Domain Specific Instruction Set Extensions

Adithya Pulli

Delft University of Technology

Domain Specific Instruction Set Extensions

Master's Thesis in Computer Engineering

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Adithya Pulli

22nd May 2014

Author
Adithya Pulli

Title
Domain Specific Instruction Set Extensions

MSc presentation
28th May 2014

Graduation Committee

Abstract

Over the past years, a considerable amount of effort has been devoted to the definition and implementation of techniques for the optimization and acceleration of applications on various computing platforms. Among these techniques, the extension of a given instruction-set architecture with custom instructions has become a common approach. Custom instructions effectively reduce the dynamic instruction count, which, in turn, leads to increase in performance and reduction in power consumption. Traditionally, existing techniques address Instruction-Set Extension (ISE) on a per-application basis. Anyhow, when many applications have to be considered at the same time, ISE on a per-application basis is, clearly, less effective, as the custom instructions have often limited re-utilization across applications. To overcome this problem, we propose a new framework for the automatic generation of domain-specific ISEs. Experimental results show that, the proposed framework, evaluated on a number of applications from various domains, can effectively generate domain-specific instructions with high utilization factor across the targeted applications. At the same time, the generated instructions reduce the instruction count, 45% on average and upto 80% in special cases. This, in turn, can lead to considerable improvements in performance and reduction of power consumption.

Preface

The idea of identifying commonality across the applications emerged during a discussion with Prof S.K. Nandy at Indian Institute of Sciences in May 2012. The discussion was about specializing a Coarse Grained Reconfigurable Array (CGRA) for an application domain by replacing ALU operations with complex operations in the Functional Units (FUs). Like any other researcher I was passionate about my work and during my attempts to understand the problem in more detail, I ended up in the website of Carlo. While glancing through his summary paper on Instruction Set Extension (ISE), I understood that the problem of identifying domain-specific FUs is equivalent to domain-specific ISE which is the central theme of this thesis.

The work presented in this thesis is the outcome of research performed as part of my MSc program in Computer Engineering. Before I continue further, I would like to mention that this piece of paper can neither capture the extent of my gratitude nor the entire list of people who have directly or indirectly helped me in carrying out this thesis.

I would like to thank my supervisor Prof. Georgi Gaydadjiev for giving me the freedom to work on the topic of my choice. Despite his extremely busy schedule, he managed to advice me at every stage of the project, right from problem formulation to writing this thesis. I am also equally grateful to Dr Carlo Galuzzi, for being my mentor and daily supervisor. I would like to thank him for sharing with me his insights on graph theory based approach to ISE and for patiently correcting and proof reading this thesis. I would like to thank Prof Henk Sips and Dr Rene van Leuken for readily accepting to be members of my thesis committee.

I would like to thank Phani, Rajeev, Varun, Sriram, Vishu, Abhi, Sumedh, Harshitha, Arun and Manju for proving me the necessary and sufficient distraction from the thesis. I would like to thank my parents for having confidence in me and for supporting me to pursue a Masters degree away from home.

Adithya Pulli

Delft, The Netherlands
22nd May 2014

Contents

Chapter 1

Introduction

Traditionally, embedded processors have been specialized for single applications or small set of applications. Anyhow, processor specialization has become a common approach to deal with every increasing computational power required by software applications. The trend in processor industry shows that, with the advent of heterogeneous computing, desktop and server processors will also contain specialized cores [9].

Over the years, many techniques have been proposed and used in both industry and academia to specialize general purpose processors for the required application(s). One such technique consists in the augmentation of a processor core with special-purpose hardware to improve the application performance. This special purpose hardware can be in the form of either a custom functional unit, or a co-processor subsystem, or an accelerator, and it is exposed to the programmer as a custom instruction. The identification of these custom instructions constitutes a major challenge, usually addressed in literature as the Instruction-Set Extension (ISE) problem.

Traditionally, the ISE problem is addressed on per-application basis. The unique combination of increasing demand for computational power and diversifying nature of the applications makes application-specific custom instructions less favourable. Custom instructions should shift from application-specific to domain-specific, in a sense that each custom instruction should have high utilization across many applications (in a domain), while still delivering the required performance improvements. This requirement for effective re-utilization of custom instructions leads to a new challenge in the custom instruction generation process.

In this thesis we propose a framework to address the problem of automatically identifying domain specific ISE. In the following, we first present an overview of the problem studied in this thesis, after that, we outline our major contributions. The chapter concludes with an overview of the thesis organization.

1.1 Problem Description

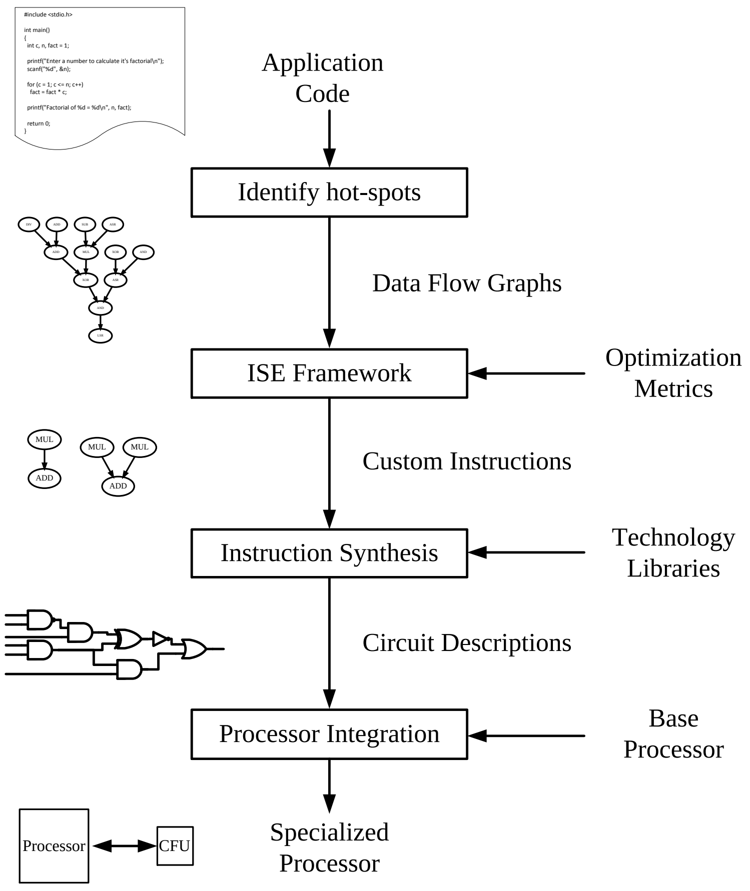
Over the years computational platforms have evolved according to the demands of the market. Need for better performance (processing speeds) has been one of the major drivers of growth in the processor industry. Frequency scaling provided an easy and straight-forward method to scale performance upto the late 90s. At the beginning of 21^{st} century, industry moved towards multi-core systems after processors hit a performance wall, due to saturation in the frequency scaling. Today, multi-core systems have penetrated into all the major segments of the market, from mobile phones to super computers.

Performance demands continue to grow across all segments of computing. In computing systems catering for consumer markets, there is a demand to enable compelling user experience. On the other end, in data-centres, there is an increasing demand for performance due to the vast amount of data that needs to be processed. Heterogeneity in the processing platforms is perceived as an answer to providing performance scaling in processors for the next decade and more [9]. Heterogeneous multi-core systems combine specialized processors and general purpose processor, on a single die. These specialized processors (cores) are optimized to efficiently support set of applications from a number of domains.

As mentioned earlier, supplementing a processor's instruction set with custom instructions is one of the many dimensions in which a processor can be specialized for a set of applications. With custom instructions, a processor can approach better performance levels compared to a GPP and can sustain this performance gain for modest changes in the application(s), such as bug fixes or incremental modifications to the standard. Over the years, many researchers have shown that automation is the key to make ISE successful [31, 4, 21, 16, 43, 20]. This ensures that the cost of the system and the overall design time are lowered.

Figure 1.1 shows the steps involved in augmenting a processor with custom instruction. After profiling an input application code, the parts of application code with high profile numbers are converted into graph representations. An ISE framework takes these graph representations as input and, based on certain optimization metric, identifies clusters of operations, which are sub-graphs of the input graph, for hardware implementation. Later, these clusters (sub-graphs), which are the custom instructions are synthesized (implemented) for a technology node. The implemented custom instructions are then integrated into a processor core. Every step shown in Figure 1.1 poses interesting research questions. The scope of this thesis is restricted to the ISE as a graph problem.

Over the years, many frameworks have been proposed for the automatic identification of custom instructions [25][29]. Most of these frameworks identify complex custom instruction(s) to improve the performance of a given application. Clearly, these application-specific custom instructions cannot cater for the future demands of specialized processors which, as mentioned before, are domain specific in nature. In this thesis, we make an effort to solve this problem. The framework proposed in this thesis takes a set of graphs, which correspond to the application hot-spots,



```
#include <stdio.h>

int main()
{
  int c, n, fact = 1;

  printf("Enter a number to calculate it's factorial\n");
  scanf("%d", &n);

  for (c = 1; c <= n; c++)
    fact = fact * c;

  printf("Factorial of %d = %d\n", n, fact);

  return 0;
}
```

Figure 1.1: A methodology for specializing a processor by using application-specific custom instructions.[1]

along with an optimization metric, and identifies a set of custom instructions. A formal description of the problem along with the algorithms involved in identification of the custom instructions are described in the next chapter.

1.2 Contributions

In this thesis, we present a methodology for the identification of custom domain-specific instructions. The custom instruction identification involves two steps: first, the candidate instructions which have better utilization across applications are enumerated. After that, based on a certain optimization metric, we select a sub-set of the enumerated instructions for hardware implementation. The selection process ensures that our instructions can produce maximum benefit for a given utilization

[1]The images are meant for illustrative purposes and they do not correspond to a real example.

factor.

More specifically the main contributions of this thesis are the following:

- the design and implementation of an ISE framework, which does not depend on the base processor and implementation technology;
- the formulation of the instruction generation problem as the Maximum Common Sub-graph problem, which is solved by transforming the problem into a maximum clique problem;
- the formulation of the instruction selection problem as a graph covering problem, which is solved by using sub-graph isomorphism and exact covering of a set;
- A theoretical proof that an application utilizing the custom instructions generated by our framework can be scheduled;
- A set of experiments to show that the custom instructions generated by our framework can reduce the dynamic instruction count, on average, by 45%.

1.3 Organization

The rest of this thesis is organized as follows.

In Chapter 2, we provide a formal definition of the ISE problem and we discuss the two steps involved: instruction generation and instruction selection. After that, we present a detailed overview of state-of-the-art in automatic ISE frameworks, which address the ISE problem at both application-specific and domain specific levels. We also provide an objective summary of some of the most popular ISE frameworks on the basis of the techniques they employ for instruction generation and instruction selection.

In Chapter 3, we extend our discussion on the ISE problem and we present our framework for domain specific ISE. We provide a formal definition of domain specific ISE and, later, describe the algorithms used by our framework. We also prove that custom instructions generated by our framework can be utilized by applications without any scheduling problem.

We evaluate the effectiveness of the custom instructions generated by our framework in Chapter 4. We first describe the experimental set-up and, later, present the experimental results. In this chapter we also present the drawbacks of evaluating ISE frameworks on basis of conventional metrics and we propose the dynamic instruction count as an effective metric to evaluate ISE frameworks in an architecture and technology independent way.

Finally, in Chapter 5, we provide concluding remarks. We outline the contributions of the thesis and we set direction for future research, which can be carried out based on our framework.

Chapter 2

Background and Related Work

The augmentation of a processor core with special-purpose hardware has become a common technique used to speed up the execution of applications. This special purpose hardware can be in the form of either a custom functional unit, or a co-processor subsystem, or an accelerator, and it is exposed to the programmer as a custom instruction. Supplementing a processor's instruction set with custom instructions has various advantages. These include, for example, reduced instruction execution cycles, improved performance, and reduced power consumption [25]. The identification of these custom instructions constitutes a major challenge, usually addressed in literature as the Instruction-Set Extension (ISE) problem. This chapter presents an overview of the ISE problem and briefly describes the existing solutions to the problem. The chapter begins by defining the ISE problem and, later, it introduces the two constituent sub-problems: custom instruction generation and custom instruction selection. The chapter continues with a description of state-of-the-art frameworks, which address the ISE problem at application-specific and domain-specific level. Finally, the chapter concludes with an overview of the existing frameworks and an account of how the framework proposed in this thesis stand against and complements the existing solutions.

