Efficient Design-space Exploration of Custom Instruction-set Extensions

Marcela Zuluaga

Thesis for the Degree of Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2010

Abstract

Customization of processors with instruction set extensions (ISEs) is a technique that improves performance through parallelization with a reasonable area overhead, in exchange for additional design effort. This thesis presents a collection of novel techniques that reduce the design effort and cost of generating ISEs by advancing automation and reconfigurability. In addition, these techniques maximize the perfomance gained as a function of the additional commited resources.

Including ISEs into a processor design implies development at many levels. Most prior works on ISEs solve separate stages of the design: identification, selection, and implementation. However, the interations between these stages also hold important design trade-offs. In particular, this thesis addresses the lack of interaction between the hardware implementation stage and the two previous stages. Interaction with the implementation stage has been mostly limited to accurately measuring the area and timing requirements of the implementation of each ISE candidate as a separate hardware module. However, the need to independently generate a hardware datapath for each ISE limits the flexibility of the design and the performance gains. Hence, resource sharing is essential in order to create a customized unit with multi-function capabilities.

Previously proposed resource-sharing techniques aggressively share resources amongst the ISEs, thus minimizing the area of the solution at any cost. However, it is shown that aggressively sharing resources leads to large ISE datapath latency. Thus, this thesis presents an original heuristic that can be parameterized in order to control the degree of resource sharing amongst a given set of ISEs, thereby permitting the exploration of the existing implementation trade-offs between instruction latency and area savings. In addition, this thesis introduces an innovative predictive model that is able to quickly expose the optimal trade-offs

of this design space. Compared to an exhaustive exploration of the design space, the predictive model is shown to reduce by two orders of magnitude the number of executions of the resource-sharing algorithm that are required in order to find the optimal trade-offs.

This thesis presents a technique that is the first one to combine the design spaces of ISE selection and resource sharing in ISE datapath synthesis, in order to offer the designer solutions that achieve maximum speedup and maximum resource utilization using the available area. Optimal trade-offs in the design space are found by guiding the selection process to favour ISE combinations that are likely to share resources with low speedup losses. Experimental results show that this combined approach unveils new trade-offs between speedup and area that are not identified by previous selection techniques; speedups of up to 238% over previous selection thecniques were obtained.

Finally, multi-cycle ISEs can be pipelined in order to increase their throughput. However, it is shown that traditional ISE identification techniques do not allow this optimization due to control flow overhead. In order to obtain the benefits of overlapping loop executions, this thesis proposes to carefully insert loop control flow statements into the ISEs, thus allowing the ISE to control the iterations of the loop. The proposed ISEs broaden the scope of instruction-level parallelism and obtain higher speedups compared to traditional ISEs, primarily through pipelining, the exploitation of spatial parallelism, and reducing the overhead of control flow statements and branches. A detailed case study of a real application shows that the proposed method achieves 91% higher speedups than the state-of-the-art, with an area overhead of less than 8% in hardware implementation.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Marcela Zuluaga*)

Related Publications

Some of the material used in this thesis has been published in the following papers:

- Marcela Zuluaga and Nigel Topham. "Resource Sharing in Custom Instruction-set Extensions". In: *Proceedings of the 6th IEEE Symposium Application-specific Processors (SASP)*. June 2008.

- Marcela Zuluaga, Theo Kluter, Philip Brisk, Nigel Topham and Paolo Ienne. "Introducing Control-flow Inclusion to Support Pipelining in Custom Instruction-set Extensions". In: *Proceedings of the 7th IEEE Symposium on Application-specific Processors (SASP)*. July 2009.

- Marcela Zuluaga and Nigel Topham. "Design-space Exploration of Resource-sharing Solutions for Custom Instruction-set Extensions". In: *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*. December 2009.

- Marcela Zuluaga and Nigel Topham. "Exploring the Unified Design-Space of Custom-Instruction Selection and Resource Sharing ". In: *Proceedings of the International Symposium on Systems, Architectures, Modeling, and Simulation (SAMOS)*. July 2010.

Acknowledgements

First of all, I am very thankful to Professor Nigel Topham for giving me the opportunity to participate in such an exciting project. His wise advice and always-positive words of support made my PhD an enriching experience in every respect.

I would like to thank the rest of the PASTA project team. It was a pleasure to work with such a talented group of researchers. The discussions and collaborations with the group were instrumental in shaping the work that constitutes this thesis.

I would also like to thank Professor Paolo Ienne, Philip Brisk and Theo Kluter for welcoming me to work with them. The experience taught me important lessons both as a researcher, and as a person.

A big thank you goes to Edwin Bonilla for his meaninful advice on machine learning.

I am also very grateful to all of the friends that I have met in Edinburgh, I learned something from everyone, and I greatly enjoyed the time I spent at their side.

And to my family, for their endless support, encouragement, and love, thank you with all my heart.

Contents

Abstract i

Declaration iii

Related Publications v

Acknowledgements vii

Contents ix

1 Introduction 1

1.1 The Problem 3

1.2 Hypotheses 7

1.3 Contributions 8

1.4 Structure 9

2 Background and Related Work 11

2.1 Extensible Processors 11

2.2 Introduction to ISEs 13

2.3 Resource Sharing in ISEs 15

2.4 ISE Identification 17

2.5 ISE Selection 20

2.6 Pipelining ISEs to Speedup Loops 24

2.7 Machine-learning Background 27

3 Resource Sharing in ISEs 31

3.1 Introduction 31

3.2 Motivation 32
3.3 Parametric Resource-sharing Heuristic 34
3.4 Area and Delay of Graphs 52
3.5 Processor Customization 55
3.6 Experimental Evaluation 57
3.7 Results 59
3.8 Conclusions 69

4 Fast Exploration of the Resource-sharing Design-space 71
4.1 Introduction 71
4.2 Formalizing the Problem 73
4.3 Input-set Features 75
4.4 Generating the Training Data 76
4.5 Building a Model 77
4.6 Experimental Evaluation of the Model 79
4.7 Practical Usage of the Model 86
4.8 Conclusions 91

5 ISE Selection 93
5.1 Introduction 93
5.2 Motivational Example 95
5.3 Local Iterative Exploration 97
5.4 Global Iterative Exploration 104
5.5 Hardware/Software Partitioning Framework 104
5.6 Experiments and Results 106
5.7 Conclusions 117

6 Pipelining ISEs to Speedup Loops 119
6.1 Introduction 119
6.2 Pipelining ISEs 122
6.3 Allowing Loop Bodies with Multiple Exits 125
6.4 AFU Implementation 127
6.5 Code Example 132
6.6 Loop ISE Generation Framework 133

6.7 Experimental Setup 137
6.8 Results 138
6.9 Conclusions 141

7 Conclusions **143**
7.1 Critical Analysis 147
7.2 Future Work 148

Bibliography **151**

Chapter 1

Introduction

Embedded systems are expected to efficiently perform a small set of specific tasks such as processing sound, video or data packets. Moreover, the rapid proliferation of electronic devices puts pressure on industry to offer not only high performance, multi-purpose devices but also long lasting battery lives, low cost, and new generations of products in short periods of time.

Thus, embedded systems designers have to customize their solutions in order meet strict requirements: performance, cost, power consumption and time-to-market. Unfortunately these requirements are, very often, conflicting. While power consumption and performance are better achieved with custom hardware implementations, cost and time-to-market might drive the design towards software running on embedded processors. Thus, designers have a variety of implementation alternatives that ranges from a custom hardware implementation whose functionality matches exactly the application's requirements, to an embedded processor that can be programmed to perform the functionality of many applications. The flexibility offered by software is a tempting alternative when design cycles and cost are restrictive. However, high performance and power efficiency, achieved by hardware implementations, are key features for the success of any embedded design.

Hardware/software partitioning is an alternative that combines the flexibility of processors with the efficiency of hardware implementations. Thus, critical portions of the application are mapped directly to hardware implementation, whereas the non-critical parts are executed in software by a processor. The partitioning

process consist of identifying the portions of the application that will be executed in hardware. A complex design space of trade-offs is then available where different partitioning decisions create solutions with particular characteristics of flexibility, performance, cost and power efficiency.

Application-Specific Instruction-set Processors (ASIPs) are often tailored to target applications with a hardware/software partitioning approach. Numerous design methodologies for ASIPs have been proposed. There are two general approaches: loosely coupled co-processors, such as loop accelerators, and customizations to the Instruction-Set Architecture (ISA) with custom Instruction-Set Extensions (ISEs).

A co-processor can provide substantial acceleration, but requires significant area and often has a large communication overhead. At present, the designer is most often responsible for partitioning the application; however, once the partition is defined, advanced behavioural synthesis methods can automatically generate the accelerators.

ISEs, on the other hand, offer speedups by exploiting fine grain parallelism. Unlike co-processors, which are loosely coupled, ISEs are tightly coupled to the processor pipeline, and can exchange scalar data with the processor's register file every cycle. ISEs have evolved to the point where they have their own local memories, and, as a consequence, can achieve speedups comparable to co-processors.

ISEs are complex instructions that perform the functionality of a group of basic arithmetic or logic operations that are dependent in the application. For instance, an application might often need to add three numbers. This would translate into an instruction stream with two add instructions, where one of the inputs of the second add instruction is the result of the first one. On the other hand, an ISE can be created to perform these chained operations. In which case, an Application-specific Functional Unit (AFU) is built to execute in hardware two chained additions and then it is attached to the execution stage of the processor. As the hardware delay of an adder is in most cases much smaller than the clock period of the processor, a chain of 2 adders can be still executed in one cycle. Thus, two instructions in the instruction stream can be replaced by one that will require three inputs and will give the result in one single cycle. Similarly, more complex instruction patterns can be found in the applications.

1.1 The Problem

Much research in recent years has focused on ASIP design automation in order to find the best partition of an application. When the hardware partition is to be implemented as ISEs, the partitioning process takes place in two stages: in the first stage potential ISEs are identified in the application, while in the second stage the best set of ISEs is selected. However, there is a gap between the partitioning process and the hardware datapath synthesis of the ISEs. Some of the datapath synthesis issues that have been ignored in the partitioning process are addressed in this thesis. Namely, resource-sharing amongst the ISEs and pipelining of the ISE hardware datapath.

Flexibility and performance can be improved by increasing the number of ISEs included in the processor, more software routines and more segments of code can be accelerated. However, to independently generate a hardware datapath for each ISE would lead to an unacceptable increase in die area. Hence, resource sharing is essential to create a customized unit with multi-function capabilities.

The problem of identifying ISEs in the application and selecting the optimal set for implementation is solved based on the gain and the cost of implementing the functionality of the ISEs in hardware. A typical selection process, under area constraints, would include ISEs in the design until the area constraint is met. When resource sharing is being used for implementation, adding one ISE to the AFU datapath may cost anywhere from a few multiplexers to implementing all of its operators. Therefore, the selection process needs to be aware of the effects of resource sharing over potential ISEs in order to make informed decisions.

On the other hand, the benefits of pipelining a datapath are well understood. When the datapath of an ISE is already implemented in hardware, a clear performance improvement is to make use of a pipelined datapath to overlap execution of consecutive calls to the same ISE. For instance, consecutive execution of the same ISE could take place when the ISE covers the execution of an entire inner loop. However, current identification techniques constrain this optimization by keeping, by all means, the control of the program flow in the software partition. To address this concern, this thesis proposes a new concept to identify ISEs that borrows ideas from zero-overhead loop instructions to permit pipelined execution

of loops in the customized unit.

Additionally, previous works also lack the ability to explore the trade-offs that can be obtained in the process of partitioning an application. Instead, the solution has been always considered to be one clear optimal. The concept of extending a processor with an AFU is already a trade-off: area is spent for the sake of obtaining application speedups, and as more area is spent, more speedups are obtained. Hence, a variety of trade-offs are exposed in the process of partitioning an application. Similarly, trade-offs between speedup and area exist when a set of ISEs is considered for an implementation that performs resource sharing. The more resources are shared, the more the speedup obtained from the ISEs is likely to decrease. Thus, it is important for the designer to be aware of all of the existing trade-offs between speedup and area in order to choose the one that better suits the design goals and constraints of the project.

Figure 1.1 and 1.2 show two hypothetical examples that illustrate some of the mentioned problems. In Figure 1.1 two ISEs, ISE_1 and ISE_2, are identified in the most critical section of an application. Each ISE groups three basic operations and thus, their software execution takes three clock cycles, while the hardware implementation of each ISE has an execution latency of one clock cycle. The areas occupied by the hardware implementation of ISE_1 and ISE_2 are 8 and 7 area units, respectively. The selection process is performed under an area constraint of 10 units. A traditional ISE selection process makes a decision based on the area required by individual ISEs. In which case, only one ISE can be selected since implementing both of them would require 15 units. As both ISEs represent the same speedup to the application, ISE_2 is chosen as it requires less area. However, when the implementation shares resources amongst the ISEs, and when the selection process is aware of such an implementation strategy, both ISEs can be selected as they can be implemented together using only 10 units. In this case, a single datapath that implements the functionality of both ISEs can be used. This simple example shows that it is important to take into account the available implementation alternatives in the selection process, as ISE area requirements can be considerably reduced by using resource sharing and thus, important speedups can be obtained.

On the other hand, both solutions: ISE_1+ISE_2 requiring 10 area units, and

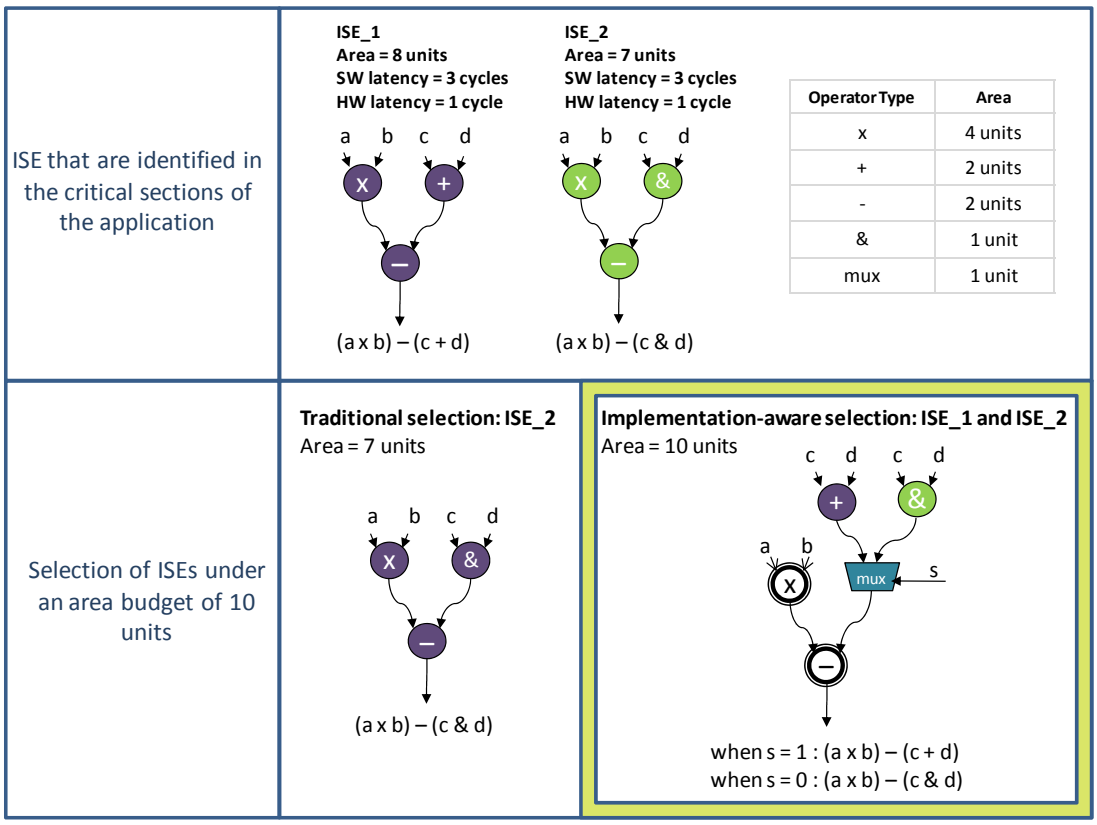

Figure 1.1: Example where two ISEs are identified in the most critical section of an application. ISE selection takes place under an area budget of 10 units. Traditional selection techniques consider ISEs that are implemented as independent hardware datapaths. Thus, only one ISE can be selected, as implementing both of them would require 15 area units. On the other hand, an implementation-aware selection process can select both ISEs by sharing resources amongst the ISEs, therefore obtaining more speedups in the given area budget.

ISE_2 requiring 7 area units, represent different trade-offs between speedup and area. When ISE_1 and ISE_2 are implemented, more area is spent in order to increase the speedup returns. As each solution is characterized by two important metrics: speedup and area, there is no single best, but instead, there are trade-offs in which more resources can be invested in return for gaining application speedups. Thus, unlike previous ISE selection methods, the techniques proposed in this thesis focus on exploring the design space of available trade-offs between speedup and area.

Figure 1.2 shows a loop in which an ISE is identified. This ISE can be implemented in hardware to perform the main operations of the loop in two cycles, in

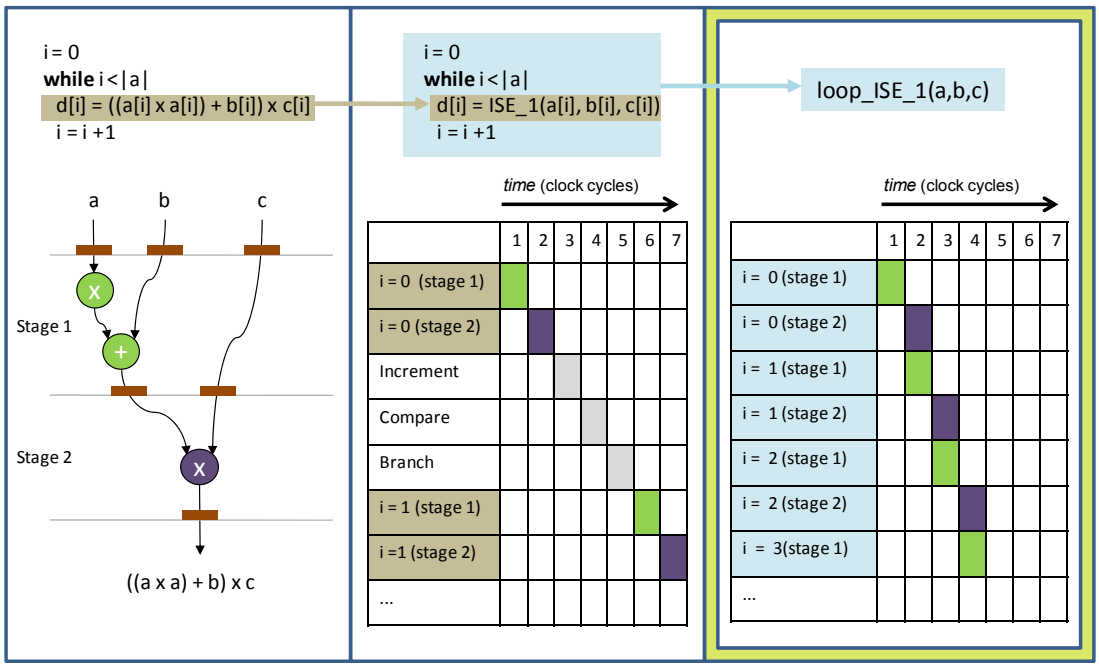

Figure 1.2: Example where an ISE is identified to perform the main operations of a loop body. A pipelined implementation of the ISE allows for the overlap of the execution of consecutive iterations. However, this is not possible due to control flow overhead. This thesis proposes an ISE that can absorb the instructions that control the iterations of the loop, in order to obtain the speedups offered by a pipelined datapath.

contrast with the three cycles required by the execution of the software sequence. Thus, one cycle is saved for each iteration of the loop. The ISE is implemented as a pipelined hardware datapath, i.e, it is executed in two independent stages. Moreover, the operations of consecutive iterations do not present data dependencies. In theory, when an iteration starts execution of stage 2, the following iteration could start the execution of stage 1. However, the first time table of Figure 1.2 shows that this is not possible for two reasons. Firstly, the instructions that control the iteration of the loop are issued after every ISE. Secondly, an ISE whose execution time takes more than one cycle typically stalls the processor pipeline and no other instruction can be issued until its completion. Thus, in order to exploit the speedup potential of pipelined implementations, important changes need to be made in the ISE identification and selection stages. The second time table of Figure 1.2 shows how the execution of the loop developes when the control of the loop is given to the ISE itself, as it is proposed in this

thesis. This example shows that important speedups can be obtained not only by overlapping the execution of loop iterations, but also by significantly reducing loop control overhead.

1.2 Hypotheses

This section presents the statements that motivated this thesis. A process of research and experimentation was carried out in order to prove the validity of these statements. Therefore, this thesis is based on the following hypotheses:

- Resource sharing amongst ISEs can be applied in order to obtain flexibility and area efficiency. Furthermore, the design space of resource-sharing solutions can be explored in order to find the available implementation trade-offs between instruction latency and area savings.
- Given a set of ISEs that is considered for implementation, the optimal trade-offs can be quickly found by learning the behaviour of previously explored resource-sharing design-spaces via predictive modeling techniques.
- Given a set of ISE candidates, the unified design space of ISE selection and resource sharing can be explored in order to find solutions that achieve maximum speedup and maximum resource utilization.
- Additional parallelism can be exploited by pipelining the hardware datapaths of the customized unit. Application loops may be sped up in this fashion when control flow statements are appropriately inserted into the ISEs.
- The aforementioned optimization processes can be implemented as an automated tool to aid and speed up the design of extensible processors.

1.3 Contributions

This thesis presents novel techniques in the field of automated processor synthesis, which advance the state-of-the-art and enable the design of more efficient processors. The main contributions of this thesis are summarized below:

- **A novel heuristic that can be parameterized to control the degree of resource sharing amongst a given set of ISEs, thereby permitting the exploration of the existing implementation trade-offs between instruction latency and area savings.** As solutions that aggressively share resources present the highest ISE datapath latency, parameters are used to control the ISE latency increments in strategic points of the resource-sharing process. Parameters are also used to enable the creation of multi-function operators such as adder-subtractors as well as the compression of ISE operators that allow synthesis optimizations to create modules such as multiply-adders and carry-save adders. The design space of implementation trade-offs can be explored by varying the parameter values within their allowed ranges. This contribution is discussed in Chapter 3.

- **An original method to quickly expose the optimal trade-offs between instruction latency and area savings, given a set of ISEs that is considered for implementation.** Based on previously explored design spaces, predictive modeling is used to generate the parameterization that the resource-sharing process requires in order to directly find the optimal trade-off solutions. This contribution is discussed in Chapter 4.

- **A complete hardware/software partitioning framework that, for the first time, combines the design spaces of ISE selection and resource sharing in ISE datapath synthesis.** This integration exposes a wide range of design points that can offer the designer previously unseen solutions that maximize utilization and speedup within a given area budget. An exploration of this unified design space is performed by guiding the selection process with resource-sharing considerations, thereby favouring combinations that are likely to share resources with low speedup losses.

The result of this exploration is a set of selection-implementation alternatives, each of which represents a unique trade-off between performance gain and cost. On the benchmarks analyzed, the proposed technique finds solutions that under a fixed area constraint, achieve speedups from 9% to 251% higher than previous selection techniques. This contribution is discussed in Chapter 5.

- **An innovative method to allow multi-cycle ISEs, implemented as pipelined datapaths, to overlap the execution of consecutive calls.** This is achieved by giving the extension unit the control over loop iterations, when ISEs can cover the entire loop body. To further expose instruction-level parallelism, the proposed loop ISE supports loops whose bodies form hyperblocks, by providing the extension unit the means to communicate to the processor the next instruction address. Loop ISEs broaden the scope of instruction-level parallelism and obtain higher speedups compared to traditional ISEs, primarily through pipelining, the exploitation of spatial parallelism, and by reducing the overhead of control flow statements and branches. A detailed case study of the JPEG application shows that the proposed method achieves a speedup of 3.1×, while the state-of-the-art solution achieves a speedup of 2.2× over pure software execution, with an area overhead of less than 8% in hardware implementation. This contribution is discussed in Chapter 6.

1.4 Structure

This thesis is organized as follows:

Chapter 2 introduces basic concepts of processor extensibility as well as previous relevant works found in academia and industry. Additionally, the techniques proposed in this thesis are further introduced and contrasted with the state-of-the-art.

Chapter 3 presents a heuristic that can be parameterized to expose a broad range of trade-off solutions to the implementation of an area-efficient AFU that is able to share resources amongst a selection of ISEs.

Chapter 4 develops a machine learning technique to quickly find the optimal trade-offs in the design space of resource-sharing solutions that can be exposed with the heuristic proposed in Chapter 3.

Chapter 5 explores a heuristic to solve the ISE selection problem while taking into account a resource-sharing-based implementation. The proposed heuristic explores the design space in order to offer the designer a set of unique trade-offs between cost and performance gain.

Chapter 6 introduces a new type of ISE that is able to take the control of an inner loop in order to exploit the pipelined datapath of ISEs that cover a complete loop body.

Chapter 7 presents concluding remarks and explores some ideas that could complement or improve the work presented in this thesis.

Chapter 2

Background and Related Work

This chapter introduces basic concepts of processor extensibility. Additionally, it explores existing architectures and tools developed by academia and industry and compares them with the techniques proposed in this thesis.

Section 2.1 introduces the concept of processor extensibility and refers to some of the extensible processors available in industry and academia. Section 2.2 defines ISEs and describes how application speedups can be achieved with their implementation. Section 2.3 describes other works that have approached the resource-sharing problem, and discusses how the techniques proposed in this thesis advance the state-of-the-art. Section 2.4 presents exisiting solutions for the ISE identification problem while Section 2.5 presents existing solutions for the ISE selection problem. Section 2.6 reviews other works that have proposed solutions to speedup the execution of loops and describes their difference with the techniques explored in Chapter 6. Finally, Section 2.7 gives an overview of the machine-learning techniques that are used in Chapter 4.

2.1 Extensible Processors

The term *extensible processors* refers to microprocessors that offer the possibility to augment their ISA with ISEs. ISEs are executed in custom functional units or AFUs that are part of the processor datapath.

There are several extensible processors available in the market. The most visible of these are provided by ARC and Tensilica.

ARC 600™and ARC 700™ are configurable processor families that can be configured using the ARChitect™Processor Configurator tool [1], from ARC. The designer can customize a processor from a wide range of configuration options such as inserting special functional units, peripherals and closely coupled memory, specifying register file type and size, interfaces, interrupts and cache typical parameters. Processors can also be extended with ISEs by specifying the AFU as a hardware module in Verilog.

Similarly, Tensilica Xtensa processor generator [2] offers the designer some configuration options such as special functional units, register files and zero overhead loop support. The processor can also be extended with ISEs. The designer can specify the ISE's functionality in the Tensilica Instruction Extension (TIE) language, based on Verilog. Alternatively, the designer can use the XPRES compiler tool to automatically scan the application code to generate *fusion* candidates [3]. *Fusion* operators are chains of basic operations compressed into one. The concept of fusion is similar to ISEs but focuses on reusability of small chains of operations.

ARC and Tensilica offer a complete tool chain for the design and development stages. The techniques proposed in this thesis can be used in conjunction with these tools to extend either of the offered core architectures. Additionally, soft processors such Altera Nios II [4] and Xilinx MicroBlaze [5] can also be coupled with the extensions created by the tools described in this thesis.

EnCore Extensible Processor

EnCore, introduced in [6], is an extensible processor based on the ARCompact™ ISA. A base-line implementation of EnCore is an in-order single-issue processor with a 5-stage pipeline. The gate count of the pipeline is approximately 25 Kgates. Gate count is a technology-independent metric that refers to an estimation of the number of 2-input NAND gates that are needed for the hardware implementation of the design. It is calculated as the standard cell area given by the synthesis tools divided by the area of a strength-1 2-input NAND gate in the same technology.

The EnCore processor can be extended with an AFU that implements ISEs and that is closely coupled with the processor pipeline.

The extension registers of the ARCompact ISA provide a route through which the main core and the AFU exchange operands and results. A collection of scalar registers are mapped to a set of short vector operands, each containing 4 scalar registers. Each extension instruction can specify up to three source vectors and up to two destination vectors. Source vector elements can also be permuted as part of the extension instruction, thereby removing positional constraints on input operands.

EnCore is the target processor for the techniques developed in this thesis.

2.2 Introduction to ISEs

ISEs are complex instructions that are added to a predefined processor ISA and execute subgraphs of the program Data Flow Graph (DFG). A DFG forms a basic block in the Control Flow Graph (CFG) of an intermediate compiler representation of a program. In the DFG, graph nodes represent basic operations and edges represent data dependencies. On the other hand, the CFG represents all of the possible execution paths of the entire program.

In order to create a single complex instruction from a subgraph S of the DFG, the three following conditions are checked:

- All of the nodes in the subgraph must be executable in the AFU. Thus, nodes that correspond to memory operations or function calls, that need to be individually handled by the processor, are usually forbidden.

- There is no path from a node $x \in S$ to another node $y \in S$ through a node $z \notin S$ This property is commonly referred to as the *convexity* of a graph. In order to exploit more parallelism, ISEs may group two or more disconnected subgraphs as long as the ISE remains convex. Figure 2.1(a) shows an example of a subgraph that does not meet this condition.

- The third condition is dependent on the architecture of the extensible processor and is referred to as *I/O architectural constraint*. As the AFU exchanges input and output values with the register file of the processor, the number of inputs and outputs of the ISE must agree with the number of

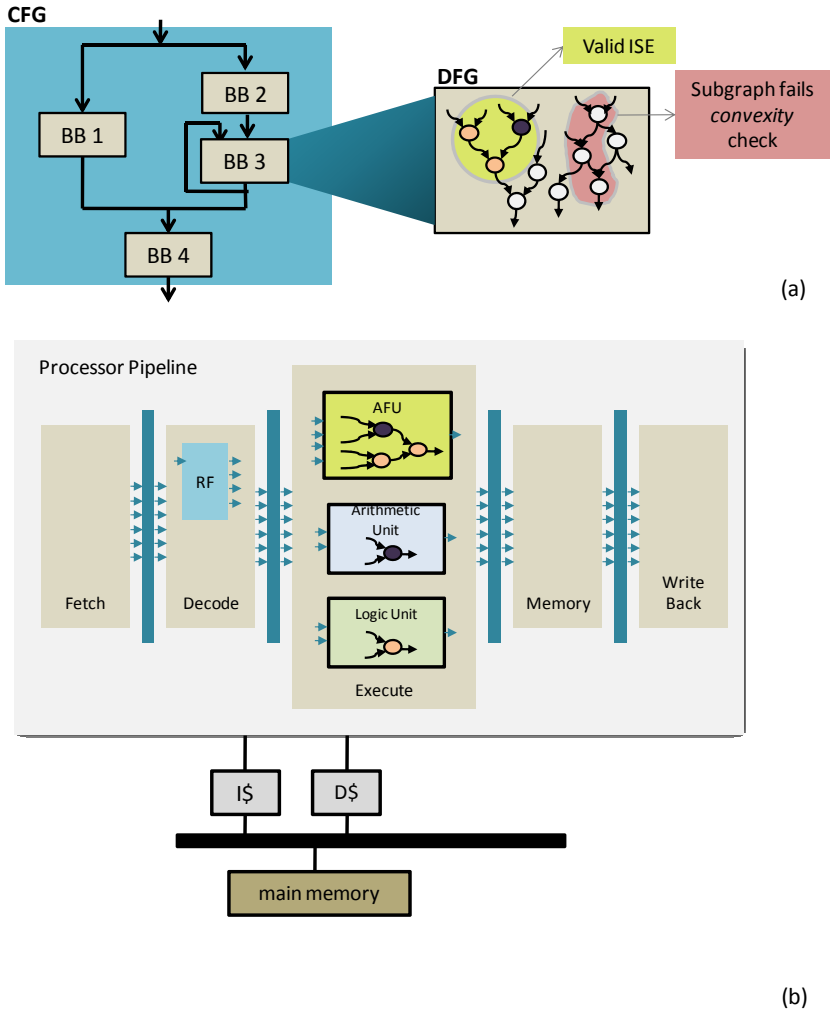

Figure 2.1: (a) Shows how ISEs are identified in the DFGs of a program. (b) Shows an extensible processor augmented with an AFU to execute an ISE.

inputs and outputs of the register file of the processor. This condition allows the AFU to obtain in a single clock cycle all of the inputs that are required for ISE execution. Similarly, it allows the processor to store in a single clock cycle all of the outputs generated by the ISEs.

