fournies dans cette thèse. En outre, une présentation du code généré utilisant les opérateurs spécifiques est également faite.

Flot de conception automatisé: un flot de conception automatisé et adaptable basé sur l'utilisation d'opérateurs spécifiques pour la synthèse de haut niveau a ainsi pu être mis en œuvre. Les résultats pour un ensemble d'applications de référence montrent que la solution proposée permet d'obtenir une réduction significative la taille du code, une réduction de la surface et une diminution de la latence du circuit généré.

Contents

List of Figures

List of Tables

List of Acronyms

Acronyms

ALAP	As Late As Possible
ALU	Arithmetic Logic Unit
ASAP	As Soon As Possible
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction Set Processor
CDFG	Control Data Flow Graph
CSE	Common Sub-expression Elimination
DAG	Directed Acyclic Graph
DFG	Data Flow Graph
EDA	Electronic Design Automation
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GA	Genetic Algorithm
GPP	General Purpose Processor
HLS	High-Level Synthesis
HTG	Hierarchical Task Graph
ILP	Integer Linear Programming
IR	Intermediate Representation
MAC	Multiply-Accumulate
MAXMIMO	Maximum Multiple-Input-Multiple-Output
MIMO	Multiple-Input-Multiple-Output
MISO	Multiple-Input-Single-Output
RTL	Register-Transfer Level
SCP	Set Covering Problem

List of Notations

Notations

IN(G,M)	The Number of Inputs of a Subgraph
PerInput(G,M)	Permanent Inputs of a Subgraph
OUT(G,M)	The Number of Outputs of a Subgraph
PerOutput(G,M)	Permanent Outputs of a Subgraph
IN_{max}	Maximal Number of Inputs
OUT_{max}	Maximal Number of Outputs
Succ(G,n)	All the Successors of a Node
Succ(G,M)	All the Successors of a Subgraph
ISucc(G,n)	Immediate Successors of a Node
ISucc(G,M)	Immediate Successors of a Subgraph
Pred(G,n)	All the Predecessors of a Node
Pred(G,M)	All the Predecessors of a Subgraph
IPred(G,n)	Immediate Predecessors of a Node
IPred(G,M)	Immediate Predecessors of a Subgraph
Btw(n, M)	The Nodes between a Node and a Subgraph
Disc(G,n)	Disconnected Nodes of a Node
Disc(G,M)	Disconnected Nodes of a Subgraph

Chapter 1

Introduction

The rapid growing size and complexity of the applications to be implemented has led to performing designing at a higher-level abstraction. Compared to register-transfer level (RTL) synthesis, high-level synthesis (HLS) may achieve better productivity. In recent years, it is increasingly common to see custom operators, which make it possible to reduce code size, improve performance and reduce area, appear in various fields of circuit design. In this thesis, we develop a custom operator based high-level synthesis flow in benefit of the advantages of custom operators.

First of all, we give a brief introduction on high-level synthesis. Then, we discuss the advantages achieved by using custom operators. Next, we present the proposed custom operator based high-level synthesis flow. Finally, the contributions and organizations of this thesis are presented.

1.1 High-Level Synthesis

In the past decades, the applications to be implemented are becoming more and more complex. As raising the design abstraction level allows to reduce the design time substantially, the use of high-level synthesis [McFarland 1988] has been increased in the electronic design automation (EDA) community. Many commercial high-level synthesis tools have been proposed by several vendors. Autopilot[Zhang 2008, Cong 2006], CatapultC [Bollaert 2008], CtoS [Bailey 2010, Cadence], Cynthesizer [Meredith 2008], Synphony-C [Synopsys , Kathail 2002] and CyberWorkbench [Kazutoshi 2008, Wakabayashi 2006] are only some of the existing high-level synthesis tools. In addition, plenty of academic high-level synthesis tools have also been introduced in recent years (e.g., Legup[Canis 2011], GAUT[Coussy 2008], Trident[Tripp 2007, Tri 2005] and

SPARK[Gupta 2003, Gupta 2004]).

The high-level synthesis, sometimes referred to behavior synthesis, is an automated design process of transforming untimed or partially timed specifications (e.g., C, C++ or SystemC) to low-level cycle-accurate register-transfer level specifications that implements the specified behavior while satisfying the design constraints. Fig. 1.1 shows an overview of high synthesis flow. The HLS tools accept high-level specifications as inputs. In general, most of current commercial HLS tools take C-based specification as design entry [Cong 2011a]. We can also find some HLS tools using other input languages such as BlueSpec [BlueSpec], Esterel[Edwards 2002] and MATLAB[Malay 2001]. In addition to the high-level specifications, the resource library containing the detail information of hardware components and specific design constraints are provided to HLS tool at the beginning [Coussy 2009]. With the provided inputs, the HLS tool carries out the following tasks and produces a hardware description language that implements the specification:

• Compilation Front End: The specifications are parsed into intermediate representations. Several preliminary transformations or code optimizations [Muchnick 1997, Gupta 2004] such as dead-code elimination, false data dependency elimination, branch balancing, constant propagation, loop transformations, speculative code motion and common subexpression elimination are performed at the earlier stage of this step [Coussy 2009]. After the optimizations, the specifications are transformed to an appropriate intermediate representation. In order to capture both the data dependencies and control dependencies between the operations in the specifications, different intermediate representations (IR) that retain all the information presented in the input specification are used. Control data flow graph (CDFG) is considered as one of the most popular IR. The CDFG is a directed graph in which a node can be either an operation or a basic block. The directed edges in a CDFG represent the transfer of a value or control from one node to another. Inside each node, a data flow graph is used to capture the data dependencies between operations. In addition, the data dependencies between basic blocks can be captured by using the hierarchical task graphs (HTGs) representation proposed in [Gupta 2003, Gupta 2004].

• Allocation: This process determines the types of hardware components and the number for each type to be included in the hardware implementation. The hardware components refer to the function units (such as adders, multipliers) for operations, the registers for storing values, the buses and the multiplexors for interconnections between

operators.

• Scheduling: Scheduling is the process of assigning operations to clock cycles or time steps so that the design constraints are satisfied. Each operation is scheduled according to the data dependencies and control dependencies between the operations. There are several commonly preferred scheduling algorithms: as soon as possible (ASAP) scheduling, as late as possible (ALSP) scheduling, list scheduling, force directed scheduling and integer linear programming formulation. The scheduling and resource allocation are usually interdependent. Scheduling is performed with respect to the resources constraints that are assigned in the allocation step, while the resource allocation can be improved if the operations that can be executed parallel are known in advance (these information can be obtained from scheduling) [McFarland 1988].

• Binding: The binding process assigns the function units to perform operations, registers to store values that pass across cycles and buses/multiplexors to realize the data/control transfers. The binding algorithm decides which function unit should be used to perform a specific operation when there are more than one functional units capable of executing the operation. The operations and the values with mutually exclusive lifetime can share the same hardware component.

• Code Transformation: At the end, the code transformation produces a RTL implementation including a data path and a control unit. The data path refers to function units such as adders, multipliers and arithmetic logic units (ALUs), storage units such as memories and registers, and connectivity components such as buses and multiplexors. The control unit is a finite state machine (FSM) that generates control signals and controls the data flow into the data logic path.

1.2 Custom Operators

Nowadays, it is common to find custom operators in various fields of circuit design. As an example, custom operator is a vital component to make trade-offs between flexibility and efficiency in extensible processors [Gonzalez 2000]. A custom operator is composed of a cluster of basic operations (e.g., primitive arithmetic operations such as add, subtract and multiply). A custom function unit is the hardwired implementation of a custom operator.

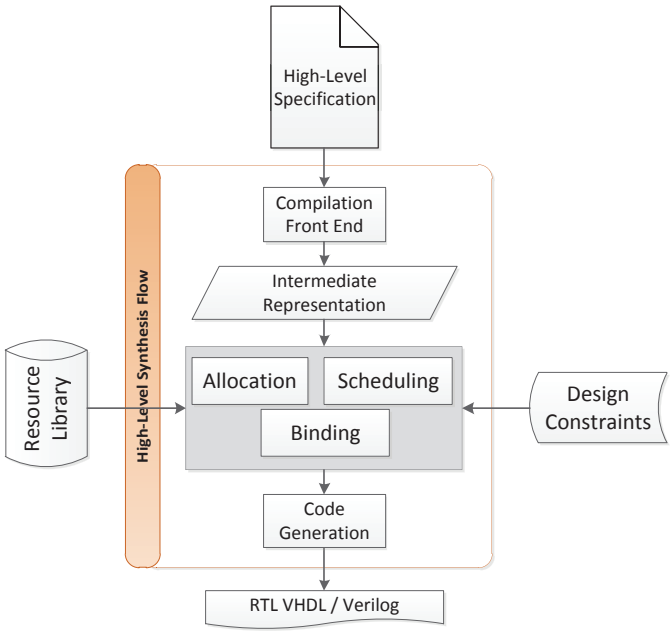


Figure 1.1: High-level synthesis flow

The first benefit brought by using custom operators is the significant compactness of the source code. Generally, the custom operator encapsulates several basic operations and compacts them into one complex operator. Thus, the size of the source code can be reduced and the granularity level of the source code is increased. Consequently, the high-level synthesis tools may give solutions in a shorter time with the compacted code.

With custom operators, considerable performance improvement can be achieved. The custom operators improve performance through four possible ways. First, the basic operations inside the custom operator are automatically chained according to the data dependencies (operation chaining). Second, the basic operations without data dependencies between each other may be parallelized (parallelization). Third, some optimization techniques can be applied to reduce the critical path of custom operators (Fig. 1.2 shows an example of reducing the critical path of a selected custom operator. Let assume each addition takes 1 clock cycle to execute. The custom operator in Fig. 1.2 (a) requires 3 clock cycles to execute, while the custom operator with reduced critical path in Fig. 1.2 (b) only takes 2 clock cycles to execute. The critical path is reduced by imposing parallelism between the two sequential additions). Finally, the dedicated hardware implementation of a custom operator is usually faster than the basic operators (e.g., the latency of a

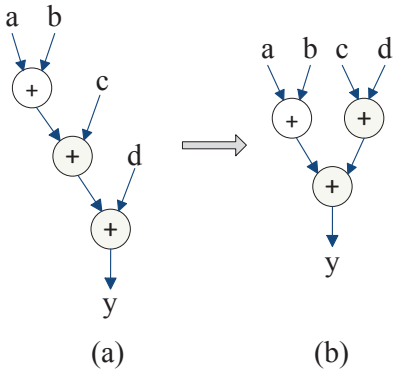


Figure 1.2: (a) A selected custom operator (b) A custom operator with reduced critical path

multiply-accumulate (MAC) is less than the sum of the latency of a multiplier and the latency of an adder [Yadav 1999]).

Moreover, the custom operators can reduce the area through the following two aspects. First, in general, the internal data flows of a custom operator are free of multiplexors [Cong 2008, Cong 2010, Cong 2011b]. Second, the dedicated hardware implementation of a custom operator leads to less area cost (e.g., the area of a MAC is less than the sum of the area of a multiplier and the area of an adder [Yadav 1999]). It is noteworthy that, using custom operators may lead to less resource sharing: more basic operations are found in a graph compared to custom operations.

1.3 An Overview of the Design Flow

With the benefits of using custom operators in circuit designs, we propose an automated custom operator based pre-synthesis design flow to transform the source code by incorporating custom operators, such that better synthesis results can be obtained with high-level synthesis tools. Fig. 1.3 illustrates our proposed framework of the custom operator based pre-synthesis design flow. Unlike the previous frameworks in the literature that require to modify the scheduling and binding algorithms of high-level synthesis tools [Cong 2008, Cong 2010, Cong 2011b], our design flow is fully independent of high-level synthesis tools. The design flow can be adapted to many commercial high-level synthesis tools.

The design flow consists of four major steps. The starting point is the high-level

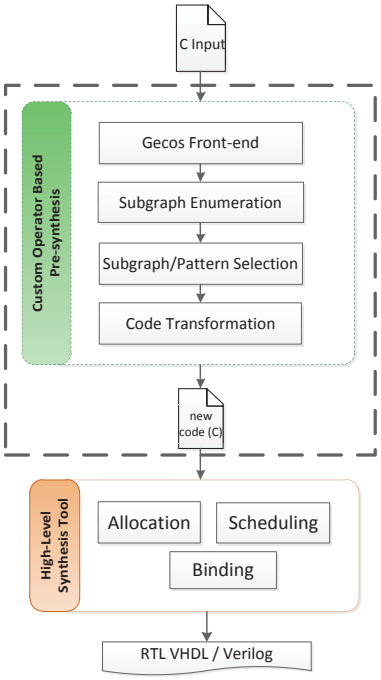


Figure 1.3: Custom operator Identification for high-level synthesis

source code (C language). In the first step, the source code is translated to a control data flow graph (CDFG) using an open source compiler GECOS [GECOS]. A CDFG is a graph that represents the data dependencies between a number of basic blocks. Next, the potential subgraphs are identified by the subgraph enumeration algorithm from the DFG corresponding to a basic block of the CDFG. Then a subset of the identified subgraphs are selected according to different strategies (minimum number of custom operators, minimum code size etc.). Finally, the original source code is transformed to a new source code by incorporating the selected subgraphs (the selected subgraphs will be implemented as custom operators). The new source code with custom operators is then provided as input for high-level synthesis tool. More straightforwardly, a summary of our approach with a simple example to custom operator based high-level synthesis design flow is shown in Fig. 1.4. In the figure, a piece of C code is provided as input. It is transformed to a DFG by GECOS. All the subgraphs (connected subgraphs) in the DFG are enumerated. After subgraph enumeration, two subgraphs are selected (we assume M_1 and M_2 are selected). Based on the graph with selected subgraphs, a piece of new source code is generated. The new source code is then provided as input to the high-level synthesis tool.

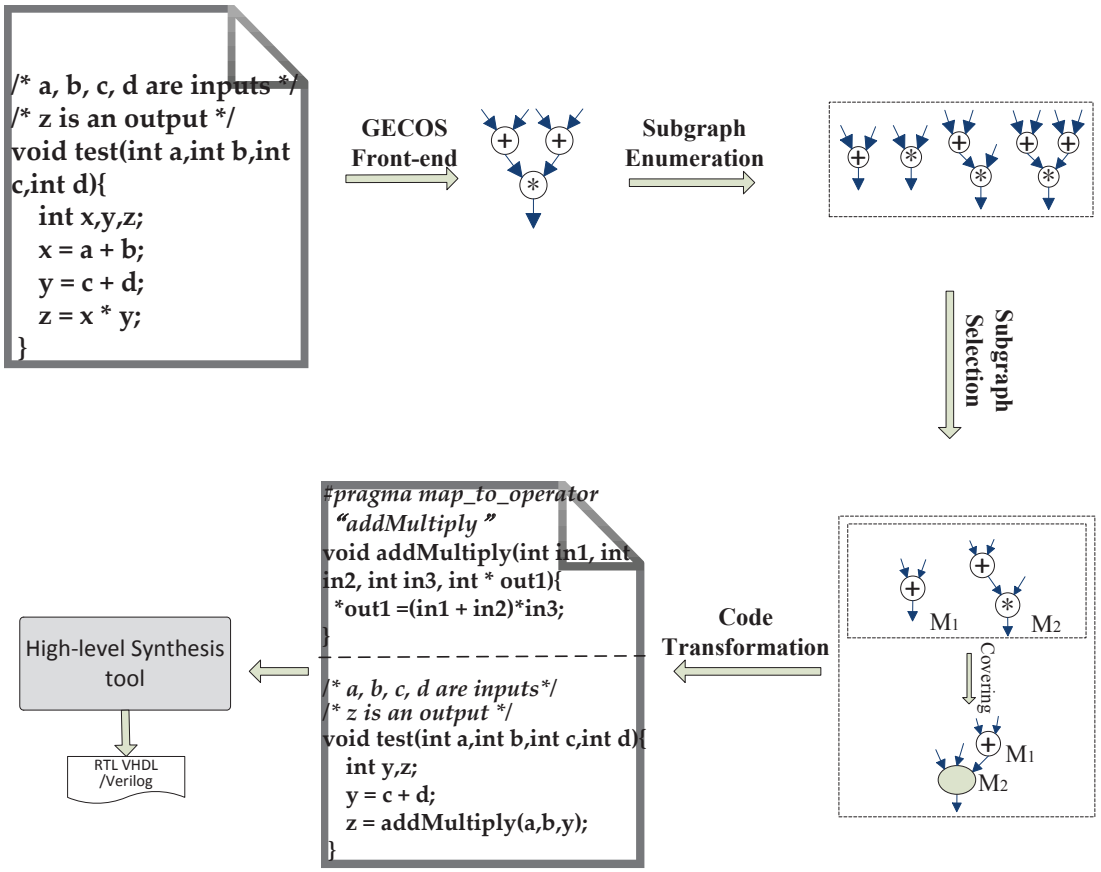


Figure 1.4: Summary of our approach to custom operator based high-level synthesis flow

1.4 Contributions of this Thesis

In this thesis, we develop a new custom operator (pattern) based high-level synthesis design flow. The key issues involved in the design flow are : automatic enumeration and selection of custom operators from a given high-level application and regeneration of source code incorporating the selected custom operators. This thesis makes the following contributions:

• Subgraph Enumeration: As the number of subgraphs in a DFG is exponential to the number of nodes of the DFG, subgraph enumeration is a computationally difficult problem. For example, in a DFG from a real world benchmark GSM that has 490 nodes, the number of subgraphs can be 341641 when input and output constraints (I/O constraints) are 4 and 2. When the I/O constraints are relaxed to 6/2, the number of subgraphs is augmented to 1454539. We present three scalable subgraph enumeration algorithms which can enumer-

ate all the subgraphs very efficiently under various design constraints. To enumerate all feasible subgraphs under size constraint, we propose a new algorithm that avoids multiple identifications of any subgraph. Based on the proposed size constrained subgraph enumeration algorithm, an extended algorithm that aims to enumerate all feasible subgraphs or only connected subgraphs is then presented. In addition, we present a polynomial-time algorithm that improves the previously proposed well-known algorithm [Pozzi 2006] for enumerating all feasible subgraphs including disjoint subgraphs and connected subgraphs. Experiments show that our algorithms can achieve orders of magnitude speedup over the well-known algorithm[Pozzi 2006].

• Subgraph Selection: Selecting a subset from the set of subgraphs generated by subgraph enumeration is also time costly, so our goal is to propose algorithms that are time efficient. Three different heuristic subgraph selection approaches targeting to different objectives are depicted and a complete comparison of them is shown in experiments. Furthermore, we present an exact algorithm and a genetic algorithm for selecting minimal number of matches to cover the data-flow graph of given application program. The proposed genetic algorithm makes trade-off between the greedy approach and the exact approach. In other words, it makes trade-off between the run-time of selection and the quality of selection results.

• Code Transformation: we extend an existing graph isomorphism algorithm to quickly reject dissimilar graphs by using some characteristics of data-flow graph. Some specific problems involved in the graph isomorphism check and subgraph isomorphism are first exposed. To our knowledge, the aforementioned problems were never considered in previous literature. The corresponding solution for the exposed problems are provided in this thesis. Moreover, a brief introduction to the code representation for custom operator in the generated code is also given.

• The Design Flow: an automated and adaptable custom operator-based pre-synthesis design flow is presented; results for a set of benchmarks show that significant code size reduction, area reduction and speedup can be achieved by our proposed design flow.

1.5 Organization of this Thesis

The rest of the thesis is organized as follows. The thesis starts with a survey of related work (Chapter 2). In chapter 2, for the sake of clarity, the intermediate representation of code used in the design flow and some important definitions are introduced. As the design flow involves three major problems: subgraph enumeration, subgraph selection and graph isomorphism, we then review the three problems respectively.

In chapter 3, the problem formulation of the subgraph enumeration is depicted. Particularly, the proof that gives a tighter upper bound of valid subgraphs under input and output constraints is presented. To efficiently solve the problem under various design constraints, three enumeration algorithms are presented in detail.

We formally formulate the subgraph selection problem and present some algorithms for this problem in chapter 4: three heuristics targeting different objectives, an exact algorithm and a genetic algorithm that select the minimal number of matches resulting in the most compacting code size.

After that, an extended graph isomorphism algorithm used to check the functional equivalence between subgraphs is introduced in chapter 5. In the same chapter, some specific problems like non-commutative problem and symmetrical problem residing in the isomorphism check are discussed and corresponding solutions are also provided.

Chapter 6 evaluates the efficiency of the proposed algorithms and the quality of results of the whole design flow. Finally, conclusions and future works are presented in chapter 7.

Chapter 2

Related Work

In this chapter, to clarify, we first present the intermediate representation used in our design flow. As there are three major problems involved in the proposed design flow, we begin the survey from the subgraph enumeration problem, which enumerates all feasible subgraphs under various constraints. Then, we review the subgraph selection problem, which try to select a subset of most profitable subgraphs in terms of different motivations. Finally, we survey the graph isomorphism problem, which aims to find the functional equivalence between subgraphs. In the survey of related works, we mainly focus on the algorithms dedicated to hardware design.

2.1 Intermediate Representation

The behavioral specification to high-level synthesis is usually composed of a sequential list of statements. The statements could be operation expressions, conditional constructs and loop constructs. To capture the data-flow dependencies in the input description, data-flow graph is a good candidate for designers. As the data-flow graph only contains the explicit data dependencies, we could use control flow graph to capture the control flow through the input description. In order to capture the both information in design description, control data-flow graphs are a well known and good intermediate representation for high-level synthesis. In this section different intermediate representations for high-level synthesis are presented. The definition, semantics and an example of construction for DFG, CFG and CDFG are shown respectively.

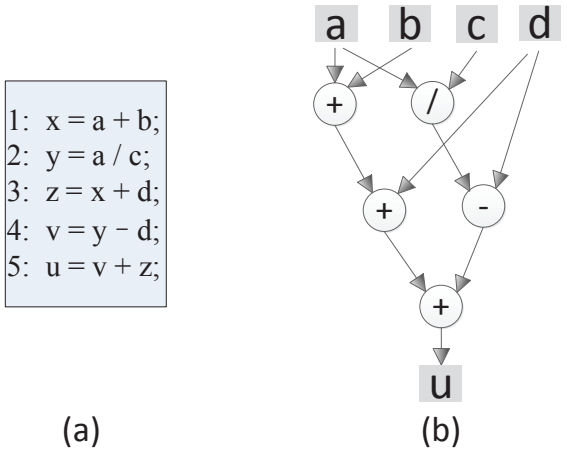


Figure 2.1: (a) Fragment of C code without condition or loop (b) Corresponding data flow graph

2.1.1 Data-Flow Graph

2.1.1.1 Definition

A data-flow graph (DFG) is a directed acyclic graph (DAG) which represents data dependencies between a number of operations. It is a graph G = (V, E), where the vertex set $V = \{v_1, ..., v_n\}$ represents primitive operations and the edge set $E = \{e_1, ..., e_m\} \in V \times V$ represents the data-flow dependencies.

2.1.1.2 Semantics

A node receives data generated by its predecessor nodes and produces new data needed by its successors. The operation of the node is performed on the data of the incoming edges only when all the incoming data are ready. The resulting data is then put on the outgoing edges. Therefore, the execution order of the operations in the graph is thus constrained by the partial ordering of the nodes as defined by the directed edges. It is clear that data-flows through the graph and each time data encounters a node, an operation is performed on it.

Operation nodes can be primitive arithmetic operations, like $+$, $-$, $*$, $++$, or Boolean like $>$, $<$, or can be more complex custom operations. The timing constraints imposed by data dependencies can be denoted as follows: Let $T = \{t_i; i = 0, 1, .., n-1\}$ be the execution start times of the operations and $D = \{d_i; i = 0, 1, .., n-1\}$ be the execution

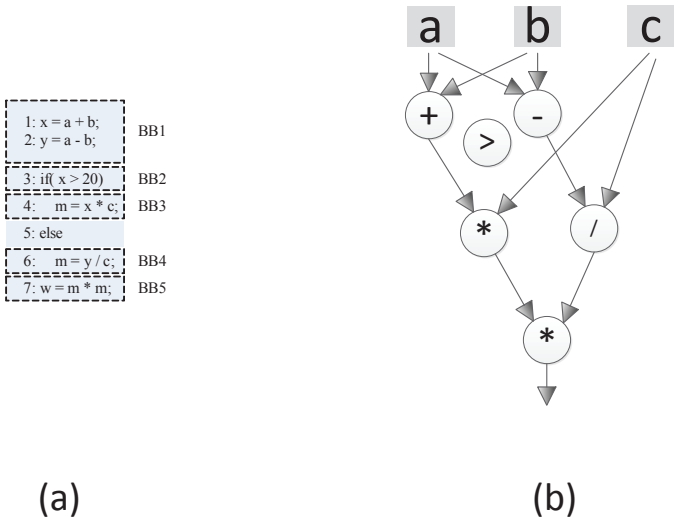


Figure 2.2: (a) Fragment of C code with condition (b) Corresponding data flow graph (with loss of consistency)

delays of each operations. Assume that an operation l reads the result of its predecessor k, then the operation l can start execution only after the predecessor k has finished execution. This can be expressed as:

$$t_l \geq t_k + d_k, \forall k, l : (v_k, v_l) \in E \tag{2.1}$$

2.1.1.3 Examples

Any algorithmic behavioral description of a system consists of ordered operations. To demonstrate the data-flow graph construction, the fragment of C description in Fig.2.1(a) is used as an example. To simplify the construction, we use the segment of C code without condition or loop in it. The example of a data-flow graph for the fragment of C description is given in Fig.2.1(b). Operations in the data-flow graph are denoted by circular nodes with the operator sign within. The input nodes are represented by squares.

The segment of C code listed in Fig.2.2(a) has condition in it. The corresponding data flow graph for the above fragment of C description is given in Fig.2.2(b). Obviously, the control information is lost during the construction of data flow graph according to the definition of data flow graph. As shown in Fig.2.2(b), the node labeled with ">" has no input or output edge. Consequently, when we regenerate the new description code

from the incrementally refined DFG which has passed through various stages of a high-level synthesis system, the new generated description code is not functionally consistent with the original source code. Thus, to support conditional constructs, loops and procedure calls, we may use control flow graph directly, besides CFG. The authors [van Eijndhoven 1992] has proposed a new form of data flow graph which moves conditional constructs into the data flow graph.

2.1.2 Control-Flow Graph

2.1.2.1 Definition

A control-flow graph represents the control dependencies among basic blocks (A basic block is a sequence of statements from the input description with no conditionals or loops between them). It is a graph G = (V,E), where $V = \{v_1, ..., v_n\}$ is the set of basic blocks which are treated as different sequences of statements in the design and $E = \{e_1, ..., e_m\} \in V \times V$ is the set of directed edges which represents the control flow between basic blocks.

2.1.2.2 Semantic

As we know, during the execution of program, each time the condition operation is executed, the control flow partitions into two control flows, one for the evaluation with value of true, the other for the evaluation value of false. As the program be executed continually, the control flows merge into a single control flow at the end of the conditional or loop construct [Gupta 2004]. Hence, in control flow graph, node can be identified as an operation node, branch node or merge node.

Operation node is a basic block which contains a sequence of operations and has one incoming edge and one outgoing edge, having no conditional check in them and having only a default output true path.

Branch node has two or more outgoing edges and one incoming edge, signifying the point at which a condition causes the control flow to branch into multiple control paths. Obviously, it also determines the control flow passes through which outgoing edges.

Merge node that has two or more incoming edges and one outgoing edge is opposite

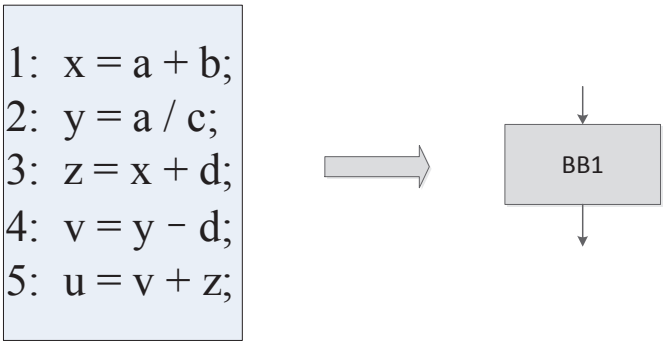


Figure 2.3: C code without condition or loop and its corresponding CFG

to the branch node. In general, branch and merge nodes are used to model algorithmic constructs like if...then...else, case, and loop constructs like while...do, for...do. A feedback edge between the branch and merge nodes can represent the loop construct [de Jong 1991]. The last basic block in the design, which does not have any output control flow, is called as end node. The first basic block in the design, which does not have any input control flow, is called as first node [Gupta 2004].

2.1.2.3 Example

As shown in Fig.2.3, the sequential operations in the fragment C code without condition or loop are aggregated into one basic block that represents these sequential operations. To understand more about CFG, we use the fragment of C code with condition 2.4(a) to construct the CFG. Fig.2.4(b) shows the control flow graph of the source code depicted in Fig.2.4(a). Each basic block aggregates a sequence of operations in the source code with no control flow between them (shown by shaded boxes in Fig. 2.4(a)). The arrow between basic blocks denotes the control flow as shown in the corresponding control flow graph in Fig.2.4 (b). The basic blocks are labeled from BB_1 to BB_5. A diamond represents a Boolean conditional check or a fork in control flow with a true path and a false path.

2.1.3 Control Data-Flow Graph

Data flow graph and control flow graph captures only explicit data dependencies and data-flow in the input description respectively. To represent the both information in the input description, we use control data-flow graphs. CDFG has been considered as one of the

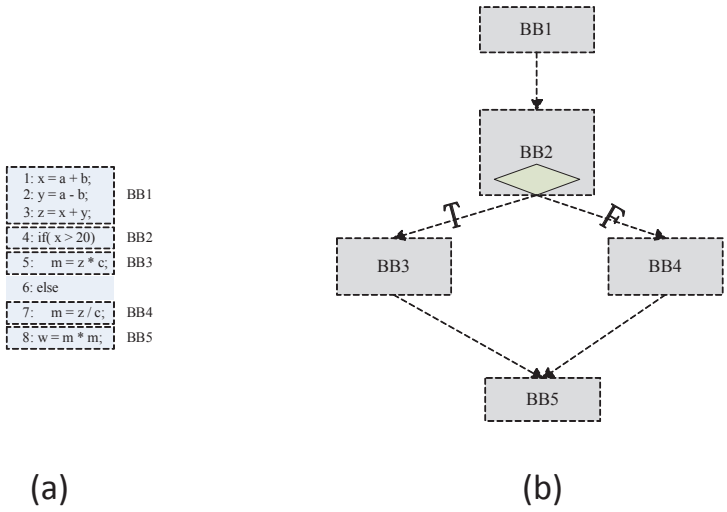


Figure 2.4: (a) Fragment of C code (b) Corresponding control flow graph

most popular intermediate representation for high level synthesis.