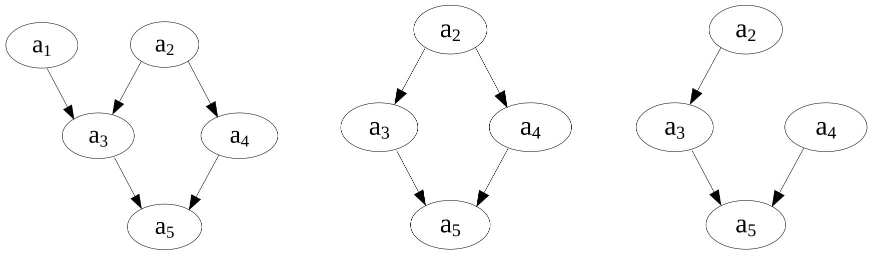
2.1 The Instruction Set Extension Problem

Many applications have the property that most of their execution time is spent in a small fraction of the source code. There is an informal rule, popularly known as 90-10 rule, which states that 90% of the execution time is spent in 10% of the code [42]. Therefore, one of the best ways to speed up an application is to accelerate its "hot spots", the portions of the application in which most of the execution time is spent.

ISE problem deals with the identification of custom instructions, which, usually, reduce the execution time of the "hot spots". Frameworks for automatic generation of custom instructions usually work on graph representations of the applications, called the Data Flow Graphs (DFGs). DFGs do not include control structures and, hence, are derived from the basic blocks[1] of the application code.

Definition 2.1. A **DFG** $G(V, E)$ is a directed acyclic graph, where vertices represent basic operations and edges represent data dependencies.

To understand the notion of complex custom instructions, we have to introduce two concepts: the induced sub-graphs and the convexity of a sub-graph. A sub-graph $G'(V', E')$ of a graph $G(V, E)$ is said to be an induced sub-graph, if $\forall u, v \in V', e = (u, v) \in E$ iff $e \in E'$. This means that an induced sub-graph G' contains all the edges over its vertex-set that appear in G, and only those. Figure 2.1b shows an example of an induced sub-graph.



(a) $G(V, E)$. (b) $G'(V', E')$. (c) $G''(V'', E'')$.

Figure 2.1: G' is an induced sub-graph of G. Anyhow, G'' is not a induced sub-graph of G as $(a_2, a_4) \in E \setminus E''$

Given a graph $G(V, E)$, a sub-graph $G'(V', E')$ is called convex if there exists no path from a node $u \in V'$ to another node $v \in V'$, which involves a node $w \in V \setminus V'$. For example, the sub-graph shown in Figure 2.2 is not convex as there exists a path from a_1 to a_4, which includes node $a_3 \in V \setminus V'$, by itself, is not part of the sub-graph.

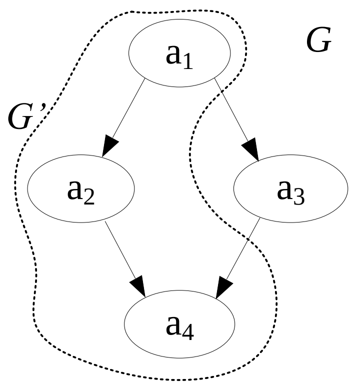


Figure 2.2: A non-convex sub-graph. $G'(V', E') \subset G(V, E)$

[1]Portions of the source code, which have only one entry point and one exit point [7].

Definition 2.2. Let G be a DFG of a given application. A custom instruction is represented by a convex induced sub-graph $G' \subseteq G$.

It can be observed from Def. 2.2 that a custom instruction, in certain cases, can be the DFG itself ($G' = G$). Although, this is a possible scenario, it's occurrence is very rare as memory operations and branch instructions are excluded from the custom instructions[2]. The identification of the convex induced sub-graphs which, together, can maximize a certain metric (e.g, performance), for a given application is a possible solutions to the ISE problem. Existing methodologies for ISE usually address the customization (extension) problem in two steps: first,they generate a suitable number of custom instructions; after that, they select, among the pool of candidate custom instructions a subset, which optimizes a given metric [25]. These steps are commonly addressed as instruction generation and instruction selection and they will be analysed in more detail in the following.

2.1.1 Instruction Generation

Instruction generation consists in the enumeration of a number of custom instructions (clusters of basic operations from the available instruction-set), which are either application- or domain-specific. This enumeration process is a *design space exploration* [24] which aims at identifying instruction that can be selected for hardware implementation. If the vertex set of a DFG has order n, then the total search space for identification of custom instructions is in the order of $O(2^n)$. Several techniques have been proposed to efficiently handle this complexity. The most common one involves efficient pruning of the search space by introducing specific constraints.

2.1.2 Instruction Selection

Once a pool of candidate instructions is available, instruction selection narrows them down by selecting a subset of instructions, which optimize (maximize/ minimize) specific metrics. These may include, for example, a reduction in the application execution time and/or a reduction in power consumption.

Usually, instruction selection is addressed as a covering problem or a 0/1 knapsack problem, which are widely known to be NP-complete [23]. The proposed solutions are either exact, whenever possible, or heuristics when a solution cannot be computed in a feasible time. In the next sections, an overview of the current state-of-the-art in ISE is described in more detail.

Traditionally, ISE problem has been addressed on a per-application basis. The analysis is done on one application at a time and one or more instructions are identified to improve the performance of execution of the application. This approach for ISE is categorized as application-specific ISE in the rest of this thesis. As outlined

[2]Though, techniques for including memory operations in custom instructions exist in literature [15], they are are seldom used in the frameworks

in Chapter 1, application-specific ISE is not a scalable approach, if the number of applications that needs to be accelerated grows. Domain-specific ISE aims at maximizing the utilization of custom instructions across a set of applications (in a domain), while still delivering required performance improvements. Some existing frameworks for ISE consider re-utilization of custom instructions as a goal. Anyhow, they only deal with instruction reuse within an application.

In the next sections, various application-specific and domain-specific ISE frameworks are investigated in more detail. The main objective of the rest of the chapter is to provide an overview of some common approaches adopted by researchers to tackle the complexity of the ISE problem and provide solutions. It does not try to present every existing technique in ISE. Extensive overviews of the different methodologies developed over the past years are presented in [29] and [25].

2.2 Application-Specific ISE

As mentioned earlier, automatic application-specific ISE frameworks identify custom instructions, which can improve the performance of execution of a given application. Some of the widely adopted approaches for application-specific ISE are presented in the following. While the early research in ISE only addressed instruction selection from a predefined set of templates, state-of-the-art ISE frameworks address both instruction generation and selection. These frameworks are based on either exhaustive search of the design space or incremental clustering of nodes to generate instructions.

2.2.1 Predefined Templates

A template is a cluster of simple ALU operations such as, for example, Multiply ACcumulate (MAC) operation, which is a very common operation in signal processing applications. Early work in automatic ISE totally side stepped the instruction generation by assuming the existence of template library [34, 8]. These frameworks require the designer to enumerate a superset of useful custom instruction candidates. Only the instruction selection phase is automated. This approach has severe limitations due to the manual design effort required for the instruction generation. The quality of the results is often determined by the expertise of the designer. Additionally, the design time is very large and, hence, is not scalable when we consider multiple applications. Most of the recent works in ISE consider automatic instruction generation and, hence, unlike the previous works, custom templates are generated after analysis of the application. Some of the most common approaches adopted by the researchers for automatic instruction generation are described later in the chapter.

2.2.2 Exhaustive Search

This process involves the exploration of the entire design space. As an exhaustive enumeration of all the sub-graphs is computationally impractical[3], researchers introduce constraints, which usually represent architectural limitations of the processor(s) targeted in their work.

In [13], Atasu *et al.* proposed a single-cut identification algorithm, which maximize the gain per custom instruction. They used a branch-and-bound algorithm with an efficient pruning technique based on the input/output (I/O) constraints for the identification of custom instructions. The constraints on the I/O degree of the sub-graphs is based on the assumption that a register file in the processor core can only have limited I/O ports. The authors also proposed an iterative selection algorithms with linear time complexity to choose a subset of identified instructions with maximum benefit.

Single-cut identification algorithm has exponential computational complexity and, hence, is not scalable. In [44], Yu *et al.* proposed a exhaustive enumeration algorithm similar to [13]. Anyhow the approach presented in [44] is quite scalable and can be applied to large data-flow graphs with relaxed micro-architectural constraints. The limitation of this approach is that the instruction generation can only identify connected sub-graphs. Additionally, the selection problem is not addressed in [44]. To reduce the run-time of the algorithms, Atasu *et al.* formulated the instruction generation as a Integer Linear Programming (ILP) problem [11]. Since ILP solvers use efficient heuristics, the run time is reduced.

All these approaches are restricted by the number of register ports in a register file. To overcome this limitation, Pozzi *et al.* [39] presented a framework, which performs I/O serialization of the instructions, with higher I/O degree than the I/O degree of the register file in a processor. In their framework, instruction identification is performed using the single-cut identification algorithm from [13] with relaxed I/O constraints. This approach for instruction identification has two major drawbacks. Firstly, the runtime increases significantly because relaxing I/O constraints also relaxes the bounds on the single-cut identification algorithm. Secondly, it is hard to determine a relaxation factor that produces optimal results. For instance, Pozzi *et al.* propose a relaxation of the I/O degree to 10 inputs and 5 outputs for the single-cut identification algorithm. Anyhow, custom instructions with a higher I/O degree can exist in the considered applications.

2.2.3 Incremental Clustering

In incremental clustering, a node is selected as a seed and, iteratively, nodes are merged together until certain constraints are satisfied. A guide function guides the clustering process by heuristically removing a number of edges during the exploration.

[3]We remind that given a graph with n nodes, there are upto 2^n possible sub-graphs (custom instructions), which can be enumerated.

In [21], Clark *et al.* proposed a framework for the identification and utilization of custom instructions. The instruction identification is based on incremental clustering under I/O and area constraints. The authors use a guide function which ranks desirability of edges based on three parameters: criticality, latency and area. The edges with smaller score are not considered during the generation of the sub-graph thus reducing the search space of the algorithm. Instruction selection is formulated as a 0/1 knapsack problem. The goal is to maximize the total value (performance gain) of a set of resources (identified custom instructions) for a given cost (area).

In [5], Alippi *et al.* described the MAXMISO partitioning. A MAXMISO is a MAXimal, Multiple Input and, Single Output sub-graph of a DFG, which is suitable for hardware implementation. Unlike the frameworks in [13, 21], their MAXMISO partitioning does not constrain the number of inputs an instruction can have and MAXMISOs are identified by recursively adding nodes to the parent node [38, 27]. Galuzzi *et al.* extended the notion of MISO clustering [24] of DFGs and presented various linear time clustering algorithms, which can identify MIMO (Multiple Input Multiple Output) clusters [26] in a DFG.

In [43], Verma *et al.* proposed a processor-agnostic approach to ISE without considering I/O degree during instruction identification. The authors formulated instruction identification as an identification of maximum cliques in a cluster graph. This approach is similar to incremental clustering. The nodes in the cluster graph represent a set of vertices of the DFG, which belong to a consistent set (see [43] for more details). The custom instructions identified using this approach are maximal convex sub-graphs in the given DFG. These custom instructions, usually, have a very high I/O degree. The authors formulate I/O serialization as a matrix problem to tackle the high I/O degree of the custom instructions.

2.2.4 Recurrence Aware ISE

All the ISE approaches mentioned before lead to the identification of large clusters of operations (custom instructions), which can provide maximum benefit for a given application. The goal of recurrence aware ISE is to increase the utilization of the identified instructions, which, in turn, leads to a reduction of the overall area. Recurrence aware ISE frameworks use either exhaustive search or incremental clustering techniques in conjunction with the recurrence criteria. Recurrence aware ISE by-itself is not a totally different technique and it can be considered as an extension of the earlier mentioned ISE techniques. While some authors [20][16] tackle recurrence problem during instruction selection, others [31, 17, 4] consider recurrence as a goal during instruction identification.

In [16], Bonzini et al. address the generation of custom instructions by considering high utilization and better gain of the custom instructions. Instruction selection is formulated as a covering problem, which aims at selecting a set of non-overlapping and recurrent custom instructions. The authors proposed both exact and approximate algorithms to solve the covering problem in conjunction with the recurrence of candidate instructions. Anyhow, they only consider the selection of custom instructions and leave their identification to the user. The covering problem is widely known to be NP-complete and, hence, the exact solution has a exponential time complexity in terms of number of custom instructions identified in the instruction generation stage.

Kastner *et al.* [31] were the first to consider recurrence in the instruction generation stage. Their algorithm performs simultaneous template generation and matching for the given application. At each step, the algorithm selects most common edges and clusters them into super-nodes. The algorithm stops when a sufficient number of templates to maximally cover the graph are generated.