Figure 2.1(a), shows an example of an ISE that is chosen to replace a chain of operators in one of the basic blocks or DFGs of the CFG of a program. ISEs are executed in a custom functional unit or AFU. The AFU has a hardwired datapath that corresponds to the functionality of the ISE. Figure 2.1(b) shows a typical scheme for an extensible processor that executes ISEs in an AFU.

Performance improvements are achieved by the parallel execution of operations, together with the chaining of basic operations in the same clock cycle. Additionally, the use of ISEs can improve energy consumption for several reasons. Firstly, the number of instruction fetches decreases as the ISEs replace several native operations in the application code. Secondly, the number of read and write operations to the register file decreases, as the chaining of hardwired operations removes the need for temporary registers. Another positive consequence of using ISEs is that the code of the application is reduced in size.

2.3 Resource Sharing in ISEs

Typically, one AFU is constructed for each ISE that is adopted to be part of the ISA of the processor. However, most embedded processors are single-issue architectures, thus, only one ISE will be in execution at any point in time. Therefore, resources can be shared amongst the adopted ISEs in order to increase the hardware utilization. As a consequence, more instructions could be allocated in a given area and the overall cost and performance of the system could be considerably improved.

Minimizing the area required to implement a set of ISEs is equivalent to the *minimum-area common super-graph* problem, where a graph is the representation of an ISE. Nodes represent operations and edges represent data dependencies. Thus, given a set of graphs G, the minimum-area common super-graph problem aims at finding a set of graphs G' where every graph $g_i \in G$ is isomorphic to at least one subgraph of a graph $g'_j \in G'$, and G' minimizes the total area of the set. A graph g_i with vertices V_i and edges E_i is isomorphic to graph g_j with vertices V_j and edges E_j, if there is an edge-preserving bijection between V_i and V_j.

A variant of the minimum-area common super-graph problem has been demonstrated to be NP-complete in [7].

A path-based heuristic approach to this problem is presented in [8], which transforms a set of ISEs into a single hardware datapath based on the classical problem of finding maximal subsequences and substrings thereof in the graph representation of ISEs. Their aim is to maximize die area reduction through the construction of a *consolidation graph* representing merged ISEs.

Similarly, [9] introduces a heuristic that uses the construction of a *compatibility graph* to reduce the problem of merging two datapaths to a maximum weight clique problem, which is NP complete. They propose non-exact methods to solve this problem in polynomial time. This approach operates on Control/Data Flow Graphs (CDFG) which correspond to application loops that are to be mapped into a reconfigurable unit which is closely coupled to a processor. Therefore, the nature of their input graphs is different as the work in this thesis focuses on acyclic graphs. Consequently, they do not prevent final graphs from containing false loops.

The heuristic proposed in [9], is extended in [10] to account for the cost of including multiplexers in the merged graphs. Based on this heuristic, [10] proposes a high-level synthesis flow for ISEs, where ISE graphs are scheduled first in order to generate from every multi-cycle ISE several single-cycle ISEs that can be merged with each other in the following binding process. As this methodology performs high level synthesis at the same time as merging the ISEs, the resulting datapaths are limited and cannot be used if a pipelined AFU is desired.

The heuristic presented in Chapter 3 of this thesis, uses a path-based approach to perform resource sharing, because it allows for control of every step during the datapath merging process. The resource-sharing heuristic proposed in Chapter 3 differentiates itself from previous approaches in the introduction of latency constraints in the merging process, while they focus only on maximizing the area savings. Furthermore, the space of possible implementation alternatives is explored instead of trying to find a unique solution.

Another approach to resource sharing in ISEs is presented in [11]. The technique proposed in [11] performs ISE identification, selection and resource sharing in the same heuristic. At first, small patters are identified in the application and referred to as *building blocks*. Building blocks are then combined in order to create more complex patterns that constitute ISEs. The motivation for creating ISEs from building blocks is the possibility to reuse the hardware implementation of these blocks. Although the heuristic proposed in [11] simplifies the ISE generation process significantly, as a consequence, the speedup potential of ISEs is not fully explored as ISEs are limited to be built from a combination of the

selected small patterns. Similarly, the resource sharing possibilities are limited since ISEs can share only the whole predefined building blocks.

Hardware resource sharing is also a design goal in the field of high-level synthesis. [12] presents an algorithm for dynamically generating templates of reoccurring patterns for resource sharing in CDFGs. The approach in [12] is meant to be used by compilers that translate complete applications described in a high-level language into detail hardware specifications. The high-level representation is first translated into an intermediate CDFG. Then, reoccurring patterns are extracted from the CDFG and selected patterns represent the hardware blocks that can be reused during in the application. As the input of the process is the complete CDFG of the application, the problem is more related to the template extraction problem in the ISE identification phase that is discussed in the following section.

Subgraph isomorphism is another field of study that has been used to find resource-sharing solutions in the domain of behavioural synthesis. The problem can be seen as enumerating all isomorphic subgraphs within a given set of graphs. This problem, however, is even more relevant at the ISE identification stage, where increasing the reusability of the chosen ISEs requires the identification of isomorphic subgraphs in the DFGs of the application. However, another generalization of the subgraph isomorphism problem: finding the Largest Common Subgraph (LCSG) within a given set of graphs, is part of the procedure to find resource-sharing solutions proposed in Chapter 3. The LCSG problem is to find the largest subgraph which appears to be common in a given set of graphs. When only two graphs are given, this problem is polynomial, while it is NP-hard for three or more graphs. In this thesis, the main objective of performing meging is to reduce area, the LCSG problem is reduced to finding the maximum-area common substring within the graphs. Nevertheless, approximation algorithms to the LCSG problem, such as [13] and [14], can be used to replace the search for the largest common substring within graphs in the algorithm proposed in Chapter 3.

2.4 ISE Identification

Extending an ISA with new complex instructions is generally divided into instruction identification and instruction selection. Instruction identification starts from the CFG of an application, extracted from high level code, such as C. Then, it operates over each basic block in the CFG to identify potential subgraphs that can be implemented as ISEs. The identification process involves clustering basic operations to create larger and more complex ones, while taking into account a set of constraints and a guide function that captures the designer's objectives. While ISE identification operates on a basic block, selection looks over the entire application to choose the set of candidates that best suits the designer's goals and constraints.

Early works on ISE identification for reconfigurable processors considered the identification of Multiple-Input Single-Output (MISO) subgraphs in the hot basic blocks of the application [15]. The goal was to find maximal MISO connected subgraphs whose implementation is feasible in a reconfigurable unit attached a base processor. The complexity of this problem is linear, however, the number of nodes that can be clustered under the single output constraint limits the potential speedup that can be achieved by a dedicated hardware datapath.

Motivated by architectures that could commit more than one value per cycle, such as the VLIW ST200 [16], later works aimed at identifying Multiple-Input Multiple-Output (MIMO) subgraphs. Relaxing output constraints allowed the identification of larger subgraphs that could exploit more parallelism. On the other hand, the problem of finding the optimal MIMO subgraph from the DFG of a basic block has a solution space that grows exponentially with the number of nodes of the DFG. Thus, enumerating all of the subgraphs from a given graph is an intractable problem. However, as in [17], it has been shown that the number of feasible subgraphs is much smaller than the exponential worse case since the feasibility of the subgraphs is constrained by factors such as I/O ports of the register file, forbidden operations and convexity.

The ISE identification phase is commonly recognized as the enumeration of all of the feasible subgraphs of a basic block such that the selection phase cannot miss the global optimum. In [17], a heuristic to quickly enumerate all of the fea-

sible subgraph from a given graph was presented. When only feasible subgraphs are to be enumerated, the design space can be pruned given I/O architectural constraints. In [17], the design space was also pruned by allowing only connected subgraphs.

An exhaustive enumeration of feasible subgraphs which allows disconnected nodes to be part of MIMO subgraphs was presented in [18]. This enumeration is performed in order to find the optimal subgraph of one basic block according to a *merit function*. The *merit* of a subgraph is taken to be the speedup obtained by its hardware implementation during the execution of a program. This algorithm, based on the violation of I/O and convexity constraints, prunes a complete binary decision tree, where decisions are to include or not a node of the DFG in the subgraph that is being created. This enumeration is extremely computationally expensive for large graphs and for very loose I/O constraints.

Other works coupled the speedup potential of the subgraphs with their capability to be reused by other sections of the program [19, 20, 21, 22, 23]. When reusability is considered, candidate subgraphs tend to be small, thus restricting the potential performance of ISEs. [19] further simplified the complexity of the search by limiting the latency of the subgraphs. Similar constraints have been used in other works, such as limiting the number of operators in the candidate subgraphs [20].

Other exact solutions to the problem of finding the optimal subgraphs from basic blocks are based on Integer Linear Programing (ILP) [24].

In [25], a polynomial time algorithm to fully enumerate all of the feasible subgraphs in a given graph is presented. This is achieved by reformulating the problem on a polynomial solution space.

[26] proposed a heuristic to find the optimal subgraph from a basic block given a *gain function* that is taken to be, as in [18], the speedup obtained by the hardware execution of the subgraph during the execution of the program. The authors in [26] refer to their heuristic as *ISEGEN*. *ISEGEN* follows the basic principles of the Kernighan-Lin (K-L) min-cut partitioning heuristic, which identifies subgraphs in a bottom-up fashion using iterative improvement. The K-L min-cut heuristic toggles nodes between software and hardware partitions based on the *gain function*. Results in [26] show that *ISEGEN* is able to find

solutions with speedups comparable to those obtained by exhaustive search.

Works such as [27] and [28] proposed solutions to overcome the limitations imposed by the number of I/O ports of the processor register file. [27] proposed to schedule input and output transactions throughout the ISE execution such that the number of reads and writes to the register file fits the number of I/O ports available for the AFU. On the other hand, [28] suggested to have internal memory in the AFU to store the data requested during ISE executions.

Previous identification techniques had used the I/O constraints to significantly prune the design space of solutions. However, as the number of I/O ports available to the AFU does not necessarily limit the number of inputs and outputs of the ISEs, as demostrated in [27] and in [28], new heuristics were needed given that full subgraph enumeration techniques do not scale under unlimited I/O conditions.

The work presented in [29] shows that the speedups generated by ISEs behave monotonically. This means that increasing the number of nodes that the ISE comprises cannot reduce its speedup gain. Under this assumption, there is only need to enumerate maximal convex subgraphs from basic blocks to guarantee that the subgraph that maximizes speedup gain is enumerated. Later, other works such as [30] and [31] have proposed algorithms that, under the same assumption, improve the runtime to enumerate all maximal convex subgraphs in a basic block.

2.5 ISE Selection

The subgraphs listed from each basic block by identification methods become the candidate ISEs to be evaluated by a selection process.

Given a set of candidates C, the selection process searches to find the subset $S \subseteq C$ such that the overall speedup of the application is maximized under a set of design constraints.

As seen in the previous section, there is a significant body of previous work on ISE identification. However, the selection phase is often neglected or combined with the identification phase, when identification provides disconnected subgraphs, by iteratively selecting the subgraph that maximizes the gain until a global design constraint is met.

A common global design constraint taken by some approaches is the maximum number of ISEs that can be selected, referred to as N_{max} in this chapter [24, 32, 26]. An optimal solution to the selection problem under this global design constraint is described in [32]. It uses an exact algorithm to enumerate the best set of disjoint subgraphs from each basic block of the application to generate the final set of ISEs. Then, all possible combinations of N_{max} or less candidates from the generated set of subgraphs represent possible solutions to the selection problem. As pointed out in [32], exact solutions are computationally expensive and heuristics need to be applied. An approximate solution proposed in [32], iteratively identified the best subgraph from each basic block in order to select the global best. Once a global best is found, their nodes are contracted into one node, thus preventing them from being included again in another candidate. The iterative process ends when N_{max} candidates are selected. Another heuristic, presented in [32], combines identification and selection using genetic algorithms. This solution appears to have better scalability when the I/O constraints get looser.

Another global design constraint that has been considered is the maximum area that the AFU can use, referred to as A_{max} in this chapter. [33] studies the different global constraints that are commonly taken during ISE selection in order to simplify the design space or to meet design goals. An ILP formulation to solve the problem of ISE selection given a set of candidates is proposed in [33], this formutation can be constrained by N_{max} or A_{max}. By experimenting with relaxed constraints it is concluded that reasonable limits for either constraint, in most cases, can perform close to the maximum observed speedup. [33] also highlights the computational expense of performing ILP candidate selection and proposes a greedy approach with three possible ranking metrics: speedup, speedup per area and software execution time of the templates.

Area, as a constraint in the selection process, has been considered by many others, which have solved the problem either under ILP formulations [34, 35] or by using greedy algorithms [36, 37]. However, implementation considerations have been limited to check the real area and delay of the ISEs in hardware.

Other attempts for a more accurate solution have been also proposed. [38] considered timing and area constraints for the selection phase. Prior to selec-

tion, an identification phase prunes inferior candidate subgraphs in order to limit the candidate list to promising subgraphs. Hardware synthesis is attempted in every candidate subgraph. A candidate subgraph can be discarded if the synthesis process fails because of timing or area constraint violations. From each candidate subgraph, several ISE candidates can be generated. Each ISE candidate corresponds to a synthesis alternative and it is characterized by its area, its critical path and the number of cycles to complete its execution. The first ISE candidate generated from a candidate subgraph is to be executed in one clock cycle, however, if its critical path is larger than the critical path of the processor, other ISE candidates are generated by increasing the number of cycles one by one until the critical path of the ISE is equal to or smaller than the critical path of the processor. Then, Pareto optimality is used to compare the generated set of ISE candidates in terms of area, critical path and execution cycles. ISE candidates dominated by others are discarded. The selection process takes place over the remaining ISE candidates. The goal of the selection phase is to select the combination of ISE candidates, selecting only one version of every candidate subgraph, that maximizes the application speedup under a given area constraint. A few assumptions are made in order to simplify the problem and solve it with a branch and bound algorithm that finds a set of best selection alternatives to evaluate accurately.

In [39], an identification heuristic makes use of a guide function to prioritize different growing directions during the exploration of the DFGs. The guide function takes into account factors such as criticality, which tends to give more priority to the nodes in the critical path of the DFG, latency and area of the operators in hardware, and I/O constraints. The candidates identified in this process are then passed through an *isomorphism check*. Graphs that are equivalent are grouped to form one AFU candidate. Then, two passes through the AFU candidates annotate information to them. The first pass records which AFU candidates can be *subsumed* by others; AFU_1 can be *subsumed* by AFU_2 if the functionality of AFU_1 can be reproduced by AFU_2 by using the identity input values on some of its operators to pass data through them with no effects. The second pass records *wildcard* options for each AFU candidate; *wildcards* are AFU candidates that are equivalent except for the operations on one node (*wild-*

cards). The addition of *wildcards* and *subsumed* subgraphs adds complexity to the subsequent AFU selection phase as identified templates may be implemented by several AFU candidates. Thus, the gain of implementing each AFU candidate has to be updated every time one AFU candidate is selected. When an AFU candidate is selected, the templates that are comprised in it are removed from the annotations of the remaining AFU candidates. For AFU candidate selection, the problem is simplified by taking a greedy approach that repetitively takes the AFU candidate with maximum gain and subsequently updates the gain of the remaining AFU candidates.

[40] proposes a complete framework, from subgraph enumeration to selection, that takes into account *technology mapping* results. Prior to selection, isomorphism is sought amongst candidate subgraphs. Isomorphism information is then annotated in the candidate list. [40] highlights the importance of performing *technology mapping* on every candidate in order to discard any clear suboptimal. A candidate may be a clear suboptimal if *technology mapping* reports that it cannot fit the target or that hardware execution is not faster than software execution. *Technology mapping* also allows the selection process to have precise information about the area and timing that can be obtained after implementing a candidate. In order to select a set of candidates for implementation, [40] reworks a greedy selection algorithm presented in [32] to deal with isomorphism information.

Thus, mapping issues, namely isomorphism between ISE candidates and delay and area of each ISE candidate in synthesized hardware, have been approached by some research groups. Isomorphism, is important in order to avoid implementing separately ISEs with equivalent functionality, and to allow a selection process aware of template reusability across the complete application.

Mapping information such as exact area and timing of each ISE candidate has been considered under the assumption that each ISE will form a separate AFU. However, current demands for more and bigger ISEs that are able to yield application speedups comparable to thread level parallelism, require sharing resources amongst ISEs to create AFUs that can be reconfigured to perform the functionality of several ISEs.

Resource sharing for ISE datapath synthesis has been also considered in [8] and [9]. However, they assume that the set of ISEs has previously been selected

under different considerations.

In summary, area constraints are a proven type of global constraint that should be considered in the selection process, since maximizing speedup as a unique constraint and goal assumes the availability of unlimited resources. Therefore, a selection process should be aware of the area requirements of a subset of ISE candidates. However, when resource sharing is used for ISE datapath synthesis, the area and profitability of the subset cannot be known until resource sharing is attempted. Furthermore, the trade-offs between speedup and area that are found in a selection of ISEs depend not only on the area and speedup that the individual ISEs require or yield but also on the way that they can be merged with each other. Thus, a clear limitation of previous selection techniques is the lack of interaction with the implementation phase of the design. In order to solve this limitation, Chapter 5 of this thesis proposes a framework in which the selection of ISEs takes into account results of ISE datapath synthesis with resource sharing. This integration allows to offer the designer solutions that use more efficiently the given resources, thus obtaining greater speedups. Furthermore, unlike previous approaches, the framework presented in this thesis explores the design space of trade-offs between speedup and area that are available to the designer at the selection level.

2.6 Pipelining ISEs to Speedup Loops

Chapter 6 of this thesis proposes a new type of ISEs, referred to as *loop ISEs*, that are able to execute entire loops in the AFU in order to leverage the pipelined datapath of ISEs that can cover entire loop bodies. In order to increase the number of loops that can be converted to loop ISEs, loops whose bodies form hyperblocks are also supported. This section presents previous works in contrast with the proposed loop ISEs.

[41] proposed a hyperblock loop acceleration model for use in the Garp reconfigurable coprocessor. They decomposed the runtime of two applications with two distinct data sets, into single-exit loops, multi-exit loops, hyperblock loops, unfruitful loops, and other. The hyperblock loops ranged from 9.0% to 57.6% of the different applications. The speedups achieved by implementing the differ-

ent loops as hardware accelerators were not reported. [42] advocated the use of hyperblock formation to implement efficient loops in VLIW multimedia processors. The loops were stored in an external loop buffer which was distinct from the instruction cache. The use of the buffer reduces instruction fetch power and eliminates branch penalties, similar in principle to zero-overhead loops.

The method proposed in this thesis to handle hyperblock side exits shares some principle similarities to both of these proposals. The reconfigurable co-processor in [41] transfers all of its data back to the main processor upon every hyperblock exit. The approach presented in Chapter 6 does not suffer from this overhead as the loop ISE is tightly coupled to the processor and writes its results to the processor's register file; if a local memory is used, then DMA transfers all written data back to the main memory after the loop executes. In [42], to handle side exits, a predicated jump instruction branches to a *decode block* that points to the appropriate destination. Rather than using a predicated jump, the approached proposed in this thesis stores the different exit points in a local register file. The appropriate index in the local register file corresponding to each side exit is stored in the control logic of the ISE.

The work on loop ISEs is also closely related to the design and implementation of application-specific loop accelerators and software pipelining using hyperblocks, which were originally proposed to facilitate predicated execution. Therefore, the following sections discuss some existing works in these two areas.

Loop Accelerators

Behavioural synthesis for loops expressed in high-level languages is a mature field [43]. Loop accelerators are typically realized as external co-processors that are connected to the main processor by a system bus, unlike ISEs, which are more tightly coupled. [44] proposes a mixed approach to accelerate applications using a mixture of coarse-grained co-processors and fine-grained ISEs.

In the technique proposed in this thesis, unlike in prior loop accelerators, there are no restrictions on the hardware implementations of the AFUs, beyond the I/O interface to the processor. In contrast, most loop accelerator architectures are modeled with traditional arithmetic units and the operations of the loop body

are map to the available units. This constrains the speedup gains, as only one operation can be executed at ever clock cycle.

For example, the *Program-in-Chip-Out (PICO)* project [45] proposed a loop accelerator that consisted of a synchronous array of customized processor datapaths, including register files and a programmable interconnect. Data transfers between datapaths in the accelerator follow the principles of systolic arrays. [46] developed modulo scheduling and hardware synthesis methods for this type of accelerator. The *Streamroller* synthesis system extended these ideas by generating multi-accelerator pipelines, connected by double buffers [47].

To reduce area costs, several groups have proposed methods to generate accelerators that can execute multiple loops [48, 49]. Loop accelerators based primarily on coarse-grained reconfigurable components have also been proposed [50, 51, 52, 53, 54, 55]. The work in this thesis is orthogonal to both of these strains of research. The datapath of loop ISEs can be shared either with the datapath of other loop ISEs or with the datapath of other traditional ISEs. Additionally, loop ISEs generated following the techniques presented in Chapter 6 can be mapped to a wide range of reconfigurable logic units. [56] compared several ASIP acceleration methods for JPEG applications. They compared traditional ISEs, and two co-processors, one generated automatically from a behavioural synthesis tool, and one developed by hand. The best result was the lattermost, which achieved a speedup of 2.57× over an original processor. It is difficult to compare results in this thesis directly against theirs, as the base processors are different, and they used cycle-accurate simulation while for the results reported in this thesis soft-processor emulation was used.

A number of other papers have focused on the appropriate use of dependence analysis and loop optimizations, such as unrolling, pipelining, and vectorization in the context of hardware synthesis of loop-based accelerators [57, 58, 59]. The work performed in this thesis emphasizes hyperblock formation and irregular control flow in the presence of loops, but could easily benefit from these analyses and transformations as well. This thesis does not consider the possibility of unrolling loops prior to pipelining, and the scheduling method is not sophisticated. More aggressive optimization and scheduling techniques could improve the quality of the loop ISEs proposed in this thesis.

Software Pipelining

Software pipelining techniques seek to improve loop performance by overlapping the execution of different iterations; typically, software pipelining is used for VLIW processors that have parallel functional units and require static scheduling at compile-time. The works most relevant to this thesis are those that focus on software pipelining loops that have multiple control flow paths or function calls [60, 61]. The solution presented in this thesis is to generate a hardware pipeline rather than a software pipeline, and to form hyperblocks to best deal with control flow within loops.

2.7 Machine-learning Background

Machine learning attempts to simulate human intelligence in order to allow machines to make accurate predictions based on past observations.

More formally, the goal is to automatically learn a functional mapping between an input that is processed and the generated output.

$$\text{Input} \rightarrow \text{Function} \rightarrow \text{Output}$$

This function can be modeled by extracting statistical phenomena from sample input-output pairs. Such samples represent incomplete information belonging to a complex pattern. The constructed model can later predict the correct output from previously unseen inputs.

Inputs and outputs are vectors whose components can be real-valued numbers, discrete-valued numbers or categorical values.

There are a great number of theories, algorithms, techniques and heuristics that tackle differently the task of learning. In practice, the incorporation of prior knowledge of the problem is crucial in choosing the statistical modeling method that best suits the problem. In this section, some of these approaches are introduced.

Clustering

Clustering is a technique to partition a set of samples into c subsets. It agglomerates samples that are similar. One of the most popular clustering algorithms is known as *k-means*. This is a procedure to classify a given data set into k clusters, where k is a number fixed *a priori*. k centroids, one per cluster, need to be found in the space. Each sample will belong to the cluster of the closest centroid.

The Expectation-Maximization (EM) algorithm [62] was used in the work reported in Chapter 4. First, it randomly assigns items to clusters and then it iterates to find a clustering solution with the smallest in-cluster sum of distances. At every iteration, the centroid of each cluster is found. Then, each item is reassigned to the cluster with the closest centroid. The algorithm terminates when no further reassignments take place. Distances are typically calculated in Euclidean space.

Probability Distribution

Given a set of samples of a variable X, one can extract a statistical model that defines the probability of X falling within a particular interval. A common probability distribution that can approximate many different naturally-occurring distributions is the normal distribution. Due to its convenient properties, data with unknown distribution are commonly assumed to be normal.

A normal distribution is characterized by a mean and a variance, which are calculated from the available samples of the variable [63]. The equation used to describe a continuous probability is the probability-density function, which for normal distributions is the following:

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

The one-dimensional normal distribution can be generalized to k dimensions. In which case, a value for the variable X is expressed as a k-vector. The parameters of such distributions are a mean k-vector μ and a k×k covariance matrix Σ.

Sampling Distribution

From any probability distribution, sample numbers can be generated at random. The objective is to generate values for a variable X that are distributed according to the nature of the original variable [63]. A common method for sampling a multi-dimensional normal distribution is used in the work reported in Chapter 4. It can generate a random vector X from a k-dimensional normal-distribution characterized by a k-vector μ and a k×k covariance matrix Σ. X is computed as follows:

$$X = \mu + LZ$$

L is the Cholesky decomposition of Σ. In other words, L is the unique lower triangular matrix such that $LL^T = \Sigma$ and the diagonal elements of L are positive.

Z is a vector of independent random samples from a normal distribution whose mean vector is a zero vector and covariance matrix is the k×k identity matrix.

Cross-validation

Cross-validation is a statistical method to evaluate or compare models by testing their accuracy with data that has not been used during the training phase [64]. In other words, cross-validation gauges the generalizability of a model. The available data are divided into two segments: one is used to train the model, and the other one is used to validate it.

A typical procedure is called *k-fold cross-validation*. The data is partitioned into k equally-sized segments. One segment is used for validation while the remaining $k - 1$ segments are used to train the model. This process is repeated k times so that every segment is validated. Finally, the validation results are averaged over the rounds.

A special case of k-fold cross-validation, where k equals the number of available training instances, is called *leave-one-out cross-validation*. Therefore, in each iteration nearly all of the data are used for training except for one sample that is used for validation. This procedure is particularly useful when the amount of data available for training is small. It avoids over-fitting the model by maintaining the training set as large as possible. On the other hand, *leave-one-out cross-validation*

tends to be computationally expensive as the training process has to be repeated a large number of times.

k-Nearest Neighbours

k-nearest neighbours is one of the simplest machine-learning algorithms. It bases its predictions on the response of the k closest training samples. Distances are measured in the feature space and, for continuous variables, these are typically calculated as Euclidean distances. The heuristic to transform the response of the nearest neighbours into a prediction for a new instance can be adapted from numerous existing heuristics according to the nature of the problem. Amongst these, a simple method predicts the output corresponding to a new input to be the same as that of its nearest training sample in the feature space. The choice of k depends on the nature of the data and can be selected by heuristics such as cross-validation.

Standard Error of the Mean

The standard error of the mean (SEM) is used to estimate how much variation can be expected from the sample mean when the true mean is unknown.

It is usually estimated as follows:

$$SEM = \frac{s}{\sqrt{n}}$$

Where n is the number of available samples and s is the sample standard deviation. The larger the sample size, the smaller the error.

If the data are assumed to be normally distributed, the SEM can be used to calculate the confidence intervals of the sample mean. For 95% confidence, the lower and upper limits of the mean are:

$$\text{Upper 95\% limit} = \bar{\text{x}} + 1.96 \text{ x } SEM$$

$$\text{Lower 95\% limit} = \bar{\text{x}} - 1.96 \text{ x } SEM$$

Chapter 3

Resource Sharing in ISEs

3.1 Introduction

Customized processor performance generally increases as additional custom instructions are added. However, performance is not the only metric that modern embedded systems must take into account; die area and energy efficiency are equally important. Resource sharing during synthesis of ISEs can significantly reduce the die area and energy consumption of a customized processor and, additionally, it can greatly increase the flexibility and reusability of the extended processor. These properties are achievable by reusing hardware datapaths, thus allowing reconfiguration of the customized unit in order to target different instructions and/or applications.

As more custom instruction can be synthesized within a given area budget, the well known performance and energy benefits of ISEs can be increased [38]. Moreover, as the utilization of the synthesized logic increases, static power consumption, dominant factor in deep sub-micron technologies, is reduced.

Resource sharing involves combining the graph representations of two or more ISEs which contain a similar subgraph. This coupling of multiple subgraphs, if performed naively, can increase the latency of the extension instructions considerably. However, as it is shown in this chapter, an appropriate level of resource sharing provides a significantly simpler design with modest increases in average latency for ISEs. Thus, this chapter presents a new heuristic that controls the degree of resource sharing between a given set of custom instructions in order to

generate optimal trade-offs between area and instruction latency.

This chaper is organized as follows. Section 3.2 presents a simple example that shows the problem that motivated the work presented in this thesis, which is solved by the resource-sharing heuristic that is discussed in detail in Section 3.3. Section 3.4 describes the methodology that is used in order to model the area and the delay of the graphs that are processed by the algorithm. Section 3.5 discusses how the customization of a processor is finalized, making used of the AFU solution that is generated by the resource-sharing heuristic. Section 3.6 describes the experiments that were carried out. Section 3.7 presents the results that were obtained, and Section 3.8 concludes this chapter.

3.2 Motivation

It is possible to find a large number of potential extension instructions in embedded applications. Also, when extending an instruction set to cover a complete class of applications, a larger number of extension instructions is expected to be identified, effectively representing the union of the extension instructions required by each application in the class. Each new instruction adds to the die area of the system. However, to avoid bloating the die area with a large number of extension instructions, it is important to identify and exploit any commonality between instructions and, where possible, to share hardware resources when this represents a good trade-off between die area and execution time. The problem addressesed in this chapter is how to merge such a collection of graphs to reduce the overall die area, whilst minimizing the increase in execution latency.

Depending on the alignment of shareable paths in ISE graphs, it might be found that the resulting latency is almost unchanged after merging, or that latency increases significantly for some or all merged operations. Naturally, it is desired to avoid merging a frequently used ISE with an infrequently used ISE if such a merge would add to the latency of the one that is frequently used. Thus, the optimization process becomes highly complex when instruction latencies may be modified by merging.