The control data-flow graph is a graph G = (V,E), where $V = \{v_1, ..., v_n\}$ is the set of basic blocks which are composed of a list of statements and $E = \{e_1, ..., e_m\} \in V \times V$ is the set of directed edges which represents the control flow between basic blocks. Inside each basic block, a data-flow graph is used to represent the data-flow dependencies between the operations. A directed edge connecting the basic blocks represents a condition of statements such as if/case or loop constructs.

To model the CDFG construction, we use previous piece of c code (Fig.2.5(a)). The corresponding CDFG is shown in Figure 2.5(b). In the example, each basic block has a data-flow graph which represent a list of operations. The only entry of the data-flow graph is through the first operation and the only exit is the last instruction. The dash line represents the control flows between blocks. The real line denotes the data dependencies between operations.

2.2 Definitions

In this section, for the sake of clarity, we first give some important definitions that are utilized throughout the thesis. Then several notations that are used for calculation are introduced.

Formally, a match is defined as, for a directed acyclic graph G = (V, E), a sub-graph

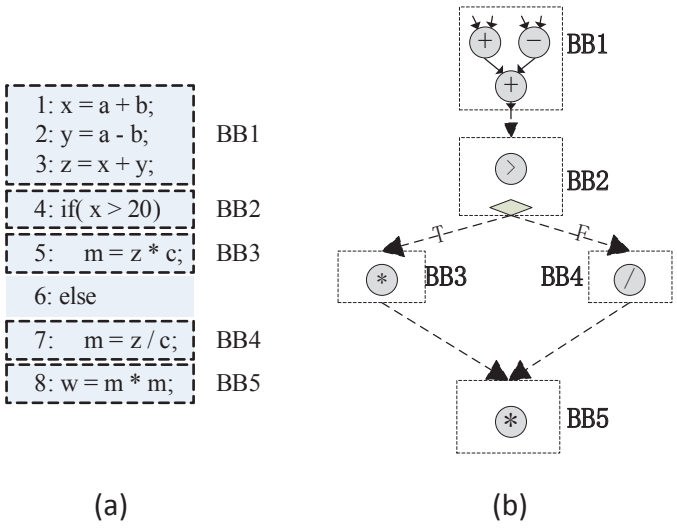


Figure 2.5: (a) Fragment of C code (b) Corresponding control data-flow graph

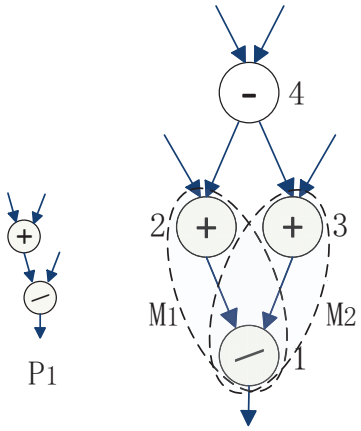


Figure 2.6: A pattern with its matches

$M = (V_m, E_m)$ of graph G, where $V_m \subseteq V$ and $E_m \subseteq E$ [1]. A pattern or a template is an induced graph of isomorphic subgraphs. A pattern or a template is a graph representation of a custom operator. In other words, a match is an instance of a pattern or a template. As an example, in Fig. 2.6, P_1 is a pattern and M_1 and M_2 are two matches of P_1. The term custom instruction is a code representation of a custom operator.

The input node set of a match (subgraph) M is denoted as $IN(G, M)$. An input node is a node in G but not in M that has at least one edge entering the match M. Similarly, we use $OUT(G, M)$ to denote the output node set of M. An output node is a node in M that has at least one edge exiting the match M and connecting to a node in G but not in

[1] we will use the term match to represent subgraph alternatively in this thesis

M.

Given a DFG *G(V,E)* and a node u, some subsets can be defined according to their relationships with u.

1. Immediate predecessors of u : $IPred(G,u) = \{v | v \in V, (v,u) \in E\}$.
2. Immediate successors of u : $ISucc(G,u) = \{v | v \in V, (u,v) \in E\}$.
3. All predecessors of u : $Pred(G,u) = \{v \in Pred(G,v) | v \in IPred(G,u)\}$.
4. All successors of u : $Succ(G,u) = \{v \in Succ(G,v) | v \in ISucc(G,u)\}$.
5. Disconnected nodes of u: $Disc(G,u) = G - (Pred(G,\ u) - Succ(G,u) - u)$.

Similarly, given a graph *G(V,E)* and a subgraph (match) $M \subseteq G$, some subsets can also be defined according to their relationships with M.

1. Immediate predecessors of the subgraph M: $IPred(G,M) = \cup_{u \in M} IPred(G,u) - M$.
2. Immediate successors of the subgraph M: $ISucc(G,M) = \cup_{u \in M} ISucc(G,u) - M$.
3. All predecessors of the subgraph M: $Pred(G,M) = \cup_{u \in M} Pred(G,u) - M$.
4. All successors of the subgraph M: $Succ(G,M) = \cup_{u \in M} Succ(G,u) - M$.
5. Disconnected nodes of the subgraph M: $Disc(G,M) = \{u | u \in V, \forall v \in M$, there is neither a path from u to v or from v to $u\}$.

2.3 Related Work on Subgraph Enumeration

As the key issues involved in our design flow are subgraph enumeration, subgraph selection and code transformation (see Fig. 2.7), we present the related works for subgraph enumeration, subgraph selection and graph isomorphism (graph isomorphism algorithm is used during code transformation to identify the functionally equivalent subgraphs) respectively. A lot of related researches on subgraph enumeration, subgraph selection and graph isomorphism have been done in recent years. The survey is restricted to algorithms that can take into account constraints related to hardware design. In this section, we start

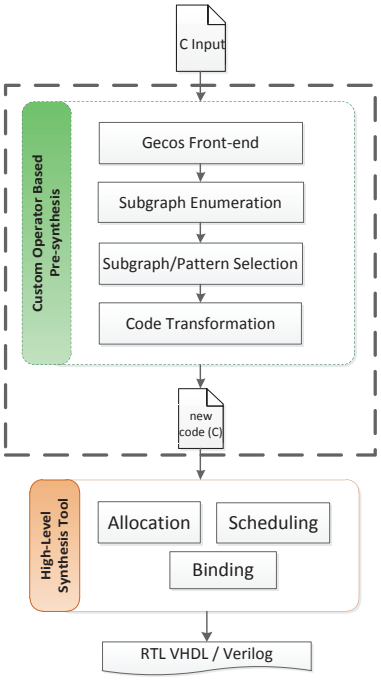


Figure 2.7: Custom operator based high-level synthesis flow

our review from subgraph enumeration and then discuss subgraph selection and graph isomorphism in the following sections.

Subgraph enumeration is the process to enumerate all the subgraphs that satisfy certain design constraints from a given application graph. It is a computationally difficult problem. The number of subgraphs in the DFG of a given application could be 2^n, where n is the number of nodes in the DFG [Jozwiak 2010]. Since the number of subgraphs is exponential with respect to the size of the DFG, efficient approaches to the subgraph enumeration problem are necessary. In order to reduce the complexity of the enumeration, several constraints are added as pruning criteria. Here, we survey the subgraph enumeration problem by classifying the previous works according to the specified constraints on the enumerated subgraphs.

Tree Sharped Subgraphs As the type of subgraphs enumerated directly relates to the enumeration problem complexity[Jozwiak 2010], some previous researches only focus on enumerating tree-shaped subgraphs [Liem 1994a, Shu 1996]. Considering only tree-shaped subgraphs can radically reduce the complexity [Yu 2008]. In [Aho 1989], a polynomial time dynamic programming method is applied to cover the DFG with the minimal number of tree-shaped subgraphs. However, enumerating only tree-shaped subgraphs may

lead to limited improvements on performance or other aspects. Compared to tree-shaped subgraphs, more internal data flows involved in net-shaped subgraphs results in saving more multiplexors [Cong 2008].

Multiple Inputs Single Outputs (MISO) In the context of extensible processors [Gonzalez 2000], the number of inputs and the number of outputs (I/O) are constrained due to the number of ports to the register files. In the scenario of high-level synthesis, the I/O constraints are considered as user-specified design constraints. The I/O constraints are important pruning criterias when performing subgraph enumeration. The tighter the I/O constraints are, the less number of subgraphs considered. In other words, the enumeration can be done in a shorter time if restricting the I/O constraints to lower values. Early works try to enumerate single output subgraphs can be found in [Alippi 1999, Pozzi 2002, Cong 2004, Galuzzi 2007b]. A subgraph with multiple inputs and only one output subgraphs is called MISO subgraph. The approach in [Cong 2004] enumerates all the K-MISO subgraphs with dynamic programming, where K is the input constraint. Another approach in [Galuzzi 2007b] iteratively considers the MISO subgraphs from the MISO subgraphs of maximal size under a specified input constraint. These approaches are efficient only when the input constraint is low. Relaxing the input constraint can lead to exponential computation.

Theoretically, the number of MISO subgraphs in a given DFG is exponential with respect to the size of the DFG [Galuzzi 2007a]. Thus, other works [Alippi 1999, Pozzi 2002] further restrict the subgraphs enumerated. These works only consider the MISO subgraphs of maximal size, which is called MAXMISO. The authors of [Alippi 1999] formally proved an important property of MAXMISOs: MAXMISOs of a given DFG cannot partially overlap. In [Pozzi 2002], the authors presented an efficient algorithm that can exhaustively enumerate MAXMISOs from a given DFG in linear complexity in terms of the number of nodes in the DFG. The algorithm generates all MAXMISO starting from a selected output node and iteratively grows the subgraphs by adding predecessors until a forbidden node is encountered (the nodes that represent memory operations like load/write). As the MAXMISO cannot overlap, each node is considered only once. Thus, a linear complexity of this algorithm is guaranteed.

Multiple Inputs Multiple Outputs (MIMO) Recent studies try to enumerate subgraphs with multiple inputs and multiple outputs. Generally, a subgraph could be it-

eratively formed by absorbing a node or nodes to a previously identified smaller subgraph. Enumerating all possible subgraphs under input and output (I/O) constraints is a computationally difficult problem, because the number of possible subgraphs grows exponentially with the size of the application graph [Galuzzi 2011]. Enumerating all possible subgraphs under I/O constraints refers to I/O constrained enumeration in the past literature. In this thesis, we roughly classify the previous approaches for I/O constrained enumeration into two groups: heuristic approaches and exact approaches.

Heuristic techniques for MIMO subgraph enumeration try to identify some promising subgraphs while discarding some less promising ones. In general, heuristic techniques only produce a subset of the candidate subgraphs. Authors of [Kastner 2002] assemble the subgraphs along the most frequently occurring edges direction. To avoid exponential blow-up, each time a bigger subgraphs is enumerated, it is collapsed into a super node. The method proposed in [Clark 2003] grows subgraphs based on previously generated smaller subgraphs by evaluating neighbor nodes with a defined cost function that calculates both performance gains and penalties in terms of input or output constraint violation. In [Wolinski 2007] the subgraph searching algorithm assembles subgraphs incrementally by adding neighbor nodes to existing subgraphs corresponding to non-isomorphic subgraphs formed in the previous iteration. A smart filtering is used to discard less "useful" subgraphs in this algorithm. Compared with the exhaustive subgraph enumeration, heuristic techniques may achieve a linear time in terms of computational time. However, the heuristic techniques cannot guarantee to produce a globally optimal set of subgraphs.

Exact enumeration approaches for MIMO subgraph enumeration offer us a chance to obtain a better result. The authors [Chen 2007] are the first ones that prove the exhaustive enumeration of subgraphs is inherently polynomial. A polynomial time subgraph enumeration algorithm is proposed in [Bonzini 2007]. However, the work in [Reddington 2009] pointed out that the polynomial time subgraph enumeration algorithm can miss enumerating up to 25% of subgraphs. Thus, the claimed polynomial time enumeration algorithm cannot provide an exact enumeration. Several other techniques have been proposed for an exhaustive enumeration of subgraphs under I/O constraints, such as dynamic programming [Arnold 2001], integer linear programming [Lee 2002, Atasu 2005] and constraint programming [Martin 2009a]. These methods are efficient for relative small applications graphs. Unfortunately, they are quite time-consuming when the application graphs be-

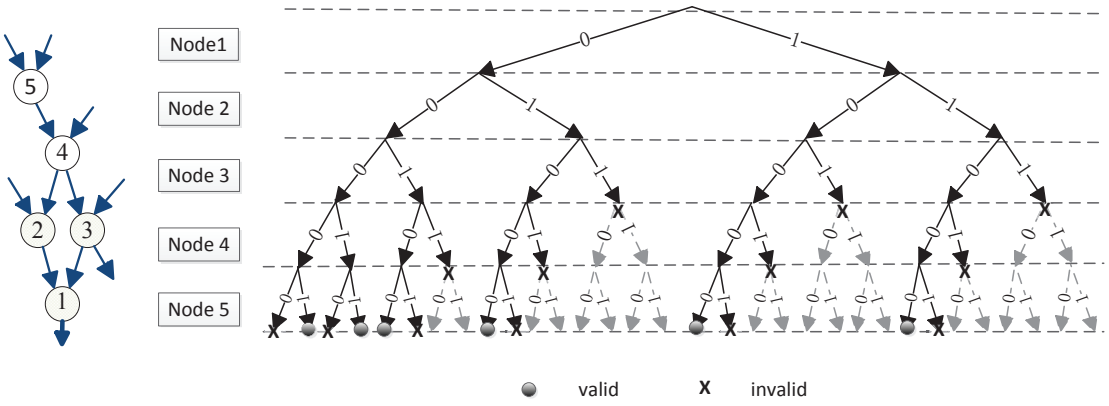


Figure 2.8: A DFG and its corresponding binary search tree

come large.

Authors of [Atasu 2003] presented an exhaustive algorithm based on a binary decision tree under convexity [2] and I/O constraints. The algorithm first assign each node a unique number according to the topological order. The subgraphs are formed by adding nodes with bigger value to smaller subgraphs. The search space can be pruned by using the monotonicity of outputs in data-flow graph. The algorithm Pozzi et al [Pozzi 2006] further improved this algorithm by adding a pruning criterion based on the number of permanent inputs. Fig. 2.8 shows an example of DFG and its corresponding binary search tree using the algorithm [Atasu 2003] when the output constraint $OUT_{max} = 1$ and there is no input constraint. There are 5 nodes in the application graph. Each node is assigned with a number with respect to the topological order. The search tree is a binary tree of nodes representing possible subgraphs building by a recursive search function based on topological order: the 1-branch and 0-branch of each level represent addition or not of the node i. Once the node is included or not in the subgraph, we can include or not the following node in accordance with topological order. Addition of a 0-branch for a node signifies the same node as its parent node.

Theoretically, there are 2^n possible subgraphs in a graph. For example, there are 2^5 possible subgraphs in the above subgraph. However, the design space is radically reduced by taking the I/O constraint and convexity into account. According to the monotonicity of the number of outputs, if inclusion of a node violates the output constraint [Pozzi 2006],

[2] convexity constraint guarantees feasible scheduling of a custom operator, the definition of convexity is given in section 3.1

the inclusion of a later node in the topological order cannot overcome the output constraint violation. Similarly, once the convexity constraint is violated, adding any node that appears later in the topological order cannot resolve the violation. As a consequence, if the constraints are violated when including a node, the sub-tree originating from that node will not be searched. For example in 2.8, the subgraph {1,2} satisfies the I/O constraints, while adding node 3 to the subgraph {1,2} results in a subgraph that violates the output constraint ($OUT_{max} = 1$). Thus, the sub-tree originating from the node 3 will not be considered. In Fig. 2.8 the gray dotted arrays represent the pruned search space. We can observe that among the 32 possible subgraphs, the algorithm considered only 19.

Yu et al. [Yu 2004b] build only connected subgraphs by enumerating upward cones and downward cones. However, in this algorithm, a subgraphs could be considered more than once. Additional redundancy checking is required. The additional redundancy checking may tremendously increase the runtime. Yu et al. [Yu 2007] proposed an algorithm targeting to enumerate disjoint subgraphs. According to the algorithm, the disjoint subgraphs are formed by merging the connected subgraphs. The connected subgraphs are provided by their previous algorithm [Yu 2004b]. However, the experiments in [Chen 2007] show that the algorithm proposed in [Pozzi 2006] has a better performance than the algorithm proposed in [Yu 2004b] when enumerating only connected subgraphs. As the disjoint subgraphs enumeration algorithm [Yu 2007] enumerates all disjoint subgraphs by combining the connected subgraphs generated by the algorithm [Yu 2004b], the algorithm [Pozzi 2006] should also has a better performance than the algorithm [Yu 2007]. Another algorithm is proposed in [Chen 2007]. The algorithm enumerates both connected feasible subgraphs and disjoint feasible subgraphs. The algorithm uses a grading method to select the next node to be included. The algorithm [Chen 2007] is comparable to the algorithm [Pozzi 2006] when enumerating all subgraphs including connected subgraphs and disjoint subgraphs. A recent research [Reddington 2011] formally proved that the algorithm [Pozzi 2006] is of polynomial complexity.

Maximal Multiple Inputs Multiple Outputs (MaxMIMO) In recent years, enumerating the maximal MIMO subgraphs has drawn a wide interest from researchers who work on the application-specific instruction-set extension processors (ASIPs). Pothineni et al. [Pothineni 2007] were the first ones to propose an algorithm for MaxMIMO enumeration. The proposed algorithm is based on an incompatibility graph. However, the

algorithm only generates connected MaxMIMOs. In [Verma 2007], the MaxMIMOs enumeration problem is reformulated as a maximal clique enumeration problem after grouping equivalence nodes and building cluster graph. Atasu et al. [Atasu 2008] proved that the number of MaxMIMOs is bounded by $2^{|V_I|}$, where V_I is the set of invalid nodes in the DFG. A top-down manner algorithm proposed in [Li 2009] solves the MaxMIMOs enumeration problem efficiently by a division operation on the DFG.

The above mentioned techniques solve the problem either in a bottom-up manner or a top-down manner. The bottom-up manner may reduce the size of the DFG by clustering equivalence nodes. The top-down manner can reduce the size of the DFG by breaking the DFG into smaller graphs. In order to take advantage of both bottom-up manner and top-down manner, we propose an efficient algorithm [Xiao 2011] that enumerates MaxMIMOs in a sandwich manner, a combination of the bottom-up manner and the top-down manner. In [Xiao 2011], we also give a tighter upper bound on the number of the MCSs within a given DFG.

Connected or Disjoint Subgraphs A computation subgraph can be a connected subgraph or a disjoint subgraph. To reduce the high computational complexity, some authors look only for the connected subgraphs [Arnold 2001, Baleani 2002, Cong 2004, Yu 2004b, Clark 2005]. Those approaches for enumerating only connected subgraphs achieve lower complexity by sacrificing the optimality. As we known, imposing the parallelism is one of the major benefits using custom operators. Compared with connected subgraph, the disjoint subgraph can exploit more parallelism when implemented as hardware function unit (custom operator). Therefore, most of recent researches mainly focus on enumerating all subgraphs including connected subgraphs and disjoint subgraphs [Atasu 2003, Pozzi 2006, Galuzzi 2006, Chen 2007, Ahn 2011].

Size Constrained Enumeration Unlike the previously mentioned approaches, some approaches restrict the size of the enumerated subgraphs [Choi 1999, Guo 2003]. This class of approaches enumerates all the subgraphs, whose number of operations (nodes) is less than the maximal number of nodes. In the context of ASIPs, the I/O constraints are hard constraints due to the micro-architecture. However, in the scenario of high-level synthesis, the I/O constraints can be viewed as soft constraints.

J.Cong et al [Cong 2008, Cong 2010, Cong 2011b] has proposed a subgraph enumer-

ation algorithm that enumerates all the subgraphs with respect to user-defined edit distance (edit distance represents the minimal sequence of edit operations that transform one subgraph to the other subgraph, it is used to measure the similarity of subgraphs) and frequency limit (frequency refers to the number of occurrences of a subgraph in the application graph). Because some profitable subgraphs may be discarded at early stage, applying frequency limit and user-defined edit distance constraint at the enumeration stage may prevent from achieving a better solution. Similar to other previous algorithms, this algorithm also has to use a extra duplication checking to exclude the redundant subgraphs.

2.4 Related Work on Subgraph/Pattern Selection

The subgraph enumeration generates every subgraph satisfying the design constraints from a given application. After the subgraph enumeration step, the subgraph selection step is performed in our design flow (see Fig. 2.7). Subgraph selection is the process that selects the most profitable subset of subgraphs from the set of subgraphs enumerated in the subgraph enumeration step. When hardware design is targeted, the subgraph candidates are selected either due to the high frequency of occurrences (to make use of resource sharing) in the application or due to their high performance compared to other subgraphs (i.e., hardware components) or due to significant area reduction. Therefore, developing a good subgraph selection method is quite vital for highest gain in performance or area for the application. Many works were proposed for selecting subgraphs with different strategies (minimal number of custom operators, minimal code size, etc.). In the following survey, we organize the survey on subgraph selection according to the objectives.

Minimal Number of Matches Selection The objective of minimal number of matches selection is to select a minimal subset of subgraphs to cover the application graph. Selecting a minimal number of matches can lead to a most compacted code. With the compacted code, the high-level synthesis tools produce a design solution in a shorter time compared to the initial code without incorporating custom operators. Examples of selecting minimal number of matches can be found in [Biswas 2003, Biswas 2005, Clark 2006]. In [Clark 2006], an exact algorithm converts the minimal number of matches selection problem to a unate covering problem. However, this algorithm is tractable only when the set of candidates is very small.

Frequency of Occurrences Based Selection Reuse is an important factor that should be considered during the selection step [Guo 2003, Kavvadias 2005, Kavvadias 2006, Lam 2009]. This is especially true when the strategy is resource constrained scheduling in high-level synthesis. In general case, small patterns have higher frequency of occurrences compared to large patterns. Extremely, the one node patterns are more likely to be chosen (selecting one node patterns brings no change). To avoid this, the authors [Guo 2003] introduced the following objective function to balance the size of selected patterns and the frequency of occurrences of selected patterns.

$$g(P) = w^{1.2} * s = w * s * w^{0.2}; \tag{2.2}$$

where w represents the size (number of nodes) of the pattern P, s represents the number of instances of the pattern P in the application graph. The extra $w^{0.2}$ factor gives the bigger pattern more weight when carrying out the selection.

Critical Path Based Selection Inappropriate selection may give rise to the delay overhead. Several interesting approaches aiming to select subgraphs along the longest path through the application graph to minimize the delay have been proposed [Liao 1995, Liao 1998, Clark 2003, Cong 2008]. J.Cong et al [Cong 2008] prefer to select the subgraphs which are flatter. A flatter subgraph imposes more parallelism than a less flat one. Although this approach can reduce the latency overhead compared to the pattern selection approaches without considering the impact on the critical path, the experiments in [Cong 2008] reveals that increase of the latency still can not be avoided in many cases compared to synthesis without custom operators (see Fig. 4.5).

Overlap Two subgraphs may have the same nodes in common. Some prior works attempt to select overlapping subgraphs to achieve maximum performance speedup [Aletà 2004, Cong 2004]. Allowing overlapping may sometimes improve the resulting execution time, while unnecessarily increase the power consumption and make the code regeneration intractable. Thus, most approaches for subgraph selection select non-overlapping set of subgraphs to cover the application graph [Guo 2003, Pozzi 2006, Li 2010]. In this thesis, we disallow overlapping between all selected subgraphs. As an instance, in Fig. 2.6, if the match M_1 is selected, then the match M_2 will not be considered. To reject overlapping subgraphs quickly, a conflict graph is built before the selection. The conflict

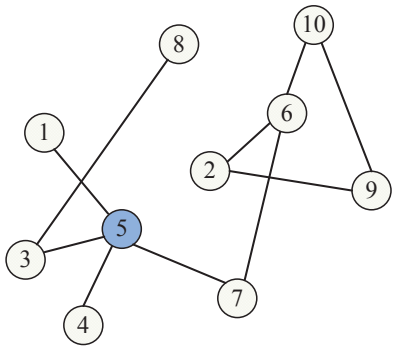


Figure 2.9: A conflict graph

graph consists of nodes and edges, where the nodes represent the subgraphs. The edges are added between nodes if two nodes have overlapping. During the selection, each time a subgraph is selected, the nodes connected to the node (corresponding to the selected subgraph) are removed from the conflict graph. Assume the subgraph (node) 5 is selected in Fig. 2.9, the nodes representing the subgraphs 1,3,4,7 will be deleted from the conflict graph, i.e., they can not be selected anymore.

Sequence of Enumeration and Selection Most of the prior works perform the subgraph enumeration and selection in two consecutive steps [Atasu 2003, Cong 2004, Galuzzi 2006]. Some other works combine the enumeration and selection into one step: the enumerated subgraphs are automatically selected [Clark 2003, Atasu 2005]. In [Clark 2003], the authors starts generating subgraphs from a seed node and use a guide function to rank the data-flow edges to be added. The edges are ranked according to three categories: criticality, latency and area. As the edges on the critical path are more likely to provide application performance by shrinking the application graph, those edges have high scores. The performance gain is usually obtained by combining two operations. The combined two operations can be executed in fewer cycles than they do individually. A latency point is used to estimate the performance gain by the equation: $(old\ latency/new\ latency) * 10$. Similarly, an area point is applied to estimate the area cost by the equation: $(old\ area/new\ area) * 10$.

Various approaches are proposed to give an optimal solution. This group of approaches use either integer linear programming (ILP) [Yu 2004a, Atasu 2005, Galuzzi 2006, Atasu 2007] or constraint programming [Martin 2009b] or branch-and-bound [Clark 2006, Dinh 2008]. ILP based approaches convert each candidate as a boolean variable. The

constraints like area budget are expressed with an equivalent linear equations. The selection objective is represented by a linear objective function. The ILP solver optimize the objective function and gives the optimal solution. A novel method was presented in [Martin 2009b], the authors try to solve the selection problem by using constraint programming. The selection of patterns is carried out with two respective scheduling strategies: time-constrained scheduling or resource constrained scheduling. This method assumes that all nodes are not able to be covered by more than one match. Similar to integer linear programming methods, the method using constraint programming also requires an appropriate modeling of the selection problem including the variable and its domain, constraints and the objective function. The difference is that the variables modeled by constraint programming are not required to only be integer or boolean value. Thus, the equations representing the constraints in constraint programming are not necessarily to be linear ones. Branch-and-bound based methods usually first generate a greedy solution. The generated greedy solution is set as the lower bound to prevent the exploration of suboptimal branches. Despite the optimality can be guaranteed by this group of techniques, the scalability becomes unsure when the size of the problem is increased.

As subgraph selection is a computationally complex problem. Finding exact solutions for subgraph selection becomes intractable and unaffordable when the application graph is large. Thus, instead of exact approaches, efficient heuristic approaches are required. Several heuristic methods were proposed in the literatures. Authors of [Kastner 2002] select the subgraphs having higher frequency of occurrence by assembling the subgraphs along the most frequently occurring edges direction. In [Guo 2003], a pattern (induced graph of isomorphic subgraphs) selection algorithm based on a conflict graph has been proposed to cover the graph with a minimum number of patterns. The algorithm uses an objective function to select the subgraphs under a prerequisite that all selected subgraphs cannot be overlapped. In the method [Bozorgzadeh 2002], the selection is guided by scheduling such that the more critical subgraphs are selected due to higher priority. The previously mentioned algorithms select subgraphs in the context of extensible processors. The selection algorithm proposed in [Cong 2008, Cong 2011b] is the only selection algorithm delicated to custom operator based high-level synthesis. However, the algorithm may select a set of custom operators that result in latency overhead.

2.5 Related Work on (Sub)Graph Isomorphism

In some scenario, it is assumed that the patterns are already provided from the pattern library, i.e., it means the components that can be used are known (given by the designer) and their corresponding subgraphs are set in the pattern library. In this case, the task is to find the occurrences of each pattern from the library and select for example the most frequent patterns to cover the application graph. Finding the occurrences of each pattern in a application graph can be viewed as subgraph isomorphism problem (pattern matching). In our design flow, the patterns are automatically extracted from the application graph by comparing the enumerated subgraphs (pattern generation). A pattern is formed to represent a group of structurally equivalent subgraphs. To detect the structure similarity between subgraphs, a graph isomorphism algorithm should be applied. As graph isomorphism problem is a subset of subgraph isomorphism problem, the algorithm targeting at subgraph isomorphism problem can be used to solve graph isomorphism automatically. We survey the related work on both subgraph isomorphism and graph isomorphism.

The subgraph isomorphism problem has been applied to many applications. For instance, in the area of robot visions, it is used to recognize 3D objects by isomorphically matching with their canonical models in a computer model base [Wong 1992]; in the area of electronics, it is applied to extract sub-circuits from large circuit [Ling 1996]. The first algorithm for subgraph isomorphism has been proposed by Ullman in [Ullmann 1976]. In pattern matching, it assumes the existence of a pattern library and finds the frequency of occurrences of each pattern in application graph by subgraph isomorphism algorithm [Rao 1992, Liem 1994b, Clark 2003]. Larossa et al [Larrosa 2002] solved the subgraph isomorphism problem for graph pattern matching using constraint satisfaction. Authors of [Wolinski 2007] presented a constraint programming based subgraph isomorphism algorithm named Jacop [Kuchcinski 2003] to get the frequency of occurrence of each generated pattern. Jacop solves subgraph isomorphism problem on top of constraint programming environment.