In [4], Ahn *et al.* proposed an isomorphism aware instruction identification technique, which can improve the utilization of custom instructions. Their work is similar to that of Kastner *et al.*, although the instruction generation in [4] can proceed with an arbitrary fitness function. Furthermore, they use a canonical representation of graphs to perform more efficiently the isomorphism test. Their incremental template generation algorithm identifies connected sub-graphs with high utilization factor in the given DFG. This leads to the identification of one instruction at-a-time for a given DFG.

An interesting aspect of recurrence aware ISE is that, its goals are in line with those of domain-specific ISE. Anyhow, all the frameworks mentioned so far only apply the recurrence criteria for instruction identification within a DFG. Although the work of Ahn *et al.* in [4] considers recurrence across DFGs, their framework is limited to the identification of connected sub-graphs one-at-a-time.

2.3 Domain-specific ISE

Domain-specific ISE deals with identification of custom instructions, which can improve the performance of a set of applications. A naive approach towards domain-specific ISE is to separately identify custom instructions for each application and, finally, collate them into a set of domain-specific custom instructions. One can immediately notice that this approach is not scalable when the number of input applications increases. One of the major requirements in domain-specific ISE is the re-utilization of the custom instructions across applications.

In [10], Arnold *et al.* presented a semi automated method for the detection and exploitation of domain-specific ISE for VLIW processors. The authors first detect frequently occurring operations and, later, group them into custom instructions. The pattern library construction is similar to recurrence aware ISE and it is performed using incremental clustering. However, the number of operations per constructed pattern is restricted to three.

In [21], Clark *et al.* presented a framework for automatic domain-specific instruction set extension. The authors initially identify application-specific custom instructions and, later, use the techniques of wild-carding and graph subsuming to generalize the identified instructions across the applications. application-specific instructions are identified using incremental clustering. This technique of identifying application-specific instructions and later generalizing them to suit a domain seldom gives optimal results. In some cases, these over-engineered custom instructions lead to limited performance benefits at the cost of a significant die area use.

Clearly, the existing frameworks for automatic ISE cannot efficiently generate optimal domain-specific custom instructions. The framework presented in this thesis is an effort towards automatic generation of custom instructions, which can be utilized by a set of applications while still delivering the required performance gain. An objective summary of the existing frameworks investigated in this thesis along with how they differ from our framework is presented next.

2.4 Discussion

Table 2.1 shows an overview of some of the existing ISE frameworks and their corresponding instruction generation and selection techniques. The ISE approaches presented in [6, 5, 13, 11, 12, 26, 27, 38, 39, 44, 45] lead to the identification of large clusters of operations (custom instructions), which drastically increase the area if more than one application is considered at a time. The goal of the framework presented in this thesis is to improve utilization of the identified instructions, which, in turn, leads to a reduction of the overall die area and, potentially, power reduction.

Unlike generalizing application-specific custom instructions [21] to improve their utilization across applications, the framework presented here considers the reutilization of the custom instructions as a goal in the instruction identification stage. Instead of growing the custom instruction from a seed node, the instruction identification is formulated as a Maximum Common Sub-graph problem.

The isomorphism aware identification of custom instructions proposed by Ahn *et al.* in [4] can be used to identify custom instructions in multiple basic blocks. By combining given set of DFGs into one DFG, the authors use their basic instruction identification technique to identify recurring custom instructions across application, one at a time. Our methodology is rather different, as our identification algorithm simultaneously works with multiple applications and, in each iteration, identifies a set of custom instructions.

The goals of recurrence aware instruction selection in [16] are inline with the goals of the domain-specific ISE described in this thesis. While [16] presents only instruction selection, this thesis addresses the full problem of ISE and also proposes an efficient instruction generation methodology, which identifies only the most "suitable" custom instructions available for the following instruction selection. As instruction selection is, in the general case, a well known NP-complete problem, by limiting the number of candidate instructions available for selection, the framework achieves an increase in scalability and, hence, it can deal with much larger applications.

Work in data path merging [18, 35, 40] is sometimes in-line with instruction identification technique presented in this thesis. One of the main steps in data-path merging is to identify the common part that can be shared across applications. The instruction identification technique used in this thesis is partly influenced by the data path merging algorithm proposed by Moreano *et al.* in [35]. Their algorithm merges a set of input graphs into a super-graph. Unlike their algorithm, we identify the maximum common sub-graph between every pair of graphs and use these sub-graphs as potential custom instructions.

2.5 Summary

In this chapter we presented an overview of various steps involved in customization of instruction-set with a set of specialized instructions. We presented a formal definition for ISE and introduced the two steps involved in ISE: instruction generation and instruction selection. The most common approaches towards ISE are analysed with the help of state-of-the-art automatic ISE frameworks. We also presented an objective summary of some of the existing ISE frameworks and qualitatively compared them against our framework. In the next chapter, the framework for generating domain-specific custom instructions is described in more detail.

ISE Framework	Instruction Generation	Instruction Selection	Complexity	Metrics	Remarks
Ahn [4]	Recurrence aware incremental clustering with IO serialization	Iterative selection algorithm	Exponential	User defined fitness function for selection.	Uses canonical representation of graphs to more efficiently perform isomorphism during instruction generation.
Alippi [6]	MAX(E)MISO enumeration by a heuristic	Genetic Algorithms	Linear	Speed-up, number of outputs, power consumption, FPGA reconfiguration time	MAXEMISO stores state information and hence can support static loops
Alippi [5]	MaxMISO identification in a DFG	Not addressed	Linear	Output degree of the custom instruction.	Only identifies connected MAXMISOs
Arnold [10]	Dynamic pattern library construction and matching	Graph covering using dynamic programming	Exponential	Implementation cost (Area)	First work to consider recurrence across applications
Atasu [13]	Branch and bound based enumeration of convex sub-graphs	Iterative selection algorithm	Exponential (generation), Linear (selection)	I/O degree of the DFG, implementation cost (area)	Also proposed a optimal selection algorithm.

Atasu [11]	ILP based template generation	Isomorphism testing and potential evaluation function	Exponential	I/O degree of the DFG, implementation cost (area)	Runtime of the framework is less due to efficient heuristics implemented in the ILP solvers
Atasu [12]	Maximal convex sub graph enumeration by incremental clustering and selective pruning	0/1 Knapsack problem	Exponential	Schedule length of custom instruction	Proposed a tighter upper-bound on the size of maximal convex sub-graph
Bonzini [16]	Not addressed	Exact and approximate algorithms to solve recurrence aware covering	Exponential (optimal), linear (approximate)	Maximizes user defined merit function	Uses instruction generation techniques from [13]
Brisk [17]	Recurrence based sequential and parallel clustering.	No explicit selection stage	Quadratic in the number of edges	Clustering stops when a stop condition is met	Uses All Pair Common Slack Graph to perform parallel clustering. Sequential clustering is same as [31]
Choi [20]	Extended sub-set sum problem	No explicit selection stage	Linear/ exponential (depending on the subset sum solver)	Number of identified instructions and their utilization	Identifies small set of instructions which have high utilization factor that can lead to performance gain pre-defined by the user.

Clark [21]	Incremental clustering	0/1 Knapsack problem	Exponential	I/O degree of the instructions, area	Uses wildcarding and graph subsuming to generalize the application-specific instruction to suit a application domain
Cong [22]	Sub graph enumeration formulated as minimum area logic covering problem	0/1 Knapsack problem	Exponential	I/O degree of the instructions, area	Considers operation duplication when instruction generation
Galuzzi [26]	Incremental clustering of SUBMAXMISOs into MIMO instructions	No explicit selection stage	Linear	Area and output degree of custom instruction	Uses SUBMAXMISO generation algorithm from [27]
Galuzzi [27]	Partitioning of a graph into SUBMAXMISO	No explicit selection stage	Linear	Output degree of the custom instruction.	Initially partition the graph into MAXMISO clusters and later decompose these clusters into SUBMAXMISO
Kastner [31]	Recurrence based incremental clustering	Graph edge covering	Linear (generation), Quadratic (covering)	Number of templates generated (area)	Clustering results in identification of more sequential patterns. This has been extended by Brisk [17]

Peymandoust [36]	MISO generation using [5] or [13]	Select most commonly used instructions after profiling mapped DFGs	Linear/ exponential	I/O degree	Uses symbolic decomposition of application code to improve utilization of custom instructions by DFGs
Pozzi [39]	Exhaustive enumeration under relaxed I/O constrains	Select n best instructions after scheduling each instruction under I/O constrains	Exponential		Introduced I/O serialization to overcome the limitations imposed by I/O degree of register file
Verma [43]	Clique enumeration in a cluster graph	Select n best instructions after I/O serialization	Exponential	Speed-up	Formulated I/O serialization as a matrix problem
Yu [44]	A two-step algorithm based on upward and downward cones to enumerate all sub-graphs	Not addressed	Exponential	I/O degree of the instructions	The algorithm approaches the solution faster than [13], but can only identify connected sub-graphs
Yu [45]	Disjoint MIMO pattern enumeration using connected MIMO patterns	Not addressed	Exponential	I/O degree of the instructions	Extends the work in [44] to support disconnected sub graphs.

Table 2.1: Summary of ISE frameworks from various researchers.

Chapter 3

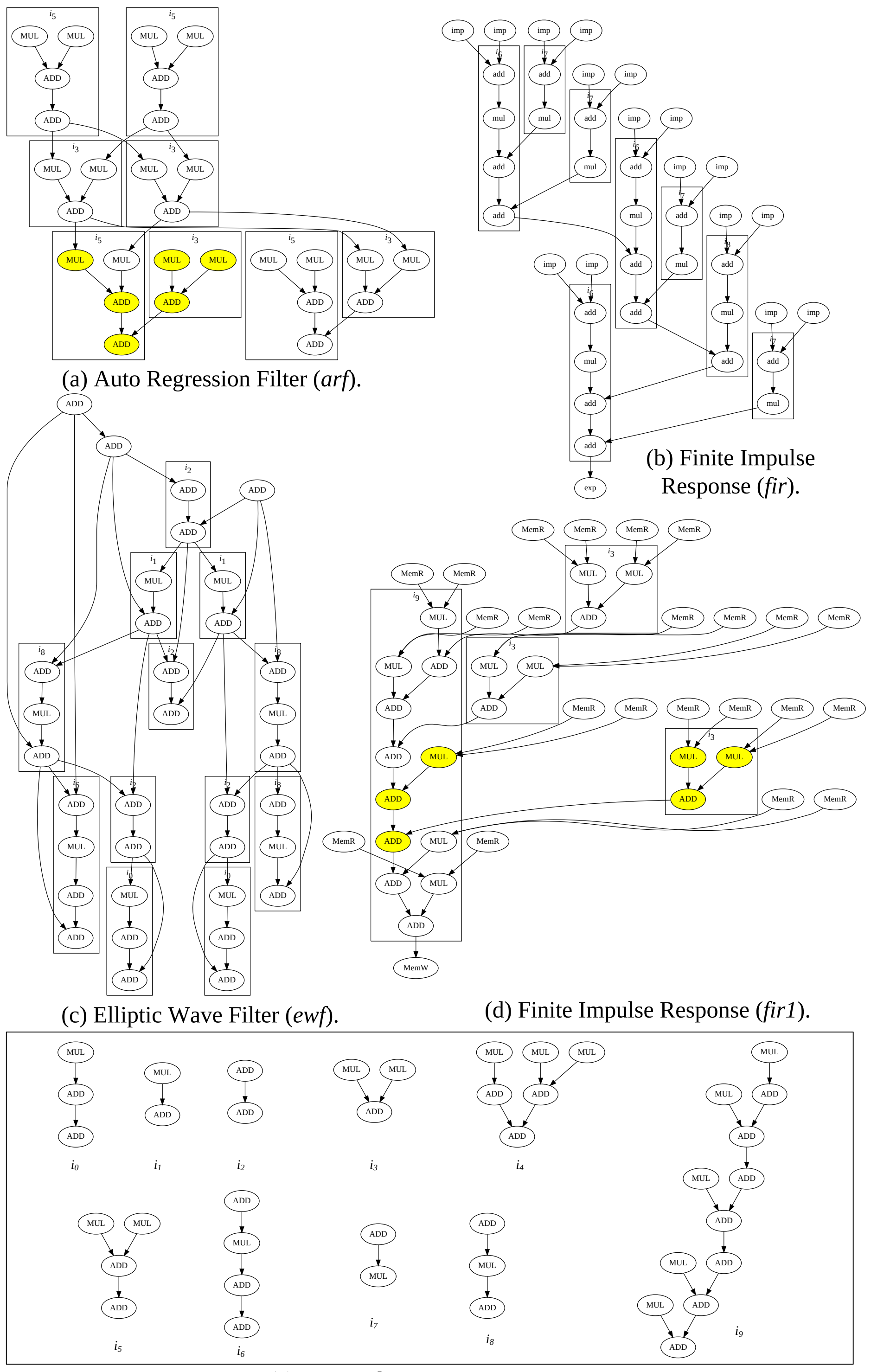
Framework for Generating Domain Specific Custom Instructions

In the previous chapter, we introduced custom instruction generation and selection and we presented some of the most common approaches for (automatic) ISE. In this chapter, we further extend our discussion and we present a framework for the automatic generation and selection of domain-specific ISE. At first; in Section 3.1, we present a motivational example to give a high level overview of the problem solved by the proposed framework. Then, the problem is formalized in Section 3.2. Finally in Section 3.3 and Section 3.4, we present the algorithms for custom instruction generation and selection used in the framework are described.