Figure 3.1 illustrates a situation that frequently occurs in the process of merging the hardware implementation of a set of ISEs. (a) and (b) show the graphs

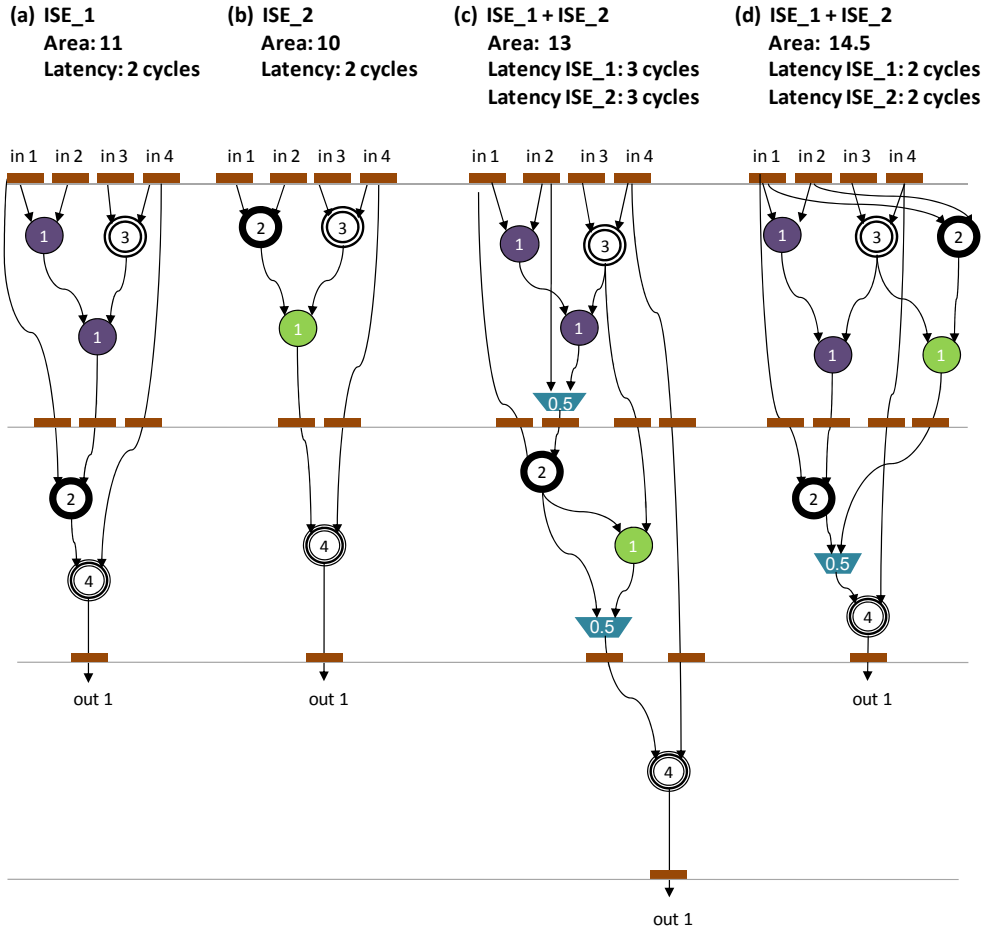

Figure 3.1: Example of path merging. ISEs in (a) and (b) can be merged into a single multi-function datapath. (c) represents the solution with minimum area, however, the latency of both ISEs is increased. Instead, (d) represents a merging solution where slightly fewer area savings are obtained but the latency of the ISEs remains unchange.

that represent ISE_1 and ISE_2 respectively. (c) corresponds to the minimum-area common-graph that is obtained in order to implement ISE_1 and ISE_2 in a single datapath. Multiplexers are used to isolate the graphs according to the operation they implement. As shown in (c), the datapath generated is significantly longer. Thus, the latency of both ISEs has increased with the merge and therefore, the speedup that each ISE yields decreases. Another solution to merging ISE_1 and ISE_2 is shown in (d). In (d), ISE_1 and ISE_2 share fewer resources, and thus, the area of the solution is greater than that of the solution in (c). However, the latency of the instructions remains unchanged after the merge. Therefore, the

speedups obtained from implementing ISE_1 and ISE_2 are maximum and at the same time area savings are obtained.

On the other hand, both solutions: (c) and (d) represent different trade-offs between ISE latency and area. (c) represents a solution that requires less area than (d), but (d) represents a solution that generates greater speedups than (c). Thus, as each solution is characterized by two important metrics: ISE latency and area, there is no single best solution. Instead, there are trade-offs in which the more resources are shared, the more the speedup obtained from the ISEs is likely to decrease. Therefore, unlike previous resource-sharing methods, the techniques proposed in this chapter focus on exploring the design space of available trade-offs between ISE latency and area.

3.3 Parametric Resource-sharing Heuristic

The proposed resource-sharing heuristic is a path-based algorithm that operates on a DFG. A DFG is, in turn, a directed graph with no cycles or a Directed Acyclic Graph (DAG). These graphs are represented by a set of vertices V and a set of edges E, where vertices can be inputs, operators or outputs, and edges indicate the data dependencies between them. A path within a DFG is a sequence of vertices that traverses the graph, through a subset of edges, from an input to an output.

A collection of such DFGs represents a set of compound operators requiring an efficient micro-architectural implementation. As it will be shown later in this chapter, there are many ways in which the operators of such DFGs can be shared, resulting in a complex design-space. The goal of this algorithm is to *expose* this design space, allowing other tools to search the space in order to select the most appropriate cost-performance point according to higher-level design constraints.

Algorithm Parameters To allow the algorithm to find many alternative solutions it is parameterized by three real-valued thresholds and two binary switches, which trigger different behaviors during the process of merging DFGs. The threshold parameters are α_T, β_T and θ_T, each taking a real value in the range $[0, 1]$. Their combined role is to limit the increase in the ISE execution delay in relation

to the area saved by merging operators. The two binary parameters are *multiOp* and *grouping*. The *multiOp* parameter controls the creation of multi-operation vertices from similar, but not identical, operators. The *grouping* parameter determines whether or not certain idiomatic operator groupings will be recognized and exploited during the merging process. Such groups are treated atomically and are expected to allow downstream optimizations amongst their components when logic synthesis is performed. These parameters are introduced in detail later in this section.

Sharing Resources Resource sharing is induced by the search for maximum-area common-substrings between two paths belonging to different graphs. A vertex is common between two paths when the operators of the two vertices are the same. There are also some special cases where vertices with different operators can be merged, as explained in Section 3.3. Area reduction is maximized by the fact that a substring is chosen according to the expected area saved by merging two instances of that substring, rather than by simply considering the substring length. The *area* of a substring is therefore defined as the sum of the areas of each operation within the substring. As the sharing of this common substring will remove one instance of the substring from the final system, this area measurement approximates the expected area savings. Function `findMaxSubstring`, in Algorithm 1, shows how the search for a maximum-area common-substring between two paths can be implemented.

Algorithm Phases The main merging process is divided into a *global* and a *local* phase. A worked example to illustrate these phases can be found in Figure 3.2. The purpose of the global phase is to locate the maximum-area common-substring between any pair of graphs. This identifies good initial candidates for merging and defines how they will be merged. The purpose of the local phase is to take a merged pair of graphs from the global phase and search for additional pairs of vertices in the merged graph that can themselves be merged. In each phase, there is an exhaustive search for a maximum-area common-substring, comparing all pairs of paths belonging to different graphs. In the global phase, a maximum-area common-substring, referred to as *MaxStrGlobal*, is chosen from all of the

available graphs. If G_x and G_y contain *MaxStrGlobal*, G_x and G_y can be merged through *MaxStrGlobal* to form a combined graph G'. The local phase performs further merging in G' by iteratively finding new common substrings that can be merged between G_x and G_y. The common substrings found during the local phase are referred to as *MaxStrLocal*.

Overview A formalized definition of the `merge` function, is given in Algorithm 4. This function operates on a set of m graphs G_{out}, where initially $m = n$. For each $G_i \in G_{out}$, a set of paths P_i is created from all possible paths in G_i. P aggregates all sets of paths from P_1 to P_m. Function `findMaxStrGlobal`, shown in Algorithm 2, finds the maximum-area common-substring *MaxStrGlobal* by comparing every path in P_i with all other paths that belong to $P_{j \neq i}$. The `merge` function then creates a candidate replacement graph G' by merging the graphs G_x and G_y that contain *MaxStrGlobal*. The local phase, defined by function `findMaxStrLocal` shown in Algorithm 3, then searches iteratively for *MaxStrLocal*. This considers all pairs of paths with one path from P_x and one from P_y. Nodes that were shared already between G_x and G_y in G' are excluded from the search. The local search excludes any *MaxStrLocal* that creates a cycle in the merged graph. This iterative local search finishes when no further *MaxStrLocal* instances can be found. Lines 15–21 of Algorithm 4 use parameter θ_T to decide whether G' should be permitted to replace G_x and G_y. The reason for introducing this threshold test is explained in Section 3.3. The sequence of global and local phases repeats until no further *MaxStrGlobal* is found during the global search, or when there is only one graph remaining in G_{out}.

Multi-operation Vertices

In practice it is common to find vertices that perform similar but different operations. In some cases these vertices could be merged with a small overhead to produce a multi-operation vertex. For example, an adder and a subtracter can be implemented by a vertex that represents a generic functional unit of similar complexity to both the adder and the subtracter.

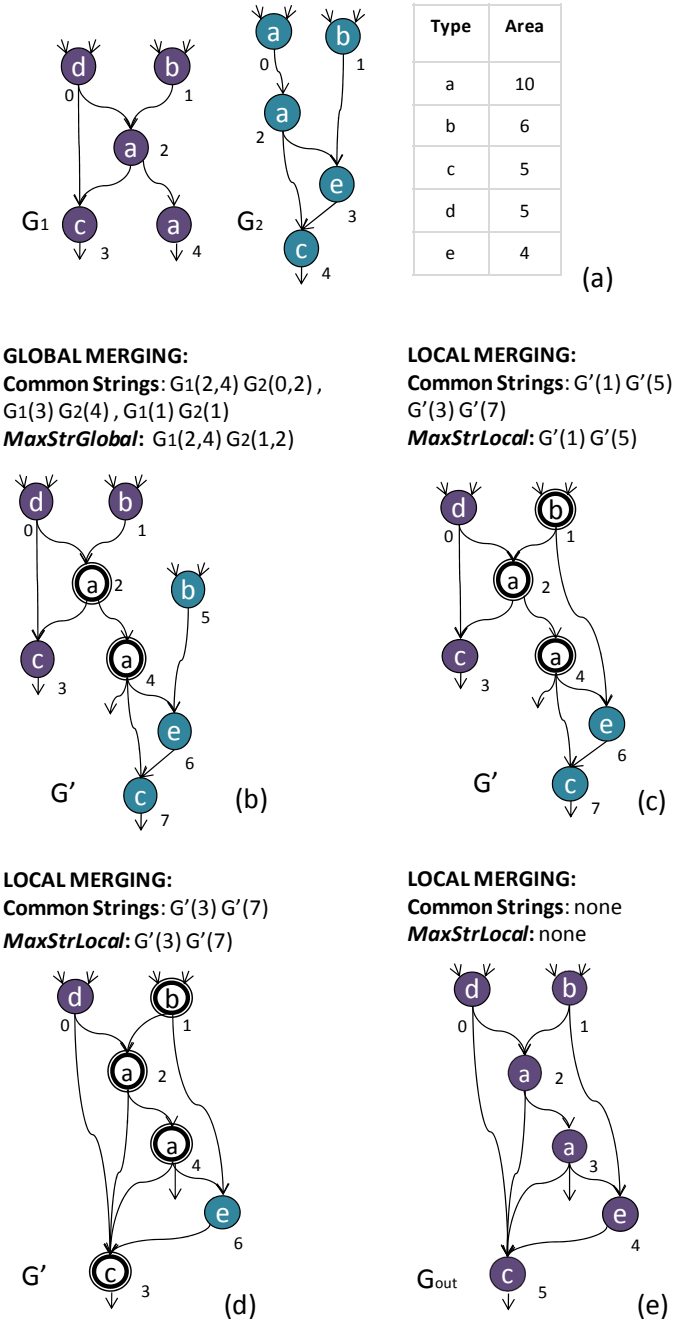

Figure 3.2: Example of merging two DFGs. In (a), two graphs, G_1 and G_2, are presented as candidates for merging, together with the area of each kind of operator. In (b), global merging is performed after finding *MaxStrGlobal* between G_1 and G_2. In (c), (d) and (e), local merging is performed iteratively until no *MaxStrLocal* is found.

Algorithm 1: Subroutine to find maximum common area substring between two paths

```
1: findMaxSubstring (p_i, p_j, multiOp)
   {Op_x represents the operator type of vertex x.
   F_x represents the operator family of vertex x.
   |X| represents the length of substring X.
   A_x represents the area of substring or vertex x.
   getAreaSavings(x, y) returns the area savings obtained from merging operator
   type x and operator type y into one multi-operation vertex, as indicated in
   equation 3.1}
2: for all common Substrings between p_i and p_j do
3:    if multiOp = 1 then
4:       newType = FALSE
5:       {∀ l from 1 to |Substring|, Substring[i] has been formed from p_i[a + l] and
         p_j[b + l] where F_{p_i[a+l]} = F_{p_j[b+l]}}
6:       for l = 1 to l = |Substring| do
7:          if Op_{p_i[a+l]} ≠ Op_{p_j[b+l]} then
8:             A_Substring += getArea(p_i[a + l], p_j[b + l])
9:             newType = TRUE
10:         else
11:            A_Substring += A_Substring[l]
12:         end if
13:      end for
14:   else
15:      {∀ l from 1 to |Substring|, Substring[i] has been formed from p_i[a + l] and
         p_j[b + l] where Op_{p_i[a+l]} = Op_{p_j[b+l]}}
16:      for l = 1 to l = |Substring| do
17:         A_Substring += A_substring[l]
18:      end for
19:   end if
20:   if NOT (newType AND (|substring| = 1)) then
21:      if A_Substring > A_MaxSubstring then
22:         MaxSubstring = Substring
23:      end if
24:   end if
25: end for
26: return MaxSubstring
```

The resource-sharing algorithm can combine a pair of vertices whose operations are different provided they can be implemented by a single unit that performs both operations with an acceptable overhead in area and delay. Vertices with different operations may be merged, if those operations belong to the same *family*. These families of operations are defined in Table 3.1.

Algorithm 2: Subroutine to find *MaxStrGlobal*

```
1: findMaxStrGlobal (P, multiOp)
   {A_X represents the area of substring X}
2: for all P_x ∈ P do
3:    for all P_{y≠x} ∈ P do
4:       for all p_i ∈ P_x do
5:          for all p_j ∈ P_y do
6:             MaxSubstring = findMaxSubstring (p_i, p_j, multiOp)
7:             if A_MaxSubstring > A_MaxStrGlobal then
8:                MaxStrGlobal = MaxSubstring
9:             end if
10:         end for
11:      end for
12:   end for
13: end for
14: return MaxStrGlobal
```

Algorithm 3: Subroutine to find *MaxStrLocal*

```
1: findMaxStrLocal (P_x, P_y, G', multiOp)
   {A_X represents the area of substring X}
2: for all p_i ∈ P_x do
3:    for all p_j ∈ P_y do
4:       MaxSubstring = findMaxSubstring (p_i, p_j, multiOp) such that
         MaxSubstring does not create cycles in G' when merged {Every vertex ∈
         MaxSubstring has not been merged between x and y}
5:       if A_MaxSubstring > A_MaxStrLocal then
6:          MaxStrLocal = MaxSubstring
7:       end if
8:    end for
9: end for
10: return MaxStrLocal
```

Family 1	Integer addition and subtraction
Family 2	FP addition and subtraction
Family 3	Shift left and shift right
Family 4	Logic operations: OR, AND, NOT, XOR
Family 5	Integer comparisons: $<$, $>$, $=$, $\leq$, $\geq$, $\neq$
Family 6	FP comparisons: $<$, $>$, $=$, $\leq$, $\geq$, $\neq$

Table 3.1: Families of operations

If two vertices are merged under these conditions there will be significant area savings since the synthesis of the multi-operation unit will be cheaper than building them separately. However, this flexibility, which increases the area, delay

Algorithm 4: Subroutine for global and local merging

```
1: Merge (G_out, θ_T, multiOp, groups)
2: S* ← ∅
3: repeat
4:    if groups = 1 then
5:       find paths in each graph of G_out that only contain recently created MAC
         and/or 4-adder type of vertices: P = {P_1...P_m}
6:    else
7:       find paths in each graph of G_out: P = {P_1...P_m}
8:    end if
9:    MaxStrGlobal = findMaxStrGlobal (P, multiOp) such that
      MaxStrGlobal ∉ S*
10:   form G' merging graphs G_x and G_y that contain MaxStrGlobal
11:   repeat
12:      MaxStrLocal = findMaxStrLocal (P_x, P_y, G', multiOp)
13:   until no MaxStrLocal is found
14:   find critical path and area of G'
15:   find θ_x and θ_y
16:   if θ_x < θ_T and θ_y < θ_T then
17:      replace G_x by merged graph G'
18:      remove G_y from G_out
19:      m ← m − 1
20:   else
21:      S* ← S* + MaxStrGlobal
22:   end if
23: until no MaxStrGlobal is found
24: return G_out
```

and complexity of the design by requiring some additional logic and control, might be unnecessary when there are sufficient vertices amongst the graphs that share the same operations to exploit resource sharing. For this reason, the use of multi-operation vertices must be controlled somehow.

The creation of multi-operation vertices is governed by the parameter referred to as *multiOp*. When this parameter is TRUE, a new multi-operation vertex can be created, but only if it belongs to a common substring containing more than one vertex. This condition enables the sharing of vertices in a multi-operation vertex only when additional savings can be found through the sharing of interconnect between vertices, which in many cases will offset the overheads of a multi-operation vertex. Every common substring is assigned an area-saving value that will be used later in the process to rank that substring against others when choosing

MaxStrGlobal and *MaxStrLocal*. The larger the area-saving of a substring the more likely it is to be chosen as *MaxStrGlobal* or *MaxStrLocal*.

The area-saving of a substring is computed as the sum of the areas associated with the operators of each vertex in the substring. However, the area assigned to a vertex created from vertices with different operators is a special case. In general, a multi-operation vertex will have an area that is slightly larger than the maximum of the areas of the two operators being combined. If vertex x and vertex y are to be merged into vertex xy, when operator x is different to operator y, the area-saving value assigned to vertex xy is given by Equation 3.1, below:

$$\texttt{getAreaSavings}\,(x, y) = \texttt{A}_x + \texttt{A}_y - \texttt{A}_{xy} \tag{3.1}$$

where $\texttt{A}_x$ and $\texttt{A}_y$ represent the area of the vertex x and vertex y respectively, and $\texttt{A}_{xy}$ represents the area of a unit that is able to perform operations of both vertex x and vertex y.

Controlling the Area-Latency Trade-off

When a pair of graphs have been selected as the next most profitable candidates to be merged, the heuristic must decide whether the increased function unit latency resulting from the merge is sufficiently offset by the area savings to make the merge *beneficial*. As there is no absolute metric of whether any given trade-off is beneficial, the metric θ to quantify the area-latency trade-off is introduced.

When candidate graphs G_x and G_y can be merged to create graph G', θ_x and θ_y are computed according to Equations 3.2 and 3.3, below:

$$\theta_x = \frac{L_{G'} - L_x}{L_{G'}} \times \left(\frac{A_{G'}}{A_x + A_y} \right) \tag{3.2}$$

$$\theta_y = \frac{L_{G'} - L_y}{L_{G'}} \times \left(\frac{A_{G'}}{A_x + A_y} \right) \tag{3.3}$$

where $L_{G'}$, L_x and L_y are respectively the critical paths of G', G_x and G_y,and $A_{G'}$, A_x and A_y are respectively the areas of G', G_x and G_y. The first term in θ represents the relative decrease in latency perceived by *not* performing the merge,

whereas the second term represents the area savings that *do* result from merging G_x and G_y.

Every time a *MaxStrGlobal* is found, a G' is formed and local merging is applied. When no further merging is possible in G', θ_x and θ_y are calculated. If either θ_x or θ_y is greater than θ_T, G' will not be considered and other opportunities for sharing will be searched for between the available graphs. Thus, the global phase starts again with G_{out} unchanged, forcing the search for another *MaxStrGlobal*. The *MaxStrGlobal* substrings exceeding the θ_T threshold, and that have therefore been discarded, are recorded in S^* and are not eligible for merging.

Figure 3.3 depicts an example containing three input graphs. Figure 3.3(a) and Figure 3.3(b) show respectively the three original graphs, and the areas and latencies of each of the operators in these graphs. In the first global phase, all common substrings amongst the three graphs are found and listed in Figure 3.3(c). The common substring with greatest area is chosen as *MaxStrGlobal*. Since several substrings have the same maximum area, any one of them could be chosen. If the substring $G_1(3)G_2(0)$ is chosen as *MaxStrGlobal*, then graphs G_1 and G_2 are merged through $G_1(3)$ and $G_2(0)$ as illustrated in Figure 3.3(d). This also shows the subsequent local phase in which vertices $G'(1)$ and $G'(4)$ are merged. Then, θ is calculated for G_1 and G_2 using Equations 3.2 and 3.3. In this example it is assumed that either θ_1 or θ_2 is greater than θ_T, so G' is rejected as a candidate for merging and is added to the set S^* as a forbidden substring. The next most profitable candidate substring is then taken from the list shown in Figure 3.3(c) and considered for merging.

In this example $G_1(0)G_2(0)$ is selected as the next *MaxStrGlobal*, suggesting that G_1 and G_2 should be merged through $G_1(0)$ and $G_2(0)$. This is illustrated in Figure 3.3(e), where again the local phase is also shown. As this merge does not affect the latency of the original graphs, $L_{G'}$=L_1=L_2, and hence θ_1 and θ_2 are zero, and the condition $\theta_1 < \theta_T$ and $\theta_2 < \theta_T$ will clearly be met. This allows the merge to be committed, creating the intermediate set G_{out} shown in Figure 3.3(f). A further iteration of the outer search loop then takes place, from which all candidate substrings are found and listed in Figure 3.3(g). According to their associated areas, the substring $G_1(1)G_3(3)$ is selected as *MaxStrGlobal*. In Figure 3.3(h), G_1 and G_3 are merged through $G_1(1)$ and $G_3(3)$ forming a new G'.

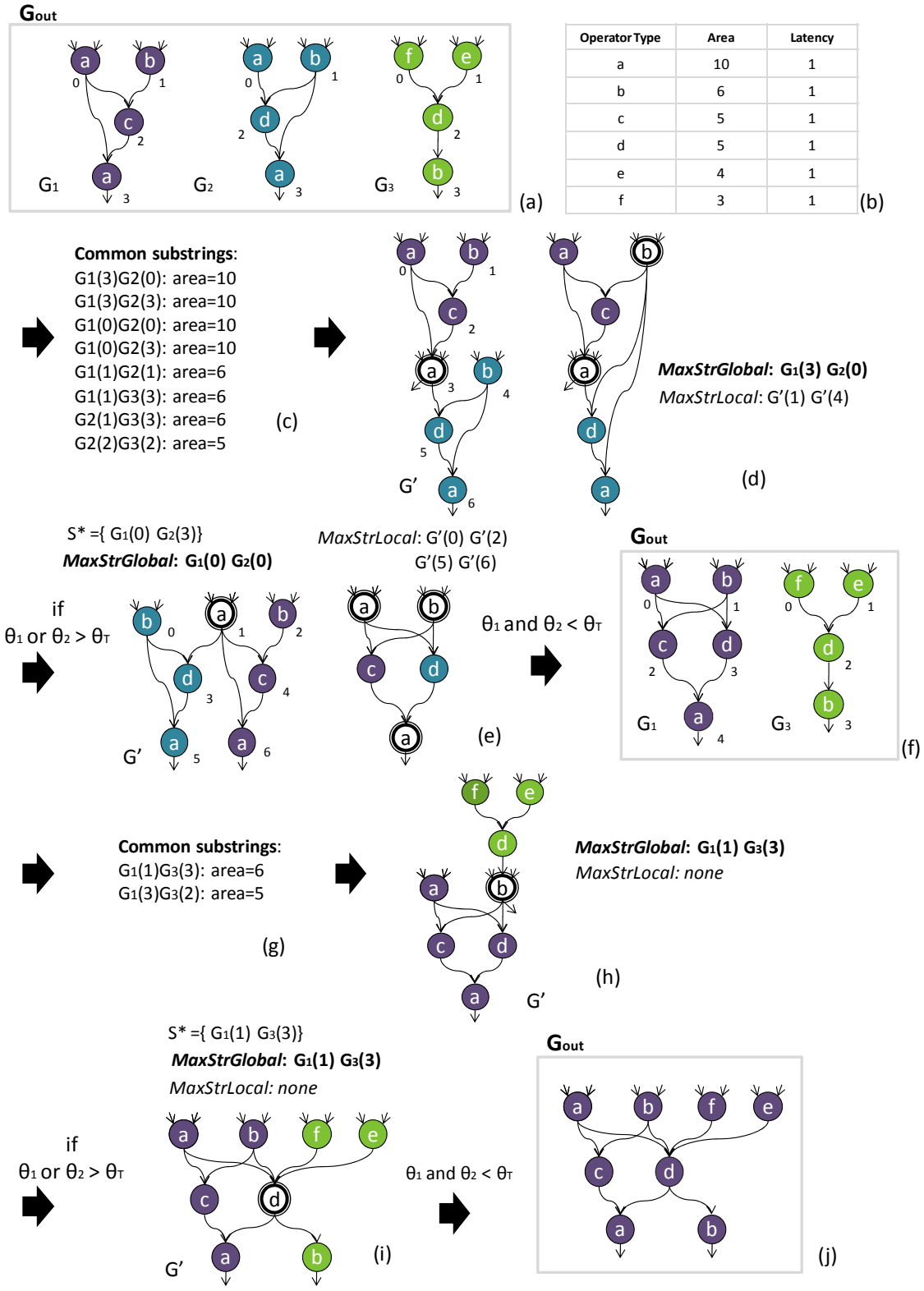

Figure 3.3: Example of merging several graphs using θ_T as a constraint to protect the latency of the instructions.

Local merging is not performed since there is no *MaxStrLocal* in this G'. θ is then calculated for G_1 and G_3, and assuming that either θ_1 or θ_3 is greater than θ_T, this merge is also rejected. The substring $G_1(1)G_3(3)$ is also recorded in S^* and another substring is selected from the list given in Figure 3.3(g). In this example, $G_1(3)G_3(2)$ is then selected as *MaxStrGlobal*, as shown in Figure 3.3(i), and hence G_1 and G_3 are merged through $G_1(3)$ and $G_3(2)$. Since L'=L_1 and L'=L_3, θ_1 and θ_3 are zero. Thus the merge is committed, forming the final merged graph shown in Figure 3.3(j). Multiplexers have been omitted from this example to simplify the representations though in practice the algorithm will insert these as required.

Note that the opportunity to merge vertices $G_1(1)$ and $G_3(3)$ in Figure 3.3(g) is *not* taken, despite the fact that it saves more area than the alternative of merging vertices $G_1(3)$ and $G_3(2)$. This illustrates how the threshold θ_T limits the increase in latency that is tolerated for a given area saving.

Controlling the Execution-time Impact of Merging

If a graph G_i originates from a frequently-executed section of code, then the degree to which the latency of G_i is increased by merging G_i with other graphs should be controlled in some way. Although the θ_T threshold test is important for preventing the pairwise merging of two graphs when it represents a poor trade-off between area savings and increased latency, this is not sufficient.

Consider the example shown in Figure 3.4(a), where G_1, G_2, G_3 and G_4 can all be merged together. This increases the latency of the original computations in G_1, and also to a lesser extent those in G_2 and G_3. If G_1 is a frequently executed graph, then the resulting G_{out} is not a good solution. However, this may not be detected by the θ_T test, as each individual merge may represent a good trade-off between area-savings and the *incremental* increase in latency. To counteract this effect, the proposed heuristic computes an additional metric α_i for each G_i in the original G_{in}, given by Equation 3.4, below:

$$\alpha_i = F_i \times \frac{L'_i - L_i}{L'_i} \times (1 - M_i) \tag{3.4}$$

where F_i is the normalized execution frequency of G_i, i.e., the execution frequency of G_i divided by the maximum execution frequency in the set G_{in}. L_i is the

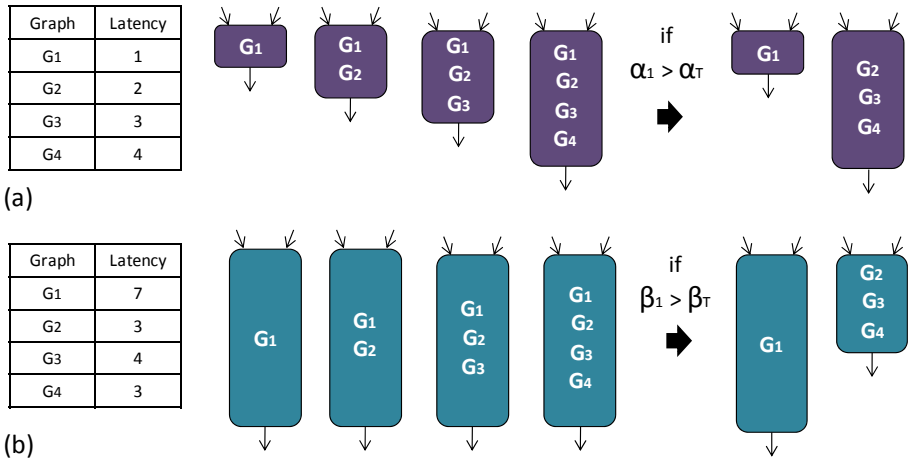

Figure 3.4: Abstract example of merging graphs using α_T (a) and β_T (b).

original latency of G_i, i.e., before the merging process. L'_i is the latency of G_i after being merged with other graphs. M_i is the percentage of area corresponding to operations in G_i that can be merged with other graphs, divided by the total area that could be merged in the whole process.

When all merging opportunities have been found, each α_i is compared with a parametric threshold α_T and, if it exceeds the threshold, the corresponding G_i is excluded from the set of input graphs when the merging process is repeated a second time. α_i depends on: the latency increase perceived by G_i in its merged form; its execution frequency; and the amount of merging that it can sustain. The effect of the α_T test is to leave G_i unmerged if the merging process would increase its latency beyond an acceptable threshold.

A similar issue exists when a graph with high latency is merged with a set of graphs that have lower latency. Consider the example shown in Figure 3.4(b), where G_1 has the highest latency and the other graphs have lower latencies that are all quite similar. Again, depending on the order in which merges take place, the first pass may merge all four graphs into one high-latency graph. Or perhaps, having merged G_1 with G_2, the opportunities for merging G_2 with G_3 or G_4 may no longer exist, due to the θ_T test. Therefore, an additional metric is introduced: β_i for each G_i in the original G_{in}. It is given by Equation 3.5, below:

$$\beta_i = \frac{|\hat{L} - L_i|}{\max_{j=1}^{m} L_j} \times (1 - M_i) \tag{3.5}$$

Algorithm 5: Subroutine to merge groups

```
1: mergeGroups (G_out, θ_T, multiOp)
2: for all G_i ∈ G_out do
3:    identify possible MAC or 4-in adder instances and group each into one vertex
4: end for
5: merge (G_out, θ_T, multiOp, 1)
6: for all G_i ∈ G_out do
7:    revert grouping of MAC and 4-in adder instances that are not shared
8: end for
9: for all G_i ∈ G_out that has been merged do
10:   repeat
11:      find all paths in G': P'
12:      MaxStrLocal = findMaxStrLocalG'(P', G', multiOp)
13:      merge MaxStrLocal in G'
14:   until no MaxStrLocal is found
15: end for
16: return G_out
```

where n is the number of input graphs and:

$$\hat{L} = \frac{\sum_{j=1}^{n} L_j}{n} \tag{3.6}$$

The β_T test is applied at the same time as the α_T test, and its effect is to exclude input graphs from the merging process if their latency is much larger than the other ISE graphs. This is characterized by the difference between the average latency of all input graphs and the latency of the graph in question. If β_i is greater than β_T, G_i will not be considered during the merging process, thus preventing G_i from affecting the latency of the other graphs.