In pattern generation, graph isomorphism algorithm is frequently used to compare the enumerated subgraphs. Kastner et al [Kastner 2002] checked whether a newly enumerated subgraph is isomorphic to previously generated patterns using an extension algorithm of Gemini [Ebeling 1983]. The extended algorithm [Kastner 2002] can possibly decrease the

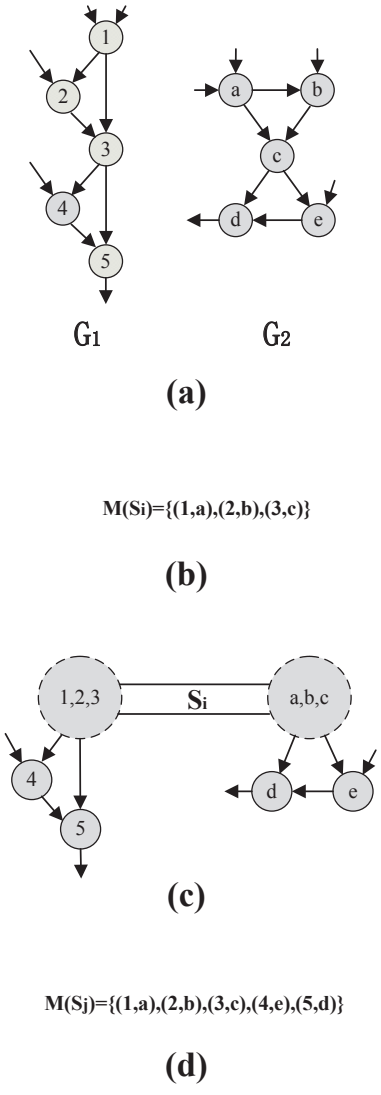


Figure 2.10: (a) Two graphs G_1 and G_2, (b) a partial mapping solution, (c) the corresponding graphic state, (d) the only full mapping solution

number of iterations by using the invariant properties of a graph to create a better initial coloring. Another application of Gemini can be found in [Guo 2003].

Subgraph isomorphism problem is a well-known NP-complete problem [Garey 1990]. Incorporating subgraph isomorphism algorithm (or graph isomorphism algorithm) into subgraph/pattern selection greatly increases the complexity. Thus, an efficient (sub)graph isomorphism is essential for the performance of the complete design flow. In this thesis, we present an extended subgraph isomorphism algorithm that is able to solve both the subgraph isomorphism problem and the graph isomorphism problem efficiently by capturing the characteristics of data-flow graph. The proposed algorithm is based on the widely

applied algorithm [P. Cordella 2004]. The algorithm [P. Cordella 2004] introduces the State Space Representation (SSR) to describe a graph match process. SSR maintains the partial mapping and is grown by adding compatible nodes pairs. To reduce the number of search space explored, a set of feasibility rules is used to detect as early as possible the following incompatible states. In Fig. 2.10 (a), two graphs to be compared are given. A partial mapping between the two graphs and corresponding graphic state are presented in Fig. 2.10 (b) and (c) (S_i represents the partial state and $M(S_i)$ represents the partial mapping). The corresponding state to the partial mapping is transited to the final state by adding two node pairs $\{(4, e), (5, d)\}$ (see Fig. 2.10 (d)). We can note that some additional checks for the number of nodes, the number of edges and the number of starting nodes can be added to further quickly reject dissimilar elements.

2.6 Summary

In this chapter, we introduced the intermediate representation that is used as input of subgraph enumeration. After that, we presented some important definitions and notations utilized throughout the thesis. Then, we reviewed the subgraph enumeration problem, the subgraph selection problem and the (sub)graph isomorphism problem respectively.

In the following chapters, we will detail each problem involved in the design flow and present the proposed algorithms.

Chapter 3

Subgraph Enumeration

In this chapter, we first give the problem formulation for subgraph enumeration. In detail, the constraints used to restrict the enumerated subgraphs are introduced. We then present an efficient size constrained subgraph enumeration algorithm. The algorithm runs efficiently by avoiding multiple identifications of any subgraph. Based on the size constrained subgraph enumeration algorithm, we propose a enumeration algorithm for enumerating subgraphs under I/O constraints. The algorithm is quite flexible and can be tuned to enumerate all feasible subgraphs or only connected subgraphs. Furthermore, a new algorithm that overcomes the drawbacks of a previously proposed well-known algorithm is also presented.

3.1 Problem Formulation

In this section, we show a generic formulation for the subgraph enumeration problem. In the problem formulation, some notations introduced in section 2.2 are utilized.

Problem 1 (subgraph enumeration): Given a DFG $G = (V, E)$, enumerate every subgraph M that satisfies the following constraints.

- Convexity: M is convex (hard constraint);
- Connectivity: M is connected or M is disjoint ;
- Size: the number of nodes in M;
- $|IN(G,M)| \leq IN_{max}$;
- $|OUT(G,M)| \leq OUT_{max}$; and
- M contains no invalid operation (hard constraint)

where $|IN(G,M)|$ represents the number of inputs and $|OUT(G,M)|$ represents the number

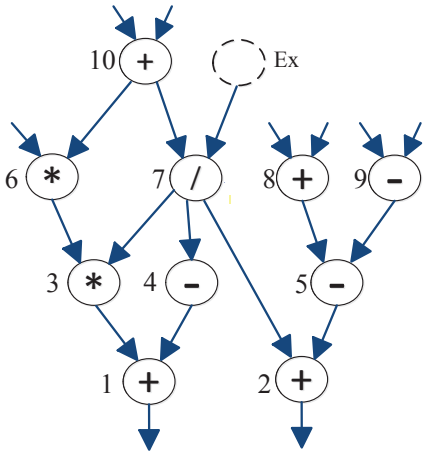


Figure 3.1: An example of data-flow graph

of outputs.

A subgraph M is said to be **convex** if there exists no path from a node $u \in M$ to another node $v \in M$, which involves a node $w \notin M$. As we know, a subgraph that can be implemented as a hardware function unit should be convex, otherwise it cannot be executed atomically. For example, in Fig. 3.1, the subgraph $\{1, 3, 4, 7\}$ is a convex subgraph, while the subgraph $\{1, 4, 7\}$ is not.

A subgraph could be **connected** or **disjoint**. If there exists at least one path (directed or undirected) between any pair of nodes in the custom instruction, we say that the subgraph is connected. Otherwise, it is a disjoint subgraph. In Fig. 3.1, the subgraph $\{1, 3, 4\}$ is a connected subgraph, while the subgraph $\{4, 5\}$ is a disjoint subgraph.

The maximum number of input and output operands of the subgraph is treated as a user-specified design constraints.

The **invalid** operations such as load/store are not considered as parts of subgraph.

Each DFG $G(V, E)$ has a number of inputs and outputs from and to some nodes (V^+) in other basic blocks. V^+ is connected to the nodes in G forming a new graph $G^+ = \{V \cup V^+, E\}$. As an example, the node Ex in Fig.3.1 belongs to V^+. In the past literature [Chen 2007], the authors first proved that the number of valid subgraphs is polynomial in the number of nodes of the DFG: $|S| \leq (|V^+|)^{IN_{max}}(|V|)^{OUT_{max}}$ when enumerating subgraphs under the I/O constraints, where S is the set of valid subgraphs, IN_{max} and OUT_{max} are the maximum number of inputs and the maximum number of

outputs respectively. As exclusion of invalid nodes in a subgraph is another important hard constraint for subgraph enumeration, a tighter upper bound exists if we take the number of invalid nodes into consideration.

Lemma 3.1.1 *Given a DFG G, associated G^+, input constraint IN_{max}, output constraint OUT_{max} and the number of invalid nodes $|V_{in}|$, if S is the set of valid patterns, then $|S| \leq (|V^+|)^{IN_{max}}(|V| - |V_{in}|)^{OUT_{max}}$*

Proof With the Lemma 6 proved in [Chen 2007], we know that a valid pattern is uniquely determined by its input node set and output node set. Thus, we choose input nodes and output nodes to get a pattern. First, we choose a number of nodes from G as output nodes. As invalid nodes can not be part of a valid pattern, we have no more than $(|V| - |V_{in}|)^{OUT_{max}}$ choices. To choose a number of nodes from G^+, we have no more than $(|V^+|)^{IN_{max}}$ choices. In total, we have no more than $(|V^+|)^{IN_{max}}(|V| - |V_{in}|)^{OUT_{max}}$ combinations of input nodes and output nodes. Therefore, we have no more than $(|V^+|)^{IN_{max}}(|V| - |V_{in}|)^{OUT_{max}}$ valid patterns in G.

As mentioned in the problem formulation, the convexity and exclusion of invalid nodes are hard constraints. All the other constraints can be viewed as user-specified constraints. In the following section, we introduce several algorithms that enumerate the subgraphs satisfying not only the hard constraint but also some of the user-specified constraints.

3.2 An Algorithm for Size Constrained Enumeration

The efficiency of the subgraph enumeration is strongly dependent on whether multiple identifications of a subgraph can be avoided. Most of previous work identify the subgraphs multiple times [Yu 2004b, Cong 2008, Pothineni 2007, Li 2009]. The runtime is unnecessarily increased. In this section, we present an efficient approach that avoids multiple identifications of any subgraph by a clever node deletion technique for enumerating subgraphs under size constraint.

In the general case, a larger subgraph could be iteratively generated by absorbing a node to a previously identified smaller subgraph. For example, a (k+1)-subgraph is

Algorithm 1 Subgraph Enumeration Algorithm

Input: Graph G
Output: CS - a complete set of enumerated subgraphs

```
1: Procedure SubgraphEnumeration()
2: R = ∅;// R is used to record the deleted nodes
3: for each node n ∈ G do
4:     M = {n};
5:     CS = CS ∪ M;
6:     call DepthFirstEnumeration(M, R);
7:     R = R ∪ n;
8: end for
9: Procedure DepthFirstEnumeration(M, R)
10: for each neighbor node n of M and n ∉ R do
11:     if SizeCheck(M) && ConvexityCheck(M, n) then
12:         M' = M ∪ {n};
13:         CS = CS ∪ M';
14:         call DepthFirstEnumeration(M', R);
15:         R = R ∪ n;
16:     end if
17: end for
```

formed by adding a neighbor node to a k-subgraph (for convenience, we call a subgraph a k-subgraph if the number of its nodes is k). However, a larger subgraph may be identified multiple times during the enumeration. Considering the DFG in the left of the Fig. 3.2, the subgraph $\{1, 2, 3\}$ could be generated from the subgraph $\{1, 2\}$ by adding the node 3 or from the subgraph $\{1, 3\}$ by adding the node 2. Thus, the subgraph $\{1, 2, 3\}$ could be identified twice. In this case, duplication check is required.

In order to avoid multiple identifications, the basic idea is to enumerate all the subgraph in a depth-first way and delete the previously considered node in each iteration. For example, we first enumerate all the subgraphs that contain the node 1 (see Fig.3.2). Next, we delete the node 1 (i.e., ~~{1}~~ in 3.2) before the enumeration of the subgraphs that contain the node 2, the node 3 or the node 4. Similarly, in the process of enumeration of all the subgraphs that contain the subgraph $\{1\}$, we first enumerate all the subgraphs that contain the subgraph $\{1\}$ and the node 2. Then, the node 2 should be deleted before the enumeration of the subgraphs that contain the subgraph $\{1\}$ and the node 3. In such way, all the subgraphs can be completely enumerated and no multiple identifications of any subgraph are performed. Fig.3.2 demonstrates the complete process of enumerating all the subgraphs from a simple DFG. [1]. Thus, at the end of the enumeration process, there are

[1] the non-convex subgraphs {1,2,4} and {1,3,4} are pruned away by the convexity check

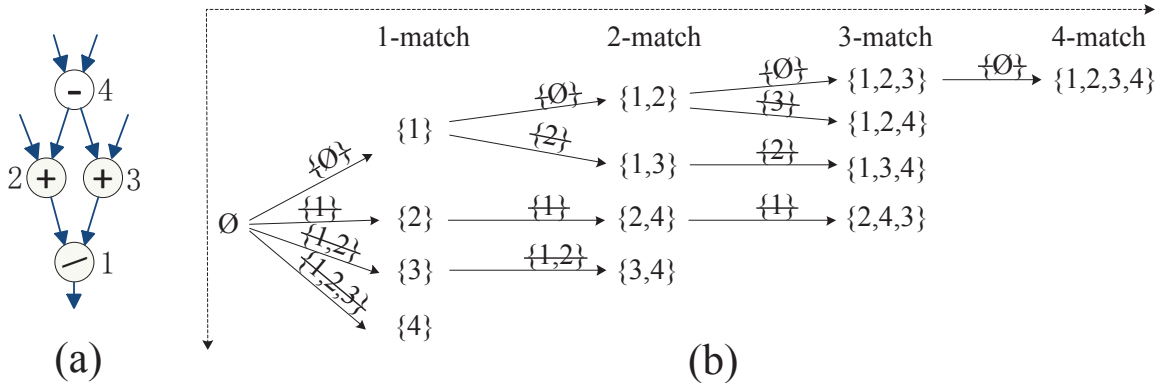


Figure 3.2: The subgraph enumeration process for a simple DFG

4 1-subgraphs ({1},{2},{3},{4}), 4 2-subgraphs ({1,2},{1,3},{2,4},{3,4}), 2 3-subgraphs ({1,2,3},{2,3,4}) and 1 4-subgraph ({1,2,3,4}).

The pseudo code of the algorithm is shown in Algorithm 1. Every time a subgraph is generated, it is checked under size constraint and convexity constraint (line 11, algorithm 1). It is worthy note that the algorithm can also tuned to generate disjoint subgraphs (replace line 10 in algorithm 1 by returning not only neighbor nodes but also disjoint nodes).

3.3 An Extended Algorithm for I/O Constrained Enumeration

In this section, we present a very flexible algorithm for exact enumeration of subgraphs under I/O constraints. The algorithm is based on our proposed size constrained enumeration algorithm . The algorithm can be tuned to generate all possible subgraphs or only connected subgraphs.

3.3.1 Overview

The subgraph enumeration algorithm we propose is depicted in Algorithm 2. It is a depth first searching algorithm by a recursive process. Our algorithm accepts directed acyclic graph as input and generates all possible subgraphs or only connected subgraphs. The algorithm traverses the nodes in the application graph. The results of the algorithm are

Algorithm 2 Pseudo code for the extended algorithm

Input: Graph G
Output: CS - a complete set of enumerated subgraphs

```
1:
2: R = ∅ // R is used to record the deleted nodes
3: for each node n ∈ G do
4:     M = {n};
5:     CS = CS ∪ M;
6:     call DepthFirstEnumeration(M, R);
7:     R = R ∪ n;
8: end for
9:
10: Procedure DepthFirstEnumeration(M, R)
11: for each node n ∈ NodeFilter(M) do
12:     M' = M ∪ n;
13:     if !permanentOutputCheck(M') then
14:         return;
15:     end if
16:     if !permanentInputCheck(M') then
17:         return;
18:     end if
19:     if !inputCheck(M')||!outputCheck(M') then
20:         ResolveIOViolation(M', R);
21:     else
22:         CS = CS ∪ M';
23:         DepthFirstEnumeration(M', R);
24:     end if
25:     R = R ∪ n ∪ Pred(G, n)orR = R ∪ n Succ(G, n)orR = R ∪ n;
26: end for
```

stored in a subgraph set.

First, 1-subgraphs are generated by single nodes of the input graph (line 4, Algorithm 2). An i-subgraph is built from an i-1 subgraph by including one of the nodes returned by the function *NodeFilter* (line 11, Algorithm 2). The algorithm checks the number of permanent outputs ($PerOutput(G, M)$) of subgraph M and the number of permanent inputs of M ($PerInput(G, M)$) (lines 13,16, Algorithm 2). The permanent outputs of a subgraph M are the nodes in M which have an ongoing edge exiting M and are connected to either: 1) a node in V^+ (external nodes) or 2) an invalid node or 3) a removed node. The permanent inputs of a subgraph M are the nodes which have an ongoing edge entering into M and belong to either: 1) V^+ (external nodes) or 2) invalid nodes or 3) removed nodes. Considering the subgraph $M = \{1, 3\}$ in Fig.3.2, node 2 is a permanent input of M (node 2 is a removed node for M {1,3}), and node 1 is a permanent output of M (node 1 is connected

an external node). As the permanent inputs and at least one successor of each permanent output have been excluded from M, it is always $|IN(G,M)| \geq |PerInput(G,M)|$ and $|OUT(G,M)| \geq |PerOutput(G,M)|$. In other words, if the number of permanent outputs of M or the number of permanent inputs of M is bigger than the output constraint or the input constraint, the output violation or the input violation cannot be resolved. In any of the two cases, the algorithm stops further searching for the subgraphs involving the current subgraph.

If the number of permanent outputs and the number of permanent inputs is smaller than the outputs constraint and the inputs constraint, the subgraph should be checked for violation of input/output constraints (line 19, Algorithm 2). If there is no violation of the I/O constraints, the subgraph is added to the subgraph set (line 22, Algorithm 2). Otherwise, a function aiming to find possible nodes to reduce the number of inputs or the number of outputs is called (line 20, Algorithm 2). In order to avoid multiple identification of each subgraph, each time the newly added node is deleted for the following iterations.

3.3.2 Convexity

In order to avoid generating non-convex subgraph, a filtering function is utilized in our algorithm. Algorithm 3 presents the pseudo-code of the function $NodeFilter$. Non-convex subgraphs are automatically ruled out by the function. In other words, each derived subgraph that is generated by absorbing one of the nodes returned by $NodeFilter$ is a convex subgraph. Therefore, the convexity check can be omitted in our algorithm. We formulize this in Lemma 3.3.1.

Lemma 3.3.1 *Given a convex DFG G, a convex subgraph $M(V_m, E_m) : M \in G$, if a node $u \in NodeFilter(M)$, then the derived subgraph $M' = M \cup u$ is a convex subgraph.*

Proof In the first case, $u \in IPred(G,M)$, $Succ(G,u) \cap IPred(G,M) = \emptyset$ and M is convex. Let assume M' is a non-convex subgraph. Then, there exists at least one path from u to another node $v \in M$, which involves a node $w \notin M'$ and w has an ongoing edge entering into M. Thus, w is a successor of u and an input of M. Obliviously, it contradicts with $Succ(G,u) \cap IPred(G,M) = \emptyset$. In the second case, $u \in ISucc(G,M)$, $Pred(G,u) \cap ISucc(G,M) = \emptyset$ and M is convex. Similarly, we can also prove it by

Algorithm 3 Pseudo code for the *NodeFilter* function

Input: M - the Match
Output: FNS - Filtered Node Set

```
1:  Procedure NodeFilter(M)
2:  FNS = ∅;
3:  for each node n ∈ IPred(G, M) do
4:      if Succ(G, n) ∩ IPred(G, M) == ∅ then
5:          FNS = FNS ∪ n;
6:      end if
7:  end for
8:  for each node n ∈ ISucc(G, M) do
9:      if Pred(G, n) ∩ ISucc(G, M) == ∅ then
10:         FNS = FNS ∪ n;
11:     end if
12: end for
13: if !CONNECTED_MATCHES_ONLY then
14:     if |OUT(G, M)| < OUT_max then
15:         FNS = FNS ∪ Disc(G, M);
16:     end if
17: end if
18: return FNS;
```

contradiction. In the third case, $u \in Disc(G, M)$ and M is convex. According to the definition of disconnected nodes of a subgraph, we know that there is neither a path from u to w or from w to u, where $w \in M$. Therefore, the derived M' is convex.

As an example in Fig.3.1, the *NodeFilter* function returns $\{7\} \cup \{1\} \cup \{2, 4, 5, 8, 9\}$ for the subgraph $\{3, 6\}$ ($\{7\} \subseteq IPred(G, \{3, 6\})$, $\{1\} \subseteq ISucc(G, \{3, 6\})$ and $\{2, 4, 5, 8, 9\} \subseteq Disc(G, \{3, 6\})$). The derived subgraphs $\{3, 6, 7\}$, $\{3, 6, 1\}$,... , $\{3, 6, 9\}$ are convex

In addition, the *NodeFilter* function can be used to generate all possible subgraphs or only connected subgraphs (line 13-17, Algorithm 3). To enumerate only connected subgraphs, the function returns only connected nodes ($IPred(G, M)$ and $ISucc(G, M)$) of the current subgraph. To enumerate all possible subgraphs, the disconnected nodes of the current subgraphs are also returned by the function.

Furthermore, we have observed that, given a subgraph M, if $|OUT(G, M)| = OUT_{max}$, the subgraph $M' = M \cup u$, where $u \in Disc(G, M)$, is an output constraint violated subgraph that cannot be resolved in most cases (only few of them can be resolved, especially in a big DFG). Considering the subgraph $\{3, 6\}$ in Fig. 3.1, let assume the output constraint is 1. As $|OUT(G, 3, 6)| = OUT_{max}$, for the subgraph $M' = M \cup u$, where $u \in Disc(G, \{3,$

6}) = {2, 4, 5, 8, 9}, $|OUT(G, M')| > OUT_{max}$. Only the derived subgraph {3, 6, 4} can be resolved by adding the node 1 ($|OUT(G, \{3, 6, 4, 1\})| = OUT_{max}$). That means if we can prune the mentioned most cases, the search spaces can be reduced radically. In the function, we use a strategy: the disconnected nodes are returned only when $|OUT(G, M)| < OUT_{max}$. This strategy is safe: if the output constraint violated subgraph M' can be resolved and M'' is the valid subgraph obtained through the resolving of M', there must be a node that is a successor of M and a successor of the disconnected node. Thus, M'' can be also obtained by continuously adding connected nodes to M. For example, the subgraph $M'' : \{3, 6, 4, 1\}$ can be obtained by adding node the 1 to $M : \{3, 6\}$ and adding the node 4 to $\{3, 6, 1\}$.

3.3.3 I/O Constraints

When the current subgraph violates the I/O constraints, we use a function to determine if a larger subgraph that satisfies the I/O constraints could be derived by adding nodes (see Fig.3.3). We utilize a similar approach as described in [Chen 2007]. We represent the nodes between a node n and a subgraph M by $Btw(n, M) = \{v | v \in (Succ(G, n) \cap Pred(G, M))$ or $v \in (Pred(G, n) \cap Succ(G, M))\}$. For example, $Btw(5, \{0, 1, 2\}) = \{3, 4\}$ in Fig. 3.3(a) and $Btw(0, \{3, 4, 5\}) = \{1, 2\}$ in Fig. 3.3(b).

If the output constraint is violated for the current subgraph M, we find out all the nodes that could possibly reduce the number of outputs (we call them as output resolving nodes). An output resolving node can only come from Succ(G, M). It has exactly two immediate predecessors in M or Succ(G, M), meanwhile, it is a successor of at least $((|OUT(G, M)| - |PerOutput(G, M)|)/(|OUT_{max}| - |PerOutput(G, M)|))$ output nodes. Once an output resolving node is found, the node and the nodes between it and M are added. Assuming $OUT_{max} = 1$, fig. 3.3(a) shows an example of resolving an output violation by adding {5} and {3,4} to M, where {5} is an output resolving node and {3,4} are the nodes between the node 5 and the subgraph M.

Similarly, we define the input resolving node n as a node that could possibly reduce the number of inputs of the current subgraph M by adding n and the nodes between n and M to M. An input resolving node can only come from $Pred(G, M)$. It has no input or a sibling in M or $Pred(G, M)$. For example, assuming $IN_{max} = 2$ in Fig. 3.3(b), node

Algorithm 4 Pseudo code for the *ResolveIOViolation* function

Input: M - the subgraph

1: **Procedure** $ResolveIOViolation(M, R)$
2: **if** $!outputCheck(M)$ **then**
3: $ORN = all\ \ the\ \ possible\ \ output\ \ resolving\ \ nodes$;
4: $resolve\_nodes = \{v | v \in ORN, \forall u \in ORN, v \notin Succ(G, u)\}$;
5: **end if**
6: **if** $!inputCheck(M)$ **then**
7: $IRN = all\ \ the\ \ possible\ \ input\ \ resolving\ \ nodes$;
8: $resolve\_nodes = \{v | v \in IRN, \forall u \in IRN, v \notin Pred(G, u)\}$;
9: **end if**
10: **for** each node $n \in resolve\_nodes$ **do**
11: $M' = M \cup \{n\} \cup Btw(n, M)$;
12: **if** $!PermanentOutputCheck(M')$ **then**
13: return;
14: **end if**
15: **if** $!PermanentInputCheck(M')$ **then**
16: return;
17: **end if**
18: **if** $InputCheck(M')\&\&OutputCheck(M')$ **then**
19: $MS = MS \cup M'$;
20: $DepthFirstEnumeration(M', R)$;
21: **else**
22: $ResolveInputViolation(M', R)$;
23: **end if**
24: $R = R \cup n \cup Pred(G, n) or R = R \cup n\ Succ(G, n) or R = R \cup n$;
25: **end for**

1 and node 2 are the resolving nodes for M, i.e., the input violation can be resolved by adding {1,2} to M.

In order to resolve I/O violation, we may add not only one node to the current subgraph, but also the nodes between the resolving node and the subgraph. Lemma 3.3.1 guarantees all the subgraphs are convex when we only add one node to the current subgraph. In this case, we could easily extend lemma 3.3.1, such that all the subgraphs are still convex when more than one nodes are added (this is formulized in lemma 3.3.2).

Lemma 3.3.2 *Given a convex DFG G (V,E), a convex subgraph* $M(V_m, E_m) : M \in G$, *a node* $u \in V$ *but* $u \notin V_m$, *if* $M_1 = \{M \cup u \cup Btw(u, M)\}$, *the following condition holds*

M_1 *is convex.*

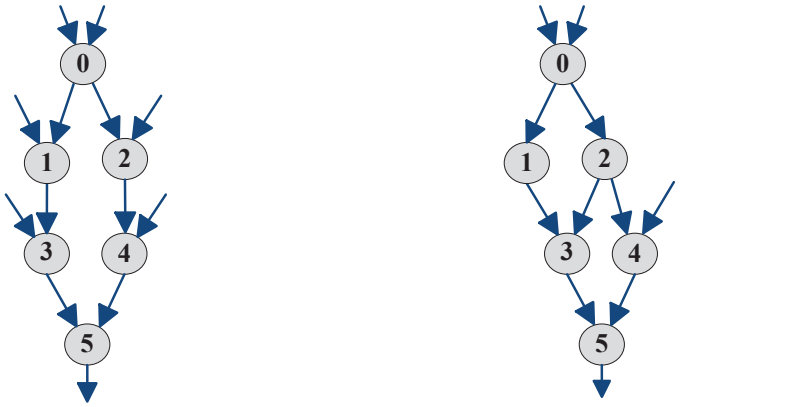


Figure 3.3: Resolve I/O constraints violation

Proof Let assume M_1 is non-convex. As M is convex, there must exist a path from a node $a \in u \cup Btw(u, M)$ to a node $b \in M$, which involves a node $w \notin M_1$. Obviously, $w \in Btw(u, M)$. As we know $Btw(u, M) \subseteq M_1$, then $w \in M_1$. Thus, a contradiction is reached. So M_1 is convex.

During the process of resolving the I/O constraint violation, more than one node could be possibly absorbed to the current subgraph. In this case, we could delete the current resolving node in the following procedures, such that all the subgraphs are still identified only once .

3.3.4 Data Structures and Calculations

During the process of subgraph enumeration, a lot of calculations have to be frequently performed. Thus, data-structures and calculations that are time efficient are required. As bit vectors are ideal for performing operations (e.g., union, intersection) on sets, we use bit vectors to store subgraphs and other subsets of the DFG. As Pred(G,P), Succ(G,P), IN(G,P) and OUT(G,P) are intensively used in the algorithm, we present the calculation of them. The calculation of Pred(G,u) and Succ(G,u) is simple and can be done recursively.

Thus, in this thesis, we do not present it.

Calculation of $Pred(G, M')$, $Succ(G, M')$, $Disc(G, M')$, $OUT(G, M')$ and $IN(G, M')$ in DepthFirstEnumeration function: In function DepthFirstEnumeration, the predecessors, successors, disconnected nodes, the outputs and inputs are calculated in each iteration. Each time a new subgraph M' is generated by adding a node u to the subgraph M, the corresponding subsets should be updated. Formulas 3.1 - 3.5 lists all the calculations required in the DepthFirstEnumeration function.

$$Pred(G,M')=\left\{\begin{array}{rl} Pred(G,M)-\{u\} & if\ u\in Pred(G,M) \\ Pred(G,M)\cup Pred(G,u)-M & if\ u\in Succ(G,M) \\ Pred(G,M)\cup Pred(G,u) & if\ u\in Disc(G,M) \end{array}\right\} \tag{3.1}$$

$$Succ(G,M')=\left\{\begin{array}{rl} Succ(G,M)\cup Succ(G,u)-M & if\ u\in Pred(G,M) \\ Succ(G,M)-\{u\} & if\ u\in Succ(G,M) \\ Succ(G,M)\cup Succ(G,u) & if\ u\in Disc(G,M) \end{array}\right\} \tag{3.2}$$

$$Disc(G,M') = G - Pred(G,M') - Succ(G,M') - M' \tag{3.3}$$

$$OUT(G,M')=\left\{\begin{array}{ll} OUT(G,M)\cup\{u\} & if\ u\in Pred(G,M), ISucc(G,u)\subsetneq M' \\ OUT(G,M) & if\ u\in Pred(G,M), ISucc(G,u)\subseteq M' \\ \{u\}\cup\{v|v\in OUT(G,M), ISucc(G,u)\subsetneq M'\} & if\ u\in Succ(G,M) \\ OUT(G,M)\cup\{u\} & if\ u\notin Dis(G,M) \end{array}\right\} \tag{3.4}$$

$$IN(G,M')=\left\{\begin{array}{rl} IN(G,M)\cup IPred(G,u)-\{u\} & if\ u\in Pred(G,M) \\ IN(G,M)\cup IPred(G,u)-M' & if\ u\in Succ(G,M) \\ IN(G,M)\cup IPred(G,u) & if\ u\in Disc(G,M) \end{array}\right\} \tag{3.5}$$

Calculation of $Pred(G, M')$, $Succ(G, M')$, $Disc(G, M')$, $OUT(G, M')$ and $IN(G, M')$ in ResolveIOViolation function: To resolve the I/O violation, more than one node could be added to the subgraph M. The set of added nodes includes a resolving node u and

the nodes $Btw(u, M)$. According to the definition of resolving node, it can only belong to $Pred(G, M)$ or $Succ(G, M)$. The $Disc(G, M')$ remains unchanged. The calculations carried out in ResolveIOViolation funtion are dipicted in formulars 3.6 -3.9.

$$Pred(G, M')=\left\{\begin{array}{ll} Pred(G, M) - \{u\} - \{Btw(u, M)\} & if\ u \in Pred(G, M) \\ Pred(G, M) \cup Pred(G, u) - M & if\ u \in Succ(G, M) \end{array}\right\} \tag{3.6}$$

$$Succ(G, M')=\left\{\begin{array}{ll} Succ(G, M) \cup Succ(G, u) - M & if\ u \in Pred(G, M) \\ Succ(G, M) - \{u\} - \{Btw(u, M)\} & if\ u \in Succ(G, M) \end{array}\right\} \tag{3.7}$$

$$OUT(G, M')=\left\{\begin{array}{ll} OUT(G, M) \cup \{v | v \in \{u\} \cup Btw(u, M), ISucc(G, u) \subseteq M'\} & if\ u \in Pred(G, M) \\ \{v | v \in \{u\} \cup Btw(u, M) \cup OUT(G, M), ISucc(G, u) \subseteq M'\} & if\ u \in Succ(G, M) \end{array}\right\} \tag{3.8}$$

$$IN(G, M')=\left\{\begin{array}{ll} IN(G, M) \cup \bigcup_{v \in \{u\} \cup Btw(u, M)} IPred(G, U) - P' & if\ u \in Pred(G, M) \\ IN(G, M) \cup \bigcup_{v \in \{u\} \cup Btw(u, M)} IPred(G, U) - P' & if\ u \in Succ(G, M) \end{array}\right\} \tag{3.9}$$

3.4 A Topology Based Algorithm for I/O Constrained Enumeration

In this section, we propose a new efficient algorithm for the automatic enumeration of all subgraphs under I/O constraint. Our algorithm solves the problem very efficiently by taking advantage of the topological property of data flow graph (DFG) as well as overcoming the drawbacks of the previously proposed algorithm.