3.1 Motivational Example

In Figure 3.1, we present the dataflow graphs of the basic blocks of four different applications from [3]. Although the graphs may look very simple, they are real examples from the literature and not synthetic custom-made applications/examples. In the first stage, our framework identifies custom instructions, which are common across the applications. We consider two input DFGs at-a-time during the identification of these instructions. As a result, for the four DFGs in the figure, the instructions are identified in six steps:

STEP 1) Between *arf* - *ewf*;
STEP 2) Between *arf* - *fir*;
STEP 3) Between *arf* - *fir1*;
STEP 4) Between *ewf* - *fir*;
STEP 5) Between *ewf* - *fir1*;
STEP 6) Between *fir* - *fir1*;



(a) Auto Regression Filter (*arf*).

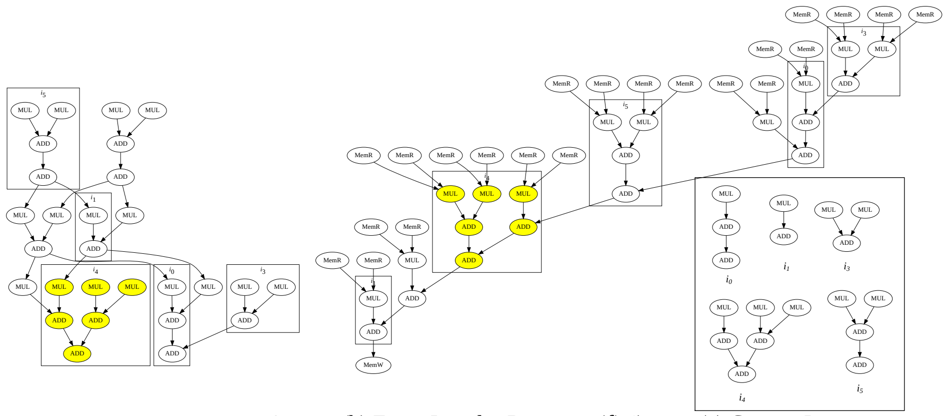
(b) Finite Impulse Response (*fir*).

(c) Elliptic Wave Filter (*ewf*).

(d) Finite Impulse Response (*fir1*).

(e) Potential custom instructions.

Figure 3.1: Motivational example. (a-d) represent four DFGs taken from the benchmark suite presented in [3]; (d) shows the custom instructions generated by our framework.



(a) Auto Regression Filter (*arf*). (b) Finite Impulse Response (*fir1*). (c) Custom Instructions.

Figure 3.2: Identification of common complex instructions across *arf* and *fir1*.

Figure 3.2 shows, as an example, the identification of instructions between *arf* and *fir1*. After the instructions are identified for each pair of graphs, they are consolidated into a set, which contains all the potential custom instructions (Figure 3.1e). If we consider the utilization of these instructions in the applications, we can notice that some of these instructions may never be utilized. This happens as each node in the DFG can be covered by multiple custom instructions. For example, when we try to cover these DFGs targeting the reduction of the dynamic instruction count, instruction i_4, which is identified as a common instruction across *arf* and *fir1* (Figure 3.2), is never utilized in both *arf* and *fir1* (Figure 3.1). It can be noticed that the nodes in *arf* and *fir1*, which correspond to instruction i_4, are covered by i_3 and i_5 in *arf*, and by i_3 and i_9 in *fir1*, respectively. Hence, the implementation of i_4 is not required. We use this as a motivation for our instruction selection stage, where we cover each DFG with custom instructions and select only those ones, which have a utilization factor better than a threshold value. As our covering aims at reducing the dynamic instruction count, *the final set of custom instructions is pareto optimal*. This means that the instruction utilization of our custom instructions can produce maximum benefit, measured as a reduction in the dynamic instruction count.

3.2 Problem Formulation

In this section, we formally describe the instruction generation and selection problem for domain-specific ISE. Unlike the general ISE formulation described in Section 2.1, domain-specific ISE should consider instruction reuse when the custom instructions are identified.

Definition 3.1. Given two graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$, the *Maximum Common Induced Convex Sub-graph* (MCICS) of G_1 and G_2, $MCICS(G_1, G_2)$, is a convex sub-graph G', such that G' is an induced sub-graph of both G_1 and G_2, and there is no other sub-graph with greater number of vertices.

In the proposed framework, the goal of instruction generation is to generate (to identify) custom instructions for which the utilization factor is maximal. These instructions, which are common across all the applications, correspond to the induced convex sub-graphs. As this formulation would lead to very small common parts, we relax our identification scheme to every pair of sub-graphs. It is important to notice that, these sub-graphs cannot contain *forbidden nodes*[1]. Therefore, for each pair of input DFGs, the instruction identification problem can be formally stated as follows.

Problem 3.1. Given two DFGs, G and H, find the maximum common induced convex sub-graphs of G and H, such that these sub-graphs do not contain any forbidden node.

The maximum common induced convex sub-graph is a disconnected graph. The connected components of this graph are the potential custom instruction candidates. After the MCICSs are identified for every pair of input DFGs, the connected components of these graphs represent a complete list of the custom instruction candidates, each of which, clearly, has utilization greater than 1.

To identify the instructions with small utilization factor, we cover each of the input DFG with the custom instructions and remove those with utilization factor smaller than a certain threshold value. The DFG covering problem, which is the crux of the instruction selection can then be stated as follows.

Problem 3.2. Given a graph G and a set of graphs $S = \{S_1, S_2, .., S_n\}$, find $S' \subseteq S$ with the smallest[2] $|S'|$ and such that the number of nodes in G not covered by elements of S' is minimal.

In the following sections, we elaborate further on both the instruction generation and the instruction selection phases.

3.3 Instruction Generation

Instruction generation corresponds to the solution of the MCICS problem (Section 3.2) for every pair of DFGs. MCICS is a variant of the Maximum Common Sub-graph (MCS) problem. MCS with a convexity constraint is equivalent to the MCICS problem. In this section, we first derive the necessary conditions to ensure convexity of the common sub-graphs and, later, we utilize these results to identify custom instructions across the applications.

[1]Memory operations cannot be part of the custom instructions. As a result, these operations are marked as forbidden (nodes).

[2]Given a set A, $|A|$ represents the number of its elements.

3.3.1 Convexity of Maximum Common Sub-graphs

A DAG G is said to be an oriented tree if the underlying undirected graph is a tree. A tree is an undirected graph in which any two vertices are connected by exactly one path.

Theorem 3.1. *Let $G(V, E)$ be a connected DAG and let $f : V \to \mathbb{N}$ be a function that determines the out degree of a given vertex. If $f(v) = 1 \ \forall v \in V$, $G(V, E)$ is an oriented tree.*

Proof. By contradiction, let us assume that G is not an oriented tree. This means that there exists at least two paths between v_i and v_j. For two paths to exist there should be a fork node and a join node. A fork node is a vertex where the two paths split. The source node of the path v_i can itself be the fork node. Without a fork node there cannot be two paths between the two vertices. The existence of the fork node contradicts $f(v) = 1, \forall v \in V$. Therefore, there cannot be two paths between any two vertices in the given graph.

□

Corollary 3.1. *Let $G'(V', E')$ be an induced sub-graph of $G(V, E)$. If, $\forall v'_i \in V', f(v'_i) = 1$, $G'(V', E')$ is a convex sub-graph.*

Proof. From Theorem 3.1, G' is an oriented tree and, therefore, there exists only one path between any pair of vertices in G'. This automatically excludes the existence of a path between two vertices of G', which contains a vertex not in G'. As a result, G' is a convex sub-graph of G. □

3.3.2 Framework for Instruction Generation

Fig. 3.3 shows the steps involved in finding the MCICS between two given graphs. In the following, we describe them in more details.

The Transformation of High Fanout Instructions

In the first step, given two graphs, each of their vertices with out degree greater than 1 is replaced with two vertices: *the operator*, with out degree of 1, and a *REP vertex*[3] (Figure 3.4). *REP* vertices are considered as forbidden nodes and they are not part of the custom instructions. All the common sub-graphs between the two input DFGs do not contain *REP* nodes and, hence, the out degree of each vertex in the common sub-graph is 1. Therefore, according to Corollary 3.1, the common sub-graphs are convex and, hence, they are potential custom instructions.

REP nodes do not correspond to any operator. They are included in the DFG to ease the generation of custom instructions. These *REP* nodes are removed from the DFGs at the end of the instruction generation stage. This does not affect the convexity of the instruction and, hence, the DFG can be scheduled using the new custom instructions.

[3]*REP* is short for Replication. These vertices have no physical meaning.

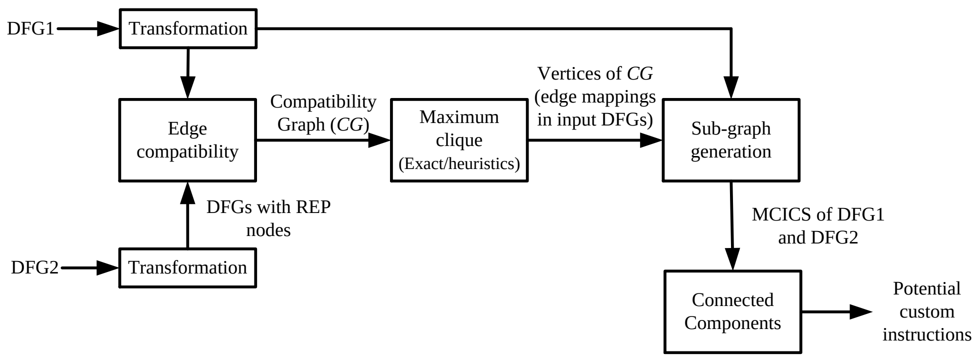


Figure 3.3: Steps involved in the generation of custom instructions for two input DFGs.

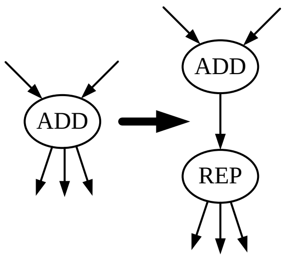
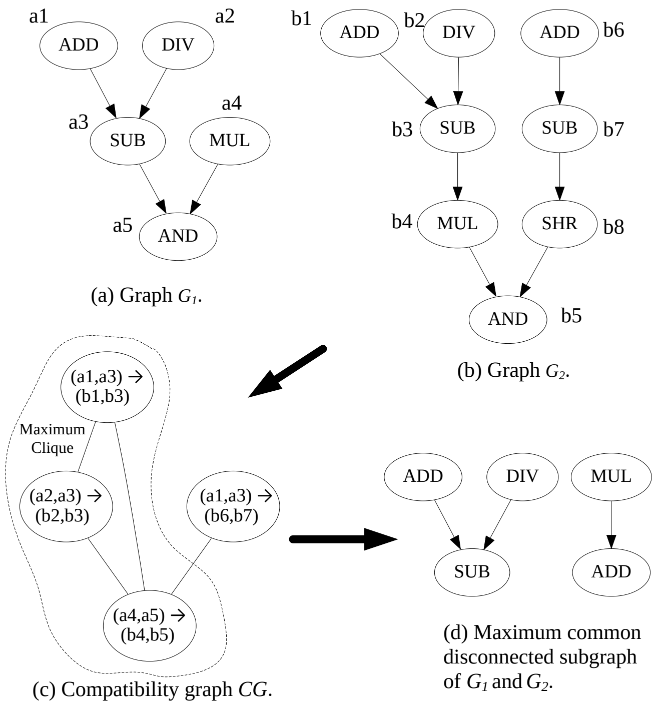


Figure 3.4: Transformation of high fanout instructions.

The Compatibility Graph

A compatibility graph of two given DFGs is an undirected graph, where the vertices represent possible edge mappings. An edge $e_1 = (u, v)$ in G_1 can be mapped to an edge $e_2 = (p, q)$ in G_2 if and only if $f(u) = f(p)$, $f(v) = f(q)$, and u, v are not forbidden nodes. The function f returns the basic operation performed at the given node. An edge exists between two vertices in a compatibility graph if both mappings represented by the two vertices are compatible.