Vertex Grouping

There are certain operator sequences that can be combined during logic synthesis to yield smaller and faster solutions than their individual components. For example, the marginal cost of an adder following a multiplier is less than the cost of an adder in isolation. [65] present examples of how arithmetic optimizations can reduce the combined latency of sequential operations. Modern logic synthesis tools have the ability to perform similar arithmetic optimizations, such as folding ADD or SUBTRACT operations into the carry-save tree of a combinational multi-

Algorithm 6: Subroutine to find *MaxStrLocal* for `mergeGroups`

```
1: findMaxStrLocalG' (P', G', multiOp)
   {A_X represents the area of substring X}
2: for all p_i ∈ P' do
3:    for all p_{j≠i} ∈ P' do
4:       MaxSubstring = findMaxSubstring (p_i, p_j, multiOp) such that
         MaxSubstring does not create cycles in G' when merged {Every vertex ∈
         MaxSubstring has been created from vertices that originate from different
         input graphs}
5:       if A_MaxSubstring > A_MaxStrLocal then
6:          MaxStrLocal = MaxSubstring
7:       end if
8:    end for
9: end for
10: return MaxStrLocal
```

plier. Similarly, multiple-input adders could be implemented as a Wallace-tree compressor followed by a carry-propagate adder.

However, if graphs are merged in order to share vertices, the resulting graph will often have multiplexers at the inputs to those shared vertices. These multiplexers will normally prevent a synthesis tool from combining the operators within adjacent vertices. For this reason, a boolean parameter, referred to as *grouping*, is introduced in the heuristic. Parameter *grouping* controls whether operator groups should be identified and retained instead of trying to merge each operator independently.

The heuristic is aware of common idiomatic groupings, recognizing a MAC for example when there is a multiplier connected uniquely to an adder, as shown in Figure 3.5(a). In the same way, a multiple-input adder is recognized when consecutive adders are connected. As more adders are grouped, the savings in area and delay increase. However, delay savings can be negated by the late arrival of the inputs of the deepest adders. Moreover, as more adders are combined, it becomes more difficult to find such instances in the input graphs. For these reasons, a 3-adder grouping is chosen for the exploration of this feature. Grouping three adders generates a 4-input adder and this can occur in two ways, as depicted in Figure 3.5(b) and 3.5(c). Vertex grouping is implemented by the `mergeGroups` function, illustrated in Algorithm 5.

Table 3.2 shows the area and delay of these operations when treated as an

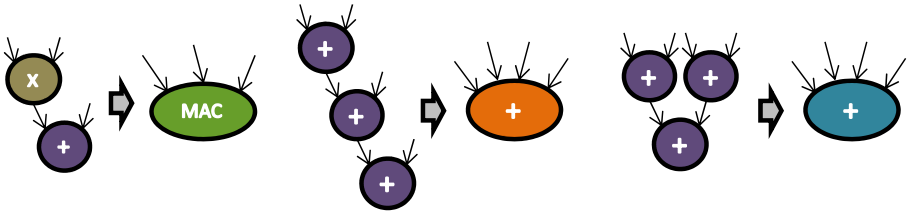

Figure 3.5: Operators that can be collapsed into one vertex when the *grouping* feature is enabled during the resource-sharing exploration.

atomic unit and when synthesized separately. These values were obtained by synthesizing the operators using Synopsys' DC Ultra synthesis tool and a 0.13μm standard cell library, optimizing for minimum delay.

Function `mergeGroups` starts by identifying and grouping all possible instances of MACs and 4-input adders in every $G_i \in G_{out}$. Then, function `merge` is called with its last parameter, *groups*, equal to 1. This causes every $P_i \in P$ to be filled with all the paths found in $G_i \in G_{out}$ that contain only the newly-created instances of MACs and 4-input adders. Hence, this function aims to merge only those groups of operators. Following this, all MAC and 4-input adder instances that have not been merged will be ungrouped, and their individual operators will be now exposed to the regular merging process.

Grouped Units	Area (gates)		Delay (ns)	
	Separate	Grouped	Separate	Grouped
MULTIPLIER + ADDER	5346	4408	3.42	2.69
3 ADDERS	1704	1362	2.64	1.39

Table 3.2: Area and delay of grouped versus separate operators

Top-level Description of the Heuristic

A top-level description of the proposed heuristic is illustrated in Algorithm 7. The main function, called `runRS`, receives as inputs a set of n DFGs G_{in}, where each $G_i \in G_{in}$ represents an ISE to be synthesized, and values for the five parameters: θ_T, α_T, β_T, *multiOp* and *grouping*. The merging processes operate on G_{out}, which is initially copied directly from G_{in}. Consequently, before any resource sharing is

Algorithm 7: Parametric resource-sharing heuristic

```
1: runRS (G_in, α_T, β_T, θ_T, multiOp, grouping)
2: G_in ← {G_1 ... G_n}
3: G_out ← G_in
4: if grouping = 1 then
5:    mergeGroups (G_out, θ_T, multiOp)
6: end if
7: G* ← ∅
8: merge (G_out, θ_T, multiOp, 0)
9: for all G_i ∈ G_out do
10:   insert multiplexers needed in G_i
11:   find critical path and area of G_i
12:   for all G_j ∈ G_in that has been merged in G_i do
13:      find β_j and α_j
14:      if α_j > α_T or β_j > β_T then
15:         G* ← G* + G_j {exclude G_j}
16:      end if
17:   end for
18: end for
19: if G* ≠ ∅ then
20:   G_out ← G_in - G*
21:   if grouping = 1 then
22:      mergeGroups (G_out, θ_T, multiOp)
23:   end if
24:   merge (G_out, θ_T, multiOp, 0)
25:   for all G_i ∈ G_out do
26:      insert multiplexers needed in G_i
27:   end for
28: end if
29: return G_out
```

applied, the number of input graphs is the same as the number of output graphs, *i.e.*, $m = n$, where $m = |G_{out}|$.

When *grouping* is enabled, the `mergeGroups` function described in Algorithm 5 identifies operator groups and merges instances of those groups where appropriate. The `merge` function then searches for opportunities to merge graphs in G_{out}, applying the local and global phases using θ_T as a constraint, as explained in Section 3.3. When no further suitable common substrings can be found between graphs in G_{out}, or when there is only one graph remaining, the values of α and β are calculated for every $G_i \in G_{in}$. These determine if each input graph is to be left unmerged according to the given threshold parameters α_T and β_T, as ex-

plained in Section 3.3. The set of graphs G^* keeps track of the graphs that are excluded from merging. If $G^* \neq \emptyset$, the `merge` function is called again to recompute the merging process over all graphs that are *not* excluded. Finally, the set of graphs G_{out} contains the result of resource sharing, where some or all of the graphs representing individual ISEs have been merged and the logic for common operators is therefore shared.

Multiplexer Insertion

Multiplexers have to be added at the inputs of a vertex when it has more than one predecessor per input as a result of sharing the resource associated with that vertex. In the case of vertices with just one input the solution is straightforward: an N-input multiplexer is added when there are N unique predecessors, and the number of selection bits is given by $\lceil log_2 N \rceil$. Vertices with two inputs can potentially have multiplexers in each input. Since multiplexers are inserted as a result of merging two different graphs, the predecessors are always from different operators. For this reason the input balancing problem exposed in [66] is not an issue in this process. Commutativity of operations is exploited where any ISE input is part of the inputs of a two-input vertex in order to balance the assignment of ISE inputs to the multiplexers. The final area of the merged graphs in the experiments presented in this chapter includes the contribution of all multiplexers inserted.

Algorithm Runtime

The runtime of the resource-sharing algorithm depends on the characteristics of each graph, which is most likely different for every graph in the input set G_{in}. In order to estimate the runtime of some of the functions, average values across the input sets will be taken. Thus, L represents the average length of a path, P represents the average number of paths per graph and V represents the average number of vertices per graph.

It has been shown in Section 3.3 that the datapath merging algorithm is divided into two phases: global and local. The global phase is executed as many times as there are graphs in G_{in}. Every time a global phase is performed, a local

phase takes place. The local phase, where two graphs are fully merged after a common string has been merged between them, is executed V times, in the worst case. This would be the case if all of the vertices are merged and all are merged individually.

The innermost and most executed function in the algorithm searches for the Maximum Common Substring (MCS) between two paths, requiring $O(L^2)$ comparisons. In the global and local phases, the MCS is computed between all pairs of paths that belong to different graphs, which requires $O(P^2L^2)$ operations. The P^2 element arises from the consideration of all possible pairs of paths, and the L^2 element is the cost of the MCS. Therefore, the runtime of the algorithm is $O(L^4P^2)$.

The value of the parameters, α_T, β_T and θ_T, also influence runtime. When $\theta_T \neq 1$, the global phase can be executed more times than the number of graphs in G_{in}. Whenever two graphs are merged, the global phase could have been executed as many times as there are strings in the set S^*, which contains banned substrings. In order to prevent a long runtime due to very low values of θ_T, which lengthens the search for eligible substrings, S^* can be limited to a maximum size. When the number of substrings in S^* reaches its limit, the merging process is finished. Conversely, when θ_T and α_T are different from 1, the runtime tends to decrease, as fewer graphs are likely to be merged.

Theoretically, the path enumeration within a graph can be exponential in the number of vertices. However, in practice, the average number of paths per graph is several orders of magnitude less than the calculated worst case.

Due to the nature of the problem tackled in this work, the runtime of the algorithm is not critical. Input graphs correspond to data-flows taken from software applications to be implemented in hardware, which means that graphs are constrained in several ways. Graphs are constrained in hardware, since they are to be implemented in silicon and area is always limited. Overall graph sizes for each ISE are also limited in practice by the disruptive influence of control flow instructions and memory operations.

Design Space

Section 3.3 has described a heuristic to find a resource-sharing solution given a set of input graphs. In addition to input graphs, the algorithm uses five parameters to guide decisions that will have an impact on the characteristics of the output graphs. These parameters (α_T, β_T, θ_T, *multiOp* and *grouping*), have been analyzed in the previous sections and create a multi-dimensional space of possible solutions as their values are changed. The next section demonstrates that this space can be explored in order to find optimal implementation alternatives based on resource sharing.

3.4 Area and Delay of Graphs

The resource-sharing heuristic depends on the area and delay of each operator instance in each graph when deciding whether to merge. However, it is infeasible to perform full logic synthesis for each graph during the execution of resource-sharing algorithm. Therefore, a piecewise-linear model of area and delay has been developed. Each operation is modeled in terms of four discrete points in its curve of area versus delay. Specific area and delay values are then found by linear interpolation from these four discrete points. Figure 3.6 shows the curves that were obtained for a selection of common operators.

In order to obtain precise area and delay values for the operators, Register-Transfer Level (RTL) synthesis was performed by using the Synopsys Design Compiler Ultra (DC) synthesis tool and a 0.13μm standard cell library. DC takes an RTL hardware description in Verilog and the standard cell library as inputs, and produces a gate-level netlist, as well as reports on area and timing of the resulting netlist. In addition, design constraints can be configured in order to create a netlist that meets the design requirements.

The four design points of the operators were obtained by synthesizing each operator four times under different constraints. Figure 3.7 shows the Verilog module used to characterize a 32-bit adder. Similar modules were used for the rest of the integer arithmetic and logic operators. Figure 3.8 shows the Verilog module used to characterize a 32-bit floating-point adder. Floating-point operations are im-

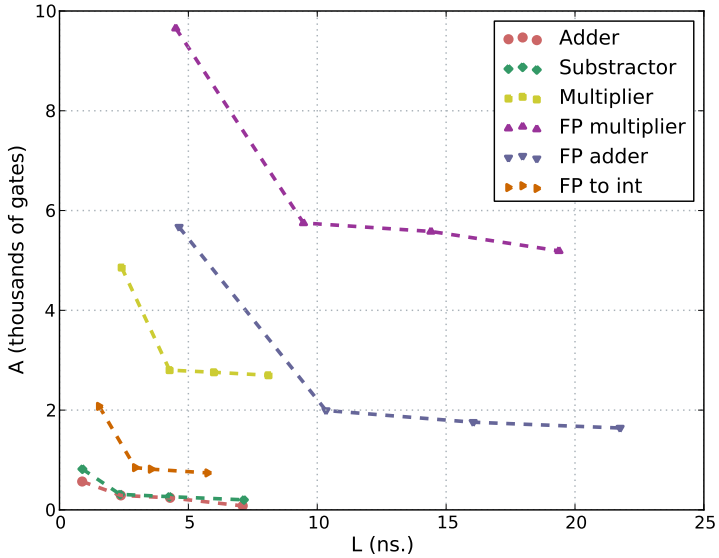

Figure 3.6: Area-delay curves for some of the operators that appear in the input graphs. The curves of area versus delay for each operator were obtained by using the Synopsys Design Compiler Ultra synthesis tool and a 0.13μm standard cell library.

plemented by instantiating a module of the DesignWare (DW) libraries provided by Synopsys [67], as shown in Figure 3.8. DW libraries provide highly optimized RTL for commonly used arithmetic components.

The fastest implementation of the operators requires maximum area, as more sophisticated circuits are used. This corresponds to the first design point of the curves in Figure 3.6. On the contrary, the minimum-area implementation of the operators is the slowest solution, which corresponds to the last design point of the curves in Figure 3.6.

In order to constrain the synthesis process such that it meets a specific area constraint, the following DC command is used:

`set_max_area` *`AREA_CONSTRAINT`*

The slowest solution is found by constraining the design with the command above, replacing *`AREA_CONSTRAINT`* with the number 0. This forces the synthesis tool to generate the minimum-area solution, and therefore the slowest solution.

```
module add ( x, y, z );

input   [31:0]  x;
input   [31:0]  y;
output  [31:0]  z;

reg     [31:0]  z;

always @(x or y)
begin
        z = x + y;
end

endmodule
```

Figure 3.7: Verilog module used for the characterization of a 32-bit adder.

In order to constrain the synthesis process such that it meets a specific timing constraint, the following DC command is used:

`set_max_delay from [all_inputs] to [all_outputs]` *`TIME_CONSTRAINT`*

The fastest solution is found by constraining the design with the command above, replacing *`TIME_CONSTRAINT`* with the number 0. This forces the synthesis tool to generate the minimum-delay solution, and therefore the largest solution.
The two intermediate design points are forced to be uniformly distributed between the fastest and the slowest solution previously found. This creates two delay constraints that can be imposed by using the DC command above, replacing *`TIME_CONSTRAINT`* with the delay that is required in each case. Thus, the synthesis tool finds an implementation for the operator that is close to timing constraints.

Figure 3.6 shows how the gate count of a synthesized 32-bit floating-point adder can vary by more than a factor of 2 as timing constraints are varied. For a 32-bit fixed-point adder the relative variation is even greater. Gate counts can easily reduce by a factor of two or more when timing constraints are relaxed. An operator that is *off* the critical path will therefore have a much lower synthesized area than a similar operator that is *on* the critical path. This phenomenon is modeled and exploited by the presented heuristic. The latency of graph G_i is estimated by the sum of the minimum modeled delays for each operator on the critical path of G_i.

The heuristic estimates the area of each graph by adapting the budget management technique presented in [68]. Each operator in G_i is initially assigned a

```
module fp_add (inst_a, inst_b, inst_rnd, z_inst, status_inst);

parameter sig_width = 23;
parameter exp_width = 8;
parameter ieee_compliance = 0;

input [sig_width+exp_width:0] inst_a;
input [sig_width+exp_width:0] inst_b;
input [2:0] inst_rnd;

output [sig_width+exp_width:0] z_inst;
reg    [sig_width+exp_width:0] z_inst;
output [7:0] status_inst;
reg    [7:0] status_inst;

DW_fp_add #(sig_width, exp_width, ieee_compliance)
        U1 ( .a(inst_a), .b(inst_b), .rnd(inst_rnd), .z(z_inst),.status(status_inst) );

endmodule
```

Figure 3.8: Verilog module used for the characterization of a 32-bit floating-point adder. A component of the DW libraries is instantiated.

latency and area given by the minimum delay point for that operator. Then a zero slack algorithm is applied to G_i in order to relax the area of the operators that are *off* the critical path. This iterative slack distribution process assigns additional latency to non-critical operators, reducing their area according to the model, until no further slack is found in the graph. After latency estimation, each operator has the smallest possible area, corresponding to the maximum delay the operator can tolerate without extending the critical path of G_i. The sum of the areas of all operators in the graph then gives the total estimated graph area.

3.5 Processor Customization

The resource-sharing process generates a set of graphs, some of which might be merged. Each graph forms an AFU that will be attached to the processor.

The AFUs that can execute the functionality of several ISEs contain multiplexers that configure the datapath accordingly. Therefore, these AFUs need to internally generate a *microcode* that constitutes the selection bits for all of the multiplexers. This *microcode* is provided by a decoding block which receives the relevant segment of the instruction opcode from the decoding stage of the processor. An example of this scenario is illustrated in Figure 3.9.

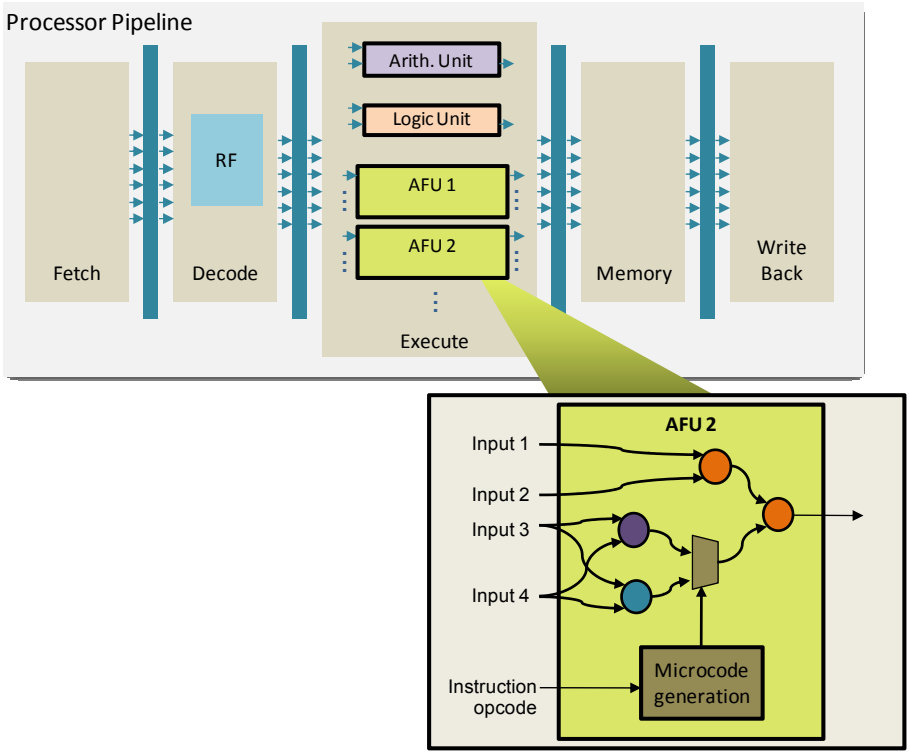

Figure 3.9: Processor pipeline extended with AFUs. The AFUs that are generated from merged graphs are configured by selecting the appropriate multiplexer inputs. The selection signals are generated from the instruction opcode provided by the decode stage of the processor.

Synthesis There are several alternatives to perform logic synthesis of the provided AFU datapaths.

Long latency ISEs can be pipelined in order to work at the same operating frequency as the target processor. In this case, registers are inserted in the datapath in order to obtain a critical path delay, from register to register, that does not exceed the clock period.

Another synthesis alternative is to use high level synthesis techniques [69] in order to reuse registers and possibly operators between the different execution cycles of an AFU. A process of scheduling and binding must take place, and further multiplexers need to be inserted and addressed according to the sequence that is generated in the scheduling and binding processes.

3.6 Experimental Evaluation

A set of 10 benchmarks were selected from a wide range of embedded application areas to evaluate the resource-sharing heuristic experimentally. The selected benchmarks were: LMS, JFDCTINT, ADPCM, LUDCMP and FIR from the SNU-RT benchmark suite [70]; LPC, FFT, SPECTRAL, COMPRESSOR and ADPCM from the UTDSP benchmark suite [71]; From these benchmarks, ISEs are identified using an existing technique and the resource-sharing heuristic is aplied under a wide range of constraints. Tables 3.3 and 3.4 show some characteristics of the set of ISEs obtained from each benchmark.

Design-space Exploration

The parameterized resource-sharing algorithm receives input graphs expressed in XML format, performs resource sharing, and outputs a description of the resulting merged logic in Verilog, for AFU implementation.

For each experiment, the algorithm was executed several times with different values of α_T, β_T and θ_T, varying from 0 to 1 in steps of 0.05. In turn, every set of values for α_T, β_T and θ_T, was run with the four combinations of values for the parameters *multiOp* and *grouping*.

This resulted in 37,044 experimental points in the design space of potential solutions. The execution time of the resource-sharing algorithm varied slightly, depending on the value of θ_T. Across the chosen sets of inputs the average execution time was 0.93 seconds. The average time to complete the whole exploration was 2.5 hours per benchmark, using four parallel threads; one for each of the four combinations of *multiOp* and *grouping*. Such times are not excessive in comparison with the time taken at the physical implementation stage of chip design.

For later reference, the static latency of a particular ISE, L'_{out}, is defined as the critical path of the (possibly merged) graph in which it appears. The evaluation of each design point is based on two metrics: AFU area, denoted $\mathcal{A}$; and average ISE latency, denoted $\mathcal{L}$. $\mathcal{A}$ is the total area of the AFU, and is given by the sum of the areas of each output graph in G_{out}. $\mathcal{L}$ is the average of the product F

Source Benchmark	LMS	JFDCTINT	ADPCM	LUDCMP	FIR
Number of ISEs	22	15	21	15	11
Largest area (gates)	73,188	35,637	38,358	19,449	66,149
Smallest area (gates)	870	4,368	403	3,369	857
Longest latency (ns)	35.07	21.50	38.36	11.73	35.00
Shortest latency (ns)	1.52	0.09	0.91	4.41	0.90
Max. num. of ops.	18	12	10	4	8
Min. num. of ops.	2	4	2	2	2
Max. num. of inputs	9	12	8	6	4
Max. num. of outputs	4	8	3	2	2
AFU input ports	12	12	8	8	4
AFU output ports	8	8	4	4	2

Table 3.3: Characterization of ISE input graphs and AFU port constraints for each benchmark application taken from the SNU-RT benchmark suite

Source Benchmark	LPC	FFT	SPECT.	COMP.	ADPCM
Number of ISEs	24	15	12	29	16
Largest area (gates)	81,680	49,485	67,171	32,548	24,876
Smallest area (gates)	1,090	1,749	2,259	366	857
Longest latency (ns)	22.82	9.03	18.31	18.22	9.78
Shortest latency (ns)	0.90	0.55	1.53	0.50	0.90
Max. num. of ops.	12	15	12	3	3
Min. num. of ops.	2	2	3	2	2
Max. num. of inputs	9	10	9	4	4
Max. num. of outputs	4	8	4	2	2
AFU input ports	12	12	12	4	4
AFU output ports	8	8	8	2	2

Table 3.4: Characterization of ISE input graphs and AFU port constraints for each benchmark application taken from the UTDSP benchmark suite

$\times L'_{out}$, over all ISEs considered as inputs. This metric quantifies the impact of merging on execution time.

Thus the design space is bounded by two opposing and extreme solutions: maximum resource-sharing, with highly compromised delay, and no resource-sharing, with minimal delay. The objective of our experiments is to expose a wide range of potentially interesting points in the design space, each showing a different trade-off between area and delay.

Input Generation

In this chapter it is assumed that all of the ISEs candidates listed by the identification phase are selected for hardware implementation. ISEs are DAGs annotated with its execution frequency obtained from profiling.

ISEs were identified from the benchmarks using an implementation of an existing identification heuristic, *ISEGEN*, described in [26]. In this implementation, ISEs can include a full range of integer, logical and floating-point operations, including divisions. The ISE identification algorithm was set up to constrain the maximum number of input and output values for each ISE, as shown in the bottom two rows of Tables 3.3 and 3.4. As the target processor connects AFUs with up to 12 inputs and 8 outputs, five benchmarks were targeted with these constraints. The other five benchmarks had varying constraints, from 8-in 4-out down to 4-in 2-out, chosen arbitrarily to illustrate the behavior of the heuristic under progressively tighter I/O resource constraints.

3.7 Results

At the beginning of this section, the results obtained using the input set extracted from the ADPCM program from the SNU-RT benchmark suite are analyzed. This input set comprises 21 initial graphs as candidate ISEs. In order to illustrate the specific effect of varying α_T, β_T and θ_T, this section shows first the resource-sharing solutions found as a result of executing the algorithm several times, while varying these three parameters, and keeping $multiOp = 0$ as well as $grouping = 0$.

Solutions found in this multi-dimensional space are plotted on a plane formed by the chosen metrics: $\mathcal{L}$ (on the horizontal axis) and $\mathcal{A}$ (on the vertical axis).

In Figure 3.10, three values of θ_T were selected, and for each value, α_T and β_T were varied across the range $[0, 1]$. As θ_T is reduced, the resulting solutions are pushed to the left. This illustrates how θ_T can be used to place a cap on the latency increase that will be tolerated when searching the design space with α and β. Similarly, in Figure 3.11, three values of α_T and β_T were chosen, and for each of them, θ_T was varied over the range $[0, 1]$. In this case, α_T and β_T tend

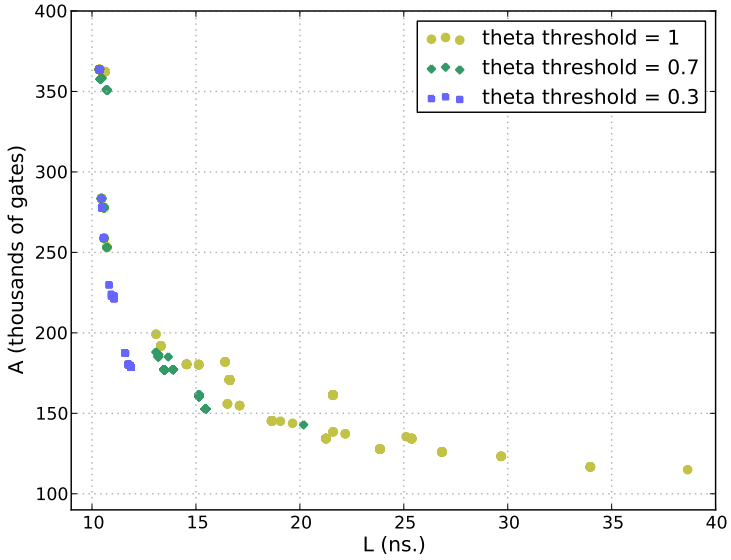

Figure 3.10: α_T and β_T variation when: $\theta_T = 1$, $\theta_T = 0.7$, $\theta_T = 0.3$.

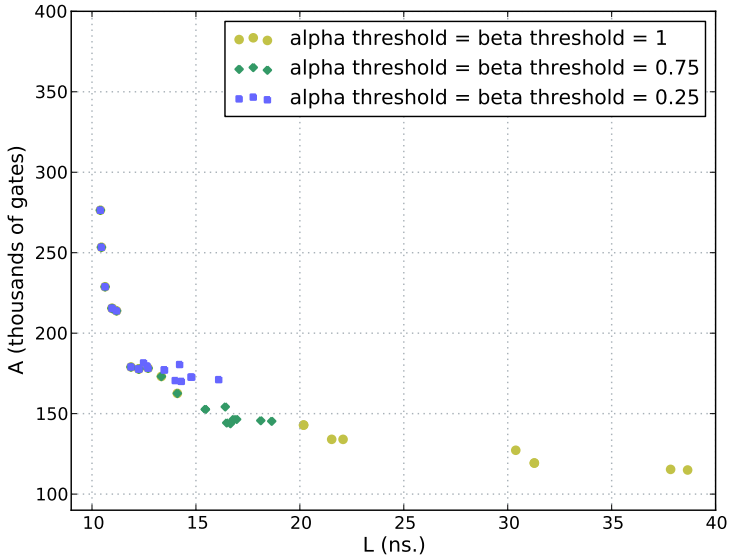

Figure 3.11: θ_T variation when: $\alpha_T = \beta_T = 1$, $\alpha_T = \beta_T = 0.75$, $\alpha_T = \beta_T = 0.25$.

to partition the design points into overlapping area ranges. Each value of α_T or β_T, by excluding some of the graphs from the resource-sharing process, decreases the minimum area threshold of the curve, thus sacrificing area for the sake of solutions with better average latency. In summary, θ_T tends to generate an *area-*

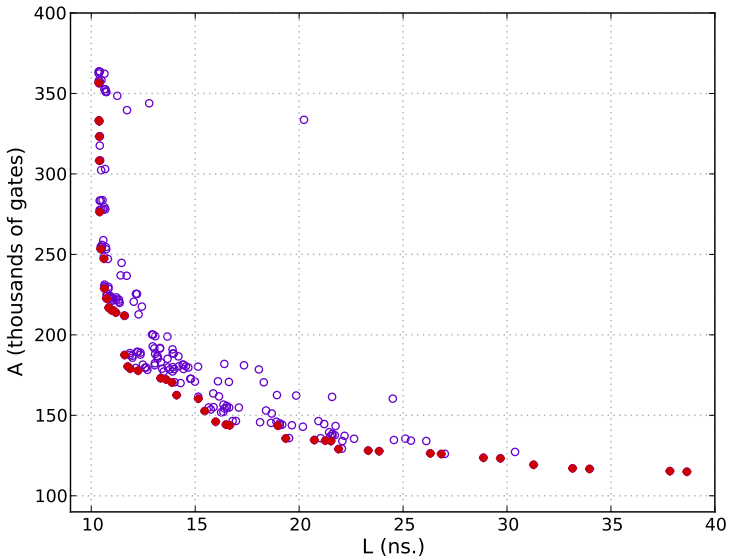

Figure 3.12: α_T, β_T and θ_T variation when $multiOp = 0$ and $grouping = 0$. Filled circles correspond to Pareto-optimal solutions.

wise exploration of the design space, for given values of α and β. Conversely, α_T and β_T generate mainly a *latency-wise* search for a given value of θ_T. Figure 3.12, shows all of the points found as a result of varying α_T, β_T and θ_T in the range [0,1] in steps of 0.05, while keeping $multiOp = 0$ and $grouping = 0$. In this plot, a curve formed by all of the Pareto-optimal solutions, delimits the space. A solution can be considered Pareto-optimal if there is no other solution found in the design space that performs at least as well on $\mathcal{A}$ and $\mathcal{L}$ and strictly better on either $\mathcal{A}$ or $\mathcal{L}$.

In contrast, Figure 3.13 shows the effect of varying all parameters, including *multiOp* and *grouping*. This illustrates the complete design space of distinct solutions exposed by the parameterized heuristic for the SNU-RT ADPCM benchmark, over the full range of all parameters.