3.4.1 Motivation

It is noteworthy that the well-known algorithm proposed in [Atasu 2003] starts with a topological sort on G: nodes of G are ordered such that if G contains an edge (u, v) then u appears after v in the ordering. Fig. 3.4 shows a topologically sorted graph. Then, the algorithms enumerate all possible subgraphs based on an implicit binary decision tree which is built according to the topological ordering. By applying the topological property

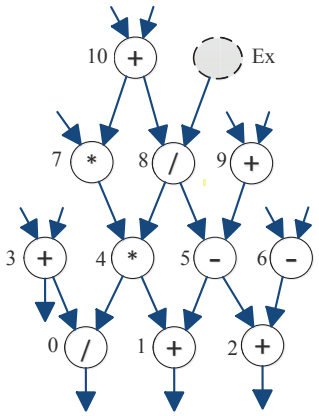


Figure 3.4: A topologically sorted data-flow graph

of G, the search space can be reduced radically. An improved algorithm [Pozzi 2006] of [Atasu 2003] further limits the search space by the permanent-input check. As an example in Fig.3.4, the subgraph {4,8} is obtained by adding the node 8 to the subgraph {4}. After inclusion of the node 8, the input node 7 becomes a permanent input (7 is smaller than the biggest number in the subgraph {4,8}). Furthermore, the external node connected to node 8 is also a permanent input. Thus, the number of permanent inputs for the subgraph {4,8} is 2. So, the subgraphs whose permanent inputs exceed the input constraint will not be considered. Therefore, the search space is further pruned.

However, we have observed that the search space still could be reduced in the following situations.

1. Given a convex subgraph M, the subgraph $M' = M \cup u$, where $u \in (Pred(G, M) - IPred(G, M))$, M' must be a non-convex subgraph. For example, the subgraph {2} is a convex subgraph in Fig. 3.4, while the subgraphs {2,8},{2,9},{2,10} are non-convex subgraphs.

2. Given a convex subgraph M, if $|OUT(G, M)| = OUT_{max}$, the subgraph $M' = M \cup u$, where $u \in Disc(G, M)$, then $|OUT(G, M')| > OUT_{max}$. In Fig. 3.4, the subgraph {0,3} has 2 outputs. Let assume the output constraint is 2. Obviously, the output constraint is violated for the subgraphs {0,3,5}, {0,3,6},{0,3,9} .

As the algorithms [Atasu 2003, Pozzi 2006] generate subgraphs based on an implicit binary search tree, they cannot prune the above mentioned search space. Therefore, we propose a novel subgraph generation algorithm that preserves all the advantages of the

Algorithm 5 Pseudo code for the topology based subgraph enumeration algorithm

Input: Graph G
Output: MS - Subgraph Set (global variable)

```
1: Procedure TopologyBasedIdentification()
2: TopologicalSort(G);
3: ONPS = FindAllOneNodeMatches(G);
4: MS = MS ∪ ONPS;
5: for each subgraph P ∈ ONPS do
6:     RecursiveMatchGeneration(M);
7: end for
8:
9: Procedure RecursiveMatchGeneration(M)
10: for each node n ∈ NodeFilter'(M) do
11:     M' = M ∪ {n};
12:     M'.SetOrder(n);//set the order of the new subgraph as n
13:     if !outputCheck(M') then
14:         return;
15:     end if
16:     if !permanentInputCheck(M') then
17:         return;
18:     end if
19:     if inputCheck(M') then
20:         MS = MS ∪ M';
21:     end if
22:     RecursiveMatchGeneration(M');
23: end for
```

algorithms [Atasu 2003, Pozzi 2006], meanwhile, the search space could be further pruned tangibly.

3.4.2 Overview

The subgraph enumeration algorithm we propose is depicted in Algorithm 5. It is a depth first searching algorithm by a recursive process. Our algorithm accepts directed acyclic graph as input and generates all possible subgraphs. The results of the algorithm are stored in a subgraph set.

The order of a node indicates the number assigned for the node by topological sort. Similarly, the order of a subgraph represents the maximum number among the nodes of the subgraph. For example, in Fig. 3.4, the order of the subgraph {0,3} is 3. Our algorithm starts with a topological sort on G (line 2, Algorithm 5): if G contains an edge (u, v) then u has a higher order than v ($u > v$). Fig. 3.4 shows a topological sorted graph.

In the algorithm, each subgraph has a tag that indicates the order of the subgraph (line 12, Algorithm 5). The node with smaller order than the order of the subgraph will not be added to the subgraph, such that the subgraph enumeration algorithm generates all the subgraphs respecting topological ordering (this implies that the node to be added to the subgraph either belongs to the predecessor nodes or the disconnected nodes of the subgraph). The algorithm traverses the sorted nodes in the application graph. First, 1-subgraphs are generated (line 3, Algorithm 5). Each generated one node subgraph has an order that is equal to the order of the only node. An (i+1)-subgraph is built from an i-subgraph by including one of the nodes returned from the *NodeFilter'* function (line 10, Algorithm 5). The order of each node returned by *NodeFilter'* is higher than the order of the i-subgraph. If the output constraint has already been violated for the i-subgraph, the algorithm stops searching (i+1) - subgraphs based on the i-subgraph: adding nodes that appear later in the topological ordering cannot reduce the number of outputs of the i-subgraph. For example, let assume the output constraint is 1, then the subgraph {0,3} (see Fig. 3.4) violates the output constraint, the violation cannot be resolved by adding nodes with higher order. This has been formulized into Lemma 3.4.1.

Lemma 3.4.1 *(Monotonicity of the Number of Outputs) : Let M_1 and M_2 be two disjunct subgraphs of G such that for every node $u_1 \in M_1$ and every node $u_2 \in M_2$, $u_1 < u_2$. Then, $OUT(G, M_1 + M_2) \geq OUT(G, M_1)$.*

Proof The proof to this lemma is straightforward and can be found in [Pozzi 2006].

The algorithm checks the number of permanent inputs (*PerInput(G,M)*) of subgraph M (line 16, Algorithm 5). If the number of permanent inputs of the current subgraph violates the input constraint, then the algorithm stops further searching for the subgraphs involving the current subgraph: the input violation cannot be resolved by further adding the nodes that appear later in the topological ordering. The permanent inputs of a subgraph M are the nodes which have an ongoing edge entering into M and either: 1) belong to V^+ (external nodes) or invalid nodes or 2) have lower order than the order of the current subgraph. Considering the subgraph {4,8} in Fig. 3.4, node 7 is a permanent input

Algorithm 6 Pseudo code for the $NodeFilter'$ function

Input: M - the Match
Output: FNS - Filtered Node Set

```
1: Procedure NodeFilter(M)
2: FNS = ∅;
3: for each node n ∈ InInput(G, M) do
4:     if Succ(G, n) ∩ IN(G, M) == ∅ then
5:         FNS = FNS ∪ n;
6:     end if
7: end for
8: if |OUT(G, M)| < OUT_max then
9:     FNS = FNS ∪ {v|v ∈ Disc(G, M), v > M.GetOrder()};
10: end if
11: return FNS;
```

of the subgraph {4,8}, and external node Ex is also a permanent input of the subgraph {4,8}. Let further assume $IN_{max} = 2$, as the number of permanent inputs of {4,8} is 2, the violation cannot be resolved anymore.

3.4.3 Search Space Pruning and Convexity

Algorithm 6 lists the pseudo-code of the function *NodeFilter'*. Due to this function, the search space is reduced radically. We define the internal inputs ($InInput(G, M) = \{u | u \in V, u \notin M, v \in M, (u, v) \in E, u > M.getOrder()\}$) of a subgraph M as the nodes that have higher order than the order of M, belong to V but not in M and have at least an edge entering into M. For example, in Fig. 3.4, node 10 is an internal input of the subgraph {4,8}. Obviously, $IN(G, M) = InInput(G, M) \cup PerInut(G, M)$. With the function, the algorithm only considers subgraph $M' = M \cup u$, where $u \in InInput(G, M)$ and $Succ(G, u) \cap IN(G, M) = \emptyset$ instead of considering subgraph $M'' = M \cup u$, where $u \in Pred(G, M)$ (lines 3-7, Algorithm 6). Furthermore, the function prevents the algorithm from considering subgraph $M' = M \cup u$, where $u \in Disc(G, M)$, when $|OUT(G, M)| = OUT_{max}$ (lines 8-10, Algorithm 6): adding a disconnected node to the current subgraph M increases the number of outputs.

In addition, non-convex subgraphs are automatically ruled out by *NodeFilter'* function. In other words, each derived subgraph that is generated by absorbing one of the nodes returned by *NodeFilter'* is a convex subgraph. Therefore, the convexity check can be omitted in our algorithm. We formulize this in Lemma 3.4.2.

Lemma 3.4.2 *Given a convex DFG G, a convex subgraph M* (V_m, E_m) *:* $M \subseteq G$*, if a node* $u \in NodeFilter'(M)$*, then the derived subgraph* $M' = M \cup u$ *is a convex subgraph.*

Proof In the first case, $u \in InInput(G, M)$, $Succ(G, u) \cap IN(G, M) = \emptyset$ and M is convex. Let assume M' is a non-convex subgraph. Then, there exists at least one path from u to another node $v \in M$, which involves a node $w \notin M'$ and w has an ongoing edge entering into M. Thus, w is a successor of u and an input of M. Obliviously, it contradicts with $Succ(G, u) \cap IN(G, M) = \emptyset$. In the second case, $u \in \{v | v \in Disc(G, M), v > M.getOrder()\}$ and M is convex. According to the definition of disconnected nodes of a subgraph, we know that there is neither a path from u to w or from w to u, where $w \in M$. Therefore, the derived M' is convex.

For example, in Fig. 3.4, the *NodeFilter'* function returns $\{5, 6\} \cup \{3, 4, 7\}$ for the subgraph $\{2\}$ ($\{5, 6\} \subseteq InInput(G, \{2\})$ and $\{3, 4, 7\} \subseteq Disc(G, \{2\})$). The derived subgraphs {2,3},{2,4},{2,7} are convex. For the subgraph {4,8}, node set {9} is returned by the function. The derived subgraph {4,8,9} is convex.

3.4.4 Data Structure and Calculations

The calculations of the subsets of the DFG involved in the algorithm are presented in this section. According to the definition of $Disc(G, M)$, we know that $Disc(G, M) = G - M - Pred(G, M) - Succ(G, M)$. We also know that the order of each node of $Succ(G, M)$ is lower than the order of the subgraph M and the order of each node of the subgraph M is lower than or equal to the order of the subgraph M. Therefore, $\{v | v \in Disc(G, M), v > M.getOrder()\} = \{v | v \in G - Pred(G, P), v > P.getOrder()\}$. So, we can use $Pred(G, M)$ instead of $Disc(G, M)$ to calculate the set $\{v | v \in Disc(G, M), v > M.getOrder()\}$.

Calculation of $Pred(G, M')$, $PerInput(G, M')$, $Output(G, M')$, $InInput(G, M')$ and $IN(G, M')$: Assuming a new subgraph M' is obtained by adding a node u to an old subgraph M. The permanent inputs, internal inputs, inputs and outputs are changed after adding the node to the old subgraph M. Thus, the calculation for these is necessary.

Our algorithm computes these by analyzing the old I/O nodes as well as the node u (see formulas (3.10)-(3.14)).

$$Pred(G,M')=\left\{\begin{array}{ll} Pred(G,M)-\{u\} & if\ u\in Pred(G,M) \\ Pred(G,M)\cup Pred(G,u) & if\ u\notin Pred(G,M) \end{array}\right\} \tag{3.10}$$

$$OUT(G,M')=\left\{\begin{array}{ll} OUT(G,M)\cup\{u\} & if\ u\in Pred(G,M), ISucc(G^+,u)\subsetneq M' \\ OUT(G,M) & if\ u\in Pred(G,M), ISucc(G^+,u)\subseteq M' \\ OUT(G,M)\cup\{u\} & if\ u\notin Pred(G,M) \end{array}\right\} \tag{3.11}$$

$$PerInput(G,M')=\{v|v\in InInput(G,M),v<u\}\cup PerInput(G,M) \tag{3.12}$$

$$InInput(G,M')=\{v|v\in InInput(G,M),v>u\}\cup IPred(G,u) \tag{3.13}$$

$$IN(G,M')=PerInput(G,M')\cup InInput(G,M') \tag{3.14}$$

Fig. 3.5 shows the search space for the generation of all feasible subgraphs involving node 1 in Fig. 3.4. To make clear the presentation, we assume the input and output constraints are not imposed. Firstly, the nodes to be added to the subgraph {1} are returned. According to the *NodeFilter'* function, only node of the set {4,5}∪ {2,3,6} can be added to the subgraph $\{1\}(\{4,5\}\subseteq InInput(G,\{1\})$ and $\{2,3,6\}\subseteq Disc(G,1))$. As the algorithm produces new subgraphs by extending previous subgraph with one node each time, the search space is split into 5 branches - {1,2},{1,3},{1,4},{1,5},{1,6}. Based on the subgraph {1,6}, the filtering function returns an empty set: $InInput(G,\{1,6\})=\emptyset$ and $\{v|v\in G-Pred(G,\{1,6\}),v>\{1,6\}.getOrder()\}=\emptyset$. Thus, the algorithm stops the searching for the subgraphs involving the subgraph {1,6}. For the subgraph {1,5}, nodes from $\{6\}\cup\{9\}$ will be added to it respectively.

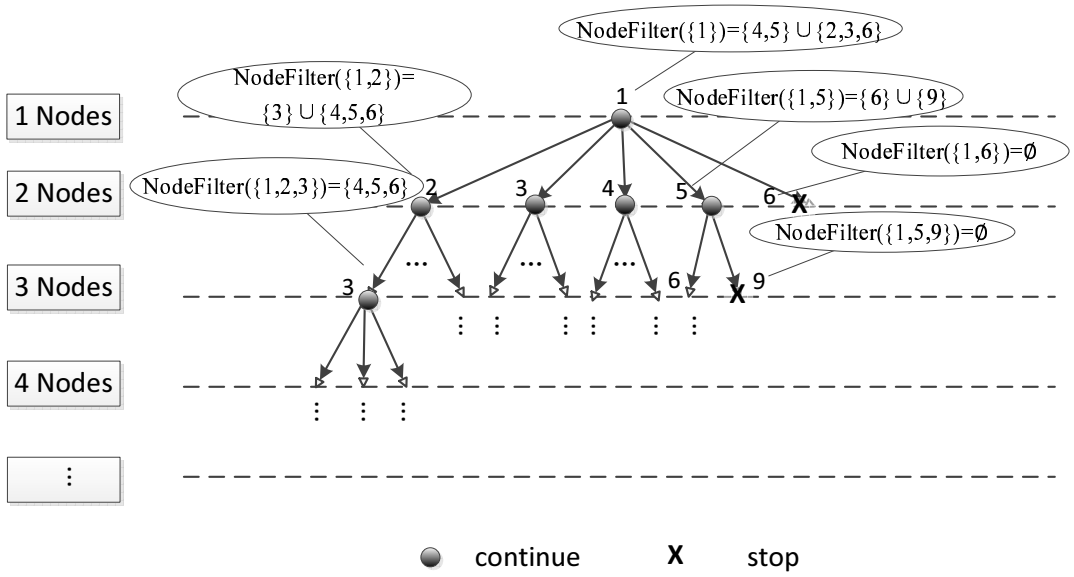


Figure 3.5: Illustration of the search tree for the generation of all feasible subgraphs involving node 1 in Fig. 3.4

3.4.5 Complexity Analysis

In order to give the worst-case bound of the proposed algorithm, we transform the pseudo code in Algorithm 5 and Algorithm 6 to a equivalent pseudo code with visible upper bound (see Algorithm 7). We first move the filter function into the recursive function. According to the algorithm, only immediate predecessor or disconnected node can be added to the current subgraph M (lines 6, 23, Algorithm 7). Adding an immediate predecessor to M may create some permanent inputs (line 11, Algorithm 7). If the immediate predecessor has at least one successor that is not in M, then a new output is created. Adding a disconnected node to M must create a new output and may create some permanent inputs (lines 25, 26, Algorithm 7).

In Algorithm 7, each iteration has three recursive calls (lines 14,17,27, Algorithm 7). Each recursive call is activated only after the increase of outputs or permanent inputs. As the $|\mathrm{OUT(G,M)}| \leq \mathrm{OUT_{max}}$ and $|\mathrm{PerInput(G,M)}| \leq \mathrm{IN_{max}}$, the maximal depth of the recursive search is $\mathrm{OUT_{max}} + \mathrm{IN_{max}}$. The while loop executes at most $|G|$ times. Thus, the number of iterations should be no more than $|G|$. Assume the union operation and intersection operation can be proceeded in O(1) time, the complexity of the algorithm is therefore $O(|G|^{IN_{max}+OUT_{max}})$. A recent research proves that the time complexity of the algorithm proposed in [Pozzi 2006] is also $O(|G|^{IN_{max}+OUT_{max}})$ [Reddington 2011].

The algorithm [Pozzi 2006] and our algorithm are the only two existing algorithms with polynomial time complexity in terms of I/O constraints.

3.4.6 Further Improvement

To reduce search space, the algorithm presented in section 3.4.2 only stops the current searching when the output constraint is violated or the number of permanent inputs exceeds the input constraint. Similar to the approach presented in subsection 3.3.3, when the current subgraph violates the inputs constraint, we use a function called *ResolveInputViolation()* (see Algorithm 8) to determine if a larger subgraph that satisfies the inputs constraint could be derived by adding nodes (thus, we add a calling to the function *ResolveInputViolation()* in Algorithm 5 when the input constraint is violated).

Calculation of $Pred(G, M')$, $Succ(G, M')$, $Disc(G, M')$ and $OUT(G, M')$, and $IN(G, M')$ in *ResolveInputViolation* function: In function *ResolveInputViolation*, a new subgraph M' is obtained by adding a resolving node u and $Btw(u, M)$ to a previous subgraph M. The calculation for subsets of M' is different from the above calculations (see formulas (3.15),(3.16)). As u is a resolving node, it can only belong to $Pred(G, M)$.

$$Pred(G, M') = Pred(G, M) - \{u\} - Btw(u, M) \tag{3.15}$$

$$OUT(G, M') = OUT(G, M) \cup \{v | v \in \{u\} \cup Btw(u, M), ISucc(G^{+}, u) \subsetneq M'\} \tag{3.16}$$

3.5 Summary

In this chapter, we presented the various subgraph enumeration algorithms that enumerate subgraphs under different design constraints. We first detailed the size constrained subgraph enumeration algorithm. The algorithm is very scalable in benefit of avoiding multiple identifications of any subgraph. After that, an I/O constrained subgraph enumeration algorithm that is based on and extends the size constrained algorithm was introduced. The algorithm can be tuned to generate all valid subgraphs or only connected subgraphs. Finally, we also presented a new I/O constrained subgraph enumeration algorithm. The

algorithm solves the problem efficiently by taking advantage of the topology property of data-flow graphs when the number of I/O is considered as design constraint.

Algorithm 7 The equivalent pseudo code with visible upper bound for the pseudo code presented in Algorithm 5 and Algorithm 6

Input: G - the Graph
Output: MS - Match Set (global variable)

1: **Procedure** $RecursiveMatchGeneration(M)$
2: $n = M.GetOrder()$;
3: **while** $(n < |G| \&\& |OUT(G,M)| \leq OUT_{max} \&\& |PerInput(G,M)| \leq IN_{max})$ **do**
4: $n = n + 1$;
5: $M' = M \cup \{n\}$;
6: **if** $n \in InInput(G,M)$ **then**//n is an immediate predecessor of M
7: **if** $Succ(G,n) \cap IN(G,M) \neq \emptyset$ **then**
8: break;
9: **end if**
10: $MS = MS \cup M'$;
11: $PerInput(G,M') = PerInput(G,M) \cup \{v | v \in InInput(G,M), v < n\}$;
12: **if** $ISucc(G,n) \subsetneq M$ **then** // not all direct successors are in M
13: $OUT(G,M') = OUT(G,M) \cup n$;
14: $RecursiveMatchGeneration(M')$;
15: **end if**
16: **if** $|PerInput(G,M')| > |PerInput(G,M)|$ **then**
17: $RecursiveMatchGeneration(M')$;
18: **end if**
19: **end if**
20: **if** $n \in Pred(G,M)$ **then** //n is a predecessor of M
21: break;
22: **end if**
23: **if** $(|OUT(G,M)| < OUT_{max}$ **then** //n is a disconnected node of M
24: $MS = MS \cup M'$;
25: $OUT(G,M') = OUT(G,M) \cup n$;
26: $PerInput(G,M') = PerInput(G,M) \cup \{v | v \in InInput(G,M), v < n\}$;
27: $RecursiveMatchGeneration(M')$;
28: **end if**
29: $M = M'$;
30: **end while**

Algorithm 8 Pseudo code for the *ResolveInputViolation* function

Input: M - the subgraph

1: **Procedure** $ResolveInputViolation(M)$
2: $IRN = all\ \ the\ \ possible\ \ input\ \ resolving\ \ nodes$;
3: $resolve\_nodes = \{v | v \in IRN, \forall u \in IRN, v \notin Pred(G, u)\}$;
4: **for** each node $n \in resolve\_nodes$ **do**
5: $M' = M \cup n \cup Btw(n, M)$;
6: $M'.SetOrder(n)$;
7: **if** $OutputCheck(M')$ **then**
8: return;
9: **end if**
10: **if** $!permanentInputCheck(M')$ **then**
11: return;
12: **end if**
13: **if** $inputCheck(M')$ **then**
14: $MS = MS \cup M'$;
15: $RecursiveMatchGeneration(M')$;
16: **else**
17: $ResolveInputViolation(M')$;
18: **end if**
19: **end for**

Chapter 4

Subgraph Selection Algorithms

So far, we get a set of subgraphs produced by the subgraph enumeration algorithms. The task now is to select a subset of the enumerated subgraphs in terms of different objectives. In this section, three heuristic approaches targeting to different objectives are first introduced. Then, an exact algorithm and a genetic algorithm for minimal number of matches selection are presented.

4.1 Problem Formulation

Various strategies can be used to guide the subgraph selection. In this section, we focus on three different strategies. First, code size can be one important optimization goal. In embedded design and processor design, only a small amount of on-chip memory is available to store the instructions. In the context of high-level synthesis, selecting a smallest set of subgraphs results in the most compacted code. Design space is reduced, so the high-level synthesis tool may produce results in shorter time. Next, the frequency of occurrences of a pattern indicate the possible resource sharing. In this paper, a pattern selection approach based on the frequency of occurrences of patterns is presented (the number of instances of patterns). Finally, selecting some subgraphs may increase the length of critical paths. So, subgraph selection that takes into account the latency overhead is very important. A critical path based selection is introduced in this section.

Problem 2 (subgraph/pattern selection): Given a DFG $G = (V, E)$ and a set of subgraphs/patterns identified by the enumeration step, select a subset of subgraphs in terms of the strategy such that every node in G is covered and the following constraints are satisfied.

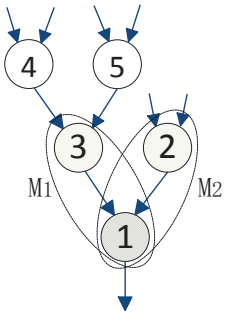


Figure 4.1: Overlapping between subgraphs

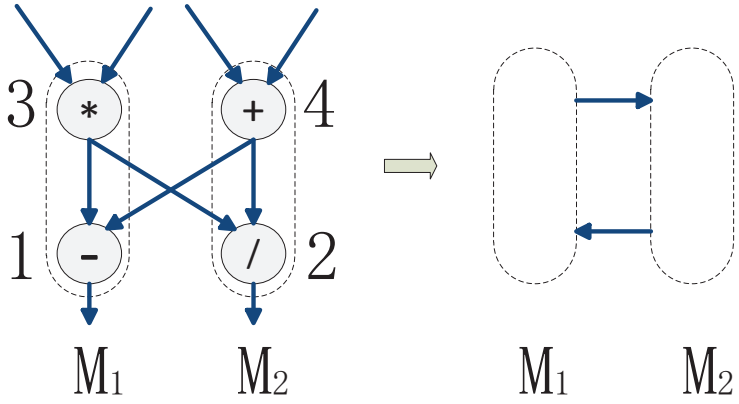


Figure 4.2: Cyclic subgraphs

- Non-overlapping;
- Acyclicity

Overlapping Constraint: Two subgraphs may have the same nodes in common. Allowing overlap may sometimes improve the resulting execution time, while unnecessarily increase the power consumption and makes the code regeneration intractable. In this thesis, we disallow overlap between any subgraph. For example, if M_1 is selected then M_2 cannot be selected in Fig. 4.1.

Acyclicity Constraint: A cycle could be generated, if two subgraphs provide data for each other. In such case, a deadlock is occurred between the two subgraphs. In Fig. 4.2, subgraphs M_1 and M_2 are cyclic subgraphs. A cycle exists between the two subgraphs.

Prior to the detail introduction of the three algorithms, the common constraints (overlapping, acyclicity) should be considered.

Overlapping: Every time a match (M) or a pattern (P) is considered as a selection

candidate, we check wether overlapping is occurred between the candidate and the already covered nodes (C).

$$M \cap C = \emptyset \tag{4.1}$$

It should be noted that the overlapping check is slightly different if the candidate is a pattern (see formula 4.2). Here, we check every match of the pattern candidate. Please note that it is possible to build a conflict graph to quickly remove overlapped subgraphs during the selection process (see section 2.4).

$$M \cap C = \emptyset, \forall M \in P \tag{4.2}$$

Acyclicity: To ensure there is no cycle between the selected matches and the current candidate (M), a cycle check should be performed. If the current candidate satisfies the following statement, then no cycle exists between it and the other selected matches.

$$Succ(G, M) \cap Pred(G, M) = \emptyset \tag{4.3}$$

If the candidate is selected and passes the overlapping check and acyclicity check, it is collapsed into a super node.

4.2 Heuristic Algorithms

As exact solution algorithm is time-consuming and usually fails to give the result due to too long runtime or memory overflow. For example, a branch-and bound algorithm (see section 4.3) takes 20 seconds to find a minimal number of matches that completely cover the application graph of the benchmark DOTPRODUCT (4×4). The application graph of DOTPRODUCT (4×4) is a small graph that contains only 8 nodes. However, the exact algorithm fails to produce results in one hour for the benchmark DOTPRODCUT (6×6) that contains 12 nodes. Thus, an efficient heuristic approach is required. In this subsection, three heuristic algorithms targeting to different objectives are shown.

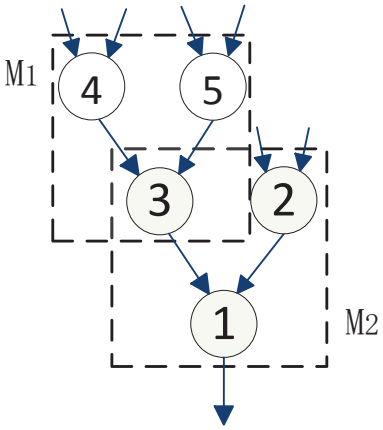


Figure 4.3: Select the subgraph with less overlapping

4.2.1 Minimal Number of Matches Selection

As we know, the number of selected matches (subgraphs) is directly related to the code size. Hence, selecting the minimal number of matches from a given data-flow graph can be an interesting strategy at the selection step.

In the approach, a priority value is given to every match. We always first select the matches with highest priority. The algorithm first calculates the priority of each match. The priority is calculated according to the following formula.

$$F_i = N + E + \alpha * 1/O(M_i) \tag{4.4}$$

where N is the number of nodes in match M_i to favorite large size matches, E is the number of internal edges of match M_i (as we know, the internal data flows are free of multiplexors, the value E can be used to roughly evaluate the save of multiplexors), and O(M_i) is the number of other matches that overlap with M_i and α is a parameter that represents the weight of overlapping. The intuition behind the using of 1/O(M_i) is to select the match that overlaps with less other matches when overlapping is disallowed such that we may have more possibilities to get a better solution. For example, assume the parameter α is 2 and the maximum size of all candidate matches is 3 in Fig. 4.3 (only connected matches are considered). We thus have 5 1-subgraphs, 4 2-subgraphs and 4 3-subgraphs. Selecting M_2 results in the selection of the set of subgraphs {M_2, {4}, {5}}. Selecting M_1 leads to the selection of a smaller set of subgraphs {M_1,{1,2}}. We prefer the match M_1 to the match M_2 since M_1 has less overlapping ($F_{M_1} = 3+2+2*1/8 > F_{M_2} = 3+2+2*1/9$).