For example, let us consider the two graphs G_1 and G_2 shown in Fig. 3.5a and Fig. 3.5b. The possible edge mappings between G_1 and G_2 includes $(a_1, a_3) \rightarrow (b_1, b_3)$, $(a_2, a_3) \rightarrow (b_2, b_3)$, $(a_1, a_3) \rightarrow (b_6, b_7)$, and $(a_4, a_5) \rightarrow (b_4, b_5)$. These mappings are the vertices of the compatibility graph CG in Figure 3.5c. Both mappings $(a_1, a_3) \rightarrow (b_1, b_3)$ and $(a_2, a_3) \rightarrow (b_2, b_3)$ can co-exist, as there is no conflict between them. As a result, they are connected by an edge in the compatibility graph. Now, let us consider the mappings $(a_1, a_3) \rightarrow (b_6, b_7)$ and $(a_2, a_3) \rightarrow (b_2, b_3)$. These two mappings cannot co-exist as vertex a_3 in G_1 can be mapped to both b_3 and b_7 in G_2. As a result, there is no edge between $(a_1, a_3) \rightarrow (b_6, b_7)$ and $(a_2, a_3) \rightarrow (b_2, b_3)$ in CG.



(a) Graph G_1.

(b) Graph G_2.

(c) Compatibility graph CG.

(d) Maximum common disconnected subgraph of G_1 and G_2.

Figure 3.5: Example for finding the MCS of two graphs G_1 and G_2.

The Maximum Clique

A clique of an undirected graph $G(V, E)$ is a subset of vertices $V' \subseteq V$, such that $(v_i, v_j) \in E \ \forall v_i, v_j \in V'$. This means that every two vertices in the subset are connected by an edge. A *maximum clique* is a clique of the largest possible size in a graph. The maximum clique problem is NP complete [30] and, over the years, researchers have proposed both exact solutions and heuristics to solve it.

The maximum clique of the compatibility graph corresponds to the maximum set of mappings that can co-exist. The maximum set of mappings corresponds to the maximum common part of the two graphs G_1 and G_2. We use a branch-and-bound based exact algorithm from [41] to solve the maximum clique problem. When the solution is not computable in a feasible time, we employ a greedy heuristic approach from [19] to solve the problem. The dotted line in Figure 3.5c highlights the maximum clique in the compatibility graph CG.

The Connected Components

After the maximum clique is identified, the mappings from the compatibility graph are projected back to the input DFGs. The common sub-graph between the two given DFGs is a disconnected graph. For example, Figure 3.5d shows the MCS between G_1 and G_2.

A connected component of an undirected graph G is defined as a sub-graph G' in which there exists a path between any pair of nodes in the sub-graph and non of those nodes is connected to nodes in $G \setminus G'$. Each connected component of the underlying undirected graph of MCS corresponds to a potential custom instruction. A Depth First Search (DFS) based algorithm with a complexity of $O(|V| + |E|)$ is used to find the connected components [28].

3.4 Instruction Selection

The goal of instruction selection is to remove unused/less-utilized custom instructions from the instructions identified in the instruction generation phase. This is done by covering each DFG with the list of instructions and, then, by removing all custom instructions, which have a utilization factor smaller than a specific threshold value defined by the user. The steps involved in the DFG covering are shown in Fig. 3.6.

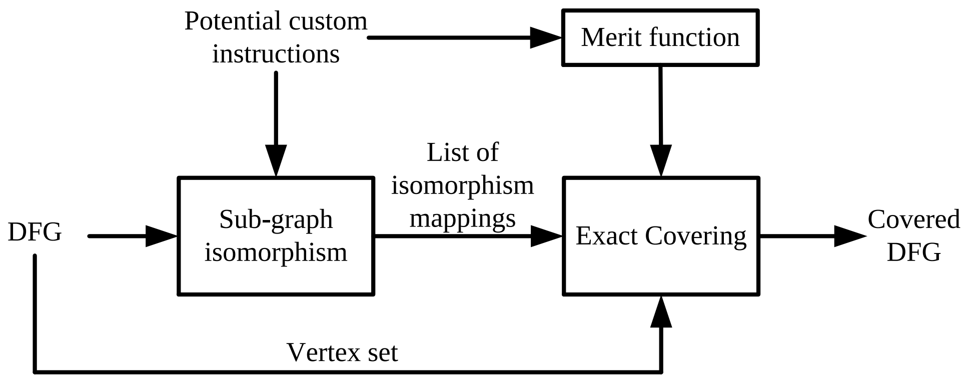


Figure 3.6: Steps in covering a DFG with a set of instructions.

The Sub-graph Isomorphism

Given two graphs $G(V, E)$ and $G'(V', E')$, an *isomorphism* between G and G' is a bijective function $f : V'(G') \to V(G)$, such that $\forall u, v \in V$, $(u, v) \in E$ if and only if $(f(u), f(v)) \in E'$. This means that f preserves the edge structure of both G and G'. f is said to be a graph-subgraph isomorphism (or just a sub-graph isomorphism) $\iff$ f is an isomorphism between G' and a sub-graph of G.

For each instruction in the custom instruction list, the isomorphism operator returns the list of vertices in the DFG, which are isomorphic to the custom instruction. It is important to note that each custom instruction can return more than one isomorphism mapping. For example, let us consider a DFG G and two instructions, i_1 and i_2, as shown in Figure 3.7. The isomorphism between G and i_1 and G and i_2, returns 8 mappings each. After the sub-graph isomorphism is performed between all the instructions and the DFG, we have a list of possible mappings. The VF2 isomorphism implementation from Boost Graph Library[2] is used to implement the isomorphism operator. Time complexity of VF2 is $O(V^2)$ in the best case and $O(V.V!)$ in the worst case.

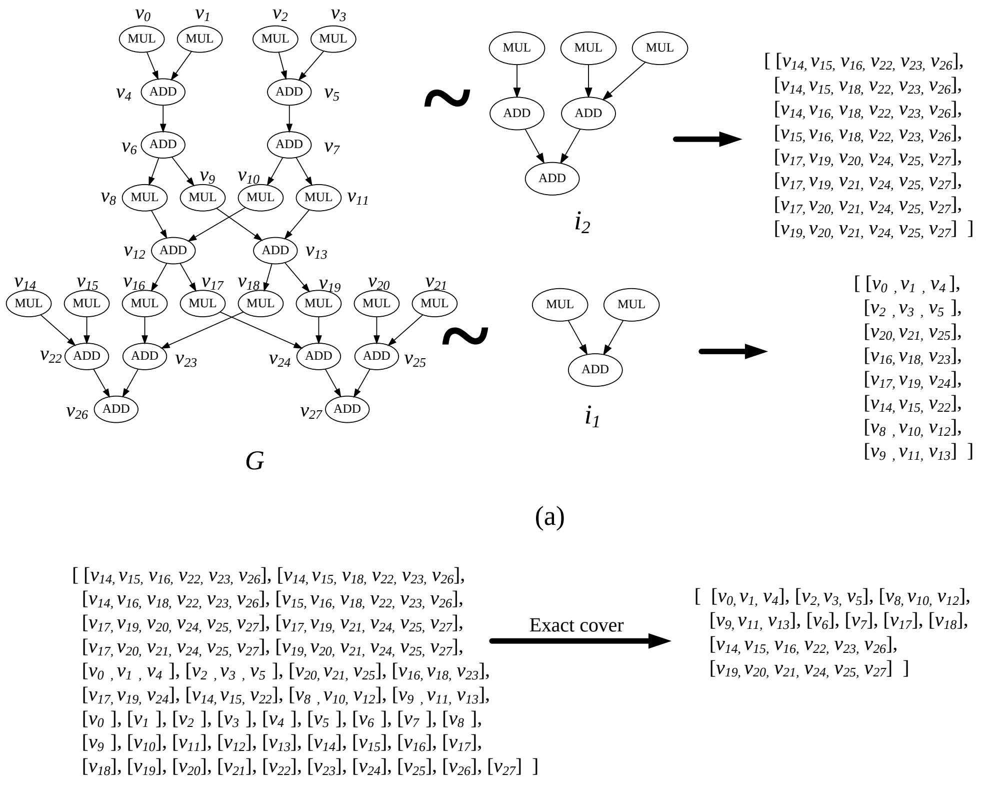

G
i2
i1
MUL
ADD
~
[[v14, v15, v16, v22, v23, v26], [v14, v15, v18, v22, v23, v26], [v14, v16, v18, v22, v23, v26], [v15, v16, v18, v22, v23, v26], [v17, v19, v20, v24, v25, v27], [v17, v19, v21, v24, v25, v27], [v17, v20, v21, v24, v25, v27], [v19, v20, v21, v24, v25, v27]]
[[v0, v1, v4], [v2, v3, v5], [v20, v21, v25], [v16, v18, v23], [v17, v19, v24], [v14, v15, v22], [v8, v10, v12], [v9, v11, v13]]
(a)
[[v14, v15, v16, v22, v23, v26], [v14, v15, v18, v22, v23, v26], [v14, v16, v18, v22, v23, v26], [v15, v16, v18, v22, v23, v26], [v17, v19, v20, v24, v25, v27], [v17, v19, v21, v24, v25, v27], [v17, v20, v21, v24, v25, v27], [v19, v20, v21, v24, v25, v27], [v0, v1, v4], [v2, v3, v5], [v20, v21, v25], [v16, v18, v23], [v17, v19, v24], [v14, v15, v22], [v8, v10, v12], [v9, v11, v13], [v0], [v1], [v2], [v3], [v4], [v5], [v6], [v7], [v8], [v9], [v10], [v11], [v12], [v13], [v14], [v15], [v16], [v17], [v18], [v19], [v20], [v21], [v22], [v23], [v24], [v25], [v26], [v27]]
Exact cover
[[v0, v1, v4], [v2, v3, v5], [v8, v10, v12], [v9, v11, v13], [v6], [v7], [v17], [v18], [v14, v15, v16, v22, v23, v26], [v19, v20, v21, v24, v25, v27]]
(b)

Figure 3.7: (a) Sub-graph isomorphism between G and i_1, and G and i_2 generates 8 mappings each. (b) The mappings generated in (a) are appended with the vertex set of G before finding the exact cover.

The Exact Covering

Given a set X and a collection S of its subsets, an *exact cover* of X is a sub-collection $S^* \subseteq S$, such that each element in X is contained in exactly one element of S^*.

To the list of mappings identified in the previous step, we append the vertex set of the DFG (see Figure 3.7b). Now, clearly, the graph covering problem can be solved as an exact covering problem. An exact cover for a set need not be unique. Whenever more than one exact cover is identified, the one which maximize the benefit function is preferred. The benefit function for a cover S^* is given by Equation 3.1.

$$M(S^*) = \sum_{n=0}^{|S^*|} m(n) \qquad (3.1)$$

where, $m(n)$ is the merit value of the instruction corresponding to mapping n.

The merit value of each instruction can be based on the parameter(s) that needs to be optimized during the covering. This can include, for example, the number of instruction savings produced by the custom instruction. More details on the merit function are given in Chapter 4. The exact cover problem is solved by using the Dancing Links implementation of AlgorithmX [32].

3.5 Summary

This chapter presented the framework for the generation and selection of domain-specific custom instructions. Instruction generation is formulated as the maximum common sub-graph problem and it is solved by identifying maximum clique in the compatibility graph. The identified custom instructions are convex by construction. Instruction selection is formulated as a graph covering problem and it is solved using sub-graph isomorphism and exact covering.

In the next chapter, the experimental validation of the framework is presented. We use the framework proposed in this chapter to identify custom instructions for various benchmarks taken from three application domains.

Chapter 4

Evaluation of the Framework

In this chapter we evaluate the framework presented in Chapter 3 with the help of benchmarks from three application domains. The chapter begins with a detailed description of the experimental set up used to evaluate the framework. In the experimental set up, we present the assumptions behind the base processor and later give the details of the benchmarks used for experiments. In addition to this, we elaborate on the metrics, and the corresponding merit functions, used to evaluate the framework. Towards the end of the chapter, experimental results for different benchmarks are discussed and, later, a summary of the conclusions is presented.

4.1 Experimental Set-up

The framework described in Chapter 3 is not restricted to any kind of architectures or sets of applications. As a matter of fact, the framework can be used for generating custom instructions for different processor models. However, in order to effectively evaluate the proposed methodology, we have to make some assumptions about the base processor described in the following. Later on, we present a set of benchmarks and the metrics used to evaluate our framework and we show how performance as a metric, is not the most suitable metric to evaluate ISE methodologies. As a result we propose a more efficient metric based on the dynamic instruction count. Finally, we present an overview of the implementation details of our framework.