The different marks on Figure 3.13 represent the 4 combinations of the binary parameters *multiOp* and *grouping*. Additionally, five solutions that correspond to Pareto points in the design space have been highlighted. Table 3.5 details the solutions represented by each of these points. The solution given by the No RS point has no resource sharing applied, and therefore has the smallest $\mathcal{L}$ but

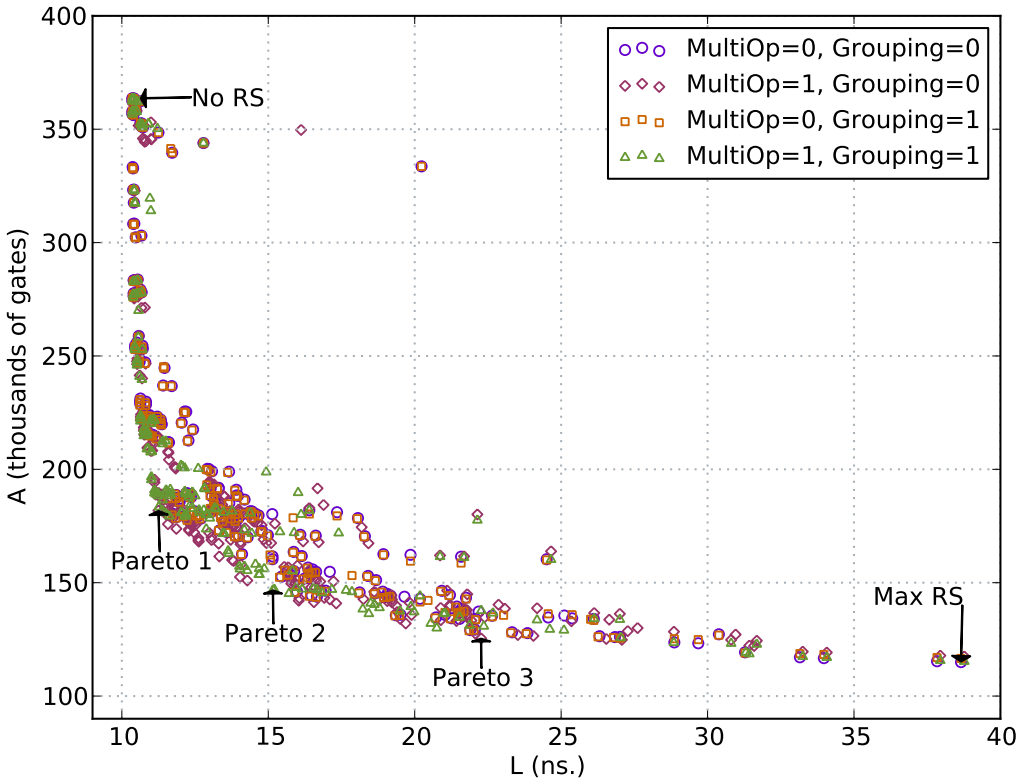

Figure 3.13: Solutions found in the design space when: $multiOp = 0$ and $grouping = 0$, $multiOp = 1$ and $grouping = 0$, $multiOp = 0$ and $grouping = 1$, $multiOp = 1$ and $grouping = 1$.

the largest $\mathcal{A}$. In contrast, for Max RS, where all of the threshold parameters are set to 1, the algorithm merges all ISEs without any restrictions, as in non-parameterized approaches. This yields the smallest $\mathcal{A}$, but also the largest $\mathcal{L}$. Points Pareto 1, Pareto 2 and Pareto 3, represent possible implementation alternatives with varying trade-offs between $\mathcal{L}$ and $\mathcal{A}$. As shown in Table 3.5, a small reduction in area savings can yield a major reduction in latency, and a small increase in latency can yield a major reduction in area, when the Pareto curve is traversed from the extreme end points.

The values of the three parameters that produce the five solutions are specified in Table 3.5. For Pareto 1, five graphs were left separate as they did not meet the constraint $\alpha_T = 0.3$ and the rest were merged with $\theta_T = 0.25$ as a constraint. Similarly, the solution that represents Pareto 2 is constrained by $\theta_T = 0.7$ and

	No RS	Pareto 1	Pareto 2	Pareto 3	Max RS
$\mathcal{L}$	10.35	11.25	15.15	22.26	38.65
Area Saved	0%	50%	60%	66%	68%
Out Graphs	21	9	4	3	1
α_T	0	0.3	1	1	1
β_T	0	1	0.75	0.75	1
θ_T	0	0.25	0.7	0.85	1
M	0	1	1	1	0
G	0	0	1	0	0
Speedup	2.31×	2.25×	1.81×	1.5×	1.04×

Table 3.5: Interesting points in the design space for the SNU-RT ADPCM benchmark

separates one graph as it did not meet the constraint $\beta_T = 0.75$. In a different way, Pareto 3 results in three output graphs that were a consequence of constraining the merging process with $\beta_T = 0.75$ and $\theta_T = 0.7$. Thus, it is apparent that the combination and variation of the three threshold parameters is useful in finding a range of good solutions.

Table 3.5 also shows the estimated speedup of the complete application in the target processor. The application speedup can be approximated from the hardware latency of the ISEs and the instruction count obtained from profile information. The latency of the ISEs has been calculated as the number of clock cycles taken in the target architecture given the chosen hardware implementation. Thus, the following equation is used:

$$speedup = \frac{\lambda_{SW_{app}}}{\lambda_{SW_{app}} - \sum_{i=ISE_1}^{i=ISE_n}(\lambda_{SW_i} - \lambda_{HW_i}) \times C} \tag{3.7}$$

where n is the number of ISEs that are included in the design, $\lambda_{SW_{app}}$ is the overall execution latency of the application. This measure does not account for control flow statements nor for cache misses. C is the number of times ISE_i is executed, λ_{SW_i} is the execution latency of the ISE in software, and λ_{HW_i} is the execution latency of the ISE in hardware and is calculated as follows:

$$\lambda_{HW_i} = \left\lceil \frac{L_i + ((t_{co} + t_{su}) \times \lceil \frac{L_i}{T} \rceil)}{T} \right\rceil \tag{3.8}$$

T is the clock period of the target architecture, e.g. 4 ns. L_i is the critical path of

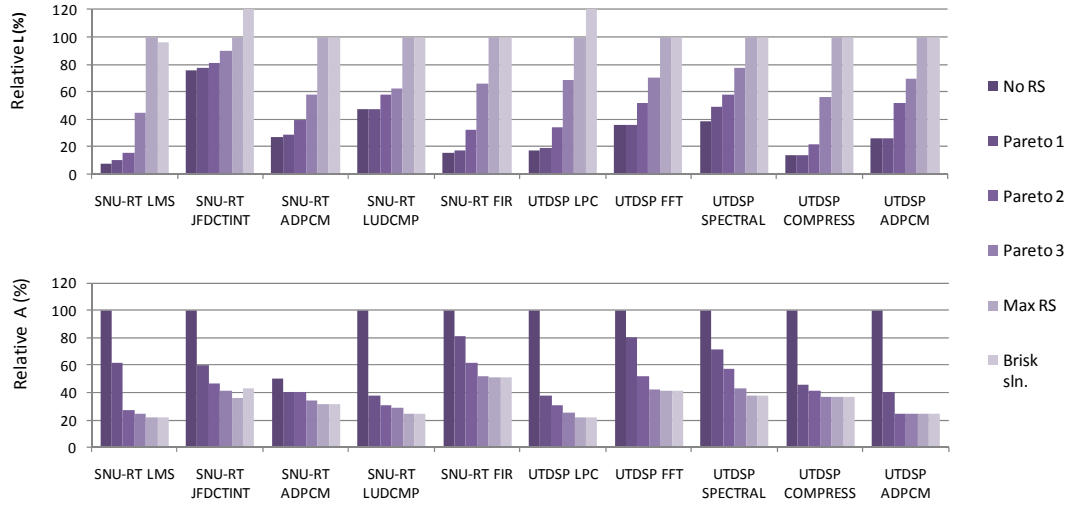

Figure 3.14: Summarized results of design-space exploration for 10 different benchmarks showing 5 Pareto points in each case. The upper chart shows the relative latency ($\mathcal{L}$) of each point, as a percentage of the latency of the solution Max RS. The lower chart shows the relative area ($\mathcal{A}$) of each point, as a percentage of the area of the solution No RS. The values of $\mathcal{A}$ and $\mathcal{L}$, to which area and latency are relativized, are presented in Table 3.7.

Source Benchmark	LMS	JFDCTINT	ADPCM	LUDCMP	FIR
$\mathcal{L}$ of Max RS (ns.)	67.5	11.6	38.65	11.7	35.9
$\mathcal{A}$ of No RS (gates)	568,270	196,100	363,64	189,784	140,000

Table 3.6: Max values of $\mathcal{A}$ and $\mathcal{L}$ found in the design space of each experiment with source benchmark taken from the SNU-RT benchmark suite

Source Benchmark	LPC	FFT	SPECT.	COMP.	ADPCM
$\mathcal{L}$ of Max RS (ns.)	29.25	10.03	23.59	18.429	14.58
$\mathcal{A}$ of No RS (gates)	512,000	247,500	278,290	124,412	157,300

Table 3.7: Max values of $\mathcal{A}$ and $\mathcal{L}$ found in the design space of each experiment with source benchmark taken from the UTDSP benchmark suite

ISE_i. Finally, t_{co} and t_{su} are the clock-to-data delay and data set-up time of the flip-flops that need to be inserted in the circuit and their values are dependent on device technology.

Pareto-optimal Solution Space

Figure 3.14 shows the summarized results of the experiments performed with all of the input sets described in Tables 3.3 and 3.4. Latency and area values of five points in the design space were extracted in each of the experiments. The five points are: Max RS, No RS and three solutions that have been taken from the resulting Pareto curve. These Pareto points are chosen to be equally distributed along the Pareto curve and are referred to as Pareto 1, Pareto 2 and Pareto 3. Additionally, in every experiment these points are compared with the solution that would be given by the heuristic proposed by Brisk *et al.* in [8] using the same input set. In the figure, this is referred to as Brisk Sln.

In order to compare the results of the different experiments in the same range, the values are taken as percentages with respect to, in Figure 3.14(a), the area of the point No RS and, in 3.14(b), the average latency of the point Max RS. These values, for each of the experiments, are presented in Table 3.7. For all of the experiments, Max RS, as expected, has long latency and small area. The point No RS shows always short latency and large area. In contrast, solutions Pareto 1, Pareto 2 and Pareto 3 represent solutions found as a result of the exploration of the parameterized space, each with a different trade-off between average ISE latency and area saving.

As expected, Brisk Sln is broadly comparable with Max RS, as both focus purely on area reduction. In four of the experiments, the minimum-area solution Max RS, yields a solution with smaller area than that found by Brisk Sln. In the case of JFDCTINT and LPC, the corresponding average latency is notably lower at the point Max RS found in our exploration. In contrast, for LMS and SNU-RT ADPCM, the average latency was slightly longer in Max RS as a result of more extensive merging.

Even when the AFU I/O constraints are low, as in the case of the experiments with UTDSP ADPCM, COMPRESSOR and FIR, the area savings are significant. Furthermore, in most cases the area could be decreased by at least 50% before a serious increase in latency is perceived. If the maximum-area values shown in Table 3.7 are reduced by 50%, the AFU would then represent between 6% and 53% of the extended target-processor, depending on the benchmark.

Distribution of Pareto-optimal Parameter Settings

Figure 3.15 illustrates how the settings of the five parameters were distributed as a function of their resulting area and latency, in order to produce the Pareto-optimal design points shown in Figure 3.14. For each benchmark, the Pareto points were initially ranked from smallest to largest area. This list was then partitioned into 10 deciles according to the magnitude of the area. For each parameter, the boxplots in Figure 3.15(a) show the maximum value, the minimum value, the lower quartile and the upper quartile within each decile across all benchmarks. The same process was used to determine how the parameter settings are distributed in order to achieve average latency results in each decile, as shown in Figure 3.15(b).

In this graphical representation, 50% of the samples have values between the lower quartile and the upper quartile. On the other hand, when the maximum value equals the upper quartile, at least 25% of the samples have values equal to the maximum. Similarly, when the minimum value equals the lower quartile, at least 25% of the samples have values equal to the minimum. In the cases where the upper quartile is very close or equal to the maximum value and the lower quartile is very close or equal to the minimum value, there is a strong bias towards extreme values: maximum and minimum. This is the case in the distributions of the binary parameters: *multiOp* and *grouping*

Pareto points benefited from α_T and β_T in the first deciles of the latency ranking and in the last deciles of area ranking. In the first decile of the latency ranking there are many points, so the effect of changing the parameters is better described over the area range. For larger areas the corresponding β value reduces. Conversely, the effect of varying θ_T can be seen throughout the whole ranges of latency and area, having small value for solutions with short latency and large area and having a greater value for solutions with long latency and small area. In the graphs corresponding to α_T and θ_T in Figure 3.15(b), these values decrease while area increases. However, in the last groups there is a slight increase in their values. This shows an inter-parameter effect; when one parameter becomes tighter the other parameter can relax or increase its value in order to obtain small delay solutions.

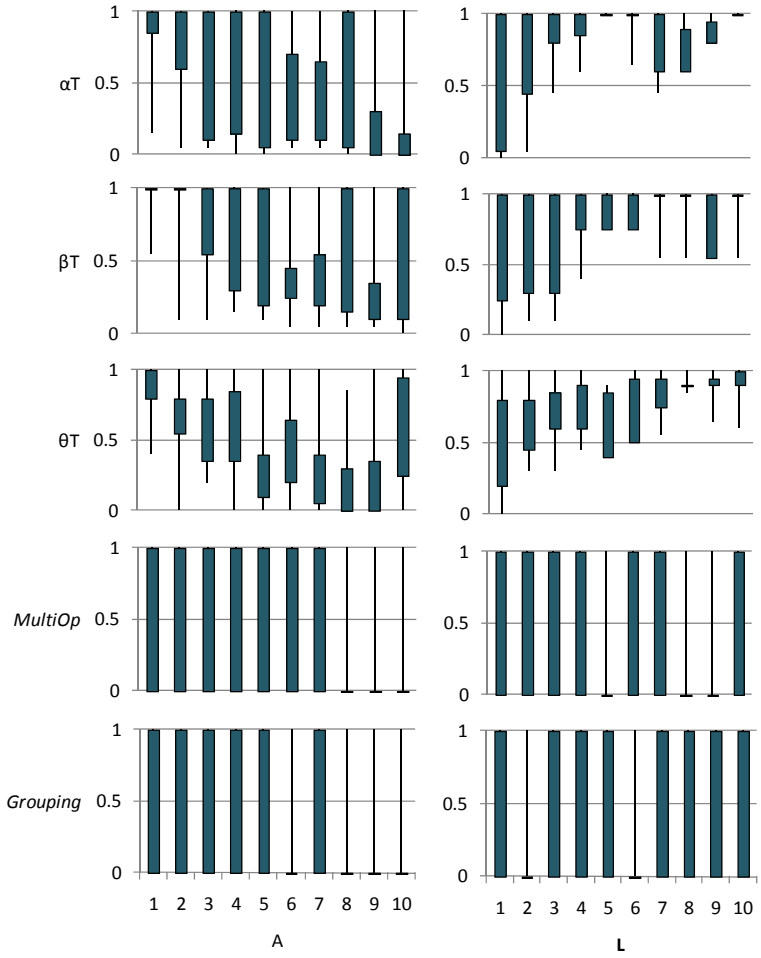

Figure 3.15: Value distribution of the 5 parameters used to find the Pareto points across all benchmarks. The boxplots indicate the maximum value, the minimum value, the lower quartile and the upper quartile for every 10th percentile. (a) First column: along the area range. (b) Second column: along the latency range.

The effect of changing the parameters *multiOp* and *grouping* on latency and area can be seen spread through both ranges, which means that a large percentage of Pareto points benefited from either *multiOp* or *grouping*.

Synthesis and Pipelining of Merged Graphs

The results presented up to this point are all derived from the model of delay and area. To establish the correlation between model and reality the output graphs of the resource-sharing algorithm were taken through logic synthesis and the areas and latencies achievable in practice were measured. Figure 3.16 shows

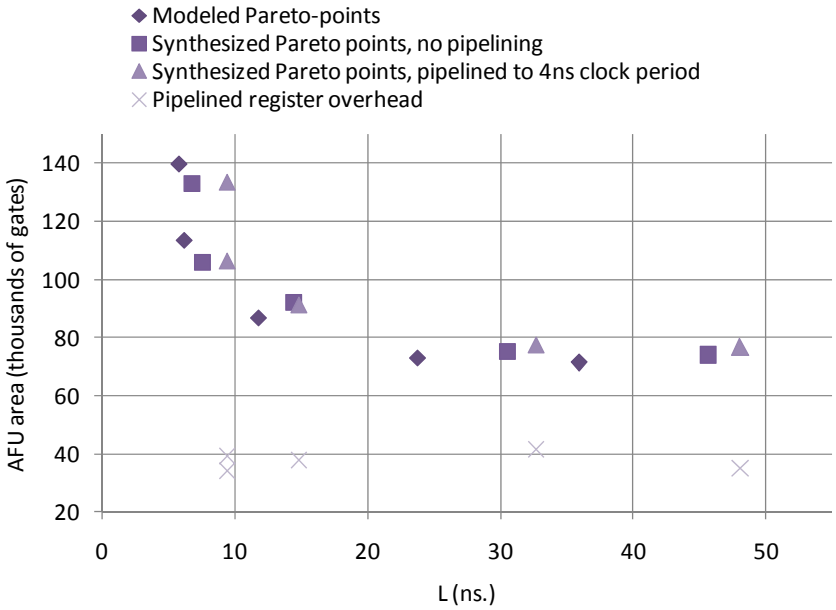

Figure 3.16: Synthesis results for the FIR benchmark, showing latency and area for the 5 Pareto-optimal points given in Figure 3.14.

the synthesis results for 5 representative points on the Pareto curve of the FIR benchmark.

The merged ISEs obtained at each of the five selected Pareto points were synthesized both non-pipelined and pipelined versions.

The non-pipelined implementation of a complex ISE will typically create a *blocking* multi-cycle instruction. This may be appropriate in some systems, but it is often more useful for long latency ISEs to be pipelined to work at the same operating frequency as the target processor. Therefore, each of the 5 Pareto points are re-timed, by pipelining them to meet the 4ns clock period of the target processor. Re-timing was achieved by determining the number of pipeline stages required for each merged ISE and, for each stage, adding the clock-to-data delay and data set-up time of a typical flip-flop to the overall timing budget. This kept the time budget unchanged for the critical path through each merged ISE after pipeline register delays are taken into account. The optimize_registers command in Design Compiler was used to automatically balance the inter-stage pipeline registers in each case.

Figure 3.16 compares the original Pareto points, obtained from the operator timing and area models within the resource-sharing heuristic, with the actual

points obtained from both the non-pipelined and the pipelined implementations. The area overheads due to the addition of pipeline registers are shown separately from the logic area of the pipelined ISEs, as this allows a direct comparison of the model and the implementation. Overall, the absolute area differences between the modeled and the synthesized design points are 5.01% and 5.66% for the non-pipelined and pipelined implementations respectively. The P1 solution yields a 20% reduction in logic area compared with No RS, with minimal increase in latency, and with the same number of pipeline stages. Conversely, the P3 solution yields a 32% decrease in latency compared with Max RS, with only a 1.7% increase in logic area.

In conclusion, the values obtained from logic synthesis follow the proportions observed when using the model. Thus, comparisons made between solutions, given their characteristics in area and average instruction latency, are likely to remain valid when logic synthesis is performed.

3.8 Conclusions

This chapter presents a new parametric algorithm for sharing hardware resources between multiple instruction set extensions that have been selected *a priori* for the performance improvements they can make to a given application. The algorithm combines a path-based resource-sharing algorithm with a timing budget management scheme to merge ISE graphs and allocate slack time in the resulting graphs so as to minimize implementation cost. The primary goal has been to develop a method by which the design space of resource sharing can be explored. Results show that die area is excessive when resource sharing is disabled, whereas very aggressive resource sharing leads to large instruction latency. This chapter presented a novel heuristic that is the first method to be able to explore the design space between these two extremes to find solutions that achieve the desired compromise between latency and area.

Previous resource-sharing techniques aim at obtaining the maximum possible area-savings. However, it has been shown that there exist other intermediate solutions with a better compromise between area savings and latency. Typically, a solution that aggressively shares resources generates one large graph that can be

configured to perform the functionality of all of the given ISEs. A single multi-functional graph, appart from greatly increasing the latency of the ISEs, causes a complex logic synthesis process due to the large number of paths, generated by multiplexers, that needs to be analyzed and optimized by the tools. Additonally, a single large AFU represents a suboptimal solution when clock gating or power gating is to be used during logic synthesis in order to reduce the dynamic power consumption of the processor extensions. This is because the entire AFU would need to be active every time any ISE is executed. Instead, when resource sharing is carefully performed, several AFUs are generated, thus permitting a finer granularity for techniques such as clock gating and power gating.

The following chapter shows how the Pareto solutions of the resource-sharing design-space can be quickly obtained based on the intrinsic characteristics of the input ISE set.

Chapter 4

Fast Exploration of the Resource-sharing Design-space

4.1 Introduction

As seen in Chapter 3, the resource-sharing algorithm is parameterized by five threshold values that are used to take decisions during its execution. As these threshold values represent trade-off decisions, such as whether or not to merge at different points, several optimal solutions might be found with different threshold values. Each parameter has a unique impact when evaluating trade-offs in the design space. Additionally, as all of the five parameters can be varied at the same time, inter-parameter effects can take place, thus creating a much larger range of possible solutions.

Solutions might be optimal or suboptimal in comparison to others in terms of the target metrics: area and latency. Every combination of parameter values affects differently each input set; therefore, the parameter space has to be explored exhaustively for every input. After the exploration is completed, the Pareto-optimal solutions can be extracted. A solution is said to be Pareto-optimal when no other solution is better in both metrics.

Exhaustive processes are extremely time-consuming and make the exploration at the ISE selection level prohibitive. However, machine-learning approaches could make use of previously explored spaces in order to predict the combination of parameters that results in Pareto-optimal solutions. The ideal scenario

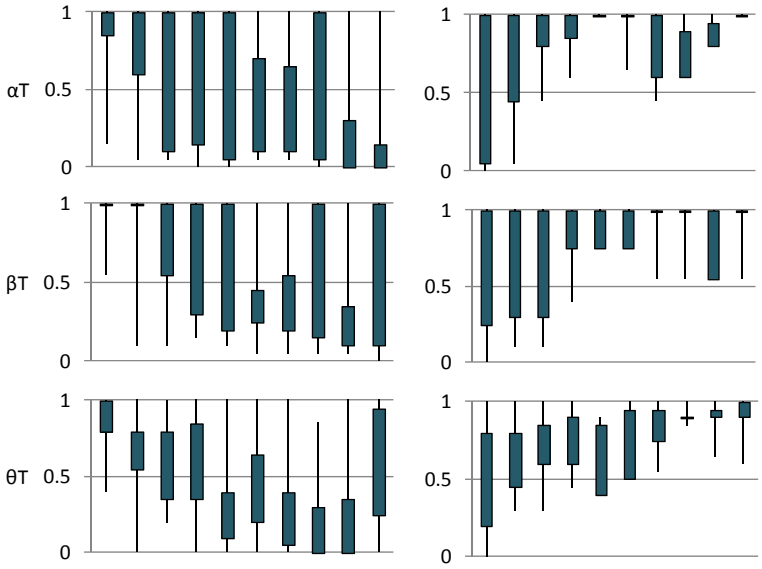

Figure 4.1: Value distribution of α_T, β_T, θ_T used to find the Pareto points across all benchmarks. The boxplots indicate the maximum value, the minimum value, the lower quartile and the upper quartile for every 10th percentile. (a) First column: along the area range. (b) Second column: along the latency range.

being that the model can predict the parameter combinations that lead to the Pareto curve. Consequently, in this ideal scenario the resource-sharing algorithm is executed only as many times as Pareto points can be found in the resource-sharing design-space. In this case, the number of executions required to find the Pareto curve would be three orders of magnitude smaller than when the space is exhaustively explored.

In the previous chapter, Figure 3.15 shows how the settings of the five parameters were distributed as a function of their resulting area and latency, when delivering the Pareto-optimal design points. This figure is partly reproduced in Figure 4.1, which shows the distribution of values, along the Pareto curves found throughout several explorations, for the parameters with continuous values: α_T, β_T, θ_T.

As shown in the figure, the values of the parameters in a given segment of the curve are likely to fall in the same range throughout different input sets. This is due to the intrinsic behaviour of each parameter during the execution of the algorithm. From these behaviours, it could be possible to derive some parameter settings that are unlikely to lead to a Pareto-optimal solution. This would prune

the parameter space, making it smaller for exploration. Alternatively, these behaviours and patterns can be automatically extracted by using machine-learning techniques. These techniques could, for instance, show that similar set of graphs respond likewise to parameter configurations. The similarities across input sets can be inferred from a number of characteristics extracted from the graphs. These characteristics can describe the set as a whole and can include details of delay and area of the comprising graphs.

In this chapter, it is shown that a predictive model can be trained, using data obtained in previous explorations, in order to speed up the exploration of the resource-sharing design-space of a set of graphs given its characteristics.

This chapter is organized as follows. Section 4.2 shows how the design-space exploration flow is formulated such that a predictive model can be inserted in order to speedup the process. Additionally, the input-output requirements of the model are discussed. Section 4.3 shows how a set of ISEs can be quantitatively represented by extracting a set of characteristics of the graphs that represent the ISEs in the set. These characteristics constitute the input of the model. Section 4.4 describes the fully-explored design-spaces from which predictions are generated. Section 4.5 presents the process carried out to design a predictive model that is able to generate the parameter configurations that lead to Pareto points in the resource-sharing design-space, given the characteristics of a set of ISEs. In Section 4.6, experimental results show the efficiency of the model. In Section 4.7 the model is used to predict the resource-sharing design-space of previously unseen inputs. Finally, Section 4.8 summarizes the results and presents some concluding remarks.

4.2 Formalizing the Problem

As seen in the previous chapter, when a selection of ISEs is to be implemented and attached to a processor, it is sensible to attempt to reduce area by sharing resources amongst the ISEs. However, in most cases, the sharing comes at the cost of degrading the critical path of the ISEs, and therefore, the speedup obtained with their implementation. It has been shown that exploring the multi-dimensional space created by the parameters: α_T, β_T, θ_T, *multiOp* and *grouping*;

gives rise to another space, whose dimensions are our metrics. This transformation is obtained by executing the resource-sharing algorithm with every point in a discretized parameter-space. The result of every execution is a netlist of hardware components that is needed to execute all of the ISEs. Every time the algorithm attempts to obtain different resource-sharing trade-offs, according to the set parameter values. The result can be characterized by the metrics: area of the solution and weighted average critical path of the ISEs.

For a designer, it is important to know the resource-sharing solutions that represent a trade-off between the metrics for a chosen set of ISEs. Figure 4.2(a), depicts the process that is required to obtain the optimal trade-offs. A set of ISEs, with each ISE represented as a DAG, is the input for the resource-sharing algorithm. The algorithm is executed iteratively for all of the possible combinations of the parameter values in the permitted ranges. This represents the hot-spot of the process, as it implies an exhaustive exploration of the resource-sharing design-space. From the solutions that are found, the designer needs to extract the solutions that are Pareto-optimal as these are the only interesting points for making a design decision.

In contrast, Figure 4.2(b) shows the corresponding process when a model is built to speedup the exploration of the parameter space. With the assistance of the model, instead of testing all of the points in the discretized parameter space, only a few points that are likely to be reflected as Pareto points in the resource-sharing space are considered. As indicated in the figure, a set of quantitative features should be extracted from the set of ISEs in order to be used as inputs for the model. Details about the features will be given in the following section. Given a set of features, the model will give a set of parameter values that are predicted to lead to the Pareto curve in the resource-sharing design-space. The parameter values suggested by the model are then tested in the resource-sharing algorithm in order to get their performance in terms of the metrics.

Therefore, the particularization of the learning goal is now represented as:

$$\text{ISE Input-set Features} \rightarrow \text{Model} \rightarrow \text{Parameter Settings}$$

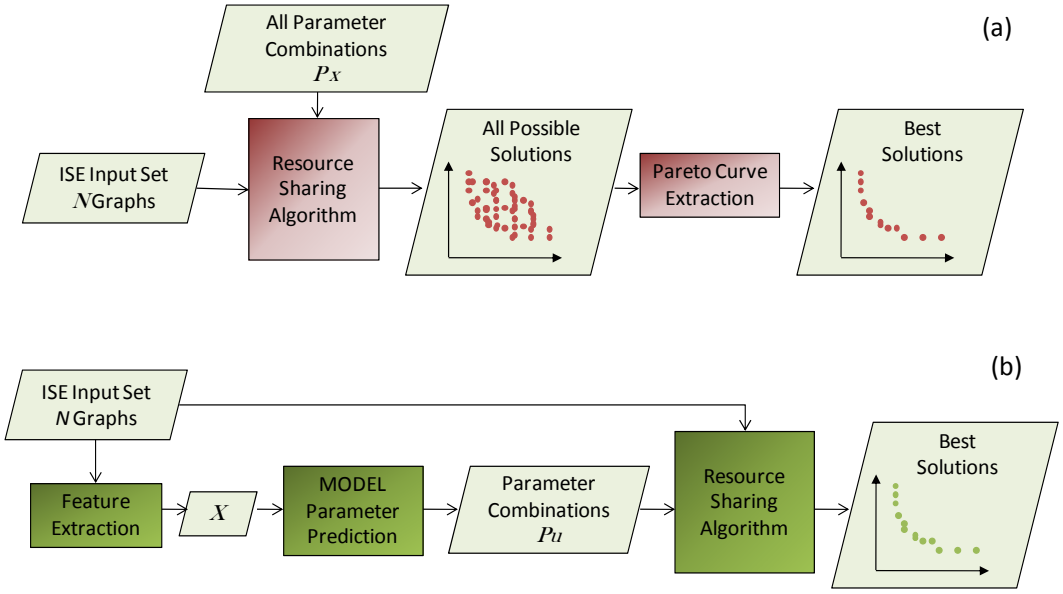

Figure 4.2: (a) Flow chart of the process to exhaustively explore the design space of resource-sharing solutions and find the optimal trade-offs. (b) Updated flow chart of the exploration with the intervention of a predictor to determine the best parameter configurations in order to directly find the optimal resource-sharing trade-offs.

4.3 Input-set Features

The input of the model needs to be a quantitative representation of the set of ISEs that will be merged and implemented.

The selection of features is important to any machine-learning-based technique as these have a direct impact on how the model differentiates between inputs and how it can gauge similarities. Features can describe the set as a whole and can include details of delay and area of the comprising graphs.

The features extracted are: number of ISE graphs, standard deviation, 1st quartile, 2nd quartile and 3rd quantile of the set of critical paths of the graphs weighted with their corresponding frequency of execution. The values are normalized so that their maximum value is 1. Thus, the features form a multi-dimensional space where each point represent a set of feature values F.

4.4 Generating the Training Data

In order to obtain explored design-spaces from which patterns can be extracted, 56 benchmarks were taken from the UTDSP [71] and SNU-RT [70] benchmark suites.

ISE identification was performed on each benchmark using an implementation of *ISEGEN* [26]. In this implementation, ISEs can include a full range of integer, logical and floating-point operations, including divisions. The identification was set to constrain ISEs to have a maximum of 12 input values and 8 output values corresponding to the register file I/O port constraints of the target processor.

For the sake of obtaining a large training set, more than one training case was extracted from each benchmark. Given a set of ISE candidates found by the algorithm, ISEs were randomly selected to be included in the set of graphs to be merged, thus forming a new training case. This was repeated more than once for each benchmark, depending on the number of ISE candidates available. In this way, 95 training sets were obtained. The smallest training set contains 5 ISEs and the largest training set contains 26 ISEs.