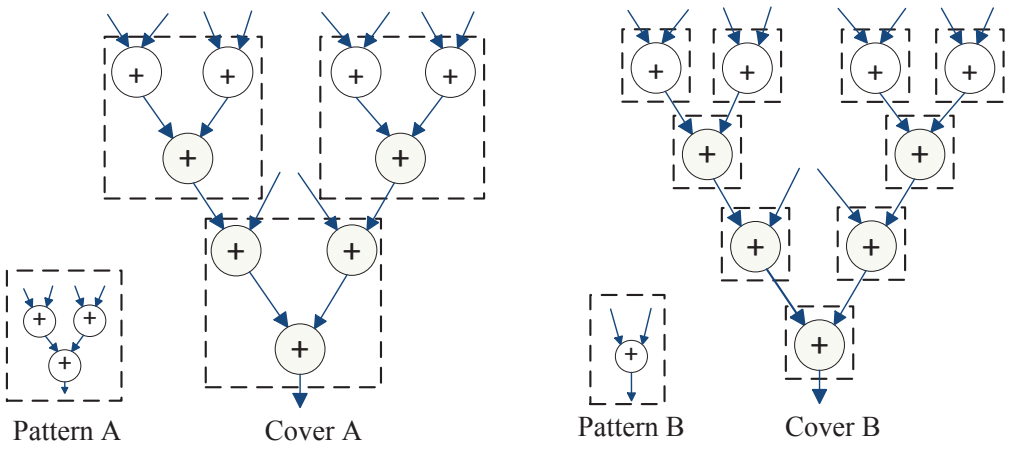


Figure 4.4: Select the pattern with more nodes

4.2.2 Frequency of Occurrences Based Pattern Selection

Other than the preceding selection approach that accepts the set of subgraphs generated by the subgraph enumeration step as input, a set of patterns should be provided to this pattern based selection strategy as input. The set of patterns is generally collected using a graph isomorphism algorithm. Given two subgraphs a and b, if a is isomorphic to b, a pattern p is created, and the subgraphs a and b are recorded in the pattern p.

Selecting a minimum set of distinct patterns may result in less area cost in high-level synthesis taking component reuse into account. In this sense, the pattern with higher frequency of occurrences seems to be more interesting. Generally, smaller patterns have higher frequency of occurrences. Nevertheless, small patterns may not lead to good performance improvement or obvious area reduction. In the extreme case, if only 1-subgraphs are selected, there is no change at all to the source code. The authors [Guo 2003] have proposed an interesting objective function. Inspired by it, we propose a more efficient objective function (formula 4.5) to balance the weight of the size and the weight of the frequency.

$$F_i = N * |M| + \alpha * N; \tag{4.5}$$

where N is the size of the pattern, and $|M|$ is the number of matches of the pattern. An extra weight is given to the size. The weight is controlled by the parameter α. For example, assuming the parameter α is 0.3, we prefer to select P_A than P_B in Fig. 4.4 ($F_{P_A} = 3 * 3 + 0.3 * 3 > F_{P_B} = 1 * 9 + 0.3 * 1$).

4.2.3 Critical Path Based Match Selection

The previous two approaches do not carefully consider the performance overhead. Although the work in [Cong 2008, Cong 2010] tried to use a measurement to select flat patterns to make use as much as possible of parallelism, results show that the length of the critical path of the DFG is still increased in many applications. Using the example in Fig. 4.5, we assume the multiplication takes 2 cycles to execute, the addition takes 1 cycle to execute and the maximum size of the matches/patterns is 2. Then, the length of the critical path of the original DFG is 4 cycles (Fig. 4.5 (a)). If the matches M_1 and M_2 are selected, let further assume each match takes 2.7 cycles. The length of the critical path is augmented to 7.4 $(2 + 2.7 + 2.7)$ cycles after selection (Fig. 4.5).

Therefore, we propose a critical paths based match selection approach that can not only avoid the increase of the length of the critical paths but also give performance improvement. As the matches that appear on the critical paths are more likely to provide performance improvement by shrinking the height of the application graph, we initially rank every match in terms of the number of its nodes occurring on the critical paths (see formula 4.6).

$$F_i = |M \cap CP|; \tag{4.6}$$

where M is the set of nodes in the match, CP represents the set of nodes on critical paths. As an example in Fig. 4.5, the matches {1,2} and {2,4} have a higher rank than the matches {1,3} and {2,5} ($F_{\{1,2\}} = 2, F_{\{2,4\}} = 2, F_{\{1,3\}} = 1, F_{\{2,5\}} = 1$). After ranking every match candidate, we evaluate the opportunity of every match. The matches that may increase the length of the critical path are not selected. Considering the same example in Fig. 4.5, the match {2,4} is not selected due to the increase of the length of the critical path. Instead, the match {1,2} along with the one node matches {3},{4} and {5} are selected by our proposed approach (if only connected subgraphs are considered). Let assume the delay of the match {1,2} is 1.4 cycles, the length of the critical path after the selection is then 3.4 cycles $(2 + 1.4)$.

As the three proposed approaches are all heuristic algorithms and the main difference is the given objective function, we use a common pseudo-code to describe them (see

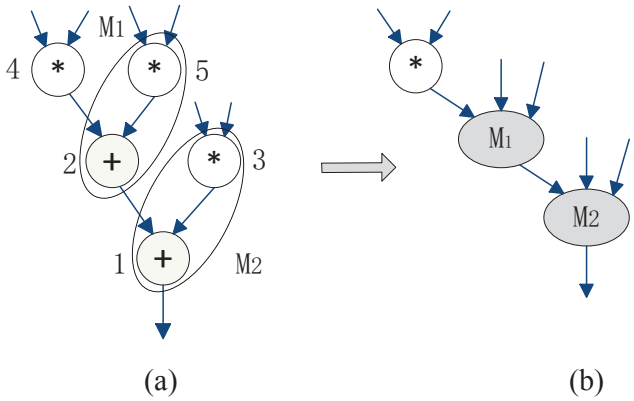


Figure 4.5: (a) The original data-flow graph (b) A selection that results in the increase of the length of the critical path

Algorithm 9 Subgraph/Pattern Selection Algorithm

Input: CS - the complete set of enumerated candidates (subgraphs/patterns); $F(i)$ - objective function

Output: SS - a subset of selected candidates

```
1: sort CS in descending order according to F(i);
2: while CS ≠ ∅ do
3:     M = the highest prioritized candidate in CS;
4:     if Overlapping(M, SS) and Cycle(M, SS) then
5:         if IncreaseCriticalPath(M) then
6:             SS = SS ∪ M;
7:         end if
8:     end if
9:     CS = CS − M;
10: end while
```

Algorithm 9).

4.3 An Exact Algorithm for Minimal Number of Matches Selection

Although the exact algorithm is time-consuming compared to heuristic algorithms or greedy algorithms, we still present a branch-and-bound algorithm to find minimal number of matches for the purpose of completeness and comparison.

4.3.1 Set Covering Problem

First of all, it is necessary to introduce the definition of the set covering problem. Then, we show that finding the minimal number of matches from a given data-flow graph is a specific case of the set covering problem.

Set Covering Problem (SCP): Given an m-row, n-column, zero-one matrix (a_{ij}), each column of the matrix is associated with a cost c_k, the set covering problem is to find a subset of the columns that covers each row of the matrix at minimal cost. Following is the integer linear programming formulation of the SCP:

$$Minimize \sum_{j=1}^{n} c_j x_j \tag{4.7}$$

$$Subject \ \ to \sum_{j=1}^{n} a_{ij} x_j \geq 1, i = 1, \cdots, m \tag{4.8}$$

$$x_i \in 0, 1, j = 1, \cdots, n \tag{4.9}$$

The variable x_j equals 1 if the column j is selected in the solution, and 0 otherwise. The equation 4.8 ensures that each row is covered by at least one column. When all the cost coefficients c_j are equal, the problem is then a particular case of SCP. It is called unicost set covering problem.

According to the definition of the set covering problem, we can easily transform the minimal number of matches selection problem to the unicost set covering problem. A simple scheme for the transformation follows :

1. Each node of the graph corresponds to a row in the matrix.
2. Each match corresponds to a column in the matrix.
3. All the matches have the same cost (c_j=1).

As SCP is a classical and well studied problem in computer science, plenty of approaches such as integer linear programming[Liao 1997], heuristic method[Caprara 1999] and ant colony optimization[Ren 2010] trying to solve it efficiently have been proposed. In

this thesis, we present a typical branch-and-bound approach that gives exact solution for SCP and a genetic algorithm that gives near-optimal solution for SCP. Please note that the value of each column (match) can be typically link with the power consumption, area cost or performance speedup. If the selection target is not reducing the code size, then the match selection problem can be transformed to the general SCP.

4.3.2 A Branch-and-Bound Algorithm

In this subsection, a typical branch-and-bound algorithm for covering the application graph with minimal number of matches is presented. Similar branch-and-bound algorithms targeting different objectives can be found in [Clark 2006, Dinh 2008]. The algorithm produces the optimal solution by using branch-and-bound manner. The pseudo code for the exact algorithm is shown in Algorithm 10.

Similarly to the heuristic algorithm, the inputs to the exact algorithm are the DFG and a list of matches. The list of matches is sorted in order of decreasing size (line 2, Algorithm 10). Each candidate match can be selected or be passed (lines 3-4, Algorithm 10). For each candidate match currently considered, it is deleted from current match list (line 7, Algorithm 10). If it is selected, we add it to the selected list (line 9, Algorithm 10). When the DFG is fully covered by the selected matches so far (line 10, Algorithm 10), we record the better solution from the current solution and the best solution so far (line 16, Algorithm 10). Then, we update the match list (line 16, Algorithm 10). A comparison is used to estimate whether the best solution based on current partial solution is better than the best solution so far (line 17, Algorithm 10). The best solution so far is the lower bound (current_best). The calculation, $|C| + (G - C)/|M_1|$, gives the best solution based on the current partial solution. The first portion of the calculation ($|C|$) is the number of matches selected so far. The rest portion ($(G - C)/|M_1|$) gives the least of number of matches that have to be used to cover the uncovered nodes ($G - C$). $|M_1|$ is the number of nodes covered by the "best" match among the left matches from the match list. If the best solution based on the current partial solution is worse than the best solution so far, all the solutions based on the current partial solution are not necessary to be explored (line 18, Algorithm 10).

Algorithm 10 The pseudo code for the exact algorithm selecting minimal number of matches

Input: G - the DFG; M - the set of all matches; P_i - the priority of the match M_i;
Output: C - the list of selected matches

```
1: Procedure Select(G, M)
2: Calculate P_i for each match;
3: Sort M in descending order in terms of P_i;
4: Branch(M_1, true, M, C);
5: Branch(M_1, false, M, C);
6:
7: Procedure Branch(S, selected, M, C)
8: M = M − S;
9: if selected == true then
10:     C = C ∪ S;
11:     if C == G then // if all the nodes are covered
12:         if |C| < current_best then
13:             current_best = |C|;
14:             return;
15:         end if
16:     end if
17:     Update M by deleting the matches overlapping with the match s;
18:     if (|C| + (G − C)/|M_1|) > current_best then
19:         return;
20:     end if
21: end if
22: Branch(M_1, true, M, C);
23: Branch(M_1, false, M, C);
```

4.4 A Genetic Algorithm for Minimal Number of Matches Selection

The idea of genetic algorithm (GA) was first introduced by Holland [Holland 1975] in the 1970s. Genetic algorithms have been widely applied to various fields including engineering, chemistry, physics, transportation, and so on. Genetic algorithms are nature-inspired optimization algorithms inspired from the genetic inheritance in the evolutionary process of nature world. In general, a genetic algorithm tries to evolve the solutions generation by generation through inheriting the good characteristics from highly adapted ancestors, while the less adapted ancestors will be eliminated and replaced by the newly generated descendant. Genetic algorithms use probabilistic search that aims at locating globally optimal solution. The most common processes taken in genetic algorithms are presented as follows.

1. Generate an initial population;

2. Evaluate the fitness of each individual in the population;

3. **Repeat**

   select the best-fit individuals from the population;

   produce children through crossover or mutation on selected individuals;

   evaluate the fitness of the children;

   replace some least-fit or all of the ancestors by the children;

4. **Until** (max number of generations or a satisfactory solution has been found)

As described in the previous section, the pattern selection problem can be transformed into the set covering problem. The set covering problem is a typical combinatorial optimization problem, while the genetic algorithm is one of the best algorithms for solving combinatorial optimization problem, especially when the problem is NP-complete problem. Therefore, we try to apply the genetic algorithm to solve the pattern selection problem in this section.

4.4.1 Encoding

The first step for any GA is to find an appropriate representation of each solution (individual, chromosome). Intuitively, we use a n-bit binary representation as the solution structure. Each bit in the representation is associated with a subgraph (matches, gene). A value of 1 for the ith bit implies the subgraph i is selected in the solution. For convenience, we use the index i to represent the subgraph i in this section. Each subgraph is assigned a unique index either randomly or according a specific sorting order (for example, in minimal number of matches selection problem, we sort the subgraphs according to descending order of the size of subgraph). The binary representation of a solution for pattern selection problem is illustrated in Fig. 4.6. This solution represents the selection of subgraphs 1,3,5,6,7.

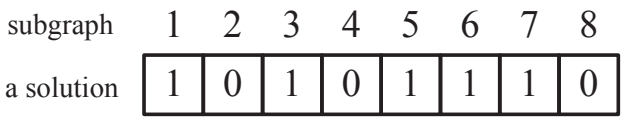


Figure 4.6: Binary representation of a solution

Algorithm 11 A randomized greedy heuristic method for generating initial population

Input: G - graph; S - subgraphs; N - the number of solutions;
Output: P - initial population

1: **Procedure** $Heursitic\_Initlialize\_Population(G, S, N)$
2: L = $\{s \in S : e_s > \gamma \times max_{s \in S}|s|\}$ //generate a candidate list
3: **while** $(n <= N)$ **do** //the number of solutions to be generated
4: $I = Generate\_A\_Solution(L, S)$;
5: $P = P \cup I$;
6: **end while**
7:
8: **Procedure** $Generate\_A\_Solution(L, S)$
9: **while** $(|L| > 1 \&\& (|I| < |G|)$ **do**
10: r = Randomly_Select_A_Subgraph(L); //randomly select a subgraph from the candidate list
11: $I[r] = 1$;
12: $L = L - r$;
13: $L = L - \{s \in L : s \cap r \neq \emptyset\}$; //delete the subgraphs overlapping with the subgraph
14: **end while**
15: $C = M - L$; //a complement list
16: **while** $|I| < |G|$ **do** //until a feasible solution is generated
17: r = Randomly_Select_A_Subgraph(C);
18: $I[r] = 1$;
19: $C = C - r$;
20: $C = C - \{s \in C : s \cap r \neq \emptyset\}$; //delete the subgraphs overlapping with the subgraph
21: **end while**
22: return I;

4.4.2 Generating the Initial Solution

Traditionally, in order to generate a diversity of initial solutions, all the initial solutions are randomly generated. However, an initial population with higher quality may speed-up the convergence and reduce the generations required to obtain near-optimal solutions. Thus, we use a randomized greedy heuristic method to generate some high quality solutions. Meanwhile, to keep the diversity of initial population, the rest initial solutions are randomly generated. The method used to generate some high quality initial solutions is presented in Algorithm 11.

A candidate list containing some promising subgraphs is created in the method (line

2, Algorithm 11). We use the following ratio to roughly evaluate the importance of each subgraph.

$$e_s = |s|/c_s \tag{4.10}$$

Where c_s is the cost of the subgraph s, $|s|$ is the number of nodes covered by the subgraph s. In the case of minimal matches selection problem, as the cost of each subgraph is equal to 1, $e_s = |s|$. Thus, the promising subgraphs are the subgraphs whose size is equal to or bigger than the value $\gamma \times max_{s \in S}|s|$. Based on the created candidate list, subgraphs are randomly selected (line 10, Algorithm 11) until the candidate list is empty or the graph is covered (line 9, Algorithm 11). As overlapping is disallowed, the subgraphs overlapping with the selected subgraph are deleted from the candidate list (line 13, Algorithm 11). If the solution so far cannot cover the graph, a random selection from the complement list is repeated until a feasible solution is generated (line 15-20, Algorithm 11).

4.4.3 Fitness Function

In the natural world, the individual that has a higher fitness to environment will have a better chance to survive and propagate. A fitness function, related to the objective function, is used to evaluate the quality of solutions. Formally, the fitness function for pattern selection is presented as follows:

$$f_i = \sum_{j=1}^{n} c_j \cdot s_{ij} \tag{4.11}$$

Where c_j is the cost of the jth subgraph (column), s_{ij} presents the value of jth bit in the binary representation of the solution s_i. In the case of minimal number of matches selection, c_j is always 1. Certainly, c_j can be associated to area cost or power consumption or delay according to the selection objectives. Clearly, a solution that has a higher fitness has less total cost.

4.4.4 Selection

During each generation, a proportion of the existing population is selected to reproduce (crossover) new solutions such that a new fitter population can be generated. In the

Algorithm 12 Pseudo code for the tournament selection

Input: T - the size of tournament; P - the existing population;
Output: P_1, P_2 - the selected parent;

```
1: Procedure tournament_selection(T, P)
2: S_1 = Draw_Solutions(T, P); //draw T solutions from existing population
3: S_2 = Draw_Solutions(T, P);
4: P_1 = Select_A_Fittest_Solution(S_1); //select the fittest solution
5: P_2 = Select_A_Fittest_Solution(S_2);
```

selection, fitter solutions are more likely to be selected to reproduce new solutions. Generally, proportionate selection and tournament selection are two widely used methods. The proportionate selection method proportionally selects the fitter solutions according to the probability rate of the fitness of each solution. As the calculation of probability rate for each solution may be very time-consuming, we prefer to use tournament selection. The tournament selection method is also a fitness-based selection. The method first creates two pools, each of which contains T solutions randomly drawn from the existing population. The fittest solution in the two pools is selected to produce new solutions. Obviously, the chance of selecting less fitness solutions can be decreased by increasing T. Algorithm 12 illustrates the pseudo code of the tournament selection.

4.4.5 Reproducing

To produce new solutions, two main genetic operators (crossover and mutation) are generally considered. In the following subsections, the two operators are introduced in details.

4.4.5.1 Crossover Operator

With crossover operator, the good characteristics (subgraph, bit, gene) of parents can be inherited by children. The most common crossover operators including one-point crossover operator and two-point crossover operator randomly choose a point or two points on parents' strings (solutions) and exchange segments of the parents' strings to produce children. An example of two points crossover operator is shown in Fig. 4.7.

To better inherit the good characteristics from parents, a guided fusion crossover operator inspired by [15] is utilized. Unlike one-point and two-point crossover operator, the fusion crossover operator only produces one child. It enables the parent solution to con-

Algorithm 13 The pseudo code for fusion operator

Input: G - graph; P_1 - the first parent solution; P_2 - the second parent solution;
Output: C - the child solution

```
1: Procedure Fussion_Crossover(G, P1, P2)
2: while (i <= N) do //the number of solutions to be generated
3:     if P1[i] == P2[i] then // the bits are identical in parents
4:         t = P1[i] = P2[i];
5:     else
6:         p = Random_Generator(0, 1);
7:         if (0 ≤ p ≤ f_p1/(f_p1 + f_p2)) then
8:             t = P1[i]; //copy the bit in the first parent
9:         else
10:            t = P2[i]; //copy the bit in the second parent
11:        end if
12:    end if
13:    if t == 1&&S_i ∩ C == ∅ then //the subgraph i does not overlap other subgraphs
       in C
14:        C[i] = 1;// the subgraph i is selected
15:    else
16:        C[i] = 0;
17:    end if
18:    i++;
19: end while
20: if |C| < |G| then// some nodes are not covered
21:    for n ∈ {G − C} do// for each uncovered node
22:        set = Get_Covering_Subgraphs(n); // get the subgraphs that cover node n
23:        j = Select_A_Graph(set);// select a subgraph
24:        C[j] = 1;
25:    end for
26: end if
```

tribute the bit level rather than segment level. Each bit (subgraph) in the child solution is created by copying the corresponding bit from one or other parent with a random number generator [0,1]. Let f_{p_1} and f_{p_2} be the finesses of the parents p_1 and p_2 respectively. The operator creates bit by bit to form a child solution. The bits which are identical in parents are copied to the child solution. Otherwise, if the random number is between [0, $f_{p_1}/(f_{p_1} + f_{p_2})$], then the bit in p_1 is copied to the child solution. If the random number is between ($f_{p_2}/(f_{p_1} + f_{p_2})$,1], the bit in p_2 is copied to the child solution. The pseudo code for fusion operator is illustrated in Algorithm 13.

However, like other crossover operators, the fusion crossover operator may also result in infeasible solutions (some subgraphs overlap with other subgraphs or some nodes are not covered). Here we propose a guided operation to maintain the feasibility of the solutions.

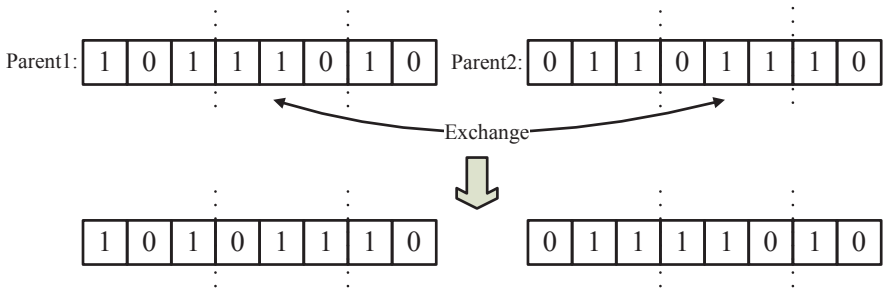


Figure 4.7: Two point crossover operator

To prevent the selected subgraph overlapping with other subgraphs in the solution, we add an additional check (lines 13-16, Algorithm 13). Once overlapping is disallowed, some nodes may not be covered. For each uncovered node, we first get a set of subgraphs that cover the node (line 22, Algorithm 13). Among the set of subgraphs, we select the subgraph i that does not overlap with the selected subgraphs in the solution C and has the highest following ratio (line 23, Algorithm 13):

$$|S_i|/c_i \tag{4.12}$$

Where c_i is the cost of the subgraph i, and S_i is the number of nodes in the subgraph i. A crossover rate ε is used to determine the number of new solutions generated by crossover operator. Assume the size of population is 100, $\varepsilon = 90\%$, as the fusion operator can only produce one child each time, then 90=100*90% pairs of parents have to be selected to produce 90 new solutions.

4.4.5.2 Mutation Operator

In order to keep the variety of solutions, a mutation operator is applied to introduce new search space. The mutation operator generally flips the chosen bit in the solution. An example of mutation operator is shown in Fig. 4.8. The bits are chosen according to a user-defined mutation rate δ . For example, assume there are 10 bits in a solution, $\delta = 0.2$, then two $(0.2 * 10)$ randomly chosen bits should be selected to invert. Similar to the crossover operator, the mutation operator may also result in infeasible solutions. We can use the methods utilized in the fusion crossover operator to make the solutions feasible.

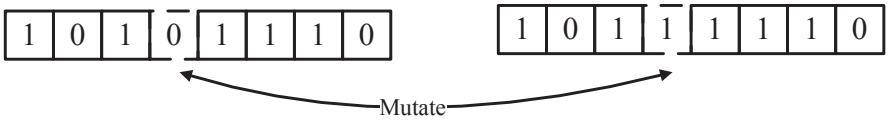


Figure 4.8: An example of mutation operator

4.4.6 Replacement

A new population is formed by using the steady-state replacement method that always keep the best solutions of the previous population and replaces the less fit solutions by the newly generated solutions (the crossover rate determines the number of less fit solutions that should be replaced). However, the genetic algorithms have a tendency to converge. When the population converges to a set of homogeneous solutions, the solutions may fall into local optimum. In order to escape from local optimum and keep the diversity of the population, a random immigrant mechanism can be used. The mechanism replaces a fraction of less fit solutions in the population by randomly generated solutions (the replacement rate β indicates the fraction of less fit solutions).

4.5 Summary

In this chapter, we introduced three different heuristics for subgraph/pattern selection. The first method targeting to select minimal number of matches that cover the whole DFG prefers to choose the bigger subgraphs with less overlapping. The second heuristic is based on the reuse of a pattern, in other word, a pattern with higher frequency of occurrences has a higher possibility to be selected. The third one takes into account the length of the critical path when a subgraph is selected.

For the purpose of comparison and completeness, a branch-and-bound algorithm for minimal number of matches selection is also proposed. However, the exact algorithm is time-consuming and the heuristics may stay sub-optimal. Thus, we presented a genetic algorithm that makes trade-off between efficiency and the quality of the results. Please note that, the genetic algorithm is flexible and can be applied to different selection targets (modification on the fitness function is enough).

Chapter 5

Code Transformation

After obtaining a set of subgraphs produced by the subgraph enumeration step, until this point, we now have a set of selected subgraphs (matches) produced by the subgraph selection step. It is required to identify whether two selected subgraphs can be implemented with the same custom function unit (this is done before selection for pattern based selection). This task can be viewed as a graph isomorphism problem. In our design, we developed a graph isomorphism algorithm. Our algorithm is an extension of the graph isomorphism algorithm VF2 [P. Cordella 2004].

After the isomorphism check, the selected subgraphs are collapsed into super nodes. Based on the collapsed graph, a piece of functionally equivalent new source code is generated. The code representation of a custom operator is also presented in this chapter.

5.1 A Graph Isomorphism Algorithm

In some cases, it is assumed that the available pattern library are already provided. Thus, a subgraph isomorphism algorithm is required to discover the occurrences of each pattern. Then, a selection is carried out according to for example the frequency of occurrences of the available patterns. However, in our design flow, the patterns are automatically extracted from the DFG of applications by performing graph isomorphism check among subgraphs: given two enumerated subgraphs G_1 and G_2, if G_1 is isomorphic to G_2, then a pattern P is created, and G_1 and G_2 are recorded in P as instances. Therefore, in this section, we only describe the graph isomorphism algorithm.

We extend the VF2 algorithm [P. Cordella 2004] by capturing some characteristics of data flow graph. The VF2 algorithm finds the mapping between the two input graphs by incrementally comparing the node pairs. Generally, the partial mappings are expanded to

Algorithm 14 A Graph Isomorphism Algorithm

Input: $G_1(V_1, E_1), G_2(V_2, E_2)$ - the two graphs;
Output: M - the mappings between the two graphs

```
1: Procedure IsomorphismCheck(G1, G2)
2: if (|V1|! = |V2|)||(|E1|! = |E2|) then
3:     return ;
4: end if
5: LN1 = all the starting nodes of G1;
6: LN2 = all the starting nodes of G2;
7: if |LN1|! = |LN2| then
8:     return ;
9: end if
10: call Match(LN1, LN2);
11:
12: Procedure Match(N1, N2)
13: Compute the set P(N1, N2) of the node pairs candidate from N1 and N2;
14: for each pair p(n1, n2) ∈ P(N1, N2)&& n1 and n2 were not considered do
15:     if NodeEquivalenceCheck(n1, n2) then
16:         M = M ∪ {(n1, n2)};
17:         call Match(ISucc(G1, n1), ISucc(G2, n2));
18:     end if
19: end for
```

new bigger partial mappings by adding a compatible neighbor node pair. A node pair is said to be compatible only when the two nodes satisfy a set of feasibility rules. The set of feasibility rules can efficiently prune the search space. The VF2 algorithm that can verify both graph isomorphism and subgraph isomorphism has been used in many application domains due to its efficiency. We further improve the algorithm by adding vertex, edge and starting node cardinality check to quickly rejects dissimilar graphs. Furthermore, the extended algorithm addresses the problem caused by non-commutative operations. In addition, the extended algorithm handles the redundant mappings when it is used as a subgraph isomorphism algorithm.

Algorithm 14 presents the overview of the proposed algorithm. The algorithm takes the two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ as input. The algorithm extends the partial mappings from top to down. It first performs the edge and vertex cardinality check. Then all the starting nodes of the two graphs are enumerated. A starting node is a node without any predecessors in the DFG. The starting node cardinality is also checked to early reject dissimilar graphs. Initially, the node pairs from the two set of starting nodes are computed. Starting from a starting node pair, the equivalence of the two nodes

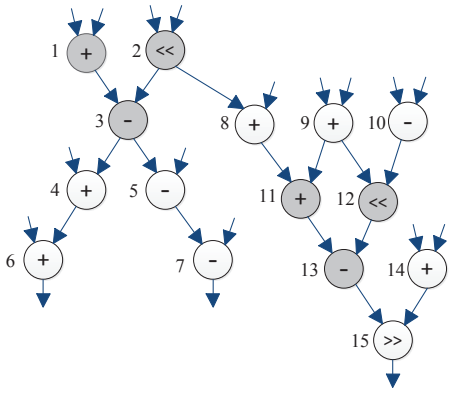


Figure 5.1: DFG from the JPEG benchmark

in the pair is evaluated. If the two nodes have the same label (the same operation) and the set of feasibility rules [P. Cordella 2004] is respected, the pair is added to the partial mapping forming a new partial mapping. A recursive process is called to perform the same computation for the successors of the two equivalent nodes.