4.1.1 Base Processor

In order to evaluate the effectiveness of our custom instructions, we assume an single-issue inorder processor model. The custom instructions are implemented in hardware as Custom Functional Units (CFUs). Figure 4.1 shows an example of how a CFU can be integrated into a base processor's execution stage. We would like to emphasize that, the scope of this thesis is limited to the identification of the custom instructions. Their implementation as CFUs is not discussed in this thesis.

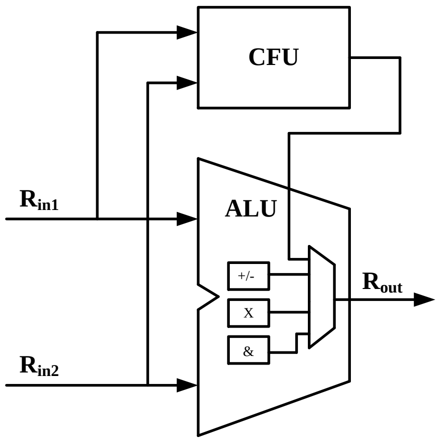


Figure 4.1: Execution stage of a processor with the custom instructions identified by our framework.

4.1.2 Input data set

The input of our framework is a set of DFGs derived from various well-known applications. More specifically, we tested our framework with seventeen DFG benchmarks from [3]. These benchmarks are further divided into three application domains and custom instructions are generated for each domain. Furthermore, we tested our framework by considering, at the same time, all applications from the different domains.

Table 4.1 shows the details of the benchmark DFGs used in the experiments. The selected DFGs (benchmarks) have very diverse topologies to consider different kinds of applications. While some of these show a very high degree of parallelism, others are more sequential by nature. By using a diverse set of DFGs to test our framework we intend to show that the proposed algorithms are not dependent on the topological structure of the graph and, hence, can be effectively utilized for generating custom instructions for any set of applications.

4.1.3 Metrics for evaluation

Most of the ISE frameworks presented in the literature [13, 4, 39, 43, 16, 31, 17] use performance and area as a metric to evaluate the quality of the generated custom instructions. The performance gain for an application can be computed using Equation 4.1.

$$G_p = \frac{(t_{original} - t_{custom})}{t_{original}}, \tag{4.1}$$

Table 4.1: Target applications from [3] and the corresponding numbers of nodes/operations.

Application	$\|V\|$	**Application**	$\|V\|$	**Application**	$\|V\|$
EPIC	56	BMP Header	106	Matrix inversion	333
FIR	44	Smooth Downsample	51	Smooth Triangle	197
FIR1	40	Forward DCT	134	Horner Bezier	18
EWF	34	JPEG: Inverse DCT	122	Interpolate Aux	108
ARF	28	MPEG: Inverse DCT	114	Matrix Multiplication	109
		Motion Vectors	32	Feedback Points	53

where G_p is the performance gain and t_{custom} and $t_{orignial}$ are the execution times of the application with and without custom instructions, respectively.

The execution time of an application on an in-order processor is rather simple to measure. Most of the current operating systems provide APIs to precisely measure the execution time of an application. In order to measure the execution time of an application, which includes custom instructions, these instructions have to be implemented and integrated into a processor. This involves a considerable amount of design time and efforts. Therefore, most of the ISE frameworks use a simplified version of Equation 4.1, represented by Equation 4.2 to measure the performance gain.

$$G_p \quad = \quad \frac{(l_{original} - l_{custom})}{l_{original}}, \tag{4.2}$$

where l_{custom} and $l_{orignial}$ are the execution latencies of the DFG under consideration, respectively.

The execution latency is the number of processor cycles that are required for executing all the operations in the DFG. This simplification is based on two assumptions:

1 the memory access latencies, which are non-deterministic (or rather had to be modelled) in a processor system with memory hierarchy, are the same with and without custom instructions;

2 the DFG corresponds to the hot-spots in the application, the parts of the application where most of the execution time is spent.

Let us assume c is the processor cycle time, l is number of processor cycles taken by the application, and t is the application execution time. Then, t, l, and c are dependent as shown in Equation 4.3:

$$t \quad = \quad l \times c. \tag{4.3}$$

Let l_{dfg} and l_{rest} be the number of processor cycles spent for the execution of the instructions in the DFG under consideration, and for the rest of the application respectively. Then:

$$l \quad = \quad l_{dfg} + l_{rest}. \tag{4.4}$$

By the second assumption, we know that $l_{dfg} >> l_{rest}$. By ignoring l_{rest} in Equation 4.4, we have

$$t \quad = \quad l_{dfg} \times c. \tag{4.5}$$

From Equation 4.5 and Equation 4.1, we can conclude that, under the two assumptions mentioned above, Equation 4.2 provides a good approximation of the performance gain.

For an in-order single issue processor, the latency of the DFG is the sum of the latencies of each of its nodes (instructions). While latencies of simple instructions are available from the processor data sheet, the latency of the complex custom instruction is not directly available. To get the latency of the custom instructions, these should be implemented (synthesized) using the same technology libraries as the base processor.

As a result, it is clear that, by using performance as a metric for the evaluation of the custom instructions generated by an automatic ISE framework the results are highly dependent on their implementation details. An effective evaluation of ISE frameworks should not depend on the architecture of the baseline processor and the implementation of the custom instructions. Anyhow, the definition of such evaluation metric is beyond the scope of this thesis.

To provide an architecture and technology independent evaluation of the framework, we use the dynamic instruction count as the evaluation metric. Dynamic instruction count is the run-time count of the number of instructions executed by an application. Assuming that the DFGs are derived from the application hot-spots, the dynamic instruction count is proportional to the number of vertices in the DFG. The reduction in the dynamic instruction count is, therefore, given by Equation 4.6:

$$G_i \quad = \quad \frac{(|N_o| - |N_c|)}{|N_o|}, \tag{4.6}$$

where G_i is the reduction in the dynamic instruction count, $|N_o|$ and $|N_c|$ are the number of vertices in the DFG, with and without custom instructions, respectively.

We cannot quantitatively measure the performance, area and power benefit of our custom instructions, without implementing them. However, the reduction in the dynamic instruction count gives a qualitative indication of how these high level metrics may vary:

- a reduction in the dynamic instruction count can potentially lead to a shorter execution time of the application and, therefore, improves the performance of the system;

- a reduced execution time directly correlates to lower energy consumption;
- a reduction in the size of DFG with custom instructions reduces, in general, instruction memory requirements.

4.1.4 Merit Functions

For a given instruction I, the merit function is a mapping $m : I \to \mathbb{R}$, which assigns a weight to instruction I. As mentioned in Section 3.4, a merit function is used by the instruction selection phase of the framework to perform graph covering. It is noteworthy that, the instruction I can be a simple ALU operations as well as a custom instruction identified by our framework. Based on the optimization goal, various merit functions can be defined. In this section, we present two examples of the merit functions. The first one tries to minimize the dynamic instruction count whereas the second one is based on performance gain.

The merit function guides our framework to produce custom instructions which, when implemented, can optimize a particular metric. Although a merit function based on performance gain is intended to improve the performance, this cannot be verified without implementing the custom instructions in hardware. As the implementation of the custom instructions is not discussed in this thesis, we did not perform any experiment with merit functions based on performance gain.

Dynamic Instruction Count

A domain-specific custom instruction is a cluster of simple ALU operations. Let I be an instruction and $V(I)$ be the number of simple ALU operations performed by the instruction. The merit function, which corresponds to the savings in the dynamic instruction count, is given by Equation 4.7:

$$m(I) \quad = \quad V(I) - 1. \tag{4.7}$$

For simple ALU operations, $V(I) = 1$ and, therefore, the merit value of using simple ALU operation in covering the DFG is 0. This means that, the utilization of only simple ALU operations during the graph covering will result in no reduction in the dynamic instruction count, as expected. On the contrary, if I is a custom instruction, $m(I) > 0$. A bigger cluster has a higher value of $m(I)$ and, therefore, it leads to a greater reduction in the dynamic instruction count. Intuitively, $m(I)$ is the instruction savings that an instruction I can produce. The instruction savings are directly proportional to the size of the custom instruction (cluster). It is interesting to observe that, the use of Equation 4.7 as the merit function results in the selection of bigger custom instruction over the smaller ones.

As mentioned before, a reduction in the dynamic instruction count is less dependent on the base-processor model and has no dependency on the implementation details of the custom instructions. As a result, the merit function in Equation 4.7 will be used in the rest of this thesis to perform the experiments.

Performance Gain

The merit value of an instruction I, which can lead to a performance gain, is equal to its latency. For simple ALU operations, the latency is specified in the ISA. In this section, we present a simple technique to estimate the latency of the custom instructions generated by our framework.

The latency of a custom instruction is the critical path of the custom instruction divided by the cycle time of the processor under consideration. Apart from the critical path, the latency is also dependent on the I/O degree of the custom instruction. If the I/O degree[1] of the custom instruction is greater than the I/O degree[2] of the register file of the base processor, the reads/writes have to be serialized. In such cases, we perform an ASAP scheduling of the custom instructions under I/O constraints and, later, measure the critical path.

Latency for custom instructions is estimated using Algorithm 1. Each instruction $I(V, E)$ is associated to a graph $G'(V \cup V^+, E \cup E^+)$, which contains additional nodes V^+ and edges E^+ (see Figure 4.2). The additional nodes represent the input and output variables of the instruction. The additional edges E^+ connect nodes V^+ to V, and nodes V to V^+. The scheduling problem can now be addressed as an assignment of each operation in G' to a time slot corresponding to a clock cycle, such that the schedule uses at most N_{in} and N_{out}, *IN* and *OUT* operations in each time slot, respectively. N_{in} and N_{out} are the number of read and write ports of the register file in the base processor. After the time slots are assigned, the critical path in the scheduled graph is determined.

From the discussion in Section 4.1.3 it is clear that the performance improvement due to the use of custom instructions is affected by parameters, such as the Instruction Set Architecture (ISA) and the memory architecture of the base processor and the implementation details of the custom instructions. Although the merit function described above tunes the graph covering to optimize performance, it is not possible to measure the performance benefit resulting from the custom instructions. Therefore, we refrain from using this merit function in our experiments.

4.1.5 Implementation details

The proposed framework is implemented as a dedicated tool chain and it can work with any given set of DFGs. We make extensive use of Boost Graph Library [2] (BGL), a header only generic library, to store the graph structures and implement graph algorithms. The use of BGL makes our implementation highly extensible and, thus, it can be easily tuned to suit different optimization goals with very few changes. Unlike the earlier ISE frameworks, which are usually built over Trimaran [1] and LLVM [33] infrastructures, our tool chain is not bound to a particular compiler infrastructure. Thus, it can be used in any proprietary HW/SW co-design

[1]I/O degree of an instruction is the number of input required and number of outputs generated by a custom instruction.

[2]I/O degree of a register file is equal to the number of its write ports and read ports.

Algorithm 1 Resource constrained ASAP scheduling algorithm

```
1: procedure SCHEDULE($I(V, E), N_{in}, N_{out}$)    ▷ Returns the latency of instruction I
2:     $G'(V', E') \leftarrow transform(I(V, E))$
3:     $s \leftarrow 0$    ▷ Number of scheduled nodes
4:     $t \leftarrow 0$    ▷ Time unit
5:     while $s \neq |V \cup V'|$ do
6:         $U \leftarrow ready\_nodes(G')$
7:         $S_{in} \leftarrow S_{out} \leftarrow 0$
8:         for all $v$ in $U$ do
9:             if $v$ is IN vertex && $S_{in} \leq N_{in}$ then
10:                $T[v] \leftarrow t$
11:                $s \leftarrow s + 1$
12:                $S_{in} \leftarrow S_{in} + 1$
13:            else if $v$ is OUT vertex && $S_{out} \leq N_{out}$ then
14:                $T[v] \leftarrow t$
15:                $s \leftarrow s + 1$
16:                $S_{out} \leftarrow S_{out} + 1$
17:            else if $v$ is operation then
18:                $T[v] \leftarrow t$
19:                $s \leftarrow s + 1$
20:            else
21:                continue
22:        $t \leftarrow t + 1$
23:    $l \leftarrow critical\_path(G', T)$
24:    return $l$    ▷ $l$ is the latency of $I$
```

tools.