The design space of resource-sharing solutions was explored for every training case. The exploration of this space was done by executing the resource-sharing algorithm several times with different values of α_T, β_T and θ_T, varying from 0 to 1 in steps of 0.05. In turn, every set of values for α_T, β_T and θ_T, was run with the four combinations of values for the parameters *multiOp* and *grouping*.

Training input-output pairs are then composed by coupling the features extracted from each set of graphs and the parameter configurations that were used by each of the Pareto points found during the exploration.

A point in the resource-sharing design-space can be generated by several parameter configurations. Therefore, for each Pareto point found in the design space, the configuration that was selected was the one with the lowest values according the following comparison priority order: α_T, β_T, θ_T, *multiOp* and *grouping*.

4.5 Building a Model

Figure 4.1, was a first attempt to analyze the behaviour of the parameter space when reflected in the resource-sharing design-space. The model needs to capture the patterns of the parameter values that generate a Pareto-optimal point in the resource-sharing design-space. This figure suggests that a point in a given section of the curve, generated with a point P in the parameter space, might be generated for a different input set with a point that is in the vicinity of point P.

As an example, the extreme cases seen in the resource-sharing design-space are, as shown in Chapter 3, Max RS and No RS. These points are generated for the majority of the input sets with $P_1 = (\alpha_T = 1,\ \beta_T = 1,\ \theta_T = 1,\ multiOp = \mathsf{X},$ $grouping = \mathsf{X})$ and $P_2 = (\alpha_T = 0,\ \beta_T = 0,\ \theta_T = 0,\ multiOp = \mathsf{X},\ grouping = \mathsf{X})$, respectively. Thus, these two points in the resource-sharing design-space: Max RS and No RS, represent two different sections of the Pareto curve and are found throughout different input sets with parameter configurations that are in the vicinity of P_1 for Max RS solutions and in the vicinity of P_2 for No RS solutions. Consequently, a set of points that represent different trade-offs in the resource-sharing design-space are likely to be generated by points from different regions of the parameter space. Therefore, as the goal of the model is to find all optimal trade-offs in the resource-sharing design-space, different regions of the parameter space need to be explored.

As suggested in the previous example, the key regions of the parameter space are given by the parameters α_T, β_T, θ_T. The values of these parameters are continuous, share the same range and complement each other during the execution of the algorithm when driving the solution from Max RS to No RS in the resource-sharing design-space. Hence, the three-dimensional space given by α_T, β_T, θ_T will be explored in several regions.

In order to generate predictions, the following procedure is adopted:

1. Given an unseen set of features F_{unseen}, located in the feature space along with each F_{train} from the training set, Euclidean distance is calculated from F_{unseen} to each F_{train} in order to find the k training case closest to the new case.

2. The parameter settings of the k closest training cases are stored in $P_{k-train}$ and are used in order to generate predictions for parameter settings P_{unseen}.

3. Points in $P_{k-train}$ are clustered into c groups, thus forming regions in the parameter space that can be independently explored. Therefore, every $P_{k-train}$ is classified into one of the c clusters and then referred to as $P_{k-c-train}$.

4. The normal probability distribution of a set of points $P_{k-c-train}$ is extracted from each cluster. Therefore, c probability distributions are obtained from this step.

5. In order to generate predictions for P_{unseen}, the c distributions are sampled one by one to generate points that follow the same trend as the training points $P_{k-c-train}$ that belong to one cluster. A prediction P_{unseen} is issued from each cluster, sampling its corresponding probability distribution, in a round-robin fashion. P_{unseen} assigns values to the parameters α_T, β_T and θ_T. The values for the parameters *multiOp* and *grouping* are predicted to be the same as the ones associated with the closest training point to P_{unseen}.

The exploration in every cluster is expected to target different regions of the Pareto curve and is based on the values of the parameters taken from the k training cases most similar to the unseen one. Figure 4.3 graphically describes these processes.

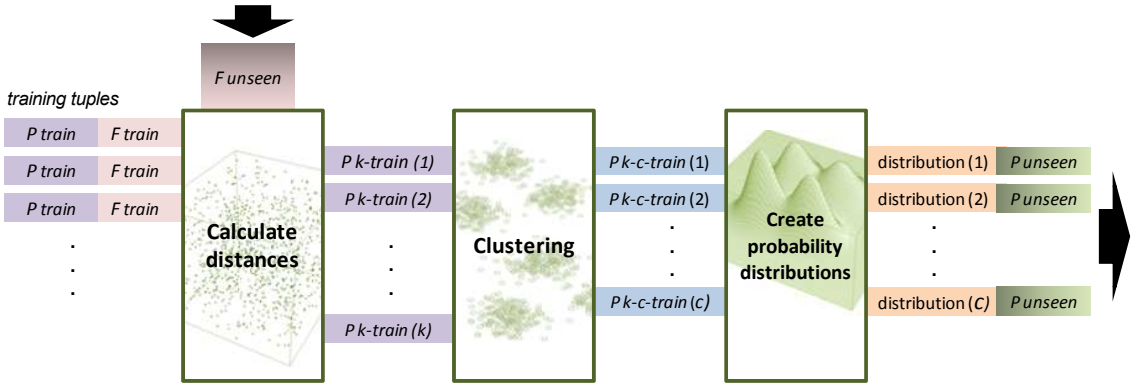

Figure 4.3: Graphical representation of the internal processes that must take place in order to predict the Pareto parameter-values given the features of the new input set of ISEs.

Number of Clusters and Neighbours

Having defined the procedure to generate a prediction, the values assigned to k (number of neighbours) and c (number of clusters) are key to the finalization of the model for future use.

Every pair of values (k, c) creates a new model that will predict differently. For this reason, several configurations should be evaluated in order to choose these values. It is impractical to evaluate all of the possible configurations for the pair (k, c). Therefore, a reasonable list of possible values was taken for consideration. The values in $k = \{10, 20, 40\}$ and in $c = \{5, 10, 15, 20\}$ were mixed in order to create a different model from every possible combination. Then, cross-validation was used for model selection.

4.6 Experimental Evaluation of the Model

A 10-fold cross-validation has been performed to evaluate the model. On the other hand, the parameters k and c have been learned on the training set by nested cross validation.

For the sake of evaluating the final configuration of the model on data that has not been used to decide the values of k and c, the training set is partitioned in two. 90% of the training set is used to evaluate the different configurations of k and c. With each configuration, this 90% is further partitioned and a *leave-one-out cross-validation* process takes place to evaluate its performance. The remaining 10% is left as a test set to later evaluate the chosen (k, c) configuration using the 90% as training set. This process is repeated with every 10% of the initial training set. Hence, every round, the remaining 90% is used to test every configuration (k, c) using *leave-one-out cross-validation*. Thus, the parameters k and c are tuned using only the training set and never the test set, in order to demonstrate the generalization of the technique by evaluating it on completely unseen cases.

Every round, a best configuration is selected. The configuration (k, c) that has been best for most rounds is selected and tested against every 10% as the final round of cross-validation.

As sampling the distributions found per cluster has a random component, each experiment was repeated 10 times and the results were averaged. The number of repetitions was determined empirically, as no changes were observed when more than 10 repetitions of the experiments were performed.

Performance Measure

The model in practice will suggest values of parameters P_{unseen} to be used in the resource-sharing algorithm given a set of features extracted from a set of ISEs that are to be merged. It will attempt to suggest P_{unseen} points in the parameter space that will lead to a Pareto-point in the resource-sharing design-space. The performance of the model can be a measure of how many P_{unseen} points need to be suggested in order to ensure that all of the possible optimal points will be found. Such points compose the Pareto-curve that can be found during an exhaustive exploration of the parameter space and will be referred to as the true Pareto-curve. Therefore, the number of P_{unseen} points that are needed in order to find the true Pareto-curve will be used as a metric and will be referred to as R. Consequently, for future explorations, the resource-sharing algorithm will have to be executed only R times. R will be determined with the experiments that evaluate the model with the training data.

Additionally, there is a need to measure, for a given R, the similarity between the true Pareto-curve and the curve that has been found so far. In Figure 4.4(a), the Pareto-curve formed by the red circles represents the true Pareto-curve, and the green triangles represent the Pareto-curve found after testing the R P_{unseen} points suggested by the model.

A metric d_{pp} is created for this purpose and it can be found by summing up the distances from each point of the true Pareto-curve to the closest point of the Pareto curve found after R executions of the resource-sharing algorithm. In Figure 4.4(b), the green lines indicate how this distance is found for each point in the true Pareto-curve. In order to normalize the distance-based metric, this sum is divided by the length of the true Pareto-curve. In Figure 4.4(b), the length of the segments that are formed between the red circles can be summed up to obtain the length of the true Pareto-curve used for normalization.

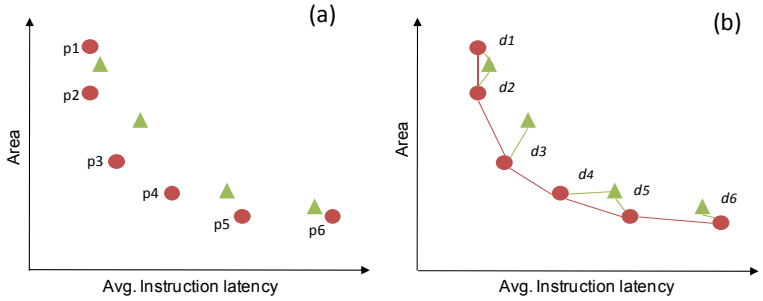

Figure 4.4: Example in the resource-sharing space to visualize the metric d_{pp} that is used to evaluate the model after R parameter predictions are tested in the resource-sharing algorithm. (a) The red circles represent the true Pareto-curve: found after exploring the entire parameter space. The green points represent the Pareto-curve found by running the resource-sharing algorithm R times with parameter configurations P_{unseen} suggested by the predictor. (b) Illustrates components to calculate d_{pp}. The green lines are the distances from each true Pareto-point to the closest green point. The segments that are formed between the red circles form the length of the curve used to normalize d_{pp}.

Choosing the Optimum Number of Neighbours and Clusters

In order to see how the performance of the model grows as a function of R, every 10 P_{unseen} points predicted by the model, d_{pp} were calculated. Figure 4.5 shows the result of 10 experiments in which cross-validation was perform on 90% of the training sets available. Every experiment corresponds to 10% left out. As explained in the previous section, the same process is repeated for every possible configuration of c and k.

All values obtained for d_{pp} in the cross-validation rounds, for a certain value of R and a certain (k, c) configuration, were averaged. Also, the 95% confidence interval of this mean is shown.

Figure 4.6 shows the averaged results across the 10 experiments, for every R, in increments of 10. The configuration $k = 10$ $c = 20$, which means 10 neighbours and 20 clusters, has smaller d_{pp} values for most values of R. Based on this observation, the number of neighbours k was fixed to 10 and the number of clusters c was fixed to 20. The model with this configurations will be referred to as *model-nearest-k10-c20*.

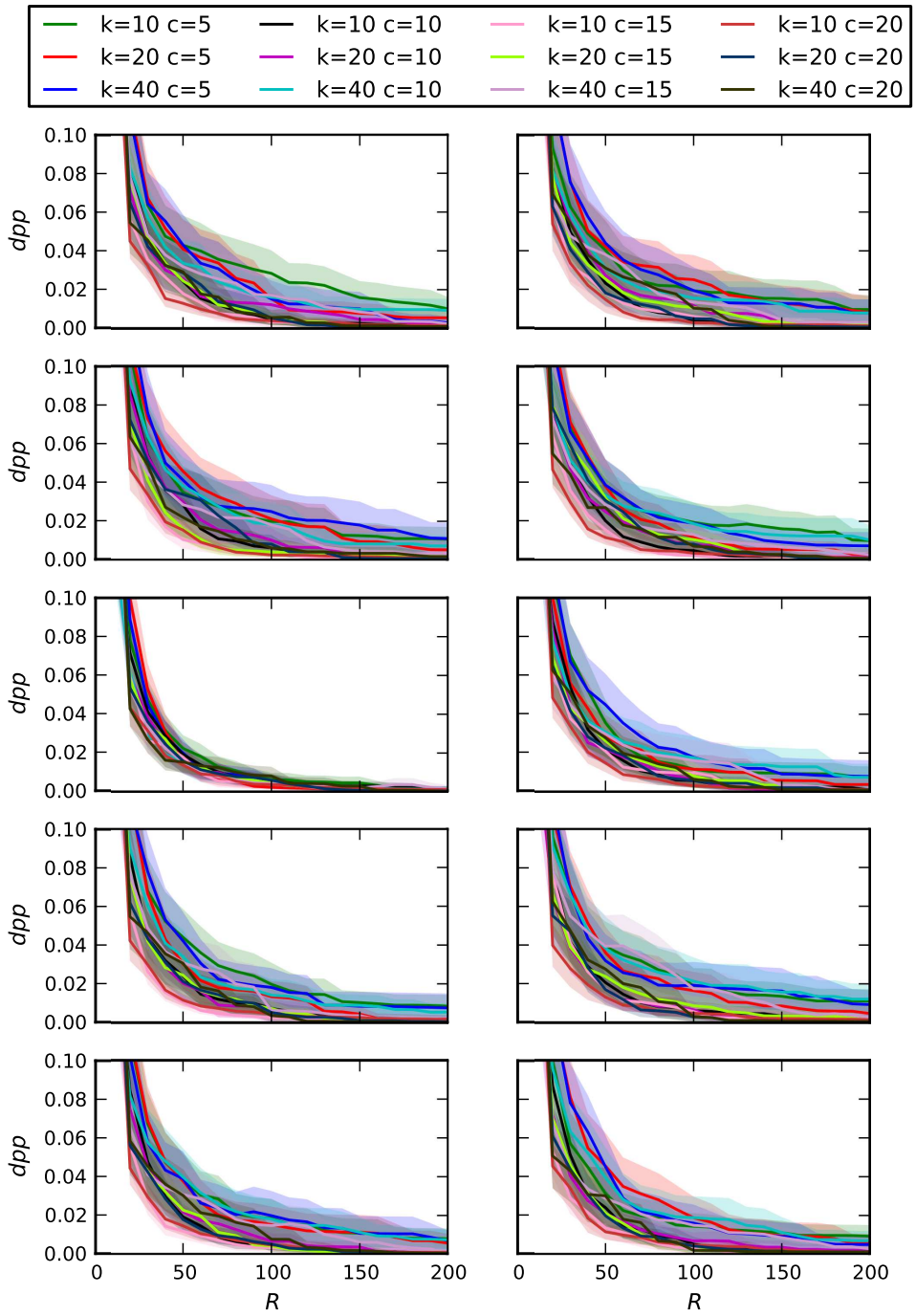

Figure 4.5: Every chart corresponds to an experiment where 10% of the training set was left out. Every remaining 90% was used to perform cross-validation on every configuration (k, c). Mean values of d_{pp} and their 95% confidence intervals are plotted as a function of R.

Performance of the Model

After finalizing the design of the model by assigning the number of neighbours to 10 and the number of clusters to 20, its performance can be tested iteratively

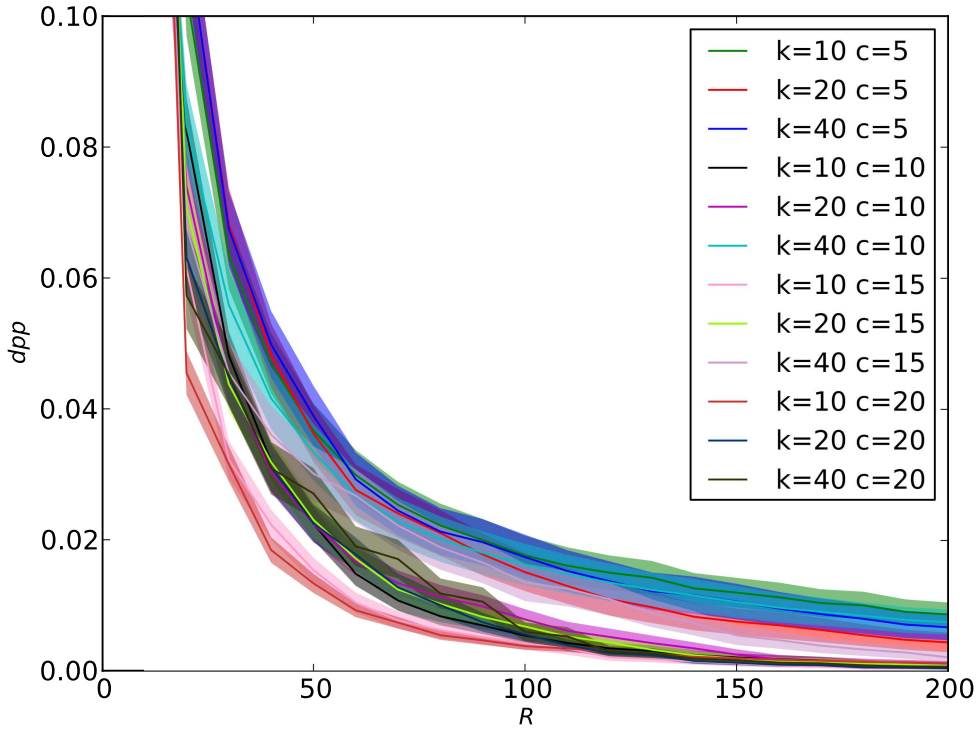

Figure 4.6: For every configuration (k, c), results obtained across 10 experiments are averaged. Mean values of d_{pp} are plotted as a function of R. Shaded areas correspond to the 95% confidence interval of the mean values.

over each 10% slice of the training set. Figure 4.7, shows mean values of d_{pp} and their 95% confidence intervals, for every R, in increments of 10. Figure 4.7 also shows the results obtained when using a modified model, referred to as *model-random-k10-c20*, that makes its predictions based on 20 random training-sets instead of the 20 nearest training-sets. As mentioned before, distances are measured in a Euclidean space formed by the features that are extracted from the sets. Comparing the results of *model-nearest-k10-c20* with *model-random-k10-c20* allows an evaluation of the effectiveness of the features in extracting the relevant characteristics of the set. In 8 of the 10 experiments, the prediction of *model-nearest-k10-c20* is better than the prediction of *model-random-k10-c20* that does not take into account input-set features. The prediction of *model-random-k10-c20* is, however, relatively close to the predictions of *model-nearest-k10-c20*. This is because the predictor does not follow the behaviour of the most similar or nearest input set, instead, it creates a distribution of the parameter values along

R	d_{pp} *model-nearest-k10-c20*	d_{pp} *model-random-k10-c20*	d_{pp} *no-model*
10	0.2419	0.2778	0.7695
50	0.0141	0.0274	0.1348
100	0.0051	0.0068	0.0587
150	0.0025	0.0031	0.0342
200	0.0005	0.0012	0.0241
500	0.0002	0.0001	0.0084
1000	0	0.0001	0.0021
2500	0	0	0.0005
3000	0	0	0.0002

Table 4.1: Performance comparison of the three predictors for various values of R.

the Pareto curve based on 10 input sets fading out the impact of feature distances.

Finally, these results are compared with a random exploration of the parameter space. Instead of using a model to suggest parameter combinations, these are generated as a vector composed of random numbers uniformly distributed in the range allowed by the parameters.

Subsequently, the results of the 10 experiments in Figure 4.7 are averaged to show the final results in Figure 4.8. When using *model-nearest-k10-c20*, at R=200, d_{pp} stabilizes to its smallest value. This means that after 200 P_{unseen} parameter configurations suggested by the model, the majority of the inputs find the true Pareto-curve in the resource-sharing design-space. When the model is used for future predictions, it will generate 200 parameter configurations to parameterize the resource-sharing algorithm. Thus, the Pareto curve found after these 200 executions will be the same or very close to the Pareto curve that would be found if the algorithm was executed exhaustively for all possible parameter configurations.

As shown in Figure 4.8, the random exploration is not a good solution to converge rapidly to the smallest possible value of d_{pp}. As it find solutions randomly located in the resource-sharing design-space, it slowly gets closer to the Pareto curve, however it fails to focus its search in the exploration of the curve.

Table 4.1 contains exact d_{pp} values for 9 R values in each of the three curves shown in Figure 4.8. Random exploration rapidly drops its d_{pp} values during

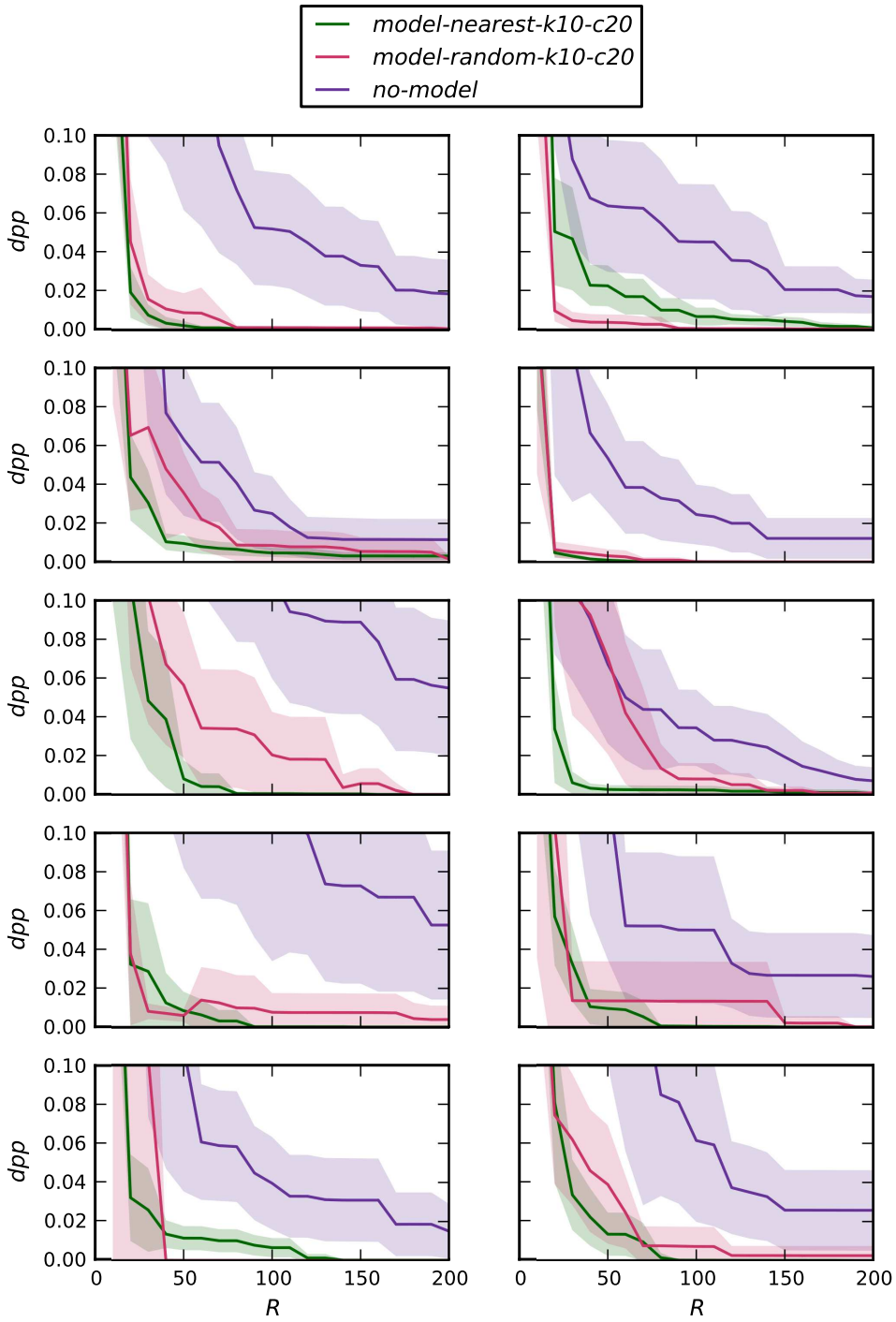

Figure 4.7: Experimental evaluation of *model-nearest-k10-c20*, in contrast with *model-random-k10-c20* and with *no-model* or random prediction. Every chart corresponds to an experiment where 10% of the training set was validated using the remaining 90% for training. Mean values of d_{pp} are plotted as a function of R. Shaded areas correspond to the 95% confidence interval of the mean values.

the first 200 runs. Afterwards, the exploration slows down reaching, only after 2500 runs, the same d_{pp} value obtained by the model after 200 runs. Thus, in comparison with random exploration, the predictive model reduces by 12.5 times the number of executions of the resource-sharing algorithm that are needed in order to find the optimal trade-offs in the design space.

Even though the number of features that the model processes is not very large, principal component analysis [63] was attempted in order to reduce the dimensionality of the feature space. However, the performance of the model was not improved; thus, the feature set was not transformed.

At this point, the model to predict Pareto parameters has been built. Given a set of ISEs, the predictive model can be used to effectively explore the design space of resource-sharing solutions, based on the patterns that are extracted from the previously exhaustively-explored spaces. The parameters of the model: $k = 10, c = 20$, and $R = 200$ have been determined, thus the model can now be used to predict unseen design spaces. In order to use the model, the quantitative features of a new set of ISEs need to be extracted. Based on these features, the model generates 200 parameter configurations that are expected to lead to the true Pareto-curve of the resource-sharing design-space. This means that the resource-sharing algorithm will be executed only 200 times instead of 37,044 times, which is the case in exhaustive exploration. This represents a speedup of more than two orders of magnitude achieved by the predictive model over exhaustive exploration.

4.7 Practical Usage of the Model

The previous sections explained in detail the design of a predictive model that speeds up the exploration of the resource-sharing design-space of a set of ISEs that is to be implemented. Once the model is designed and evaluated, it can be used to explore the resource-sharing design-space of new sets of ISEs. The sets used for training and evaluation of the model were extracted from the SNU-RT and UTDSP benchmark suites. Training sets comprised between 5 and 26 ISEs.

In this section, *model-nearest-k10-c20* is evaluated on 4 ISE sets extracted from benchmarks of a different suite, namely EEMBC [72] and CoreMark [73]. These 4 benchmarks have been chosen to be large in comparison with the bench-

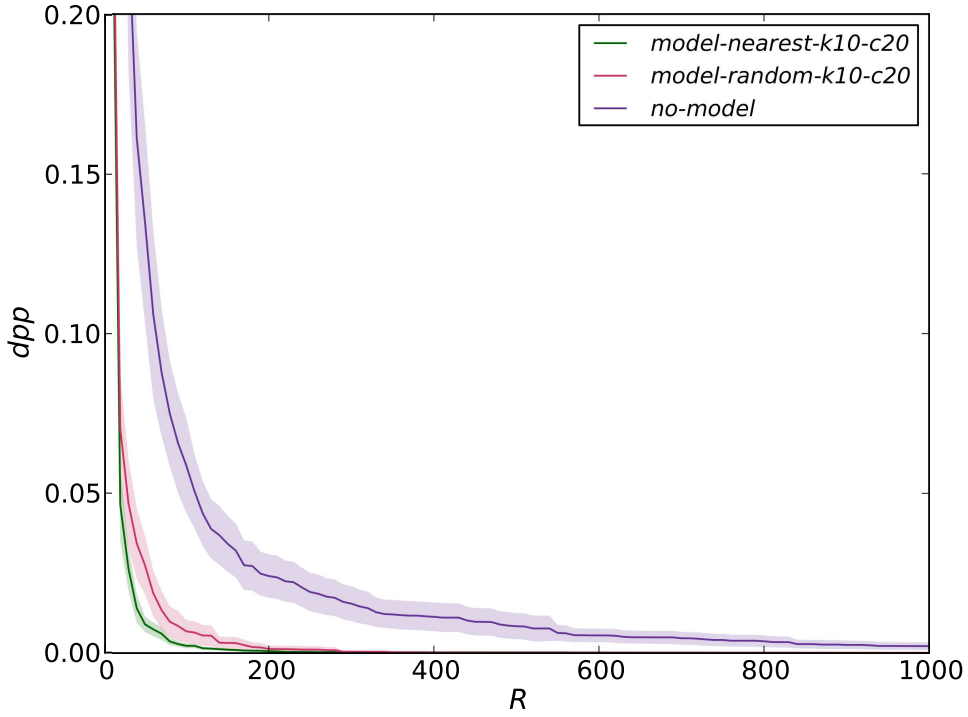

Figure 4.8: Experimental evaluation of *model-nearest-k10-c20*, in contrast with *model-random-k10-c20* and with *no-model* or random prediction. Results obtained across 10 experiments are averaged. Mean values of d_{pp} are plotted as a function of R. Shaded areas correspond to the 95% confidence interval of the mean values.

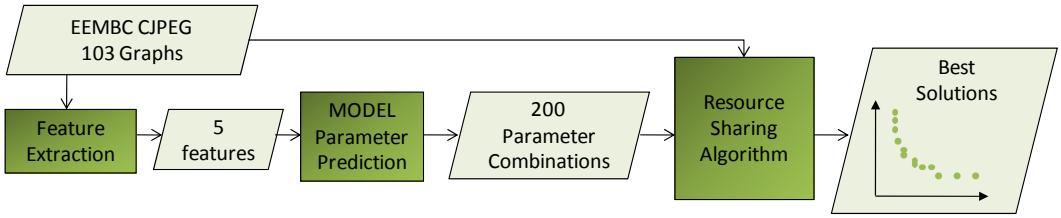

Figure 4.9: Practical usage of the predictive model to speedup the exploration of the resource-sharing design-space, given a set of ISEs extracted from the CJPEG benchmark.

marks used for training. This was done in order to assess the effectiveness of the model in scaling its predictions to new and vastly different input sets, thus demonstrating the generalization of the techninque. The chosen bencharks are: TTSPRK, QoS, CJPEG and CoreMark.

From TTSPRK a set of 20 ISEs was formed, from CoreMark a set of 40 ISEs,

from QoS a set of 49 ISEs and from CJPEG a set of 104 ISEs. Given the set of ISEs that will improve the execution time of the application or benchmark, the processor designer needs to be aware of the optimal trade-offs between area and speedup that represent alternatives for the hardware implementation of the ISEs. In order to find the optimal area-speedup trade-offs, the resource-sharing design-space needs to be explored by varying the parameters of the resource-sharing algorithm explained in Chapter 3. The exhaustive exploration of the design-space can be reduced by discretizing the values of the parameters with continuous values. Thus, the exhaustive exploration of the space implies running the resource-sharing algorithm 37,044 times. As this is impractical, specially when a large ISE set is to be merged, the model constructed in this chapter can be used.

The input of the model is the set of features extracted from each of the sets of ISEs. These features are: number of ISEs, standard deviation, 1st quartile, 2nd quantile, 3th quartile of the set of critical paths of the ISEs, weighted with their corresponding frequency of execution. Based on these characteristics, the predictive model suggests 200 parameter configurations that attempt to find the optimal trade-offs in the resource-sharing design-space. Then, the resource-sharing algorithm is run 200 times, one for each parameter configuration suggested by the model. At that point, the majority or the totality of optimal area-speedup trade-offs of the resource-sharing design-space will have been found.

In summary, the practical effect of having a model to predict the best parameter configuration for a given set is the reduction of the number of times that the resource-sharing algorithm has to be parameterized and executed. Instead of 37,044 executions, 200 is sufficient to have a set of optimal alternatives.

Figure 4.9 shows the process needed to use the model to explore the resource-sharing design-space of one of the EEMBC benchmarks and Algorithm 8 formalizes the process.