Fig. 5.1 shows part of a DFG from the the JPEG benchmark. We use this simple but realistic example to show the isomorphism checking process. Given the two enumerated subgraphs {1,2,3} and {11,12,13} in the DFG, a set of starting nodes of them are obtained respectively: {1,2} and {11,12}. The node pairs between the two sets of starting nodes are then computed: {(1,11),(2,12)}. Assuming the node pair (1,11) is considered, the two nodes have the same label and have the same number of immediate successors. Thus, an initial partial mapping {(1,11)} is created. The partial mapping is then grown toward the immediate successors of the node 1 and the immediate successors of the node 11. At this point, a bigger partial mapping {(1,11),(3,13)} is obtained. As the node 3 and the node 13 have no immediate successors in the subgraph {1,2,3} and the subgraph {11,12,13}, the algorithm turns back to the other leading node pair (2,12). The leading node pair is compatible and can be added to the partial mapping. Finally, an entire mapping {(1,11),(2,12),(3,13)} is obtained. Therefore, the subgraph {1,2,3} is isomorphic to the subgraph {11,12,13}. In other words, the two subgraphs could be implemented with the same custom operator.

Different from the general graph isomorphism check, non-commutative operations in the graphs should be carefully considered. Commutativity states that changing the order of the operands of an operation does not change the result. For example, $a * b = b * a$.

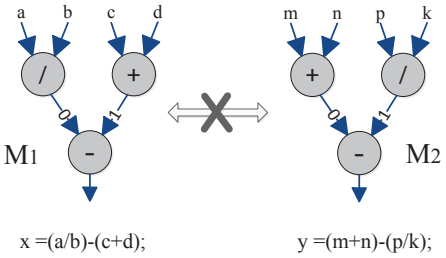


Figure 5.2: Two graphs with a non-commutative operation

However, for the graphs that contain the non-commutative operations like substraction or division, different orders of the operands may affect the result of the graph isomorphism checking. Fig 5.2 illustrates an example of two graphs with a non-commutative operation. Obviously, the graph M_1 is not isomorphic to the graph M_2 (M_1 is not functionally equivalent to M_2). Yet, with a general graph isomorphism checking, M_1 is functionally equivalent to M_2. To cope with this case, we simply assign the first edge that holds the first operand (minuend or dividend) a value "0", on the contrary, the second edge that holds the second operand (subtrahend or divisor) is assigned a value "1". Moreover, the values of the incoming edges of non-commutative operations should be checked when graph isomorphism checking is performed.

It is noteworthy that the graph edit distance used to represent the similarity of subgraphs [Cong 2008, Cong 2010] can also be measured by partial match of the graph isomorphism algorithm. In addition, the proposed graph isomorphism algorithm is able to solve subgraph isomorphism problems as it is based on the VF2 algorithm (a minor modifications required). When it is used to detect the occurrences of a pattern in a given DFG as a subgraph isomorphism algorithm, a redundancy check should be performed to guarantee the correctness. Considering the example in Fig. 5.3, without a redundancy check the pattern P_A has 4 occurrences ({(a,4),(b,6),(c,7)},{(a,4),(b,7),(c,6)},{(a,5),(b,8),(c,9)},{(a,5),(b,9),(b,8)}), however, the correct number of occurrences should be 2. In fact, the matches {(a,4),(b,6),(c,7)} and {(a,4),(b,7),(c,6)} refer to the same occurrence. This is caused by the symmetry of the pattern. Thus, if a pattern is symmetrical or partial symmetrical, a redundancy check should be performed.

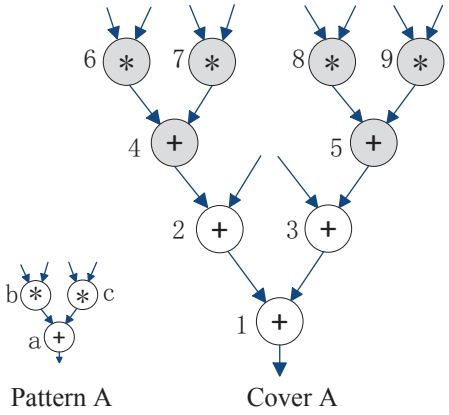


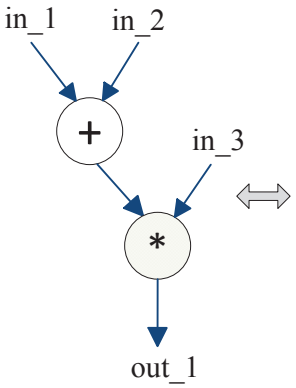
Figure 5.3: Finding the matches of a symmetrical pattern

5.2 Code Representation of Custom Operator

After mapping functionally equivalent subgraphs to an identical custom operator and selecting a subset of subgraphs, the nodes inside a selected subgraph will be replaced by a new super node. The super node represents the selected subgraph. To correctly replace the selected subgraphs and to maintain the program semantics, the super nodes should be placed in a right manner. As an example, the subgraph {1,2,3} in Fig. 5.1 will be replaced with a super node. First, the three nodes are removed along with the edges among them (edges: 1->3 and 2->3). Then, a super node is placed. The incoming edges of the node 1 and the node 2 are reconnected to the super node. The edges 3->4, 3->5, 2->8 are reconnected to the super node as outgoing edges. This replacement processing is repeated for every selected subgraphs.

In order to not loose the semantics of the original code, the super nodes contain all the information of the corresponding replaced subgraphs. In the phase of code regeneration, the information inside the super nodes is traversed and is appropriately translated to code.

Once the selected subgraphs are replaced, in the generated new source code, a specific pragma may be included in front of each custom operator to indicate the occurrence of a custom operator for high-level synthesis tools. The content of the custom operator is presented as a function. With the specific pragma, the high-level synthesis tools (e.g., CatapultC (Mentor Graphics) [Graphic]) will schedule and bind the custom operators as they do for the other basic operators. For the high-level synthesis tool CtoS (Cadence) [Cadence], all the non-inlined functions are considered as custom operators by default




```
#pragma map_to_operator "pattern_1"
void pattern_1(int in_1, int in_2, int in_3, int* out_1){
        *out_1 = (in_1+in_2)*in_3;
}
```

Figure 5.4: The code representation for a custom operator

(the pragma is not required). Fig. 5.4 shows an example of the code format for a custom operator.

5.3 An Example of Using the Complete Design Flow

Fig. 5.5 shows a simple example of using the complete design flow. A piece of C code including control flow is provided as input to the proposed design flow. The C code is first parsed to CDFG by GECOS front-end. Then, the subgraph enumeration algorithm tries to enumerate all the subgraphs from, for example, the basic block BB1 (in this example, we enumerate connected subgraphs without other constraints). We now have a set of subgraphs. Next, a subset of subgraphs is selected during the subgraph selection step (here, the biggest subgraph M_1 is selected). After the selection, a collapsed DFG is generated. Based on the CDFG and the collapsed DFG, a new piece of C code incorporating the selected subgraph is generated. Finally, the new piece of C code will be provided to the HLS tool as input.

5.4 Summary

In this chapter, a (sub)graph isomorphism algorithm based on the VF2 algorithm was presented. The VF2 algorithm is extended by checking some additional information of data-flow graph such that dissimilar graphs can be rejected quickly. Moreover, some specific problems existing in the (sub)graph isomorphism check for data-flow graphs were

exposed and corresponding solutions were given. We then briefly described the phase of code regeneration. The code representation of custom operator was also introduced.

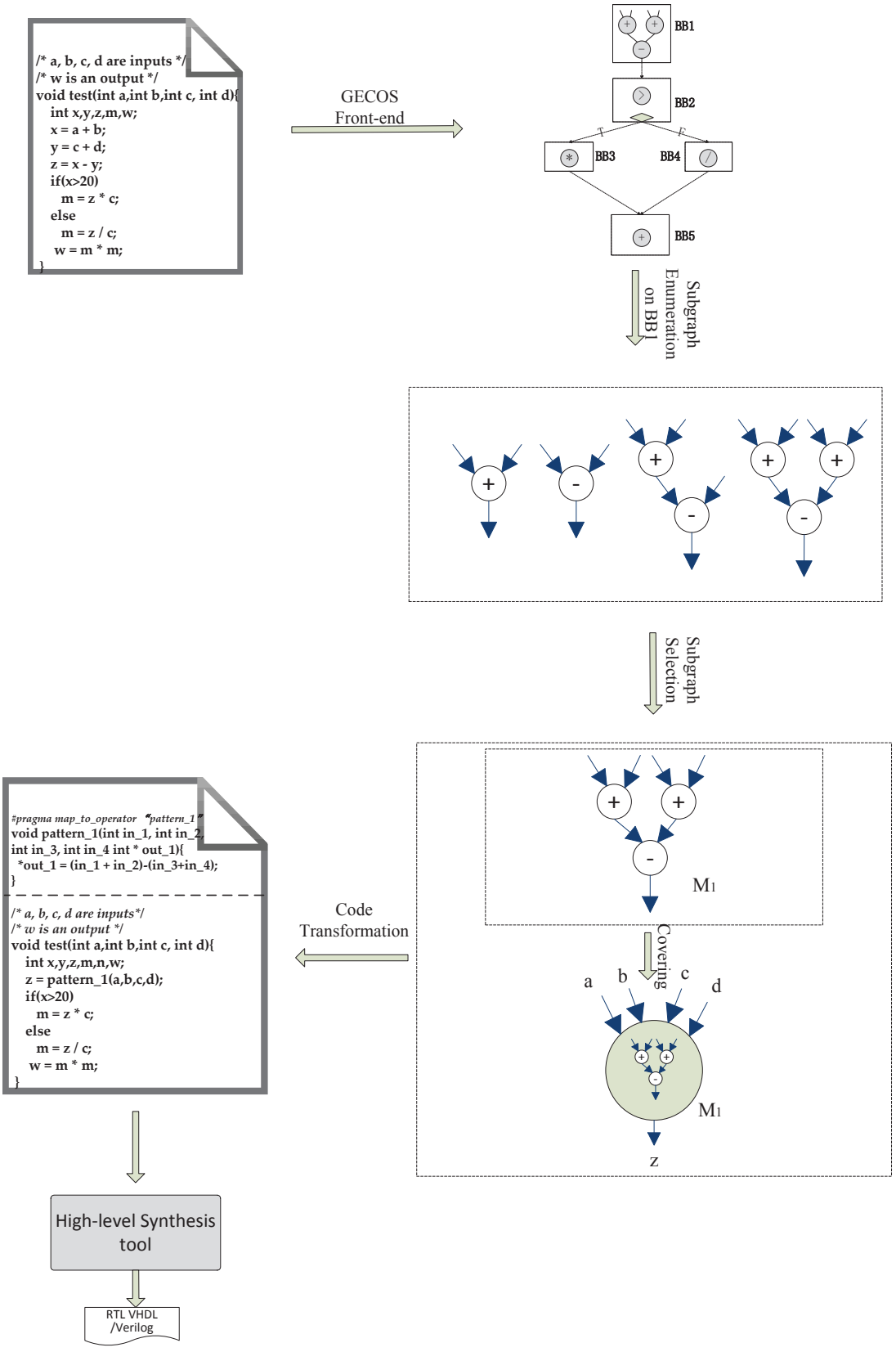


Figure 5.5: An simple example of using the design flow

Chapter 6

Experiments and Results

In this chapter, we present the runtime performance of the proposed subgraph enumeration algorithms and the quality of results achieved by the proposed subgraph selection algorithms respectively. We first evaluate the efficiency of the size constrained subgraph enumeration algorithm with a set of real-life benchmark programs. With the same benchmarks, our two proposed I/O constrained subgraph enumeration algorithms are compared to an efficient well-know algorithm[Pozzi 2006]. Based on the experimental results, we detail the runtime improvement and search space reduction achieved over the previous algorithm.

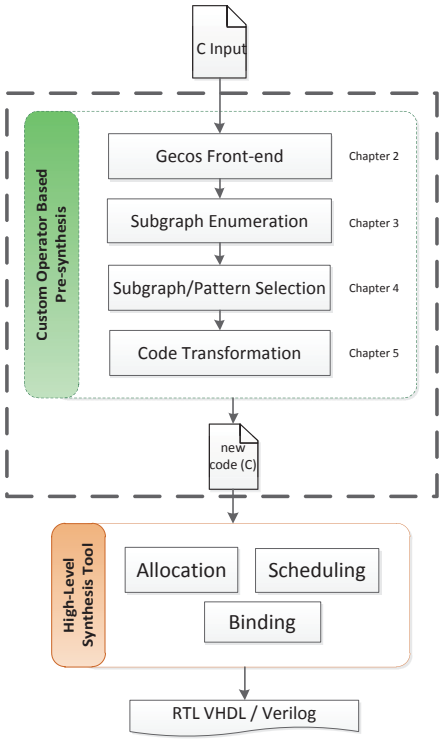


Figure 6.1: Custom operator based high-level synthesis flow

To test the quality of results achieved by the three proposed subgraph selection algo-

rithms, the regenerated source codes and the original source codes are provided as inputs of a high-level synthesis tool. The quality of results are measured in terms of the area cost and the latency. In addition, we compare the proposed genetic algorithm with the proposed heuristic algorithm for minimal number of matches selection.

To recall, Fig. 6.1 shows the detailed framework of the proposed design flow. In chapter 2, we presented the intermediate representation used in our design flow. The GECOS is used to parse the application code to a CDFG. We presented algorithms for enumerating subgraphs under various constraints in chapter 3. Then, the selection methods are described in chapter 4. Finally, the code transformation and an extended graph isomorphism algorithm are shown in chapter 5.

6.1 Experimental Setup

All the experiments were carried out on a PC with a P9400 processor running at 2.4 GHz. A set of real-life benchmark programs were selected. These benchmarks were compiled to CDFGs by a generic compilation platform GECOS developed in Cairn team [GECOS]. The high-level synthesis tool CtoS (Cadence) is used to evaluate the custom operator based pre-synthesis. We use the built-in file tutorial.lbr of CtoS as the technology library, and the clock frequency is 50 MHz.

6.2 Runtime of the Enumeration Algorithms

We have carried out extensive experiments to evaluate the performance of the subgraph enumeration algorithms. In order to evaluate the performance of our subgraph enumeration algorithms, we obtained the DFGs from the benchmarks in MediaBench [Lee 1997] and MiBench[Guthaus 2001]. Table 6.1 describes the DFGs used in our experiments. In the table, the size refers to the number of nodes. The tightness of each DFG is indicated by $|E|/|V|$, where $|E|$ is the number of edges connected valid nodes and $|V|$ is the number of valid nodes in the DFG. In the experiments of enumerating all feasible subgraphs, we choose one computation-intensive basic block's DFG from each benchmark. In the experiments of enumerating only connected subgraphs, we choose the biggest region in each basic block. A region is a connected graph that is either a disjoint subgraph of a basic

Table 6.1: Characteristics of the benchmarks for the evaluation of enumeration algorithms

Benchmark	Domain	BB Size	Region's Size	|E|/|V|
Blowfish	Security	362	{361,1}	0.98
EPIC	Security	37	{33,2,1,1}	0.64
GSM	Telecommunication	490	{479,10,1}	0.87
JPEG	Consumer	187	{185,2}	1.02
DES3	Security	94	{94}	0.93

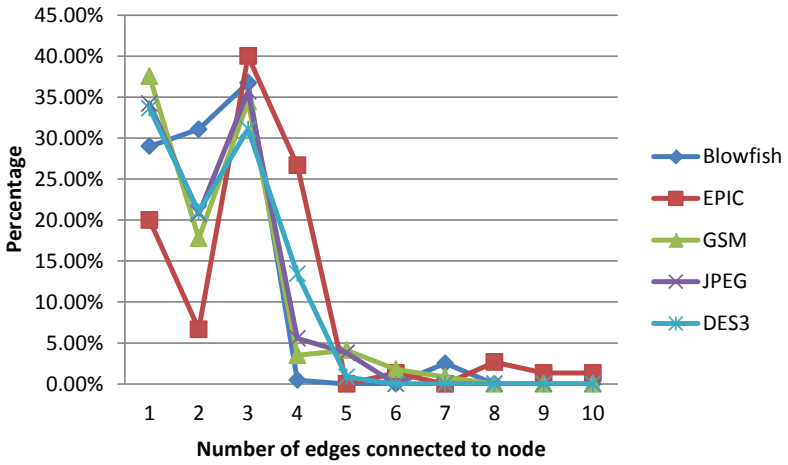


Figure 6.2: The shape profile of the benchmarks

block's DFG or is generated by dividing the DFG with invalid nodes. For example, with the benchmark Blowfish, the chosen basic block is made of 362 nodes. Two regions are included in this basic block: one with 361 nodes and one with 1 node. Fig. 6.2 shows the shape profile of each benchmark. The percentage of the nodes connected to different number of edges is calculated. As an example, with the benchmark EPIC, the nodes connected to 3 edges account for 40% over all the nodes in the DFG of EPIC.

In this section, we evaluate the proposed enumeration algorithms in terms of runtime. First, the runtime of the size constrained enumeration is presented. Then, the performance of the proposed two I/O constrained enumeration algorithms and the comparison between them are shown in subsection 6.2.2. The time unit of runtime is second.

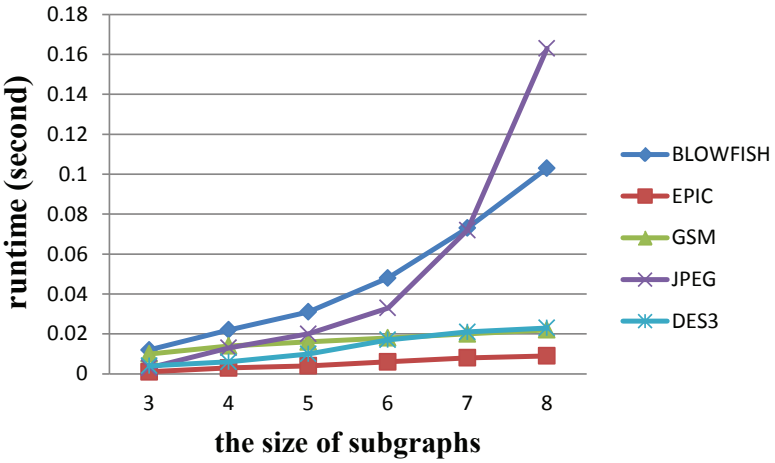


Figure 6.3: The runtime performance of the size constrained enumeration algorithm when enumerating connected subgraphs

6.2.1 Runtime of the Size Constrained Enumeration Algorithm

Our enumeration algorithm enumerates all the subgraphs in a incremental way. Accordingly, the size of the subgraphs is treated as a user defined option. Fig. 6.3 shows the runtime performance of the proposed subgraph enumeration algorithm under different sizes of connected subgraphs. It can be seen that our algorithm can completely enumerate the subgraphs within one second for all the benchmarks when the maximum size of subgraphs is set to 8. We can also observe that the runtime is increased with the relaxing of the size of the subgraphs. However, enumerating subgraphs from the larger DFG may require less time compared to smaller DFGs. For example, the enumeration runtime on the benchmark JPEG which has only 185 valid nodes is higher than the enumeration runtime on the benchmark of GSM which has 479 valid nodes. The topology of the DFGs impacts the runtime of our enumeration algorithm most. The valid nodes in the DFG of JPEG are tightly connected, while the valid nodes in the DFG of GSM are dispersed (see the column $|E|/|V|$ in Table 6.1). Thus, under the same size constraint, the number of subgraphs in the DFG of JPEG is more than the number of subgraphs in the DFG of GSM. Due to the same reason, the increase of runtime for the benchmark JPEG tends to exponential. Table 6.2 shows the number of patterns (P) and the number of subgraphs (S) enumerated under the size constraint. We can see that the number of subgraphs in the DFG of JPEG is 26758 when the maximum size of subgraphs is set to 8, while the number of subgraphs in the DFG of GSM is only 2118.

Table 6.2: Number of connected subgraphs and patterns under the size constraint

Benchmark	P	S	P	S	P	S	P	S	P	S	P	S
BLOWFISH	36	944	73	1924	139	3781	245	6730	394	10697	580	15290
EPIC	26	80	40	132	54	208	67	304	79	396	88	459
GSM	23	799	36	1094	53	1415	71	1708	88	1947	101	2118
JPEG	94	435	249	839	635	1811	1705	4254	4475	10512	15732	26758
DES3	42	189	88	380	162	568	271	934	419	1462	612	2203
Size Constraint	3		4		5		6		7		8	

The proposed size constrained subgraph enumeration algorithm can also be tuned to generate all feasible subgraphs including connected subgraphs and disjoint subgraphs. As the number of all feasible subgraphs in a DFG can be exponential, enumerating all feasible subgraphs under size constraint becomes very time-consuming. In the experiments, we found that the enumeration is unaffordable even for the smallest benchmark EPIC when the size constraint is set to 6.

6.2.2 Runtime of the I/O Constrained Enumeration Algorithms

The connectivity is one of the design constraints that should be considered when we perform enumerating subgraphs. Thus, in the experiments, we evaluate the I/O constrained enumeration algorithms for enumerating all feasible subgraphs and enumerating only connected subgraphs respectively.

6.2.2.1 All Feasible Subgraphs Enumeration

The authors of [Chen 2007] have shown with experiments that the well-known algorithm [Pozzi 2006] (denoted as a) is faster than the algorithm [Yu 2004b] in most situations when enumerating connected valid subgraphs. Since the algorithm [Yu 2007] targeting disjoint subgraphs is based on the algorithm [Yu 2004b], the algorithm a should be also faster than it when enumerating disjoint valid subgraphs. Furthermore, the algorithm a has comparable performance to the algorithm [Chen 2007]. Therefore, we compare our instruction enumeration algorithms with the algorithm a. The flexible algorithm presented in section 3.4 is denoted as b and the topology based enumeration algorithm that we

previously presented in section 3.5 is denoted as c.

Table 6.3 shows the performance of the algorithms in enumerating all feasible subgraphs under different I/O constraints. For different input and output constraints, the three algorithms produce the same subgraphs. The first column shows the number of nodes for each tested benchmarks. The column I/O indicate the maximum number of inputs and the maximum number of outputs that we set as constraints. In this table, the number of identified subgraphs is recorded in the column feasible matches. The search space of the three algorithms is the total number of subgraphs they considered. The columns runtime x represent the runtime of the proposed algorithms.

The results show that the number of enumerated subgraphs increases with the size of the DFGs of the benchmarks. This is not surprising: in general, bigger DFGs enable more combinations of nodes. We also noticed that the number of enumerated subgraphs increases rapidly with the relaxation of I/O constraints. The Lemma 3.1.1 presented in section 3.1 is well supported by this observation.

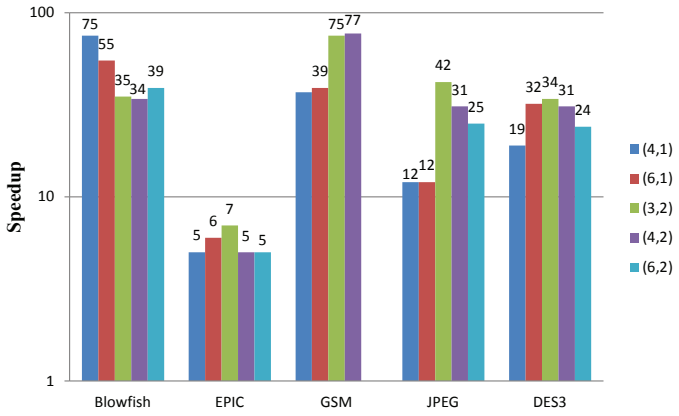


Figure 6.4: Runtime speedup achieved by the algorithm b over the algorithm a for enumerating all feasible subgraphs

According to the experimental results, our algorithms have a significant better performance (see Fig. 6.4 and Fig. 6.5). Based on the experiments, we can see that the larger DFG it is, the more significant speedup our algorithms achieve over the algorithm a. It can be observed that for the very small benchmark EPIC, our algorithm c achieves the speedup ranging from 5 to 10 times over the algorithm a. For the medium DFG JPEG, the speedup achieved by our algorithm is more significant, ranging from 32 to 80. For the large DFG GSM, our algorithm c is orders of magnitude faster than the algorithm a

Table 6.3: Comparison of the subgraph enumeration algorithms - all feasible subgraphs (*a*: the algorithm proposed in [Pozzi 2006], *b*: our algorithm presented in section 3.3, *c*: our algorithm based on topology presented in section 3.4)

Benchmark (size)	I/O	feasible matches	search space *a*	search space *b*	search space *c*	runtime *a*	runtime *b*	runtime *c*
	4/1	1647	1248639	3173	2051	4.26	0.057	0.024
Blowfish	6/1	5477	2163001	12209	7580	7.758	0.14	0.044
(362)	3/2	29891	5178645	62302	46780	16.61	0.465	0.203
	4/2	62397	9672021	108300	81449	31.02	0.903	0.349
	6/2	193154	34075910	357771	232719	106.8	2.753	1.301
	4/1	76	1321	99	140	0.005	0.001	0.001
EPIC	6/1	78	1327	99	142	0.006	0.001	0.001
Collapse	3/2	137	18205	1557	1897	0.059	0.008	0.007
(37)	4/2	885	24994	2869	2990	0.085	0.016	0.01
	6/2	2343	29057	4257	4369	0.125	0.023	0.013
	4/1	1972	576377	2790	2847	2.289	0.062	0.018
GSM	6/1	4926	974419	6354	6154	3.704	0.094	0.034
(490)	3/2	106633	68906481	375733	320013	269.9	3.599	1.519
	4/2	341641	141329786	793307	669502	559.8	7.242	3.916
	6/2	1454539	/	2719019	2306023	/	32.4	13.69
	4/1	633	76909	1083	973	0.273	0.023	0.007
JPEG	6/1	1305	123097	1963	1789	0.42	0.034	0.013
(187)	3/2	9182	3901494	40793	38022	12.71	0.3	0.158
	4/2	31871	6618871	75830	74647	21.9	0.703	0.312
	6/2	123187	16459941	217206	211381	54.34	2.153	1.078
	4/1	628	113853	1232	993	0.391	0.021	0.008
DES3	6/1	2790	513502	5528	3686	1.746	0.053	0.022
(94)	3/2	3393	608054	10218	8979	2.019	0.06	0.038
	4/2	9837	1495275	22443	19665	5.037	0.161	0.082
	6/2	47139	6275022	101185	78423	21.76	0.897	0.345

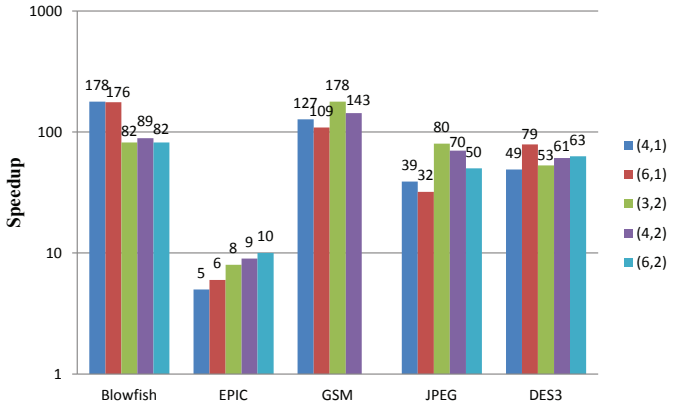


Figure 6.5: Runtime speedup achieved by the algorithm c over the algorithm a for enumerating all feasible subgraphs

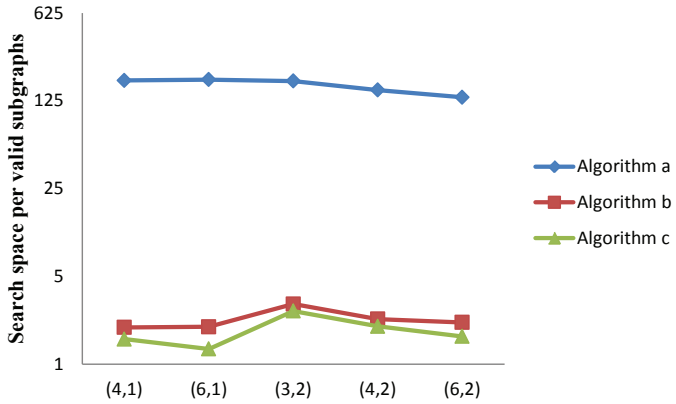


Figure 6.6: Search space per feasible subgraph for the benchmark DES3 under different I/O constraints

in all situations. We can also see that our algorithm c has better performance than the algorithm b.

The speedup achieved by our algorithms is directly due to the reduction in the search space as well as the convexity of the subgraphs guaranteed by the construction of the subgraphs. Fig. 6.6 shows the search space per feasible subgraphs of the algorithm a, the algorithm b and the algorithm c for benchmark DES3 under different I/O constraints. We note that the algorithm b has successfully reduced the search space ranging from 60 to 93 times over the algorithm a. The algorithm c has further reduced the search space by 68 to 139 times over the algorithm a. At the I/O constraint such as (6,1), the reduction factor achieved by c over a is 139 (513502/3686), i.e. the best one, this is why in fig. 6.5 we can also observe the higher runtime speedup for DES3.

The reduction of the search space achieved by the algorithm c over the algorithm a can be mainly attributed to three parts: 1) non-convex subgraphs are filtered away in an earlier stage, 2) due to a specific sequence of growing subgraphs, a large number of output constraint violated subgraphs are not considered and 3) only few of input violated subgraphs are considered by using the function that aims to finding resolving nodes when inputs constraint is violated. Compared with the algorithm b, the algorithm c can reduce more the search space by constructing subgraphs in a topology order such that only a small number of output constraint violated subgraphs are considered.