4.2 Results

In this section, we present the results concerning the reduction in the dynamic instruction count of the benchmarks described in Section 4.1.2. Figure 4.4 shows the normalized dynamic instruction count for different application domains. On average, we observe a 45% reduction in the dynamic instruction count, when the custom instructions identified by our framework are utilized. This reduction is higher, if the application can utilize greater number of custom instructions. For instance, a 72% reduction can be observed in *arf*. In this case, the entire DFG is covered by just 8 complex instructions. In most of the cases, the input DFG is completely covered by custom instructions (if not, as mentioned before, basic ALU operations are added to the list of operations.). For some applications, such as *fdct* and *idct*, the reduction is only 20%. This happens because of the higher number of

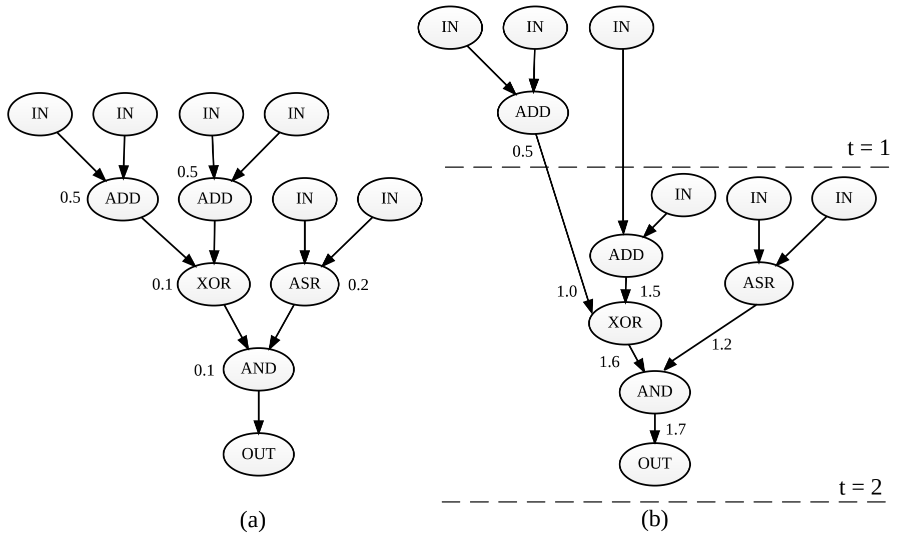


Figure 4.2: (a) Instruction with *IN* and *OUT* vertices. The normalized values of the critical path of each instruction are only indicative values. (b) Scheduled graph assuming 3 read ports and 1 write port for the register file.

memory instructions in the two applications. Memory instructions are marked as forbidden nodes and, therefore, they are not covered by custom instructions.

To improve the utilization of instructions, we may remove the instructions with limited utilization factor (below a certain threshold value defined by the user) and repeat the covering process with the new set of custom instructions. When this threshold is set to 2, we observe only a slight increase in the dynamic instruction count (see Figure 4.4). This small change is an indicator that most of our custom instructions have a very good utilization factor.

Figure 4.5 shows the dynamic instruction count, when all applications are considered simultaneously in the custom instruction generation process. Figure 4.6 shows the percentage in reduction of the dynamic instruction count for various experimental scenarios. It is interesting to see that the reduction is almost similar when applications are either considered domain wise or simultaneously. This means that the quality of custom instructions generated by our framework is the same irrespective of the number of applications under consideration, which makes our approach application independent.

The number of custom instructions identified for each of our experiments is shown in Table 4.2. On average, our framework identified 12 instructions for each domain. This low value ensures that ISE does not bolt up the op-code space. By setting a higher threshold (2), the number of custom instructions for hardware implementation is reduced, on average, by 30%, which, in-turn, can result in additional area and power savings.

Table 4.2: Number of Custom Instructions per application domain.

Application domain	Threshold = 1	Threshold = 2	% reduction
Filters	10	6	40
Image Compression	13	10	23.1
Matrix Operations	14	10	28.6

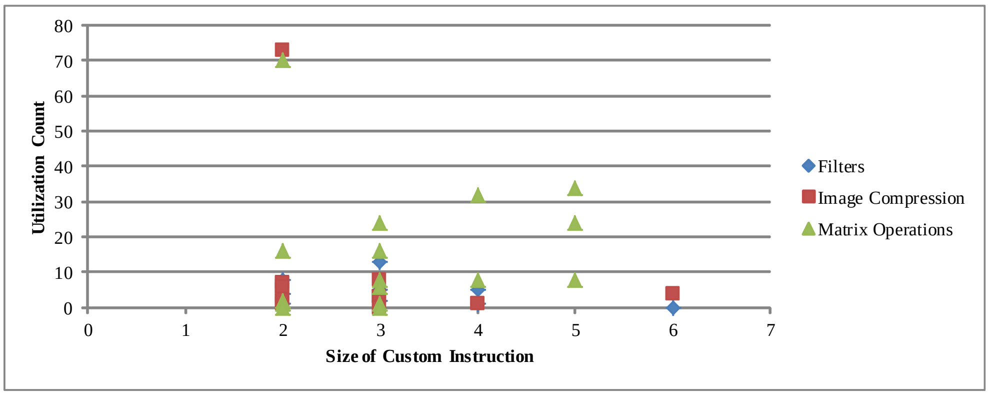
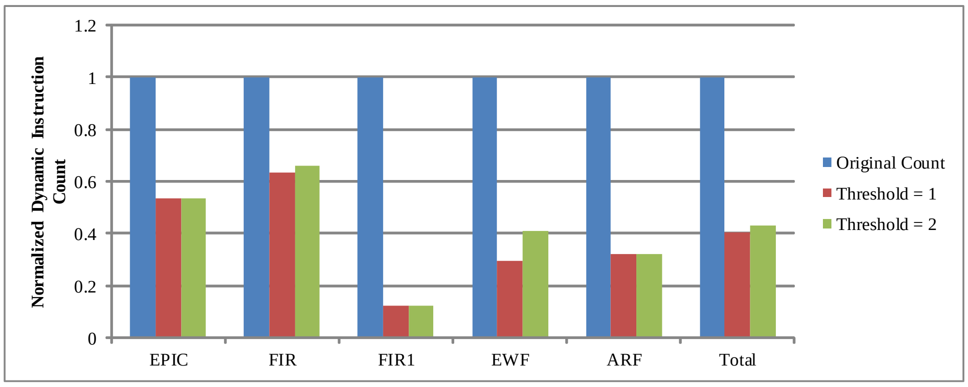


Figure 4.3: Utilization of custom instructions of various sizes.

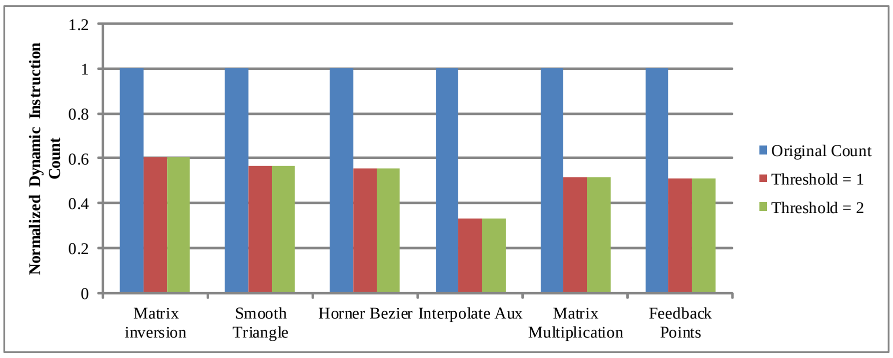
Figure 4.3 shows the distribution of the custom instruction utilization with its size. As expected, smaller instructions have better utilization over larger instructions. The two instructions with very high utilization factors are the MAC operations which is a very common operation in both image compression and matrix operations. This provides some empirical evidence that our framework is making intelligent decisions. While some custom instructions, such as MAC, are trivial to be manually identified, others, such as the ones shown in Figure 3.1d, are too unusual for manual identification.

4.3 Summary

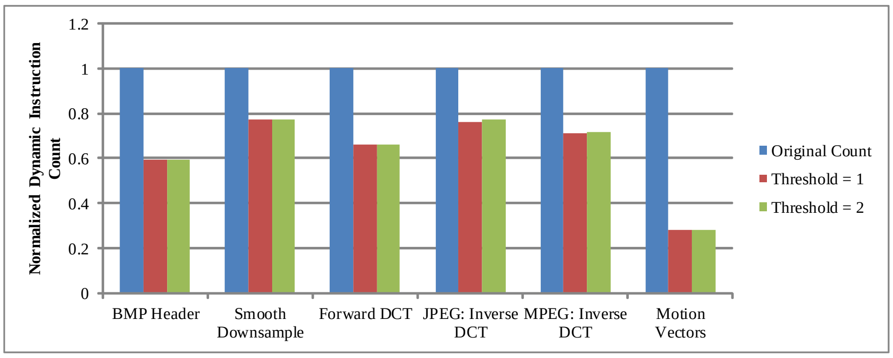
In this chapter, we evaluated the framework proposed in this thesis. We described various aspects of the experimental set-up such as the base processor model, the benchmarks, the metrics for evaluation, the merit functions and implementation details. Additionally, we showed that the conventional performance evaluation metrics cannot be used for an effective evaluation of the ISE frameworks without actually implementing the custom instructions. Experimental results are discussed in detail. We have shown that by using, on average, 12 custom instructions, we can reduce the dynamic instruction count of every application domain by 45%. In the next chapter, we provide concluding remarks and propose future research directions.



(a) Filters



(b) Matrix Operations.



(c) Image Compression.

Figure 4.4: Normalized dynamic instruction count for various application domains.

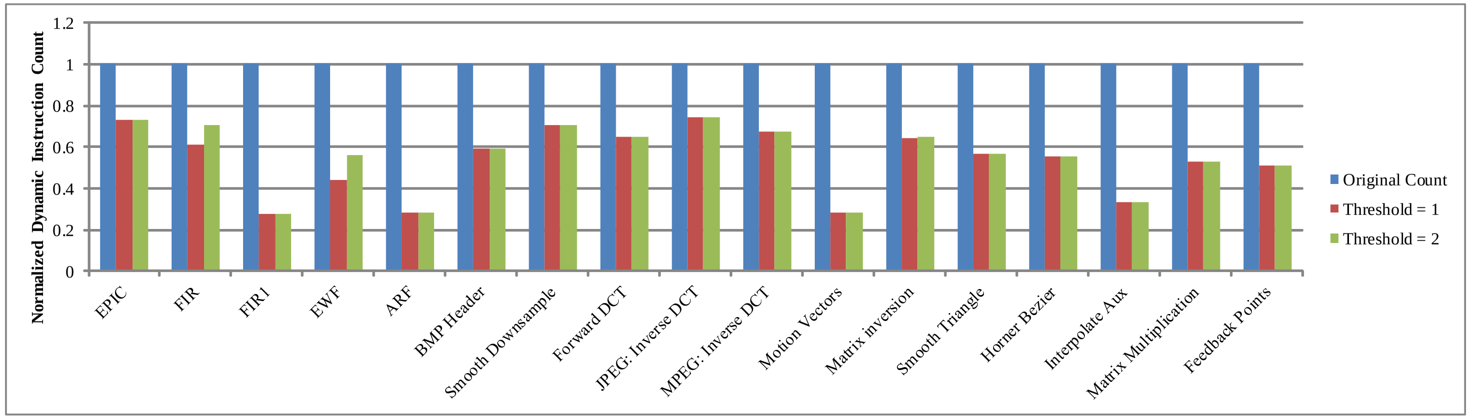


Figure 4.5: Dynamic instruction count when all the applications are given as input to our framework at the same time.

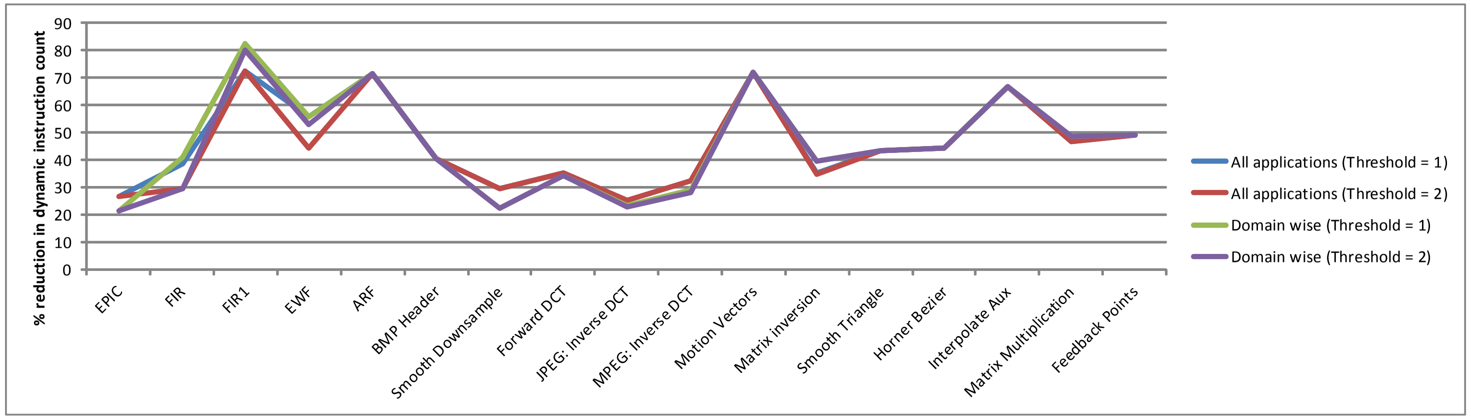


Figure 4.6: Reduction in dynamic instruction count for various benchmarks from [3].