In order to see the effectiveness of the predictions, the resource-sharing design-space was also explored exhaustively and the predictor was configured to give 800 additional parameter configurations for a total of 1000 configurations. This experiment was performed for each of the 4 benchmarks and the results are shown in Figure 4.10. The figure also shows the results of predicting the parameter

Algorithm 8: Subroutine to generate predictions with *model-nearest-k10-c20* given a set of ISEs expressed as DAGs.

```
1: predictParetoParams (S) {extractFeatures(S) returns a vector of feature
   values extracted from the set of graphs in S.
   getNearestPtrain(F_S) returns a vector that contains all parameter settings that
   belong to the 10 F_train nearest to F_S.
   cluster(P_{k-train}) returns the elements P_{k-train} clustered in 20 groups.
   getDistributions(kClusters) returns the probability distribution of each cluster
   in kClusters.
   getParamSet(iDistribution) returns parameter settings p from a sample of the
   probability distribution iDistribution.}
2: ParetoParams ← ∅
3: F_S = extractFeatures(S)
4: P_{k-train} = getNearestPtrain(F_S)
5: kClusters = cluster(P_{k-train})
6: kDistributions = getDistributions(kClusters)
7: repeat
8:    for all iDistribution ∈ kDistributions do
9:       p = getParamSet(iDistribution)
10:      ParetoParams = ParetoParams + p
11:      iterationCount = iterationCount + 1
12:   end for
13: until iterationCount ≥ 200
14: return ParetoParams
```

configurations with *model-random-k10-c20* and with *no-model* or random prediction. As stated in previous sections, *model-random-k10-c20* randomly chooses 20 training sets to generate its predictions instead of picking the 20 most similar training sets.

The value of d_{pp} was measured at every 10th execution of the resource-sharing algorithm. As explained in previous sections, d_{pp} measures the distance between the Pareto curve found after an exhaustive exploration of the space and the Pareto curve found after R executions of the resource-sharing algorithm parameterized by the model.

As there are random components in the predictor, each experiment was performed 10 times. Figure 4.10 shows the mean values and the 95% confidence intervals of the measured d_{pp} values over the repetitions.

When the predictive model was used, TTSPRK and QoS reached d_{pp}=0 much faster than CoreMark and CJPEG. For all 4 benchmarks, after 200 parameter configurations tested, d_{pp} values were low enough to conclude that the majority of

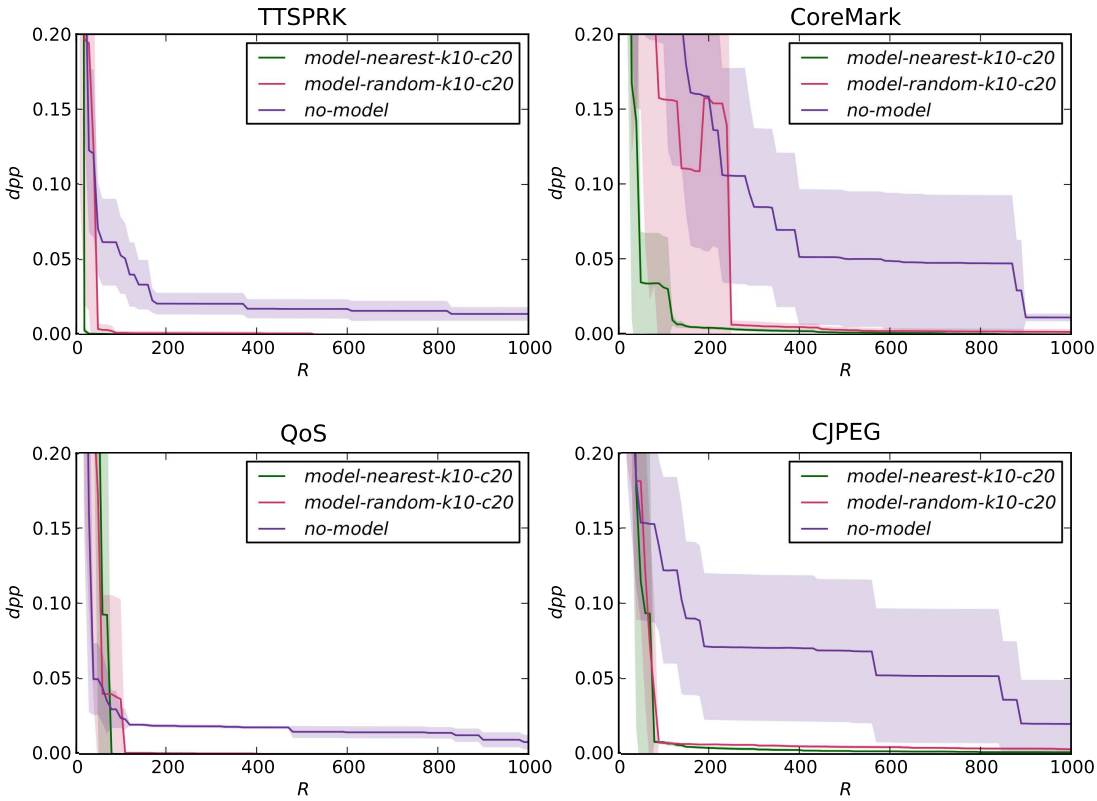

Figure 4.10: Results of predicting parameter values with *model-nearest-k10-c20* over 3 programs from the EEMBC benchmark suite and CoreMark. This is constrasted with results obtained with *model-random-k10-c20* and with *no-model* or random prediction. Mean values of d_{pp} are plotted as a function of R. Shaded areas correspond to the 95% confidence interval of the mean values.

the area-speedup trade-offs in the resource-sharing design-space had been found. On the other hand, for *model-random-k10-c20* d_{pp} values dropped slightly slower than *model-nearest-k10-c20*, and *no-model* was the worst search method in all of the experiments.

In conclusion, *model-nearest-k10-c20* predicted parameter values as expected for benchmarks that vastly differ from the ones that were used for training. This confirms that fixing the number of suggested parameter configurations to 200 adjusts well to a variety of input sets and leads to an acceptable approximation of the Pareto curve of the resource-sharing design-space.

Benchmark	d_{pp}	*model-nearest-k10-c20*		*no-model*		Speedup
		R	Running time	R	Running time	
TTSPRK	0.0002	200	21 min.	6950	605 min.	28×
CoreMark	0.004	200	111 min.	5440	2314 min.	21×
QoS	0	200	75 min.	5190	1937 min.	26×
CJPEG	0.0039	200	305 min.	1900	2850 min.	9×

Table 4.2: Running time to explore the resource-sharing design-space for each benchmark. Speedups are calculated as the running time of the random exploration: *no-model*, over the running time of the exploration with the model: *model-nearest-k10-c20*

Design-Space Exploration Running-time

Table 4.2 shows the running time that was measured on the actual exploration of the resource-sharing design-space for each benchmark. Experiments were perfomed on a workstation equiped with 4 Xeon processors running at 3 GHz, and 4 GB of RAM. The running time shown for *model-nearest-k10-c20* corresponds to the time spent in the exploration of the 200 parameters suggested by the model. The d_{pp} that is calculated at $R = 200$ on the exploration with *model-nearest-k10-c20*, was used to find the value of R for *no-model*. This is, the value of R for *no-model* is the number of randomly-generated parameter configurations that need to be explored in order to reach the d_{pp} that the model achieved at $R = 200$. The running time shown for *no-model* corresponds to the time spent on these exploration.

Speedups are calculated as the running time of the random exploration: *no-model*, over the running time of the exploration with the model: *model-nearest-k10-c20*. The predictive model achieved speedups between 9× and 28× on running time over random exploration, thus confirming the predictive power of the model, and demonstrating important time savings in the exploration of new design spaces.

4.8 Conclusions

This chapter has explored a practical and novel predictive model that can be used to quickly find the optimal implementation trade-offs in the resource-sharing

design-space of a set of ISEs. Compared to an exhaustive exploration of the design space, the predictive model is shown to reduce by two orders of magnitude the number of executions of the resource-sharing algorithm that are required in order to find Pareto-optimal solutions.

The next chapter makes use of this model in order to explore the design space at the ISE selection level. More specifically, the alternatives found for the implementation of a set of ISEs are used to decide what ISEs to select, given a large set of candidates and some area constraints.

The power of the design-space exploration at the selection level lies in the capability to actively interact with the implementation stage, where resource sharing takes place. This interaction is only possible with the presence of the proposed predictive model, as repeated calls to an exhaustive exploration would be absolutely infeasible.

Thus, it is shown that learning techniques that extract patterns from previously-explored spaces can be effectively used in order to solve complex problems, that create large design spaces but that are likely to give rise to more efficient designs.

Chapter 5

ISE Selection

5.1 Introduction

Previous chapters have assumed that a fixed set of ISEs is to be implemented. This is the case either when all of the ISEs that can be identified in the application are chosen to be implemented, or when a subset of them is chosen to fit a given area constraint.

The selection problem has been, so far, tackled by considering that each ISE is to be implemented individually. This consideration dismisses important implementation issues such as resource sharing, by assuming that speedups and area requirements of the ISEs remain unchanged after hardware implementation.

However, when resource sharing is considered for implementation, the area requirements of a set of ISEs could vary significantly. Moreover, for every possible subset that can be formed from the ISE candidate set, there is a subspace that can be explored, and the result of this exploration is only available after attempting to share resources amongst the selected ISEs under different constraints. The exploration of this subspace, as seen in Chapter 3, finds many implementation solutions to the given subset of ISEs. This design space, however, is not complete as the outcome of its exploration would be different if one or several ISEs were removed and/or added to the subset.

If n instructions are identified in the application code, the design space of solutions would only be completely discovered when the 2^n selection alternatives are considered and their corresponding resource-sharing trade-offs are explored.

The trade-offs between speedup and area that are found in a selection of ISEs depend not only on the area and speedup that the individual ISEs require or yield but also on the way that they can be merged with each other.

The unpredictability of the solutions that can be obtained from a given selection implies that the problem cannot be formulated as a conventional optimization problem. Thus, exhaustive exploration is the only option to guarantee the finding of all possible trade-offs. However, as the number of alternatives is exponential in the number of ISE candidates, an exhaustive exploration is intractable.

This chapter presents a heuristic to iteratively explore the design space of selection alternatives given a set of ISE candidates and a given area range that is available for the AFU. This heuristic is in turn used to perform a global exploration of the selection alternatives that is able to show the designer a wide range of trade-offs between speedup and area that can be obtained from a set of ISE candidates.

This chapter is organized as follows. Section 5.2 presents two motivational examples that illustrate the trade-offs that can be found in the design space of selection alternatives when resource sharing is used for implementation. Section 5.3 presents a new heuristic where the ISE selection problem is combined with the implementation process in order to maximize utilization and speedup within a given area budget. Section 5.4 shows how this heuristic can be extended in order to explore the entire design space of solutions. The result of this exploration is a set of selection-implementation alternatives, each of which represents a unique trade-off between performance gain and cost. Section 5.5 describes a complete hardware/software partitioning framework that uses the techniques presented so far in this thesis, and that represents a complete solution to efficiently map an application onto an extensible processor. Section 5.6 presents the experiments that were performed and the results that were obtained, thus demonstrating that the heuristic proposed in this chapter advances the state-of-the art in the ISE selection problem. Finally, Section 5.7 concludes this chapter.

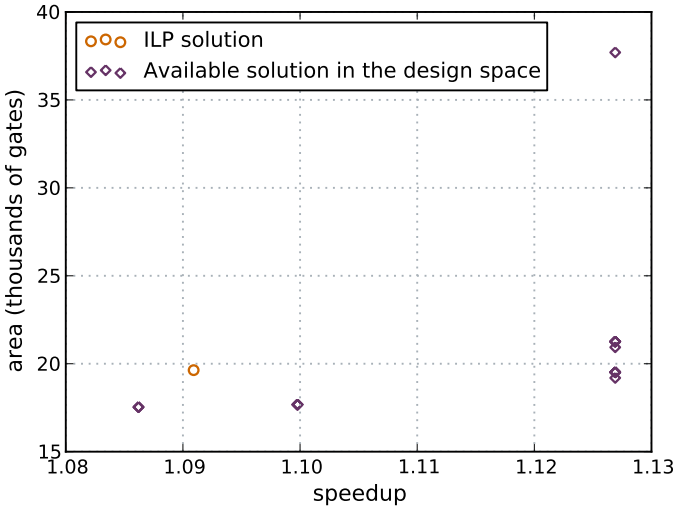

Figure 5.1: Example from the AES application. The ILP solution to the selection problem under an area constraint of 20 Kgates is contrasted with a better solution that is available in the selection + resource-sharing design-space.

5.2 Motivational Example

In order to compare the solutions obtained from different subsets of the ISE candidate set, these are characterized by their area requirements and by the application speedup that yields their implementation.

Figure 5.1 shows an example taken from the AES application of the EEMBC benchmark suite. A set of 44 ISEs is identified from the application code. It is assumed that the area constraint imposed on the design is 20 Kgates. If the ISEs were to be implemented individually, a conventional ILP solution[1] would select 10 of the ISEs that fit in 20 Kgates and yield a speedup of 1.09×. However, when resources can be shared amongst the ISEs, one could find that a different selection of 10 ISEs can yield a higher speedup in the given area. When no resources are shared amongst the ISEs the area occupied by that selection would be far beyond the area constraint. However, when the resource-sharing design-space is explored, it is found that, the 10 ISEs are highly compatible and can be merged to share resources with no penalties in speedup occupying only 18 Kgates and yielding a significantly higher speedup of 1.13×.

[1]ILP formulations are solved using an implementation of a dynamic programing algorithm that makes use of a recursive procedure to obtain an exact solution [74].

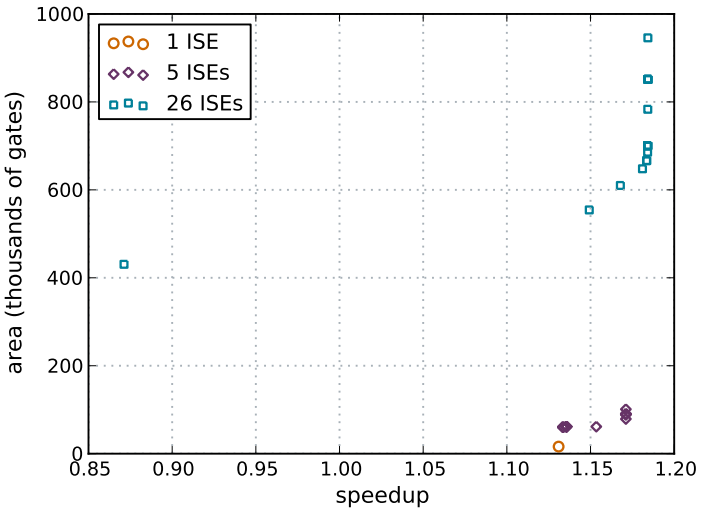

Figure 5.2: Example from the LPC application. The resource-sharing trade-offs found for 3 different ISE selections are compared.

Another example is presented in Figure 5.2. This figure shows a set of speedup-area solutions that can be obtained from a set of 26 ISEs identified in the LPC application from the UTDSP benchmark suite. When all of the 26 ISEs are chosen for implementation, a solution with no resource-sharing requires 945 Kgates yielding 1.18× speedup. After exploring the resource-sharing design-space, it is found that the same speedup can be obtained using only 647 Kgates. When the resource-sharing algorithm attempts to share more resources in order to further reduce area, it can get down to 430 Kgates, but at that point, the application is slowed down as the critical path of the ISEs is drastically increased as a result of an aggressive merging process. On the other hand, when only 5 of the ISE candidates are selected, a maximum of 1.17× speedup can be achieved using 101 Kgates without resource sharing and using 78 KGates with resource sharing. This area requirement can be further reduced to 60 Kgates to yield just 1.15× speedup. Finally, Figure 5.2 also shows that by selecting one of the ISEs, a 1.13× speedup can be obtained using only 16 Kgates. This means that 92% of the maximum speedup achievable with the 26 ISE candidates can be obtained by implementing only one ISE demanding 2.5% of the area that the maximum-speedup solution requires. Similarly, 98% of the maximum speedup can be obtained by implementing 5 ISEs demanding 9.2% of the area that the maximum-speedup solution

requires. In this case, selecting all of the ISEs for implementation does not seem a good alternative, even if there is enough die area available for the custom unit. The area that needs to be spent to achieve the last 2% of the maximum speedup might very likely give better returns by considering other alternatives such as caches, registers or functional units.

In conclusion, important trade-offs can be found when selecting different combinations of ISEs to form the candidate set, and when these ISEs can share resources in their implementation. Therefore, a thorough exploration of the speedup-area space at the selection level should be considered before committing to any particular ISE selection.

5.3 Local Iterative Exploration

This section describes a heuristic to explore the trade-offs between speedup and area that can be generated from a set of ISE candidates, C, extracted from an application, within a given area range. Given C, speedup-area solutions can be obtained by selecting any combination of ISEs. This means that there are $2^{|C|}$ selection alternatives. Each of these alternatives, in turn, generates a set of resource-sharing trade-offs, where each trade-off represents a solution in terms of speedup and area.

Each ISE is characterized by the area that it requires for hardware implementation, and by the maximum speedup that the application can obtain by including this ISE. These characteristics are extracted under the consideration that only the ISE in question is to be implemented.

The set of ISEs selected from C for implementation will be denoted as S. Initially, S is an empty set and all of the candidates extracted from the application belong to the set C. The algorithm then iterates to form the set S that generates maximum speedup within the area range.

Selecting an ISE to be moved from C to S

This decision is based on the speedup associated with the ISEs in C and their structural compatibility with the ISEs already in S. The metric η is used to guide

this decision. It is calculated for each ISE in C, c_i, as follows:

$$\eta_i = w_1 \cdot compatibility_{(c_i,S)} - w_2 \cdot speedup_i \tag{5.1}$$

$compatibility_{(c_i,S)}$ expresses how a candidate ISE c_i matches the set S in terms of potential area savings and its impact on the critical path of the ISEs. It is the average amongst the maximum θ values found in merging c_i with every s_j. It is computed as follows:

$$compatibility_{(c_i,S)} = \frac{\sum_{j=1}^{|S|} max(\theta_{c_i}, \theta_{s_j})}{|S|} \tag{5.2}$$

As seen in Chapter 3, the metric θ is a quantification of the trade-off between area and latency when merging two ISEs. When the graphs corresponding to c_i and s_j are merged into G', θ_{c_i} and θ_{s_j} are calculated according to the following equations:

$$\theta_{c_i} = \frac{L_{G'} - L_{c_i}}{L_{G'}} \times \left(\frac{A_{G'}}{A_{c_i} + A_{s_j}} \right) \tag{5.3}$$

$$\theta_{s_j} = \frac{L_{G'} - L_{s_j}}{L_{G'}} \times \left(\frac{A_{G'}}{A_{c_i} + A_{s_j}} \right) \tag{5.4}$$

L_{c_i}, L_{s_j} and A_{c_i}, A_{s_j} are respectively the critical path and area of c_i and s_i, and $L_{G'}$ and $A_{G'}$ are respectively the critical path and area of G'. The first term in θ represents the relative decrease in latency perceived by *not* performing the merge, whereas the second term represents the area savings that *do* result from merging c_i and s_j.

Smaller values of $compatibility_{(c_i,S)}$ express a better match between a candidate c_i and S, in terms of area savings and critical path implications. In contrast, higher values of speedup indicate that the selection of c_i might have a good impact on the solutions obtained from merging the set $S + c_i$. Therefore, relatively small values of η_i imply greater chances to include c_i into the selected set S.

Selecting an ISE to be moved from S to C

This decision is based on the speedup associated with the ISE in S and on a measure of how well they merge with the other ISEs in S. The metric γ is used to guide this decision. It is calculated for each ISE in S, s_j, as follows:

$$\gamma_j = w_1 \cdot \beta_j - w_2 \cdot speedup_j \tag{5.5}$$

As seen in Chapter 3, β_j is a metric that quantifies the effect of merging s_j on the average critical path of S. This, in turn, is amortized by the area savings obtained by including s_j in the merging process.

β_j can be found after merging the set S by executing the resource-sharing algorithm with no restrictions. That is, the execution is parameterized with the maximum values of α_T, β_T and θ_T. β_j is calculated as follows:

$$\beta_j = \frac{|\hat{L} - L_j|}{\max_{k=1}^{|S|} L_k} \times (1 - M_j) \tag{5.6}$$

where:

$$\hat{L} = \frac{\sum_{k=1}^{|S|} L_j}{|S|} \tag{5.7}$$

and M_i is the percentage of area corresponding to operations in s_j that can be shared with other ISEs in S, divided by the total area that could be shared in the whole process.

Higher values of β express a worse match between s_j and S in terms of area savings and critical path implications. In contrast, higher values of speedup indicate that s_j has a good impact on the solutions obtained from merging S. Therefore, relatively large values of γ_j imply greater chances to exclude s_j from S.

The optimal relation between the weights w_1 and w_2 has been determined experimentally to be $w_1 = 1.2 \cdot w2$, favoring slightly the speedup component when calculating η and γ. By construction, $compatibility_{(c_i,S)}$ in equation 5.1 and β_j in equation 5.5 have values in the range [0,1]. On the other hand, speedup values are normalized and scaled to have values of $speedup_j$ and $speedup_j$ in the range [0,1].

Iterative Search for Maximum-speedup Solutions

The set S is initialized with one ISE randomly chosen from C. Then, in every iteration only one ISE is removed or added to S. The iterations are set to add ISEs to S when the implementation of S with maximum area requirements falls below the given area range. Similarly, the iterations are set to remove ISEs from S when the implementation of S with minimum area requirements falls below the given area range. Either of these two cases means that the solutions found with S do not contribute to the exploration of the area range. Thus, ISEs candidates that by themselves require area out of the area range cannot be included in S.

At the start of the process the iterations are set to add ISEs to S. The number of iterations are ideally continued until no better solutions can be found in the area range. As this is unknown when the exploration is being performed, the number of iterations is constrained to a maximum value. Experimentally, a maximum value of 100 iterations gave the same results as higher values.

If the iteration is set to add ISEs to S, η_i is calculated for every c_i and the c_i with smallest η_i will be selected to be part of S. On the other hand, if the iteration is set to remove ISEs from S, γ is calculated for every s_j and the s_j with greater γ_j will be selected to return to C.

Subsequently, the minimum and the maximum area requirements of S are checked to confirm that solutions found in the exploration of the resource-sharing space of S contribute to the exploration of the given area range. The minimum area requirement is found by merging the ISEs belonging to S with no constraints by executing the resource-sharing algorithm described in Chapter 3 Algorithm 7 with parameters: α_T=1, β_T=1, θ_T=1, $multiOp$=0, $grouping$=0. Under these parameters, the process maximizes sharing amongst the input ISEs in order to obtain the minimum-area solution. The maximum area required to implement S can be found by summing up the area of the ISEs belonging to S, and corresponds to the case when no resource sharing is performed amongst the ISEs in S. The maximum area requirement of S can also be found by executing the resource sharing algorithm with parameter values that do not allow any merging in the process: α_T=0, β_T=0, θ_T=0, $multiOp$=0, $grouping$=0.

Algorithm 9: Local Exploration

```
1: localExploration (C, areaUpperBound, areaLowerBound, ThetaArray)
   {getParetoPoints(X) returns the Pareto points in a set X of (area,speedup)
   pairs. getRandom(x, y) returns a random number uniformly in the range [a,b].
   getSpeedup(x) returns the speedup corresponding to ISE x}
2: S ← c_getRandom(1,|C|)
3: ParetoPoints ← ∅
4: setting = ADD
5: repeat
6:    if setting = ADD then
7:       for all c_i ∈ C do
8:          compatibility_(ci,S) = getCompatibility (C, S, i, ThetaArray)
9:          η_i = (w_1 · compatibility_(ci,S) − w_2 · getSpeedup(c_i)) · getRandom(0.9, 1.1)
10:         if η_i < η_k, ∀ k < i then
11:            min = i
12:         end if
13:      end for
14:      S ← S + c_min
15:      C ← C − c_min
16:   end if
17:   if setting = REMOVE then
18:      BetaVector = runRS(S, α_T = 1, β_T = 1, θ_T = 1, multiOp = 0,
         grouping = 0)
19:      for all s_j ∈ S do
20:         γ_j = (w_1 · BetaVector[j] − w_2 · getSpeedup(s_j)) · getRandom(0.9, 1.1)
21:         if γ_j > γ_k, ∀ k < j then
22:            max = j
23:         end if
24:      end for
25:      S ← S − s_max
26:      C ← C + s_max
27:   end if
28:   maxArea_(S) = runRS(S, α_T = 0, β_T = 0, θ_T = 0, multiOp = 0, grouping = 0)
29:   if maxArea_(S) < areaLowerBound then
30:      setting = ADD
31:   else
32:      minArea_(S) = runRS(S, α_T = 1, β_T = 1, θ_T = 1, multiOp = 0,
         grouping = 0)
33:      if minArea_(S) > areaUpperBound then
34:         setting = REMOVE
35:      else
36:         SolutionsRS_(S) = exploreRS(S)
37:         ParetoPoints = getParetoPoints(ParetoPoints + SolutionsRS_(S))
38:      end if
39:   end if
40:   iterationCount = iterationCount + 1
41: until iterationCount ≥ 100
42: return ParetoPoints
```

When either the minimum or the maximum area requirement of S is proved to fall in the given area range, the resource-sharing design-space of S is completely explored in order to find the optimal implementation trade-offs between area and speedup that can be obtained from S.

Optimal implementation trade-offs are found by using the predictive model, presented in Chapter 4, to generate a set of parameter combinations that will parameterize consecutive executions of the resource-sharing algorithm. Every execution will generate a potentially different solution, with a specific trade-off between area and speedup.

Thus, information of resource-sharing compatibility amongst the ISEs is used in order to drive exploration of the selection design-space towards implementation solutions that are likely to increase the utilization of the given area resources. This iterative process is detailed in Algorithm 9.

Prior to the iterations, every possible pair of candidate ISEs (c_i, c_j) is merged in order to find θ_i and θ_j. This corresponds to the function described in Algorithm 10. The maximum value between θ_i and θ_j will be stored in $ThetaArray[i][j]$ and $ThetaArray[j][i]$. $ThetaArray[i][j]$ values are used during the algorithm iterations to calculate η. The initialization of $ThetaArray$ implies executing the resource-sharing algorithm $\frac{|C|^2-|C|}{2}$ times.

The inputs of the main routine are the set of ISE candidate identifiers C, annotated with their corresponding area and application speedup, the area range to explore, and the $ThetaArray$ array that can be previously found from C using the function in Algorithm 10. The resulting output is the set of Pareto points, expressed as (area,speedup) pairs, that have been found throughout the iterative search.

As γ and η are used to guide an iterative search, and are not absolute metrics to guarantee an optimal choice, every time their values are calculated they are randomly modified by a maximum of 10% of its value. This is done by multiplying the original γ and η values by a randomly generated number between 0.9 and 1.1. This transformation implies that the instruction c_i with minimum original η value has the highest likelihood of having the minimum transformed η value. And similarly, the instruction s_j with maximum original γ value has the highest likelihood of having the maximum transformed γ value. This transformation

Algorithm 10: Subroutine for function `getThetaArray`.

```
1: getThetaArray (C)
   {max(x,y) returns the greatest value between x and y. }
2: for i = 1 to i = |C| do
3:    for k = i + 1 to k = |C| do
4:       (θ_i, θ_k)=runRS({c_i, c_k}, α_T=1, β_T=1, θ_T=1, multiOp=0, grouping=0)
5:       ThetaArray[i][k] = max(θ_i, θ_k)
6:       ThetaArray[k][i] = max(θ_i, θ_k)
7:    end for
8: end for
9: return ThetaArray
```

Algorithm 11: Subroutine for function `getCompatibility`

```
1: getCompatibility (C, S, i, ThetaArray)
2: compatibility_(ci,S) = 0
3: for all s_j ∈ S do
4:    compatibility_(ci,S) = compatibility_(ci,S) + ThetaArray[i, j]
5: end for
6: compatibility_(ci,S) = compatibility_(ci,S)/|S|
7: return compatibility_(ci,S)
```

Algorithm 12: Subroutine for function `exploreRS`

```
1: exploreRS (S)
2: ParetoParams=predictParetoParams(S)
3: SolutionsRS_(S) ← ∅
4: for all p ∈ ParetoParams do
5:    α_T = p[0]
6:    β_T = p[1]
7:    θ_T = p[2]
8:    multiOp = p[3]
9:    grouping = p[4]
10:   (area, speedup) = runRS(S, α_T, β_T, θ_T, multiOp, grouping)
11:   SolutionsRS_(S) = SolutionsRS_(S) + (area, speedup)
12: end for
13: return SolutionsRS_(S)
```

prevents the iteration from falling into local optima or deadlocks that explore repetitively the same solutions.

Function `runRS`(S, α_T, β_T, θ_T, *multiOp*, *grouping*), in Algorithm 7 of Chapter 3, can be modified to return internal values of the process such as: the values of θ found during the last merge, the values of β corresponding to each ISE of the input set, the area of the resulting solution, and the speedup of the resulting

solution.

Function `exploreRS`(S), in Algorithm 12, executes the parameterization of the merging process. Function `predictParetoParams`, in Algorithm 8 of Chapter 4, is used in order obtained the predicted parameters that lead to find the optimal implementation trade-offs available from a selection set S.

Finally, function `getCompatibility`, in Algorithm 11, is called for the calculation of $compatibility_{(ci,S)}$.

5.4 Global Iterative Exploration

Implementation solutions derived from C may have area requirements ranging from the area of the smallest ISE to the area of all of the ISE candidates implemented without sharing resources. This is the case where at least one ISE is chosen for implementation. This area range can be explored incrementally from the minimum area to the maximum area using the local iterative exploration described in Algorithm 9. Area subranges for local explorations were chosen to be as wide as the area of a 32-bit multiply-adder. Algorithm 13 describes the routine that can be used to explore globally the design space of selection alternatives given a set of ISE candidates C. The function `getMaxMinArea` in Algorithm 14 obtains the values of minimum area and maximum area from the candidate set C in order to find the area range in which the exploration should be performed. The result of the exploration is the set of Pareto-optimal solutions, which represent the trade-offs between speedup and area, that are available to the designer given the ISE candidates extracted from the application.

5.5 Hardware/Software Partitioning Framework

This thesis proposes a framework that represents a complete solution to efficiently map an application onto an extensible processor. Given a software application, a set of trade-off solutions are exposed to the designer. Each solution corresponds to a selection of ISEs and its corresponding hardware implementation. This hardware implementation describes the low level details of the AFU and can be connected to the extensible processor.

Algorithm 13: Global Exploration

```
1: globalExploration (C)
   {getParetoPoints(X) returns the Pareto points in a set X of (area,speedup)
   pairs.
   getArea(x) returns the area required by operator x.}
2: ThetaArray =getThetaArray(C)
3: ParetoPoints ← ∅
4: (maxArea, minArea) = getMaxMinArea(C)
5: areaStep = getArea(MULTIPLY-ADDER)
6: areaLowerBound = minArea
7: areaUpperBound = minArea + areaStep
8: repeat
9:    ParetoPointsLocal = localExploration(C, areaUpperBound,
      areaLowerBound, ThetaArray)
10:   ParetoPoints = getParetoPoints(ParetoPoints + ParetoPointsLocal)
11:   areaLowerBound = areaLowerBound + areaStep
12:   areaUpperBound = areaUpperBound + areaStep
13: until areaUpperBound ≥ maxArea
14: return ParetoPoints
```

Algorithm 14: Subroutine for function `getMaxMinArea`

```
1: getMaxMinArea (C)
   {getArea(x) returns the area required by operator x.}
2: for i = 1 to i = |C| do
3:    maxArea = maxArea + getArea(c_i)
4:    if getArea(c_i) < minArea, ∀ k < i then
5:       minArea = getArea(c_i)
6:    end if
7: end for
8: return (maxArea,minArea)
```

The proposed flow of an *extension generation unit* is depicted in Figure 5.3. An ISE identification phase generates ISE candidates from the source code of an application. Identification algorithms such as [26] and [29] can be adopted for this stage. For each basic block, the most profitable ISE is selected iteratively until no more subgraphs are feasible. Since the ISEs that are listed as candidates are non-overlapping, a subsequent isomorphism check is straightforward. ISEs with equivalent functionality are combined and their execution frequency is accumulated.