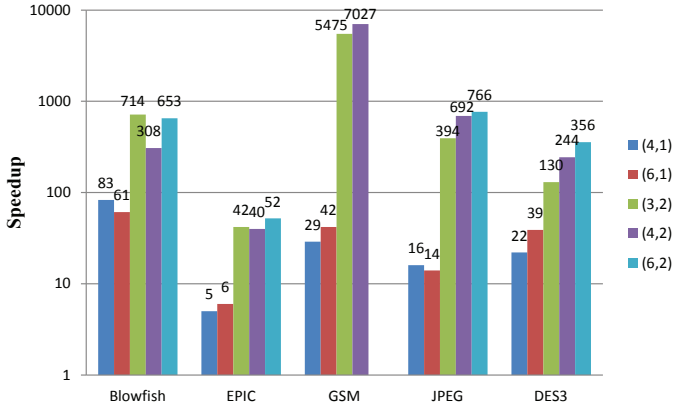


Figure 6.7: Runtime speedup achieved by the algorithm b over the algorithm a for enumerating connected subgraphs

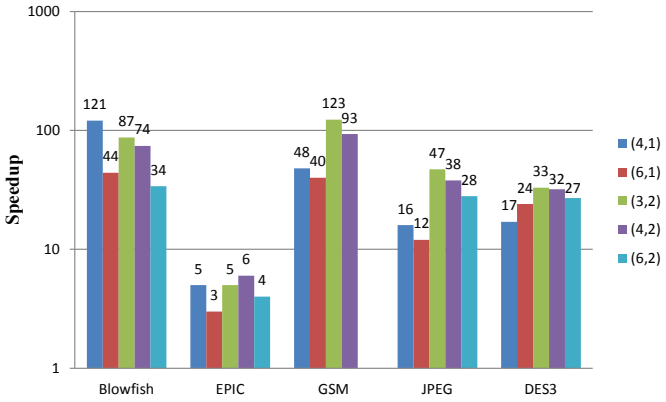


Figure 6.8: Runtime speedup achieved by the algorithm c over the algorithm a for enumerating connected subgraphs

6.2.2.2 Connected Subgraphs Enumeration

As the algorithm [Pozzi 2006] (the algorithm a) is faster than the only algorithm [Yu 2004b] dedicated to enumerate connected subgraphs, we only compare our algorithms with the algorithm a) in the experiments. In order to filter away disjoint subgraphs, we

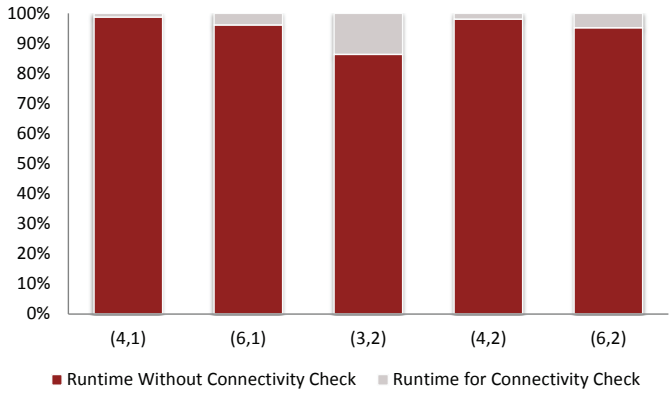


Figure 6.9: Runtime for connectivity check and runtime for subgraph enumeration for the benchmark DES3 using the algorithm *a*

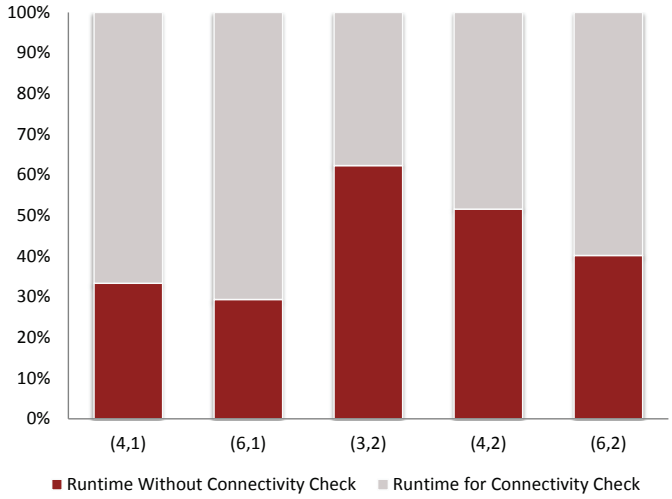


Figure 6.10: Runtime for connectivity check and runtime for subgraph enumeration for the benchmark DES3 using the algorithm *c*

add a connectivity check step to algorithms *a* and *c* after a feasible subgraph is obtained [1].

Table 6.4 compares our algorithms with the algorithm *a*. The speedup of our algorithm *b* over the algorithm *a* in enumerating connected feasible subgraphs is far more significant than that in Fig.6.4 (compare Fig.6.4 and Fig. 6.7). This is reasonable: the algorithm *b* only considers connected subgraphs, on the contrary, the algorithm *a* considers connected subgraphs and disjoint subgraphs. We also see that the algorithm *b* outperforms the algorithm *c* in most situations (Fig. 6.7 and Fig. 6.8). Similar to the algorithm *a*, the algorithm *c* considers connected subgraphs and disjoint subgraphs and performs additional

[1] In the algorithm *b*, the NodeFilter function can be used to generate all feasible subgraphs or only connected subgraphs (see algorithm 3 in section 3.3.2)

Table 6.4: Comparison of the subgraph enumeration algorithms - connected subgraphs (*a*: the algorithm proposed in [Pozzi 2006], *b*: our algorithm presented in section 3.3, *c*: our algorithm based on topology presented in section 3.4)

Benchmark (size)	I/O	feasible matches	search space *a*	search space *b*	search space *c*	runtime *a*	runtime *b*	runtime *c*
	4/1	1646	1209950	3084	2055	4.589	0.056	0.038
Blowfish	6/1	5476	2157792	11933	7579	7.58	0.124	0.173
(361)	3/2	1016	4771198	2000	46321	26.42	0.037	0.303
	4/2	2148	9268081	4568	80279	51.769	0.057	0.691
	6/2	8488	31937667	20196	231910	104.5	0.16	3.106
	4/1	71	1051	96	135	0.005	0.001	0.001
EPIC	6/1	73	1057	142	137	0.006	0.001	0.002
Collapse	3/2	77	14185	137	1601	0.042	0.001	0.009
(33)	4/2	113	18427	242	2662	0.079	0.002	0.014
	6/2	137	21760	311	4023	0.104	0.002	0.025
	4/1	1911	550552	2719	2767	1.877	0.065	0.039
GSM	6/1	4831	921762	5991	6055	3.821	0.091	0.095
(479)	3/2	1333	57143751	2062	303162	224.5	0.041	1.822
	4/2	2071	130897602	3183	632323	435.7	0.062	4.348
	6/2	5483	/	7176	2173815	/	0.095	11.373
	4/1	630	75395	992	971	0.279	0.018	0.018
JPEG	6/1	1302	119458	1878	1787	0.417	0.029	0.035
(185)	3/2	526	3612205	966	37230	9.849	0.025	0.21
	4/2	897	6495394	1960	72991	21.45	0.031	0.567
	6/2	3024	16061030	5887	205946	51.33	0.067	1.844
	4/1	628	113853	1232	993	0.396	0.018	0.024
DES3	6/1	2790	513502	5528	3686	1.81	0.047	0.075
(94)	3/2	398	608054	735	8979	2.347	0.018	0.071
	4/2	842	1495275	1749	19665	5.121	0.021	0.159
	6/2	3558	6275022	7007	78423	22.81	0.064	0.859

connectivity check to filter away disjoint subgraphs. Fig. 6.9 and Fig. 6.10 compare the time used for connectivity check and the time used for subgraph enumeration using the algorithm *a* and the algorithm *c* respectively. It can be seen that the connectivity check time accounts for less 10% of the total runtime when using the algorithm *a*. The connectivity check time is half of the total runtime when using the algorithm *c*.

6.3 Evaluation of the Selection Approaches

To evaluate the quality of the subgraph selection approaches, a set of real-life benchmark programs which are rich in arithmetic/logical operators is used in our limited study. As our design flow focuses on identifying subgraphs inside each basic block, we choose computation-intensive functions as benchmarks in our experiments. Those benchmarks are featured with various sizes ranging from tens to hundreds of valid nodes and different shapes of the DFGs. Table 6.5 describes the characteristics of the benchmarks used in our experiments. The function *dotProduct* computes the product of two 50-coefficient vectors. The function *imdct* performs the inverse modified discrete cosine transform for MP3 audio encoding. The function *idct* computes a 8-point one dimensional inverse discrete cosine transform. The function *invert_matrix* computes the inverse of 4x4 matrices. The function *fft* is actually a 2-nested loop function used to compute both real and imaginary parts of a n point fast Fourier transform. The function *arf* implements a 8-point autoregressive filter. The function *iir* computes a 8-tap infinite impulse response filter. In the latter subsections, we evaluate the proposed design flow in terms of area reduction, performance (latency of the critical path) improvement and code size reduction. The runtime performance of subgraph enumeration step for the benchmarks in table 6.5 under size constraint is shown in Annex A.

6.3.1 Area and Performance Evaluation

6.3.1.1 Results of Connected Subgraphs under Size Constraint

For comparison, the original code and the new code generated by the custom operator based pre-synthesis flow are provided to the CtoS high-level synthesis tool as inputs respectively. Table 6.6 shows the quality of the results obtained by our proposed design

Table 6.5: Characteristics of the benchmarks for the evaluation of selection results

Benchmark	BB Size	Valid Nodes
dotProduct (50×50)	303	100
imdct	1290	297
idct	130	55
invert_matrix	533	148
fft	340	72
arf	84	28
iir	525	176

flow (only connected subgraphs are considered) compared to the traditional flow without custom operator based pre-synthesis. As the runtime of enumerating all feasible subgraphs under size constraint is unaffordable (see section 6.2.1), the results of all feasible subgraphs under size constraint are not presented in this section.

In the table 6.6, the critical paths based match selection algorithm, the frequency of occurrence based pattern selection algorithm and the minimal matches selection algorithm presented in section 4.2 are denoted as CS, PS, MS respectively. The second column and the third column record the number of patterns and the number of matches (subgraphs) enumerated. The symbols X_P, X_M, X_Area and X_Per represent the number of selected patterns, the number of selected matches, the area reduction rate and the performance improvement rate achieved by each selection algorithm respectively (the maximum number of nodes of the subgraphs is set to 6 in the experiments).

To summarize, the PS algorithm achieves the best area reduction among the three proposed algorithms (on average 19.1%, and up to 37.1% area reduction over the traditional high-level synthesis flow). The CS algorithm always leads to positive and the best performance improvement rate, on the contrary, the other two algorithms may have negative performance improvement rate. This is mainly owing to the careful evaluation of the length of the critical paths carried out by the algorithm CS, whereas sometimes with PS and MS algorithms a selection results in a critical path increase (see the example in Fig. 4.5). With the CS algorithm, on average 22.3% and up to 59.4% performance improvement can be obtained.

From the results, we can see that both the area reduction rate and the performance

Table 6.6: Area Reduction and Performance Improvement with Connected Subgraphs (maximum size of subgraphs is 6)

Benchmark	Patterns	Matches	CS_P	CS_M	CS_Area	CS_Per	PS_P	PS_M	PS_Area	PS_Per	MS_P	MS_M	MS_Area	MS_Per
dotProduct	33	1568	3	57	5.5%	59.4%	2	19	6.7%	5.1%	3	17	6%	-1.3%
imdct	195	2979	9	163	41.6%	18.1%	7	99	33.9%	-33.8%	10	99	38%	-17%
idct	3101	3716	13	16	30.8%	12.2%	13	21	37.1%	-35%	11	9	25%	8%
invert_matrix	539	30473	15	46	6.3%	18.9%	10	35	7.1%	-3.9%	18	31	7.1%	-5.2%
fft	65	248	7	19	10%	10.6%	7	19	10%	10.6%	7	19	10%	10.6%
arf	80	500	4	8	5.0%	27.8%	2	12	16.6%	16.7%	4	6	5.3%	9.0%
iir	118	998	13	75	3.0%	8.9%	6	72	22.1%	-11.0%	11	68	1.0%	-13.4%
Average					14.6%	22.3%			19.1%	-7.0%			13.3%	-1%

improvement rate vary with the benchmarks. For example, with the CS algorithm, we can achieve 41.6% area reduction for the benchmark imdct, while the area reduction rate is only 5.5% for the benchmark dotProduct. To explain the difference, we studied the shape of the DFGs of the two benchmarks. The DFG of dotProduct is a tree-shaped graph. The DFG of imdct is a net-shaped graph. Generally, net-shaped graphs have higher density of internal data flows. As the internal data flows may roughly indicate the number of multiplexors, extracting custom operators from the net-shaped DFG which has more internal data flows may save a large number of multiplexors.

The results also show difference on performance improvement for the benchmarks. As an example, with the CS algorithm, the performance improvement rate reaches 59.4% for the benchmark dotProduct. Yet, the performance improvement rate reaches only 12.2% for the benchmark idct. This observation is supported by the following analysis. Most of the selected subgraphs from the DFG of the benchmark dotProduct are composed of associative operations. As shown in Fig. 1.2, the length of the critical path can be reduced due to the associativity attribute of operations. Therefore, most of the selected subgraphs are optimized with the critical path reduction technique (In the design flow, this optimization is performed by CtoS that invokes RTL synthesis tool). Nevertheless, the selected subgraphs of the benchmark idct have few opportunities to be optimized in such a way.

6.3.1.2 Results of Connected Subgraphs under I/O Constraints

As previously discussed, the number of inputs and the number of outputs can be user defined constraints. In this subsection, we evaluate the area and performance of the

Table 6.7: Area Reduction and Performance Improvement with Connected Subgraphs under I/O Constraints (6/2)

Benchmark	Patterns	Matches	CS_P	CS_M	CS_Area	CS_Per	PS_P	PS_M	PS_Area	PS_Per	MS_P	MS_M	MS_Area	MS_Per
dotProduct	20	963	2	60	4.3%	57.7%	1	25	6.1%	-29.2%	5	33	5.7%	3.2%
imdct	131	2191	9	154	42.9%	7.3%	8	126	32.9%	-14.1%	12	108	39.2%	-24.8%
idct	116	367	12	20	26.1%	4.4%	13	20	30.4%	2.1%	11	18	30.7%	-3.8%
invert_matrix	293	1647	17	43	8.0%	6.1%	9	59	7.2%	-16.9%	16	39	7.1%	-16.2%
fft	46	214	7	24	8.2%	6.8%	7	24	9.1%	7.4%	8	23	9.0%	6.8%
arf	29	206	4	10	4.4%	19.2%	2	12	5.2%	-6.5%	5	8	4.7%	-17.3%
iir	38	480	7	74	2.1%	9.0%	6	72	18.4%	-12.3%	6	72	1.0%	-12.0%
Average					13.7%	15.8%			15.6%	-9.9%			13.9%	-9.2%

obtained results when the subgraphs are connected and are respected to I/O constraints. Table 6.7 shows the area reduction and performance improvement achieved when the I/O is set to 6/2.

During the experiments, we noticed that the quality of the results obtained when the subgraphs are connected and are respected to I/O constraints is similar to the quality of the results obtained when the subgraphs are connected and are respected to size constraint. Precisely, the quality of the results of the connected subgraphs with I/O constraints is slightly lower than that of the connected subgraphs with size constraints. To explorer the difference, we looked at the enumerated subgraphs and patterns. We further noticed that a larger set of candidates (matches and patterns) is obtained when the maximum size of enumerated subgraphs is set to 6 than when the maximum I/O of enumerated subgraphs is set to 6/2. The larger set of candidates may offer more opportunities to achieve a better solution.

6.3.1.3 Results of All Feasible Subgraphs under I/O Constraints

Compared to connected subgraphs, disjoint subgraphs enable more parallelism. In the experiments, we first evaluate the results of all feasible subgraphs with I/O constraints. Then, we compare the results of all feasible subgraphs with the results of connected subgraphs. Table 6.8 shows the result of all feasible subgraphs under I/O constraints (I/O is set to 6/2).

Comparing the results in the Table 6.7 with the results in the Table 6.8, an interesting phenomenon can be observed. The performance of the results in Table 6.8 is better than the results in the Table 6.7, while the area is quite similar. We have carefully examined

Table 6.8: Area Reduction and Performance Improvement with All Feasible Subgraphs under I/O Constraints (6/2)

Benchmark	Patterns	Matches	CS_P	CS_M	CS_Area	CS_Per	PS_P	PS_M	PS_Area	PS_Per	MS_P	MS_M	MS_Area	MS_Per
dotProduct	27	9294	2	35	4.1%	57.7%	1	25	6.1%	-29.2%	5	30	4.8%	17.2%
imdct	96	93273	7	225	37.8%	10.4%	5	189	30.3%	-17.1%	6	162	30%	-6.6%
idct	222	2217	13	15	25.1%	4.8%	9	17	25.8%	12%	11	13	34.9%	-2.5%
invert_matrix	682	59217	17	43	8.0%	14.2%	15	38	6.1%	20.1%	14	29	2.1%	22.2%
fft	84	6310	15	29	15.4%	14.7%	10	40	14.1%	1.2%	8	19	16.5%	-7.3%
arf	50	1049	4	7	4.5%	19.2%	2	12	3.3%	-2.6%	2	6	5.5%	-9.0%
iir	93	31434	15	58	9.0%	15.8%	7	103	25.7%	-4.9%	10	52	8.1%	-1.0%
Average					14.8%	19.5%			15.9%	-2.9%			14.6%	4.5%

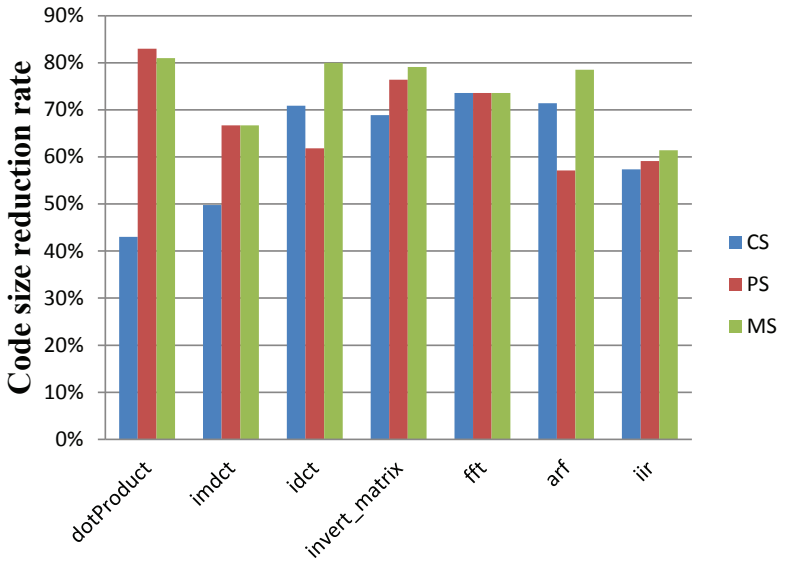


Figure 6.11: Code size reduction rate achieved by the three selection approaches (maximum size of subgraphs is 6)

the shape of the selected subgraphs that are disjoint or connected. The selected disjoint subgraphs are usually the subgraphs which have more nodes in parallel than the connected subgraphs. Thus, the critical path may be further reduced through the parallelism of the operation nodes in the selected subgraphs.

6.3.2 Code Size Reduction

To show the code size improvement achieved by the proposed algorithms, we define the improvement of the code size of the generated functionally equivalent code over the code

size of the original source code as follows.

$$imp = ((|G| - |G'|)/|G|) * 100\% \tag{6.1}$$

where $|G|$ represents the number of operations in the original source code, $|G'|$ represents the number of operations in the generated code that collapses the selected subgraphs.

Fig. 6.11 shows the code size improvement obtained using the proposed three selection algorithms when the maximum size of connected subgraphs is set to 6. Based on the results, the minimal number of matches selection produces the most compacted code in most situations. This is reasonable: it is because the minimal number of matches selection algorithm prefers to select bigger subgraphs, whereas the other two algorithms may select smaller subgraphs with the consideration of frequency of occurrence or criticality. The code size reduction achieved by the minimal number of matches selection algorithm is very significant. As an example, for the benchmark dotProduct, the reduction rate is up to 81%. On average, a reduction rate of 74% can be archived.

With the compacted code, HLS tool should be able to produce a design solution in a shorter time. Unfortunately, with CtoS, we are not able to measure the exact time reduction achieved by the compacted code (Ctos also performs the synthesis of custom operators during the high-level synthesis of the compacted code, thus, we cannot measure the time for the high-level synthesis of compacted code). However, the code size reduction may at least proportionally reflect the time reduction.

6.3.3 Comparison of the MS Algorithm and the Genetic Algorithm

As MS algorithm is targeted to produce the most compacted code, we evaluate the MS algorithm by comparing with the proposed Genetic algorithm in terms of code size reduction. The exact algorithm can not produce solutions in most situations (it takes more than one hour to give a solution), we only use the genetic algorithm to evaluate the MS algorithm. In our experiments, a best set of parameters was chosen for the genetic algorithm according to extensive tests. The population size was set to 500 for all the tests. Each time the genetic algorithm is terminated when 20 generations has produced. The detail of the set of parameters is given in the Table 6.9.

Table 6.9: A set of parameters chosen for the genetic algorithm

Parameter	Description	Value
N	the size of population	500
G	the number of generations	20
T	the size of the tournament	20
γ	the composition of initial solutions	80%
ε	crossover rate	95%
δ	mutation rate	1/number of genes
β	replacement rate	10%

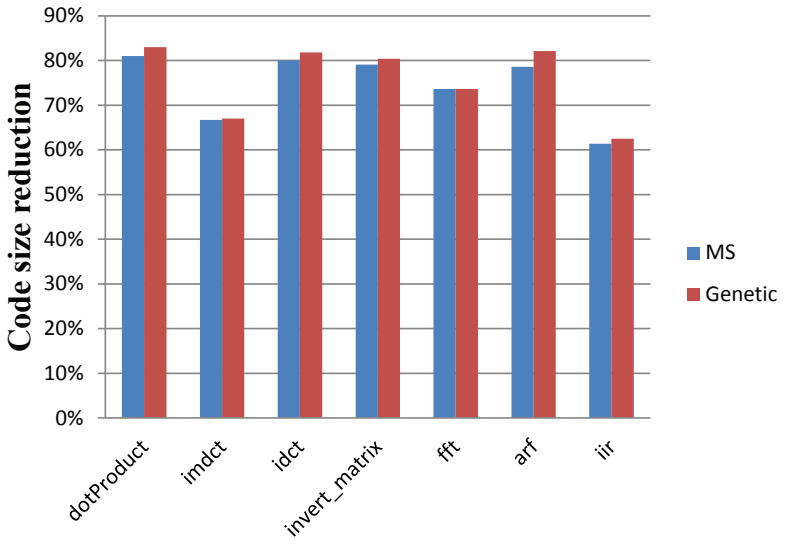


Figure 6.12: Code size reduction rate achieved by the MS algorithm and the Genetic algorithm (maximum size of subgraphs is 6)

Fig. 6.12 depicts the code size reduction rate achieved by the MS algorithm and the genetic algorithm when the maximum size of subgraphs is set to 6. It can be seen that the results generated by the MS algorithm are close to the results generated by the genetic algorithm in all the tests. Fig. 6.13 shows the runtime required for the MS algorithm and the genetic algorithm. We notice that the MS algorithm is more efficient than the genetic algorithm. The experiments tell us that the proposed MS algorithm can produce fast a good result. However, the genetic algorithm can still be interesting due to its flexibility. Actually the cost function of the genetic algorithm can be easily tuned to target other objectives (area or power consumption for example).

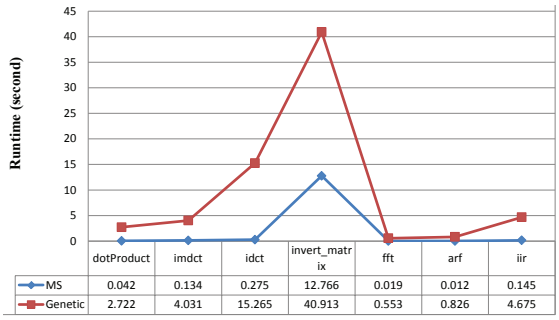


Figure 6.13: Runtime of the MS algorithm and the Genetic algorithm (maximum size of subgraphs is 6)

6.4 Discussion and Summary

In this chapter, all the proposed enumeration algorithms and selection algorithms were evaluated and analyzed. We first evaluated the size constrained subgraph enumeration algorithm. For all the benchmarks, the algorithm can completely enumerate all the subgraphs within one second. The experimental results confirmed the efficiency of the algorithm. We have examined the influences of different factors of the DFGs. The examination tells us that the size of DFGs is not the only factor that affects the enumeration algorithm. The topology of the DFGs sometimes plays a more important role in influencing the enumeration. In detail, the algorithm requires much more time to enumerate subgraphs from a tighter connected DFG. Furthermore, a tighter connected DFG may have more connected subgraphs compared to a loosely connected DFG.

We then carried out extensive experiments to test the efficiency of the two proposed I/O constrained subgraph enumeration algorithms (the one based on the size constrained subgraph enumeration algorithm and the topology based I/O constrained subgraph enumeration algorithm). We compared our algorithms to an efficient and well-known algorithm. The experiments were designed to two catalogues: enumerating all feasible subgraphs under I/O constraints and enumerating only connected feasible subgraphs under I/O constraints. In the experiments of enumerating all feasible subgraphs, the topology based I/O constrained subgraph enumeration algorithm outperforms all the other two algorithms in all situations. While in the experiments of enumerating only connected subgraphs, the one based on the size constrained subgraph enumeration algorithm has the best performance in most cases. We have done some analysis of the difference in performance. The analysis reveals that the runtime difference is mainly due to the size of the search space explored

and connectivity check that is included or not in the strategy.

The subgraph/pattern selection algorithms directly decide the quality of the final results. First, we evaluated the results obtained by the three proposed selection algorithms. From the results (area, performance and code size), the PS algorithm can generally achieve better area reduction compared with the other two algorithms, the MS algorithm lead to the most compacted code in most situations and the CS algorithm can always result in positive performance improvement. The performances are very different. The performance improvement achieved by the CS algorithm is always positive. However, the other two selection algorithms may result in performance overhead. An interesting point is found in our study, for different benchmarks the performances improvement achieved are quite different. We have carefully looked at the selected subgraphs from each benchmark. The study shows that some benchmarks enable a great chance to optimize the critical path by the critical path reduction of the selected subgraphs based on the associativity attribute of operations. Conversely, the other benchmarks offer few opportunities to optimize the critical path of selected subgraphs. We also compared the results obtained when the constraints set to subgraphs are different. We learned that the results of the subgraph under the I/O constraints and the results of the subgraph under size constraint are similar on both area and performance. Nevertheless, the results of the all feasible subgraphs and the results of the connected subgraphs are quite different on performance. This gives us an important information that selecting all feasible subgraphs can increase performance by enabling more parallelism. In section 6.3.3, we observed that the MS algorithm is quite runtime efficient (the runtime is less than one second mostly, see Fig. 6.13). As the PS algorithm and the CS algorithm are also heuristic methods and the only difference compared to the MS algorithm is the guide function used, the two algorithms have similar runtime performance as the MS algorithm. Thus, the proposed heuristic algorithms provide a solution in a short time.

In addition, we tested the heuristic algorithm, the genetic algorithm and the exact algorithm for selecting minimal number of matches. Based on the experiments, the following conclusion can be drawn. The exact algorithm guarantees the optimum of the solution, but most of the time it fails to produce a solution in a reasonable time. It is the reason why in this chapter we did not report experimental results about the exact algorithm for minimal number of matches selection that we presented in section 4.3. The greedy algo-

rithm provides result in a shorter time, however, the result may be sub-optimal. Overall, the genetic algorithm makes trade-off between greedy algorithm and the exact algorithm and can be easily tuned to target other subgraph selection objectives like area or power consumption.

Chapter 7

Conclusions

In this chapter, a summary of this thesis is presented in section 7.1. We then discuss the future work in section 7.2.

7.1 Conclusion

Custom operators are of great interest in various fields of circuit design. Custom operators that combine several primitive operations into one single operator may lead to high code compaction, performance improvement and area reduction. However, automatical using custom operators in high-level synthesis is still an emerging research.

This thesis presented a custom operator based high-level synthesis design flow. Our design flow involves compiler front end transformation, subgraph enumeration, subgraph selection and code transformation. Given a high-level specification such as C or C++, an open source compiler infrastructure named GECOS is used to transform the high-level specification to an intermediate representation. The intermediate representation used in the design flow is CDFG which captures both the data-dependencies and control dependencies of an application code. The generated CDFG is then passed to the subgraph enumeration step as input. The subgraph enumeration step exhaustively enumerate all the possible subgraphs from the DFGs in the CDFG. After the subgraph enumeration, the most profitable subset of the set of enumerated subgraphs is selected. Next, a new functionally equivalent specification that incorporates the selected subgraphs is generated. Finally, the new specification is provided as input for the high-level synthesis tool.

The main challenges faced in developing the design flow are how to efficiently enumerate subgraphs from the given CDFG and select the most profitable subset of the enumerated subgraphs. To address the subgraph enumeration problem, we produced three algorithms

for enumerating subgraphs under different constraints. The size of the custom operator (subgraph) can be a design constraint. The subgraphs are incrementally enumerated. Yet, the duplication enumeration is occurred very often. Pruning the duplicated subgraphs is very time-consuming. Therefore, we developed a very efficient size constrained subgraph enumeration algorithm, which can avoid multiple identification of any subgraph by applying a clever node deletion. The experiments demonstrate that the algorithm is capable to deal with large data-flow graphs with hundreds of nodes within one second.

The number of inputs and the number of outputs can also be a user-specified design constraint. In this case, the I/O constraints should be considered. To inherit the advantages achieved by the size constrained algorithm, we extend it by utilizing the I/O constraints as pruning criterias. This algorithm can be tuned to generate only connected subgraphs or all possible subgraphs. We shows with experiments that it outperforms the state-of-the-art algorithms when enumerating only connected subgraphs, as it can be adapted to enumerate connected subgraphs without considering disjoint subgraphs and multiple identifications of any subgraphs.

Based on the study of existing work, we find that the well-known algorithm [Atasu 2003, Pozzi 2006] for enumerating all MIMO subgraphs can be still improved. The well-known algorithm iteratively enumerates subgraphs in a binary search way. However, the specific search sequence prevent the algorithm from avoiding visiting a large number of non-convex subgraphs or output constraint violated subgraphs. Therefore, we produce an efficient algorithm that breaks the binary search and still takes advantage of topological properties of data-flow graph. We also analyzed the time complexity of the proposed algorithm. To our knowledge, it is one of the two algorithms (the other one is [Atasu 2003, Pozzi 2006]) for exhaustive enumeration MIMO subgraphs with polynomial time complexity. Furthermore, the algorithm is very easy to implement.