Chapter 5

Conclusions

In this thesis, we presented a framework for the automatic generation of domain-specific custom instructions, which can be utilized in various computing platforms, to improve the dynamic instruction count of the applications. The problem is divided into instruction generation and instruction selection. In the former, we identify instructions, which can potentially have high utilization across different applications. Later, custom instructions for hardware implementation are selected in such a way that, the dynamic instruction count is reduced. In our experiments, we considered various applications from different domains and we have shown that dynamic instruction count can be reduced 45%, on average, and upto 80% in specific cases. In this chapter we summarize our contributions and list some future research directions that can be pursued based on the our framework.

5.1 Contributions

The main contributions of this thesis can be summarized as follows.

1 **A processor and technology independent automatic ISE framework.** The ISE framework presented in this thesis does not make any assumptions or impose any restrictions on the architecture of the base processor. In addition to this, all the steps in the framework are fully automated and requires no manual intervention;

2 **Instruction generation as the Maximum Common Sub-graph problem.** The instruction generation is formulated as a MCS problem and is solved by detecting maximum clique in the compatibility graph;

3 **Instruction selection as a graph covering problem.** Instruction selection is driven by a merit function which optimizes a metric. The graph covering is solved using sub-graph isomorphism and exact covering. After choosing a cover that produces highest benefit, the instructions with utilization factor greater than a threshold are selected for hardware implementation;

4 **Convexity of the sub-graphs.** The sub-graphs identified by our framework are convex and therefore can be utilized by applications without any scheduling[1] problems. As elaborated in Section 3.3.1, the convexity of the sub-graphs is theoretically guaranteed;

5 **Experimentation and validation.** We validated the framework presented in this thesis using 17 benchmarks from three different domains. Our experimental evaluation showed that the custom instructions generated by our framework can reduce the dynamic instruction count on average by 45%.

5.2 Future Work

In this section we present some enhancements which can be taken up in future to improve/extend our framework. We also propose a research direction can make use of the algorithms presented in this thesis.

MIMO instructions

The instruction generation stage of our framework is restricted to enumeration of MISO instructions. This can be extended to support identification of MIMO instructions. The granularity (size of a cluster) of MIMO instructions is higher compared to MISO instructions [24]. Therefore, maximal common MIMO sub-graphs across a pair of applications may lead to further reduction in the dynamic instruction count.

In the current framework, we impose an out-degree restriction of 1 for each node that can be part of a custom instruction. For the generation of MIMO instructions, this restriction should be removed. When the restriction is removed, the MCS of two input graphs is not necessarily convex. To identify MIMO instructions, convexity has to be considered during the generation of the compatibility graph. Apart from checking the compatibility of two edge mappings, the construction process should also check if the convexity constraints are honoured if both mappings under consideration are part of MCS. Checking for convexity involves enumeration of all paths between every pair of nodes in the mapping. Enumeration of all paths between any two vertices in a graph is known to be NP hard. Solutions to this problem have exponential time complexities and incorporating them into our framework would drastically increase the execution time by a great extent. Further investigation is required to solve the problem of identifying overlapping MIMO instructions.

[1]Non-convex sub-graphs result in presence of loops in the DFG which in-turn makes scheduling impractical

Processor Customization

As mentioned in Chapter 1, the scope of this thesis is restricted to ISE problem during processor customization. In future, other steps in processor customization, such as DFG extraction, instruction synthesis, processor integration may be explored in greater detail. Each of these steps pose interesting research questions such as:

- Can we transform control dependencies in a program to data dependencies? This enables us to work with a larger part of an application and therefore would result in better speed-up. This problem has been addressed for Tagged Token Data Flow Architectures in [14].

- Resource sharing across instructions in custom instruction synthesis is another interesting aspect. Many researchers have proposed techniques for custom instruction synthesis [18, 46].

- What is an efficient way to integrate domain-specific custom instructions to different processor architectures? Should these custom instructions have direct access to the register file of the base processor? How is state shared between execution unit of the processor and the CFU?

- Does the computation model of the base processor have any influence on the benefit derived from custom instructions?

Domain Specific Coarse Grained Reconfigurable Arrays

A coarse grained reconfigurable array (CGRA) is a aggregation of Functional Units (FUs) which are connected by an interconnect. Compared to FPGAs, CGRAs are known to have higher performance, lower power consumption, lesser configuration overheads and ease of mapping applications [37]. Traditionally, FUs in a CGRA are simple ALU operations. Mapping of applications on a CGRA is equivalent to mapping each node in the DFG of an input application to a FU in the CGRA. Increasing the granularity of the FUs reduce the communication latency between them and therefore increase the performance of the CGRA. The granularity of the FUs can be increased by replacing the simple ALU operations with clusters of ALU operations. If a CGRA is domain-specific, FUs have to be utilized by multiple applications in the domain.

The custom instructions generated by our framework are clusters of ALU operations. These custom instructions have high utilization factor across applications and therefore, are good FU candidates in a domain-specific CGRA. While automatic generation of CGRA is a completely different research question, our framework can provide a solution for its FU identification problem.

Bibliography

[1] An Infrastructure for Research in Backend Compilation and Architecture Exploration. `http://trimaran.org/`. Accessed: 2014-05-02.

[2] Boost Graph Library. `http://www.boost.org/doc/libs/1_55_0/libs/graph/doc/index.html`. Accessed: 2014-02-01.

[3] DFG Benchmarks. `http://express.ece.ucsb.edu/benchmark/`. Accessed: 2014-02-01.

[4] Junwhan Ahn and Kiyoung Choi. Isomorphism-aware identification of custom instructions with i/o serialization. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 32, pages 34–46, Jan 2013.

[5] 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, 1999.

[6] Cesare Alippi, William Fornaciari, Laura Pozzi, and Mariagiovanna Sami. Determining the optimum extended instruction-set architecture for application specific reconfigurable vliw cpus. In *12th International Workshop on Rapid System Prototyping*, pages 50–56, 2001.

[7] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization.*, pages 1–19. ACM, 1970.

[8] A. Alomary, T. Nakata, Y. Honma, J. Sato, N. Hikichi, and M. Imai. Peas-i: A hardware/software co-design system for asips. In *Proceedings of Design Automation Conference, 1993.*, DAC '93, pages 2–7. IEEE, Sep 1993.

[9] AMD. Heterogeneous Computing. `http://developer.amd.com/resources/heterogeneous-computing/`. Accessed: 2014-05-10.

[10] M. Amold and H. Corporaal. Designing domain-specific processors. In *Proceedings of the Ninth International Symposium on Hardware/Software Codesign, 2001.*, CODES '01. ACM, 2001.

[11] K. Atasu, C. Ozturan, and G. Dundar. An integer linear programming approach for identifying instruction-set extensions. In *Third IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, 2005.*, CODES+ISSS '05, pages 172–177, Sept 2005.

[12] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proceedings of Design Automation Conference, 2003.*, pages 256–261, June 2003.

[13] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *International Conference on Application-Specific Systems, Architectures and Processors, 2008.*, ASAP 2008, pages 1–6, July 2008.

[14] Micah Beck, Richard Johnson, and Keshav Pingali. From control flow to dataflow. *J. Parallel Distrib. Comput.*, 12(2):118–129, June 1991.

[15] P. Biswas, N. Dutt, P. Ienne, and L. Pozzi. Automatic identification of application-specific functional units with architecturally visible storage. In *Proceedings of Design, Automation and Test in Europe, 2006.*, DATE '06, pages 1–6. IEEE, March 2006.

[16] P. Bonzini and L. Pozzi. Recurrence-aware instruction set selection for extensible embedded processors. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, volume 16, pages 1259–1267, Oct 2008.

[17] Philip Brisk, Adam Kaplan, Ryan Kastner, and Majid Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems.*, CASES '02, pages 262–269. ACM, 2002.

[18] Philip Brisk, Adam Kaplan, and Majid Sarrafzadeh. Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. In *Proceedings of Design Automation Conference, 2004.*, pages 395–400, June 2004.

[19] M. Brockington and J. C. Culberson. Camouflaging Independent Sets in Quasi-Random Graphs. In *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, pages 75–88. AMS, 1994.

[20] 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. In *IEEE Transactions on Computers*, volume 48, pages 603–614, Jun 1999.

[21] N.T. Clark, Hongtao Zhong, and S.A. Mahlke. Automated custom instruction generation for domain-specific processor acceleration. In *IEEE Transactions on Computers*, volume 54, pages 1258–1270, Oct 2005.

[22] 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. ACM, 2004.

[23] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[24] Carlo Galuzzi. *Automatically Fused Instructions*. PhD thesis, Delft University of Technology, Delft, The Netherlands, May 2009.

[25] Carlo Galuzzi and Koen Bertels. The instruction-set extension problem: A survey. In Roger Woods, Katherine Compton, Christos Bouganis, and PedroC. Diniz, editors, *Reconfigurable Computing: Architectures, Tools and Applications*, volume 4943 of *Lecture Notes in Computer Science*, pages 209–220. Springer Berlin Heidelberg, 2008.

[26] 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, March 2007.

[27] Carlo Galuzzi, Koen Bertels, and Stamatis Vassiliadis. A linear complexity algorithm for the generation of multiple input single output instructions of variable size. In *SAMOS*, pages 283–293, July 2007.

[28] John Hopcroft and Robert Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, June 1973.

[29] Paolo Ienne and Rainer Leupers. *Customizable Embedded Processors: Design Technologies and Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[30] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Springer US, 1972.

[31] R. Kastner, A. Kaplan, S. Ogrenci Memik, and E. Bozorgzadeh. Instruction generation for hybrid reconfigurable systems. In *ACM Trans. Des. Autom. Electron. Syst.*, volume 7, pages 605–627, Oct 2002.

[32] Donald E. Knuth. Dancing links. *Millenial Perspectives in Computer Science*, pages 187–214, 2000.

[33] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.

[34] C. Liem, T. May, and P. Paulin. Instruction-set matching and selection for dsp and asip code generation. In *Proceedings of European Design and Test Conference, 1994.*, EDAC-ETC-EUROASIC '94, pages 31–37. IEEE, Feb 1994.

[35] N. Moreano, E. Borin, Cid de Souza, and G. Araujo. Efficient datapath merging for partially reconfigurable architectures. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 24, pages 969–980, July 2005.

[36] A. Peymandoust, L. Pozzi, P. Ienne, and G. De Micheli. Automatic instruction set extension and utilization for embedded processors. In *Application-Specific Systems, Architectures, and Processors, 2003. Proceedings. IEEE International Conference on*, pages 108–118, June 2003.

[37] K. Pocek, R. Tessier, and A. DeHon. Birth and adolescence of reconfigurable computing: A survey of the first 20 years of field-programmable custom computing machines. In *FCCM20*, 2013.

[38] Laura Pozzi. *Methodologies for the design of Application-Specific Reconfigurable VLIW Processors*. PhD thesis, Politecnico di Milano, Milan, Italy, Jan 2000.

[39] Laura Pozzi and Paolo Ienne. Exploiting pipelining to relax register-file port constraints of instruction-set extensions. In *Proceedings of the 2005 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems.*, CASES '05, pages 2–10. ACM, 2005.

[40] M. Stojilovic, D. Novo, L. Saranovac, P. Brisk, and P. Ienne. Selective flexibility: Creating domain-specific reconfigurable arrays. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 32, pages 681–694, May 2013.

[41] Etsuji Tomita, Yoichi Sutani, Takanori Higashi, and Mitsuo Wakatsuki. A simple and faster branch-and-bound algorithm for finding a maximum clique with computational experiments. *IEICE Transactions*, 96-D(6):1286–1298, 2013.

[42] Jeffery D. Ullman and Alfred V. Aho. *Foundations of Computer Science*. W. H. Freeman, New York, USA, 1994.

[43] A.K. Verma, P. Brisk, and P. Ienne. Fast, nearly optimal ise identification with i/o serialization through maximal clique enumeration. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 29, pages 341–354, March 2010.

[44] 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. ACM, 2004.

[45] 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.

[46] M. Zuluaga and N. Topham. Design-space exploration of resource-sharing solutions for custom instruction set extensions. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(12):1788–1801, Dec 2009.