Given the set of enumerated subgraphs, a subset of it is selected according to different objectives. As the set of enumerated subgraphs are usually very big, sometimes consisting of millions of subgraphs, exact approach are not affordable in terms of time. Hence, in this thesis, three heuristic methods targeting different objectives are presented. The first heuristic aims at compacting the original source code most. In other words, it tries to cover the application graph with minimal number of subgraphs. During the process of selection the heuristic method always select the subgraphs with big size and less overlapping. As

overlapping is disallowed in our design, selecting a subgraph with more overlapping results in the deletion of the large number of overlapped subgraphs such that less opportunities left to achieve a better result.

Reuse of resources is an important point when performing circuit design. A heuristic selects the most frequently occurred patterns is depicted in this thesis. The heuristic uses an objective function to rank the candidate patterns. The objective function is well designed to balance the size of the selected patterns and the frequency of occurrences of the selected patterns.

However, inappropriate selection can easily result in performance overhead. To handle the performance overhead, we develop a heuristic method that carefully considers the length of critical paths. The heuristic prefers to select the subgraphs along the critical paths. In addition, the subgraphs that may increase the length of critical path are never considered.

Although the subgraph selection is a computational difficult problem, for the purpose of completeness, we also presented a typical branch-and-bound algorithm for producing an optimal solution for the minimal number of matches selection. Note that the exact algorithm can also be applied to subgraph selection problems with other objectives. As a compromise between the proposed heuristics and exact methods, a genetic algorithm is developed. The genetic algorithm provides us a trade-off between runtime and quality of solution. This algorithm can also be tuned to target other subgraph selection objectives.

7.2 Future Work

There are still many works that can be done for the custom operator based high-level synthesis flow. To summarize these, we list the future work as following:

• In our experiments, we evaluated the quality of the generated code using the high-level synthesis tool CtoS. As different high-level synthesis tools may produce different results with the same inputs, therefore, it is interesting to use other high-level synthesis tools to evaluate the quality of the code generated by our design flow. We expect for example to use CatapultC from mentor graphics, but due to licence issues, we are not able to use it yet.

• Currently, our subgraph enumeration is only performed on data-flow graphs corresponding to the basic blocks. Considering the subgraphs across basic blocks may provide a more global view and introduce more opportunities for optimizations. As a consequence, a better result may be achieved in terms of reuses.

• In the thesis, we carry out the selection step considering the area cost and the performance. Power consumption has attracted much attention in recent years. However, the power consumption is not taken into consideration when selecting the subgraphs in this thesis. What is the impact on power consumption of using custom operators and how to estimate and model the energy consumption during the subgraph selection remains an open question. However, it should be said that the heuristic based selection algorithm and the genetic algorithm can be easily tuned to take power consumption into account owing to the objective function.

Appendix A

Runtime of the Subgraph Enumeration Step

A.1 The runtime performance of the subgraph enumeration step under size constraint

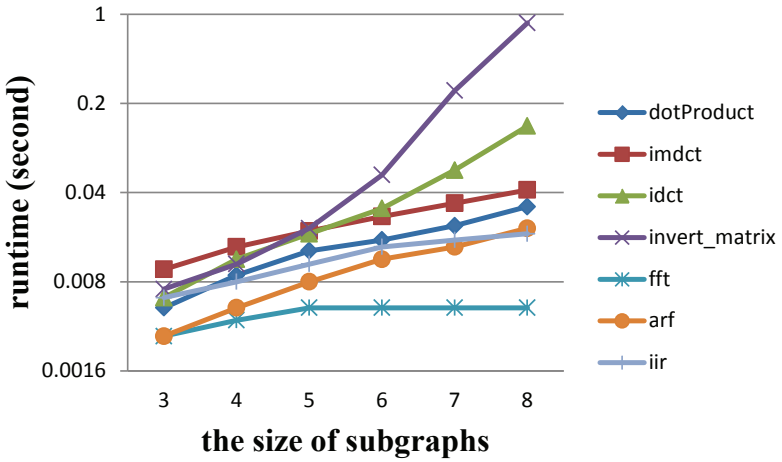


Figure A.1: The runtime performance of the subgraph enumeration step under size constraint (connected subgraphs)

Bibliography

[Ahn 2011] Junwhan Ahn, Imyong Lee and Kiyoung Choi. *A polynomial-time custom instruction identification algorithm based on dynamic programming.* In Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific, pages 573 –578, jan. 2011.

[Aho 1989] Alfred V. Aho, Mahadevan Ganapathi and Steven W. K. Tjiang. *Code generation using tree matching and dynamic programming.* ACM Trans. Program. Lang. Syst., vol. 11, no. 4, pages 491–516, October 1989.

[Aletà 2004] Alex Aletà, Josep M. Codina, Antonio González and David Kaeli. *Removing communications in clustered microarchitectures through instruction replication.* ACM Trans. Archit. Code Optim., vol. 1, no. 2, pages 127–151, June 2004.

[Alippi 1999] Cesare Alippi, William Fornaciari, Laura Pozzi and Mariagiovanna Sami. *A DAG-based design approach for reconfigurable VLIW processors.* In Proceedings of the conference on Design, automation and test in Europe, DATE '99, New York, NY, USA, 1999. ACM.

[Arnold 2001] Marnix Arnold and Henk Corporaal. *Designing domain-specific processors.* In Proceedings of the ninth international symposium on Hardware/software codesign, CODES '01, pages 61–66, New York, NY, USA, 2001. ACM.

[Atasu 2003] Kubilay Atasu, Laura Pozzi and Paolo Ienne. *Automatic application-specific instruction-set extensions under microarchitectural constraints.* In Proceedings of the 40th annual Design Automation Conference, DAC '03, pages 256–261, New York, NY, USA, 2003. ACM.

[Atasu 2005] Kubilay Atasu, Günhan Dündar and Can C. Özturan. *An integer linear programming approach for identifying instruction-set extensions.* In CODES+ISSS, pages 172–177, 2005.

[Atasu 2007] Kubilay Atasu. *Design Methodologies For Intruction-Set Extensible Processors.* PhD thesis, 2007.

[Atasu 2008] Kubilay Atasu, Oskar Mencer, Wayne Luk, Can Ozturan and Gunhan Dundar. *Fast custom instruction identification by convex subgraph enumeration.* In Proceedings of the 2008 International Conference on Application-Specific Systems, Architectures and Processors, ASAP '08, pages 1–6, Washington, DC, USA, 2008. IEEE Computer Society.

[Bailey 2010] B. Bailey, F. Balarin, M. McNamara, G. Mosenson, M. Stellfox and Y. Watanabe. *Tlm-driven design and verification methodology.* Lulu Enterprises Inc., 2010.

[Baleani 2002] Massimo Baleani, Frank Gennari, Yunjian Jiang, Yatish Patel, Robert K. Brayton and Alberto Sangiovanni-Vincentelli. *HW/SW partitioning and code generation of embedded control applications on a reconfigurable architecture platform.*

In Proceedings of the tenth international symposium on Hardware/software codesign, CODES '02, pages 151–156, New York, NY, USA, 2002. ACM.

[Biswas 2003] Partha Biswas and Nikil Dutt. *Reducing code size for heterogeneous-connectivity-based VLIW DSPs through synthesis of instruction set extensions.* In Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems, CASES '03, pages 104–112, New York, NY, USA, 2003. ACM.

[Biswas 2005] Partha Biswas and Nikil D. Dutt. *Code Size Reduction in Heterogeneous-Connectivity-Based DSPs Using Instruction Set Extensions.* IEEE Trans. Computers, vol. 54, no. 10, pages 1216–1226, 2005.

[BlueSpec] BlueSpec. *http://www.bluespec.com.*

[Bollaert 2008] Thomas Bollaert. *Catapult Synthesis: A Practical Introduction to Interactive C Synthesis High-Level Synthesis.* In Philippe Coussy and Adam Morawiec, editeurs, High-Level Synthesis, chapitre 3, pages 29–52. Springer Netherlands, Dordrecht, 2008.

[Bonzini 2007] Paolo Bonzini and Laura Pozzi. *Polynomial-time subgraph enumeration for automated instruction set extension.* In Proceedings of the conference on Design, automation and test in Europe, DATE '07, pages 1331–1336, San Jose, CA, USA, 2007. EDA Consortium.

[Bozorgzadeh 2002] Elaheh Bozorgzadeh, Seda Ogrenci, Memik Ryan and Kastner Majid Sarrafzadeh. *Pattern selection: customized block allocation for domain-specific programmable systems.* In in Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, 2002.

[Cadence] Cadence. *http://www.cadence.com/rl/resources/technical_papers Cadence C-to-Silicon White Paper.*

[Canis 2011] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown and Tomasz Czajkowski. *LegUp: high-level synthesis for FPGA-based processor/accelerator systems.* In Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays, FPGA '11, pages 33–36, New York, NY, USA, 2011. ACM.

[Caprara 1999] Alberto Caprara, Matteo Fischetti and Paolo Toth. *A Heuristic Method for the Set Covering Problem.* Oper. Res., vol. 47, no. 5, pages 730–743, May 1999.

[Chen 2007] X. Chen, D. L. Maskell and Y. Sun. *Fast Identification of Custom Instructions for Extensible Processors.* Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 26, no. 2, pages 359 –368, feb. 2007.

[Choi 1999] Hoon Choi, Jong-Sun Kim, Chi-Won Yoon, In-Cheol Park, Seung Ho Hwang and Chong-Min Kyung. *Synthesis of Application Specific Instructions for Embedded DSP Software.* IEEE Trans. Comput., vol. 48, no. 6, pages 603–614, June 1999.

[Clark 2003] Nathan Clark, Hongtao Zhong and Scott Mahlke. *Processor Acceleration Through Automated Instruction Set Customization.* In Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, MICRO 36, pages 129–, Washington, DC, USA, 2003. IEEE Computer Society.

[Clark 2005] Nathan T. Clark, Hongtao Zhong and Scott A. Mahlke. *Automated Custom Instruction Generation for Domain-Specific Processor Acceleration.* IEEE Trans. Comput., vol. 54, no. 10, pages 1258–1270, October 2005.

[Clark 2006] Nathan Clark, Amir Hormati, Scott Mahlke and Sami Yehia. *Scalable subgraph mapping for acyclic computation accelerators.* In Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems, CASES '06, pages 147–157, New York, NY, USA, 2006. ACM.

[Cong 2004] Jason Cong, Yiping Fan, Guoling Han and Zhiru Zhang. *Application-specific instruction generation for configurable processor architectures.* In Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays, FPGA '04, pages 183–189, New York, NY, USA, 2004. ACM.

[Cong 2006] J. Cong, Yiping Fan, Guoling Han, Wei Jiang and Zhiru Zhang. *Platform-Based Behavior-Level and System-Level Synthesis.* In SOC Conference, 2006 IEEE International, pages 199 –202, sept. 2006.

[Cong 2008] Jason Cong and Wei Jiang. *Pattern-based behavior synthesis for FPGA resource reduction.* In FPGA, pages 107–116, 2008.

[Cong 2010] Jason Cong, Hui Huang and Wei Jiang. *A generalized control-flow-aware pattern recognition algorithm for behavioral synthesis.* In DATE, pages 1255–1260, 2010.

[Cong 2011a] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers and Zhiru Zhang. *High-Level Synthesis for FPGAs: From Prototyping to Deployment.* Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 30, no. 4, pages 473 –491, april 2011.

[Cong 2011b] Jason Cong, Hui Huang and Wei Jiang. *Pattern-Mining for Behavioral Synthesis.* IEEE Trans. on CAD of Integrated Circuits and Systems, vol. 30, no. 6, pages 939–944, 2011.

[Coussy 2008] Philippe Coussy, Cyrille Chavet, Pierre Bomel, Dominique Heller, Eric Senn and Eric Martin. *GAUT: A High-Level Synthesis Tool for DSP Applications.* In Philippe Coussy and Adam Morawiec, editeurs, High-Level Synthesis, chapitre 9, pages 147–169. Springer Netherlands, 2008.

[Coussy 2009] Philippe Coussy, Daniel D. Gajski, Michael Meredith and Andres Takach. *An Introduction to High-Level Synthesis.* IEEE Design & Test of Computers, vol. 26, pages 8–17, 2009.

[de Jong 1991] Gjalt G. de Jong. *Data flow graphs: system specification with the most unrestricted semantics.* In Proceedings of the conference on European design automation, EURO-DAC '91, pages 401–405, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[Dinh 2008] Quang Dinh, Deming Chen and Martin D. F. Wong. *Efficient ASIP design for configurable processors with fine-grained resource sharing.* In Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays, FPGA '08, pages 99–106, New York, NY, USA, 2008. ACM.

[Ebeling 1983] C. Ebeling and O. Zajicek. *Validating VLSI circuit layout by wirelist comparison.* In Proceedings of the International Conference on Computer-Aided Design, 1983.

[Edwards 2002] Stephen A. Edwards. *High-Level Synthesis from the Synchronous Language Esterel.* In IWLS, pages 401–406, 2002.

[Galuzzi 2006] Carlo Galuzzi, Elena Moscu Panainte, Yana Yankova, Koen Bertels and Stamatis Vassiliadis. *Automatic selection of application-specific instruction-set extensions.* In Proceedings of the 4th international conference on Hardware/software codesign and system synthesis, CODES+ISSS '06, pages 160–165, New York, NY, USA, 2006. ACM.

[Galuzzi 2007a] Carlo Galuzzi, Koen Bertels and Stamatis Vassiliadis. *A Linear Complexity Algorithm for the Automatic Generation of Convex Multiple Input Multiple Output Instructions.* In ARC, pages 130–141, 2007.

[Galuzzi 2007b] Carlo Galuzzi, Koen Bertels and Stamatis Vassiliadis. *A linear complexity algorithm for the generation of multiple input single output instructions of variable size.* In Proceedings of the 7th international conference on Embedded computer systems: architectures, modeling, and simulation, SAMOS'07, pages 283–293, Berlin, Heidelberg, 2007. Springer-Verlag.

[Galuzzi 2011] Carlo Galuzzi and Koen Bertels. *The Instruction-Set Extension Problem: A Survey.* ACM Trans. Reconfigurable Technol. Syst., vol. 4, no. 2, pages 18:1–18:28, May 2011.

[Garey 1990] Michael R. Garey and David S. Johnson. Computers and intractability; a guide to the theory of np-completeness. W. H. Freeman & Co., New York, NY, USA, 1990.

[GECOS] GECOS. *Generic compiler suite - http://gecos.gforge.inria.fr/.*

[Gonzalez 2000] R.E. Gonzalez. *Xtensa: a configurable and extensible processor.* Micro, IEEE, vol. 20, no. 2, pages 60 –70, mar/apr 2000.

[Graphic] Mentor Graphic. *http://www.mentor.com/esl/catapult/overview. the high-level synthesis tool Catapult-C.*

[Guo 2003] Yuanqing Guo, Gerard J. M. Smit, Hajo Broersma and Paul M. Heysters. *A graph covering algorithm for a coarse grain reconfigurable system.* In LCTES, pages 199–208, 2003.

[Gupta 2003] Sumit Gupta, Nikil Dutt, Rajesh Gupta and Alex Nicolau. *SPARK : A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations.* VLSI Design, International Conference on, vol. 0, page 461, 2003.

[Gupta 2004] Sumit Gupta, Nikil D. Dutt and Rajesh Gupta. *Spark: : A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Springer, June 2004.

[Guthaus 2001] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge and R.B. Brown. *MiBench: A free, commercially representative embedded benchmark suite*. In Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on, pages 3 – 14, dec. 2001.

[Holland 1975] John H. Holland. *Adaptation in natural and artificial systems*. Univ. of Mochigan Press, 1975.

[Jozwiak 2010] Lech Jozwiak, Nadia Nedjah and Miguel Figueroa. *Modern development methods and tools for embedded reconfigurable systems: A survey*. Integration, the VLSI Journal, vol. 43, no. 1, pages 1 – 33, 2010.

[Kastner 2002] R. Kastner, A. Kaplan, S. Ogrenci Memik and E. Bozorgzadeh. *Instruction generation for hybrid reconfigurable systems*. ACM Trans. Des. Autom. Electron. Syst., vol. 7, no. 4, pages 605–627, October 2002.

[Kathail 2002] V. Kathail, S. Aditya, R. Schreiber, B. Ramakrishna Rau, D.C. Cronquist and M. Sivaraman. *PICO: automatically designing custom computers*. Computer, vol. 35, no. 9, pages 39 – 47, sep 2002.

[Kavvadias 2005] N. Kavvadias and S. Nikolaidis. *Automated instruction-set extension of embedded processors with application to MPEG-4 video encoding*. In Application-Specific Systems, Architecture Processors, 2005. ASAP 2005. 16th IEEE International Conference on, pages 140 – 145, july 2005.

[Kavvadias 2006] N. Kavvadias and S. Nikolaidis. *A flexible instruction generation framework for extending embedded processors*. In Electrotechnical Conference, 2006. MELECON 2006. IEEE Mediterranean, pages 125 –128, may 2006.

[Kazutoshi 2008] Wakabayashi Kazutoshi and Schafer Benjamin C. *All-in-C: Behavioral Synthesis and Verification with CyberWorkBench High-Level Synthesis*. In Philippe Coussy and Adam Morawiec, editeurs, High-Level Synthesis, chapitre 7, pages 113–127. Springer Netherlands, Dordrecht, 2008.

[Kuchcinski 2003] Krzysztof Kuchcinski. *Constraints-driven scheduling and resource assignment*. ACM Trans. Des. Autom. Electron. Syst., vol. 8, no. 3, pages 355–383, July 2003.

[Lam 2009] Siew-Kei Lam and Thambipillai Srikanthan. *Rapid design of area-efficient custom instructions for reconfigurable embedded processing*. J. Syst. Archit., vol. 55, no. 1, pages 1–14, January 2009.

[Larrosa 2002] Javier Larrosa and Gabriel Valiente. *Constraint satisfaction algorithms for graph pattern matching*. Mathematical. Structures in Comp. Sci., vol. 12, no. 4, pages 403–422, August 2002.

[Lee 1997] Chunho Lee, M. Potkonjak and W.H. Mangione-Smith. *MediaBench: a tool for evaluating and synthesizing multimedia and communications systems*. In Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on, pages 330 –335, dec 1997.

[Lee 2002] Jong-eun Lee, Kiyoung Choi and Nikil Dutt. *Efficient instruction encoding for automatic instruction set design of configurable ASIPs*. In Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design, ICCAD '02, pages 649–654, New York, NY, USA, 2002. ACM.

[Li 2009] Tao Li, Zhigang Sun, Wu Jigang and Xicheng Lu. *Fast enumeration of maximal valid subgraphs for custom-instruction identification*. In Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems, CASES '09, pages 29–36, New York, NY, USA, 2009. ACM.

[Li 2010] Tao Li, Wu Jigang, Siew Kei Lam, Thambipillai Srikanthan and Xicheng Lu. *Selecting profitable custom instructions for reconfigurable processors*. Journal of Systems Architecture - Embedded Systems Design, vol. 56, no. 8, pages 340–351, 2010.

[Liao 1995] Stan Liao, Srinivas Devadas, Kurt Keutzer and Steve Tjiang. *Instruction selection using binate covering for code size optimization*. In Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design, ICCAD '95, pages 393–399, Washington, DC, USA, 1995. IEEE Computer Society.

[Liao 1997] Stan Liao and Srinivas Devadas. *Solving covering problems using LPR-based lower bounds*. In Proceedings of the 34th annual Design Automation Conference, DAC '97, pages 117–120, New York, NY, USA, 1997. ACM.

[Liao 1998] S. Liao, K. Keutzer, S. Tjiang and S. Devadas. *A new viewpoint on code generation for directed acyclic graphs*. ACM Trans. Des. Autom. Electron. Syst., vol. 3, no. 1, pages 51–75, January 1998.

[Liem 1994a] C. Liem, T. May and P. Paulin. *Instruction-set matching and selection for DSP and ASIP code generation*. In European Design and Test Conference, 1994. EDAC, The European Conference on Design Automation. ETC European Test Conference. EUROASIC, The European Event in ASIC Design, Proceedings., pages 31 –37, feb-3 mar 1994.

[Liem 1994b] C. Liem, T. May and P. Paulin. *Instruction-set matching and selection for DSP and ASIP code generation*. In European Design and Test Conference, 1994. EDAC, The European Conference on Design Automation. ETC European Test Conference. EUROASIC, The European Event in ASIC Design, Proceedings., pages 31 –37, feb-3 mar 1994.

[Ling 1996] Zong Ling and D.Y.Y. Yun. *An efficient subcircuit extraction algorithm by resource management*. In ASIC, 1996., 2nd International Conference on, pages 9 –14, oct 1996.

[Malay 2001] Haldar Malay, Nayak Anshuman, Choudhary Alok and Banerjee Prith. *A system for synthesizing optimized fpga hardware from matlab.* In in Proc. of the 2001 IEEE/ACM international conference on Computer-aided design ICCAD'01, pages 314–319, 2001.

[Martin 2009a] Kevin Martin, Christophe Wolinski, Krzysztof Kuchcinski, Antoine Floch and François Charot. *Constraint-Driven Identification of Application Specific Instructions in the DURASE System.* In SAMOS, pages 194–203, 2009.

[Martin 2009b] Kevin Martin, Christophe Wolinski, Krzysztof Kuchcinski, Antoine Floch and François Charot. *Constraint-Driven Instructions Selection and Application Scheduling in the DURASE system.* In ASAP, pages 145–152, 2009.

[McFarland 1988] Michael C. McFarland, Alice C. Parker and Raul Camposano. *Tutorial on high-level synthesis.* In Proceedings of the 25th ACM/IEEE Design Automation Conference, DAC '88, pages 330–336, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[Meredith 2008] Michael Meredith. *High-Level SystemC Synthesis with Forte's Cynthesizer High-Level Synthesis.* In Philippe Coussy and Adam Morawiec, editeurs, High-Level Synthesis, chapitre 5, pages 75–97. Springer Netherlands, Dordrecht, 2008.

[Muchnick 1997] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[P. Cordella 2004] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone and Mario Vento. *A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs.* IEEE Trans. Pattern Anal. Mach. Intell., vol. 26, no. 10, pages 1367–1372, October 2004.

[Pothineni 2007] Nagaraju Pothineni, Anshul Kumar and Kolin Paul. *Application Specific Datapath Extension with Distributed I/O Functional Units.* In Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference: Embedded Systems, VLSID '07, pages 551–558, Washington, DC, USA, 2007. IEEE Computer Society.

[Pozzi 2002] L. Pozzi, M. Vuletic and P. Ienne. *Automatic topology-based identification of instruction-set extensions for embedded processors.* In Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings, page 1138, 2002.

[Pozzi 2006] Laura Pozzi, Kubilay Atasu, Student Member and Paolo Ienne. *Exact and approximate algorithms for the extension of embedded processor instruction sets.* IEEE Trans. on CAD of Integrated Circuits and Systems, pages 1209–1229, 2006.

[Rao 1992] D. S. Rao and F. J. Kurdahi. *Partitioning by regularity extraction.* In Proceedings of the 29th ACM/IEEE Design Automation Conference, DAC '92, pages 235–238, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[Reddington 2009] J. Reddington, G. Gutin, A. Johnstone, E. Scott and A. Yeo. *Better Than Optimal: Fast Identification of Custom Instruction Candidates.* In Computational Science and Engineering, 2009. CSE '09. International Conference on, volume 2, pages 17 –24, aug. 2009.

[Reddington 2011] Joseph Reddington and Kubilay Atasu. *Complexity of Computing Convex Subgraphs in Custom Instruction Synthesis*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, pages 1–5, 2011.

[Ren 2010] Zhi-Gang Ren, Zu-Ren Feng, Liang-Jun Ke and Zhao-Jun Zhang. *New ideas for applying ant colony optimization to the set covering problem*. Comput. Ind. Eng., vol. 58, no. 4, pages 774–784, May 2010.

[Shu 1996] J. Shu, T.C. Wilson and D.K. Banerji. *Instruction-set matching and GA-based selection for embedded-processor code generation*. In VLSI Design, 1996. Proceedings., Ninth International Conference on, pages 73 –76, jan 1996.

[Synopsys] Synopsys. *http://www.synopsys.com/systems/blockdesign/hls. the high-level synthesis tool Synphony*.

[Tri 2005] *Trident: an FPGA compiler framework for floating-point algorithms*, 2005.

[Tripp 2007] J.L. Tripp, M.B. Gokhale and K.D. Peterson. *Trident: From High-Level Language to Hardware Circuitry*. Computer, vol. 40, no. 3, pages 28 –37, march 2007.

[Ullmann 1976] J. R. Ullmann. *An Algorithm for Subgraph Isomorphism*. J. ACM, vol. 23, no. 1, pages 31–42, January 1976.

[van Eijndhoven 1992] J.T.J. van Eijndhoven and L. Stok. *A data flow graph exchange standard*. In Design Automation, 1992. Proceedings., [3rd] European Conference on, pages 193 –199, mar 1992.

[Verma 2007] Ajay K. Verma, Philip Brisk and Paolo Ienne. *Rethinking custom ISE identification: a new processor-agnostic method*. In Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems, CASES '07, pages 125–134, New York, NY, USA, 2007. ACM.

[Wakabayashi 2006] Kazutoshi Wakabayashi. *Unified Representation for Speculative Scheduling: Generalized Condition Vector*. IEICE Trans. Fundam. Electron. Commun. Comput. Sci., vol. E89-A, no. 12, pages 3408–3415, December 2006.

[Wolinski 2007] C. Wolinski and K. Kuchcinski. *Identification of Application Specific Instructions Based on Sub-Graph Isomorphism Constraints*. In Application -specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on, pages 328 –333, july 2007.

[Wong 1992] E. K. Wong. *Model matching in robot vision by subgraph isomorphism*. Pattern Recogn., vol. 25, no. 3, pages 287–303, March 1992.

[Xiao 2011] Chenglong Xiao and E. Casseau. *Efficient maximal convex custom instruction enumeration for extensible processors*. In Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on, pages 1 –7, nov. 2011.

[Yadav 1999] N. Yadav, M. Schulte and J. Glossner. *Parallel saturating fractional arithmetic units*. In VLSI, 1999. Proceedings. Ninth Great Lakes Symposium on, pages 214 –217, mar 1999.

[Yu 2004a] Pan Yu and Tulika Mitra. *Characterizing embedded applications for instruction-set extensible processors.* In Proceedings of the 41st annual Design Automation Conference, DAC '04, pages 723–728, New York, NY, USA, 2004. ACM.

[Yu 2004b] Pan Yu and Tulika Mitra. *Scalable custom instructions identification for instruction-set extensible processors.* In Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems, CASES '04, pages 69–78, New York, NY, USA, 2004. ACM.

[Yu 2007] Pan Yu and T. Mitra. *Disjoint Pattern Enumeration for Custom Instructions Identification.* In Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on, pages 273 –278, aug. 2007.

[Yu 2008] Pan Yu. *Design Methodologies For Intruction-Set Extensible Processors.* PhD thesis, 2008.

[Zhang 2008] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang and Jason Cong. *AutoPilot: A Platform-Based ESL Synthesis System.* pages 99–112, 2008.

List of publications

Journal

Chenglong Xiao, Emmanuel Casseau: *Exact custom instruction enumeration for extensible processors.* Elsivier, Integration, the VLSI journal , 2012 (Invited: Special Issue on Best Papers from ACM GLSVLSI 2011)

International Conferences

Chenglong Xiao, Emmanuel Casseau: *Efficient maximal convex custom instruction enumeration for extensible processors.* DASIP 2011, Tampere, Finland

Chenglong Xiao, Emmanuel Casseau: *Efficient custom instruction enumeration for extensible processors.* IEEE ASAP 2011, Santa Monica, USA

Chenglong Xiao, Emmanuel Casseau: *An efficient algorithm for custom instruction enumeration.* ACM GLSVLSI 2011, Lausanne, Switzerland

National Conference

Chenglong Xiao, Emmanuel Casseau: *Pattern Extraction for Digital Design.* Colloque GDR SOC-SIP 2010, Paris, France.

Custom Operator Identification for High-level Synthesis

Abstract: It is increasingly common to see custom operators appear in various fields of circuit design. Custom operators that can be implemented in special hardware units make it possible to reduce code size, improve performance and reduce area. In this thesis, we propose a design flow based on custom operator identification for high-level synthesis. The key issues involved in the design flow are: automatic enumeration and selection of custom operators from a given high-level application code and re-generation of the source code incorporating the selected custom operators. Unlike the previously proposed approaches, our design flow is quite adaptable and is independent of high-level synthesis tools (i.e., without modifying the scheduling and binding algorithms in high-level synthesis tools). Experimental results show that our approach achieves on average 19%, and up to 37% area reduction, compared to a traditional high-level synthesis. Meanwhile, the latency is reduced on average by 22%, and up to 59%. Furthermore, on average 74% and up to 81% code size reduction can be achieved.

Identification d'opérateurs spécifiques pour la synthèse de haut niveau

Resumé : Il est de plus en plus fréquent de faire appel à des opérateurs spécifiques en conception de circuits. Les opérateurs spécifiques peuvent être mis en œuvre par des unités matérielles dédiées, en vue de réduire la taille du code, d'améliorer les performances et de réduire la surface du circuit. Dans cette thèse, nous proposons un flot de conception basé sur l'identification d'opérateurs spécifiques pour la synthèse de haut niveau. Les points clés de ce flot de conception sont l'énumération automatique et la sélection des opérateurs spécifiques à partir d'un code de l'application de haut niveau et la re-génération du code source intégrant les opérateurs spécifiques sélectionnés. Contrairement aux approches proposées précédemment, notre flot de conception est adaptable et est indépendant des outils de synthèse de haut niveau (il ne nécessite pas d'intervenir sur les algorithmes d'ordonnancement et de projection des outils de synthèse de haut niveau). Les résultats expérimentaux montrent que notre approche permet de réduire la surface du circuit de 19% en moyenne, et jusqu'à 37% dans certains cas, par rapport à une synthèse de haut niveau traditionnelle. La latence du circuit est réduite en moyenne de 22%, et atteint jusqu'à 59%. De plus, la taille du code est réduite de 74% en moyenne.