The modified ISE candidate list is then passed to the selection unit. The selection process can obtain information about the resource-sharing implementation

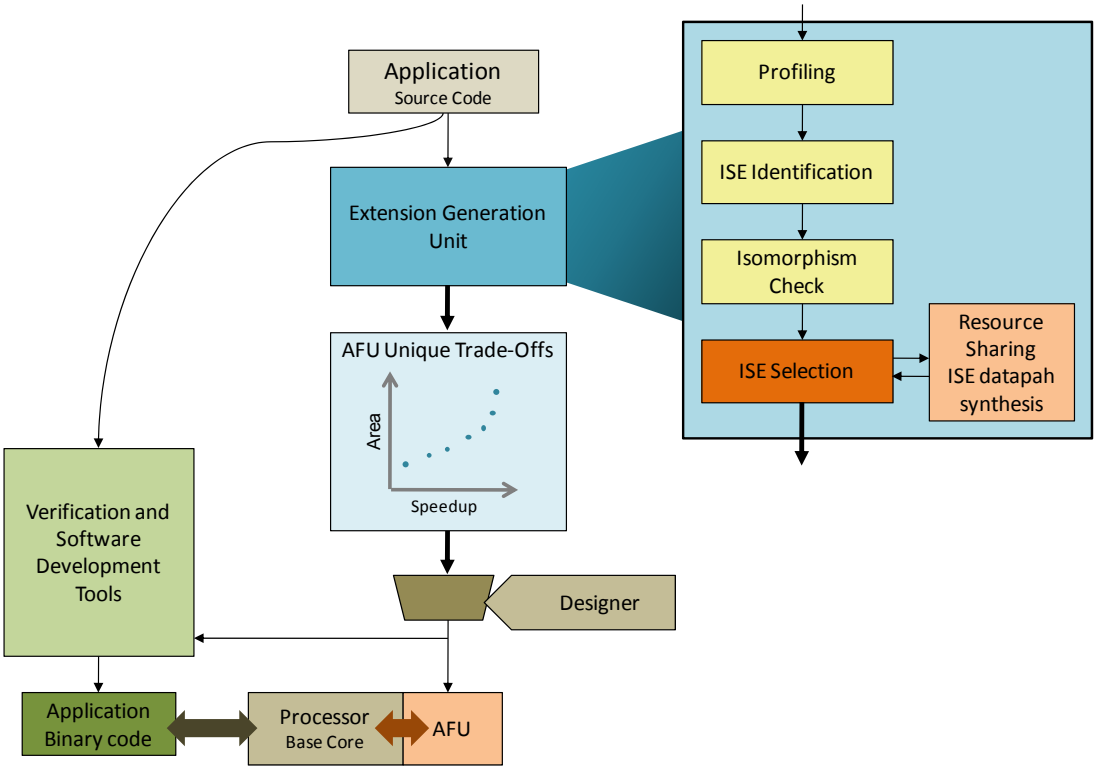

Figure 5.3: Proposed design flow to extend a processor with an AFU tuned in order to speedup an application.

of any preliminary selected subset. From the selection unit, the designer obtains a set of solutions that represents trade-offs between speedup and area from which the designer can choose the one that best suits the design goals and constraints.

The *extension generation unit* is in turn connected to the processor design framework. Since the ISA of the processor is to be extended, the compiler must be extended accordingly such that the application stream can take advantage of the newly-generated functional unit. The complete processor design framework is depicted in Figure 5.3.

5.6 Experiments and Results

Six benchmarks were taken for the evaluation of the proposed approach: LPC and ADPCM from the UTDSP benchmark suite [71], FIR and LMS from the SNU-RT benchmark suite [70], and AES from the EEMBC benchmark suite [72] and from CoreMark [73]. ISE identification was performed on each benchmark using an

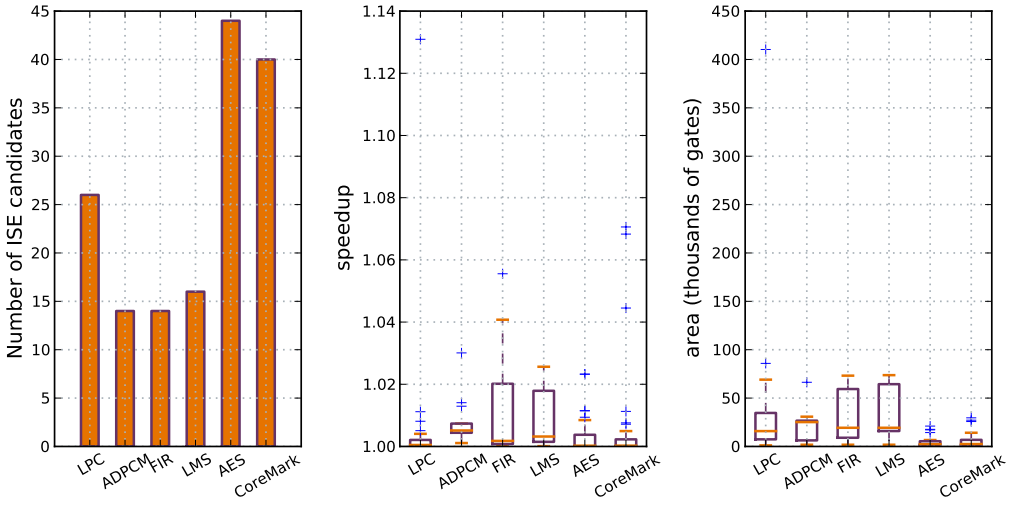

Figure 5.4: Characteristics of the set of ISE candidates extracted from the chosen benchmarks. The first plot shows the number of ISE candidates extracted per benchmark. The second plot shows the distribution of the speedups that can be obtained from individually implementing the ISE candidates. The third plot shows the distribution of the areas required to individually implement the ISE candidates. Blue + marks represent outlier values.

implementation of *ISEGEN* [26]. In this implementation, ISEs can include a full range of integer, logical and floating-point operations, including divisions. The identification was set to constrain ISEs to have a maximum of 12 input values and 8 output values corresponding to the register file I/O port constraints of the target processor.

Figure 5.4 shows the characteristics of the set of ISE candidates that was obtained from each benchmark: the number of candidates that was obtained from the identification phase, the distribution of the area requirements of the ISE candidates, and the distribution of the speedups that are expected with the separate implementation of each ISE candidate.

In order to evaluate the local exploration, an area constraint is imposed for the selection of ISEs given the candidates obtained from the chosen benchmarks. The area constraint or the area that the designer can invest on ISEs was chosen to be the same as the area occupied by the baseline target processor which is approximately 25 Kgates. The local exploration was performed using the routine described in Algorithm 9, in order to find the selection that maximizes

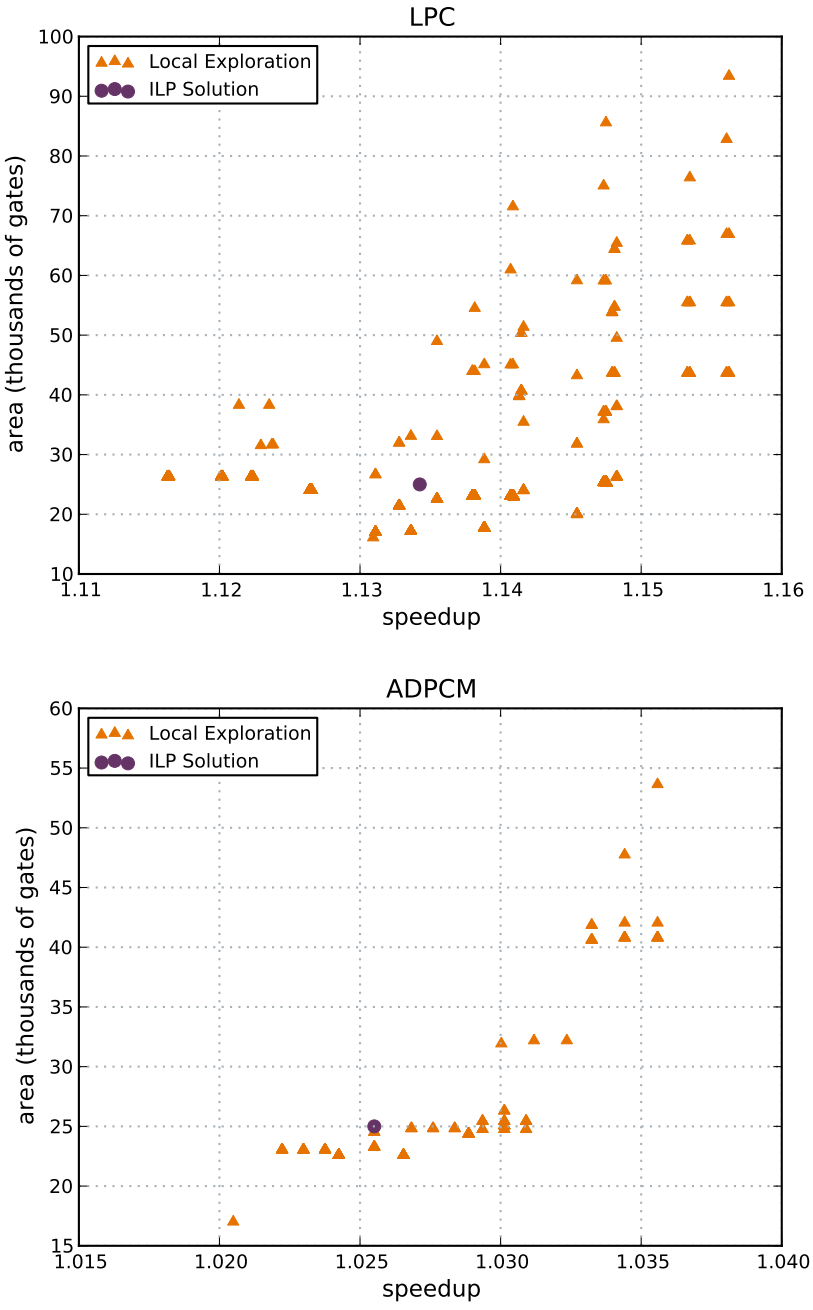

Figure 5.5: Results of the local exploration and of the ILP solution, given an area constraint of 25 Kgates.

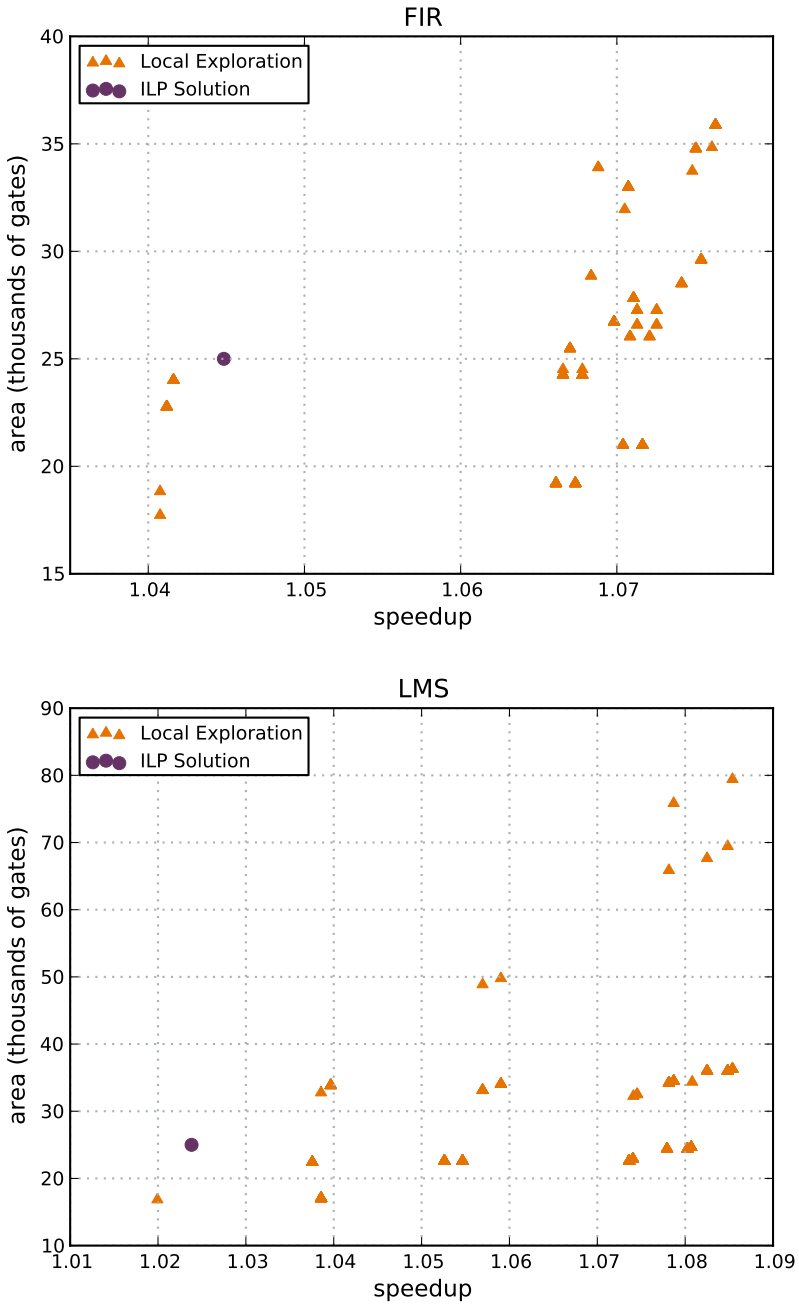

Figure 5.6: Results of the local exploration and of the ILP solution, given an area constraint of 25 Kgates, for the benchmarks chosen from the UTDSP benchmark suite.

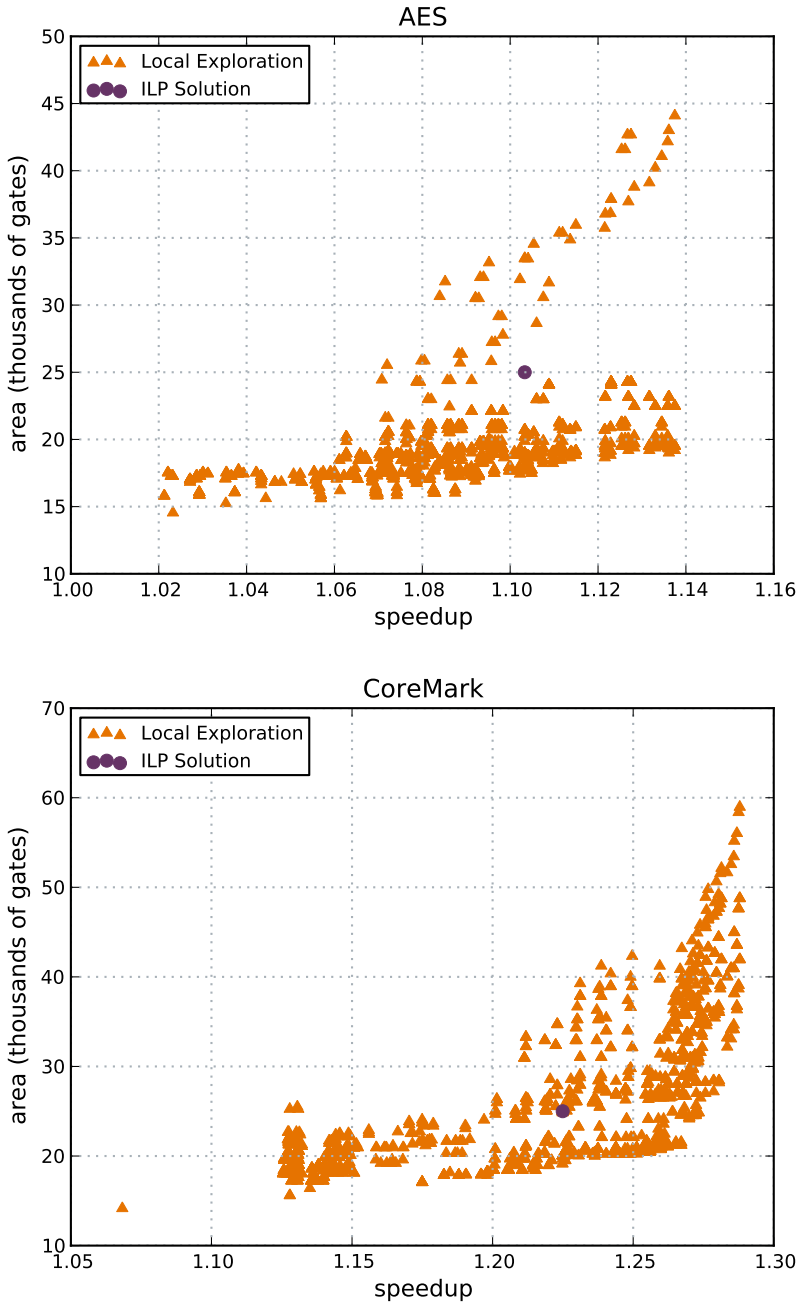

Figure 5.7: Results of the local exploration and of the ILP solution, given an area constraint of 25 Kgates, for the benchmarks chosen from the SNU-RT benchmark suite.

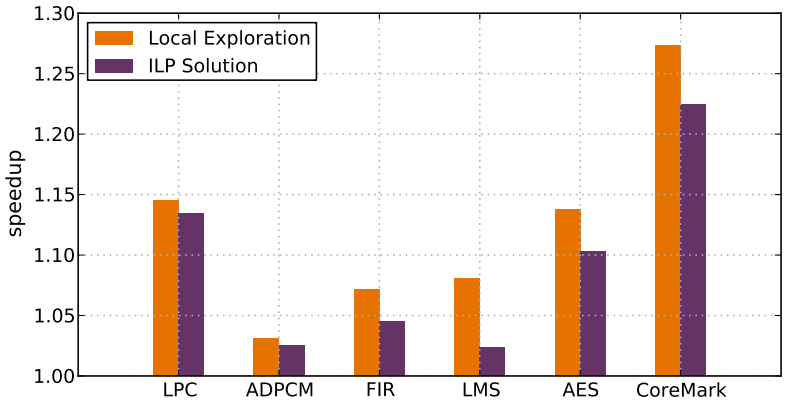

Figure 5.8: Speedups obtained with the local exploration and with the ILP solution, under an area constraint of 25 Kgates.

speedup using the given area and considering resource sharing for ISE implementation. The *areaUpperBound* provided for the exploration was 25 Kgates and the *areaLowerBound* was 25 Kgates minus the area of a 32-bit multiply-adder. For the sake of comparison, the selection problem was solved using a standard ILP formulation [33], which maximizes the obtained speedup under an area constraint of 25 Kgates, given a set of candidates with their corresponding areas and speedups.

Figures 5.5 to 5.7 shows the results of this experiment performed for each of the benchmarks. Each plot shows the speedup-area solutions that were obtained from the local exploration and one speedup-area solution obtained from the ILP formulation. In all cases, higher speedups were obtained, within the given area constraint, with the local exploration. In comparison with the ILP solutions, the local exploration showed speedup improvements from 8% in LPC to 238% in LMS, as shown in Figure 5.8. Hence, the integration of selection and resource sharing in ISEs yields better solutions by driving the exploration of the selection design space towards implementation alternatives that are likely to increase the utilization of the given area resources.

Due to the varied characteristics of the ISE candidate sets obtained from different benchmarks, local explorations develop differently. For instance, as displayed in Figure 5.7, more solutions were found in the local exploration of AES and CoreMark. This is due to the higher number of candidates and the small size

of most of them, as shown in Figure 5.4. As candidates to include in a selection need to be smaller than the area constraint, candidate sets with large ISEs such as in LPC, ADPCM, FIR and LMS, present a reduced set of candidates under area restrictions. Thus, the more ISE candidates, the more selection alternatives and resource-sharing opportunities to explore.

Unlike traditional solutions [33], the local exploration is able to give the designer a range of trade-offs in the vicinity of the given area from which the designer can choose. The visibility of such trade-offs can, for instance, show that less area needs to be spent in order to get the maximum speedup. Such is the case in AES, where the maximum speedup that is obtainable with 25 Kgates can be achieved with 18 Kgates. Thus, a local exploration has already hinted that constraining the solution to an exact area requirement does not always lead to the best utilization of the available resources, and therefore, other area requirements might offer a better trade-off between speedup and area.

In order to see all of the trade-offs that can be obtained from the ISE candidate set, one can perform global exploration using the routine described in Algorithm 13. The global exploration was stopped when 99.999% of the maximum obtainable speedup is achieved at the end of a local exploration. The result of performing a global exploration for each of the benchmarks is shown in Figures 5.10 to 5.12. These results show that local explorations provide a part, small in some cases, of the design space. Figure 5.9 shows the maximum speedups found in global exploration in comparison with the speedups found in local exploration.

An advantageous trade-off was unveiled in the global exploration of ADPCM in Figure 5.10. Solutions obtained with a local exploration that constrained the solution to use less than 25 Kgates yielded a maximum speedup of 1.03×. However, it is discovered that a speedup of 1.07× can be obtained with 26 Kgates. As the local exploration was constrained to 25 Kgates, ISEs that required more than 25 Kgates were never included in the selection. As seen in Figure 5.4, a large number of ISE candidates in ADPCM require from 25 to 30 Kgates. Thus, when the area constraints get slightly looser, the number of selection alternatives increase, enabling the combination of larger ISEs that can potentially absorb each other, thus increasing the speedup with very little area increments. Similarly, the global exploration in AES, shown in Figure 5.12, can show the designer that with

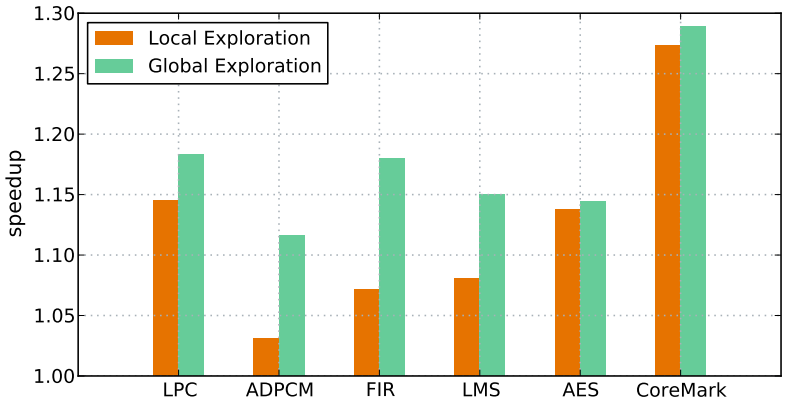

Figure 5.9: Speedups obtained with local exploration in comparison with the maximum speedup found in global exploration.

only 6 Kgates, half of the obtainable speedup can be reached.

Figures 5.10 to 5.12 also highlight the Pareto points that can be offered to the designer. They represent all of the optimal trade-offs that have been found. Thus, choosing any of the given Pareto points guarantees that no other solution is better in speedup and area. Nevertheless, the shape of the curve can be interpreted by the designer in order to choose the most suitable trade-off. Steep sections of the curve can be interpreted as costly speedup gains, since large increments in area yield small increments in speedup. On the contrary, flat sections of the curve can be interpreted as low-cost speedup gains, since small increments in area yield large increments in speedup. Therefore, points of the curve that show a visible change from flat sections to steep sections can be of special interest to the designer. In the case of the design space found in CoreMark, interesting trade-offs can be noted at an area expense of 11 Kgates where a speedup of 1.18× is obtained and at 22 Kgates where a speedup of 1.27× is obtained. These advantageous trade-offs are not displayed in local explorations. For instance, if the designer was given 80 Kgates for ISEs, the maximum speedup is achieved. However, the resource utilization of the solutions that require 80 Kgates is notably poor, since 90% of the speedup can be obtained with 25% of the area.

Thus, it is demonstrated that rather than finding a unique solution to the selection problem, exploring the space is more important in order to find the implementation solution that best suits the complete design.

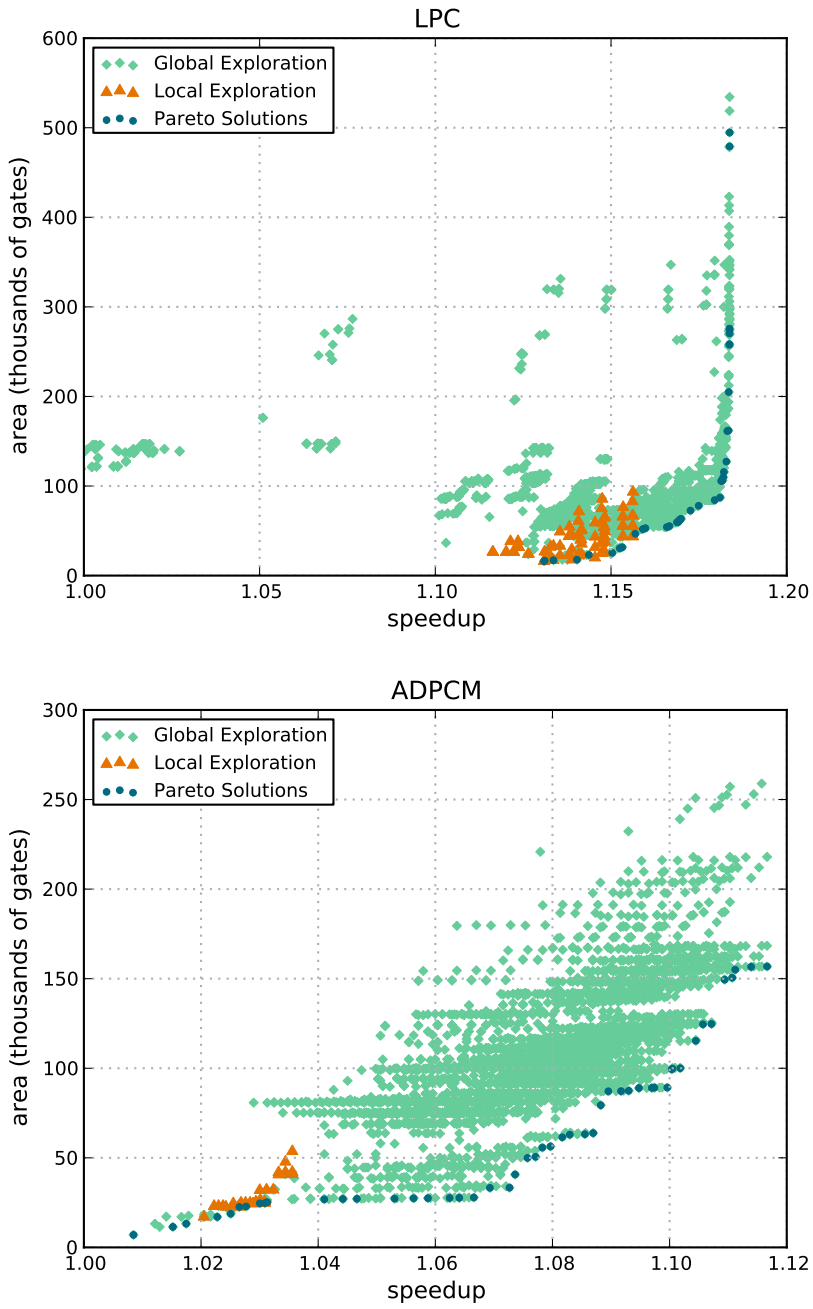

Figure 5.10: Results of globally exploring the design space of solutions derived from the ISE candidates obtained from the two benchmarks of the UTDSP benchmark suite. This is contrasted with the results of local exploration. The Pareto solutions that can be offered to the designer are also highlighted.

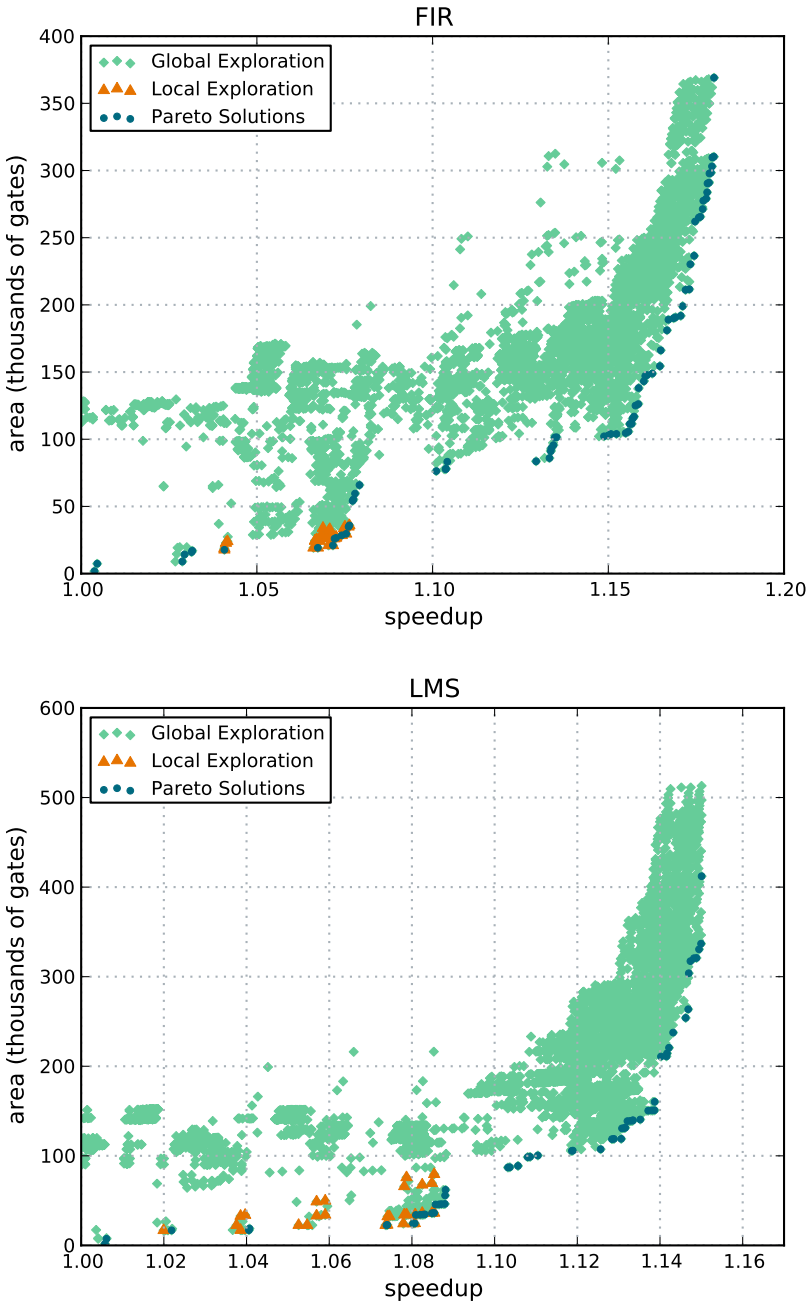

Figure 5.11: Results of globally exploring the design space of solutions derived from the ISE candidates obtained from the two benchmarks of the SNU-RT benchmark suite. This is contrasted with the results of local exploration. The Pareto solutions that can be offered to the designer are also highlighted.

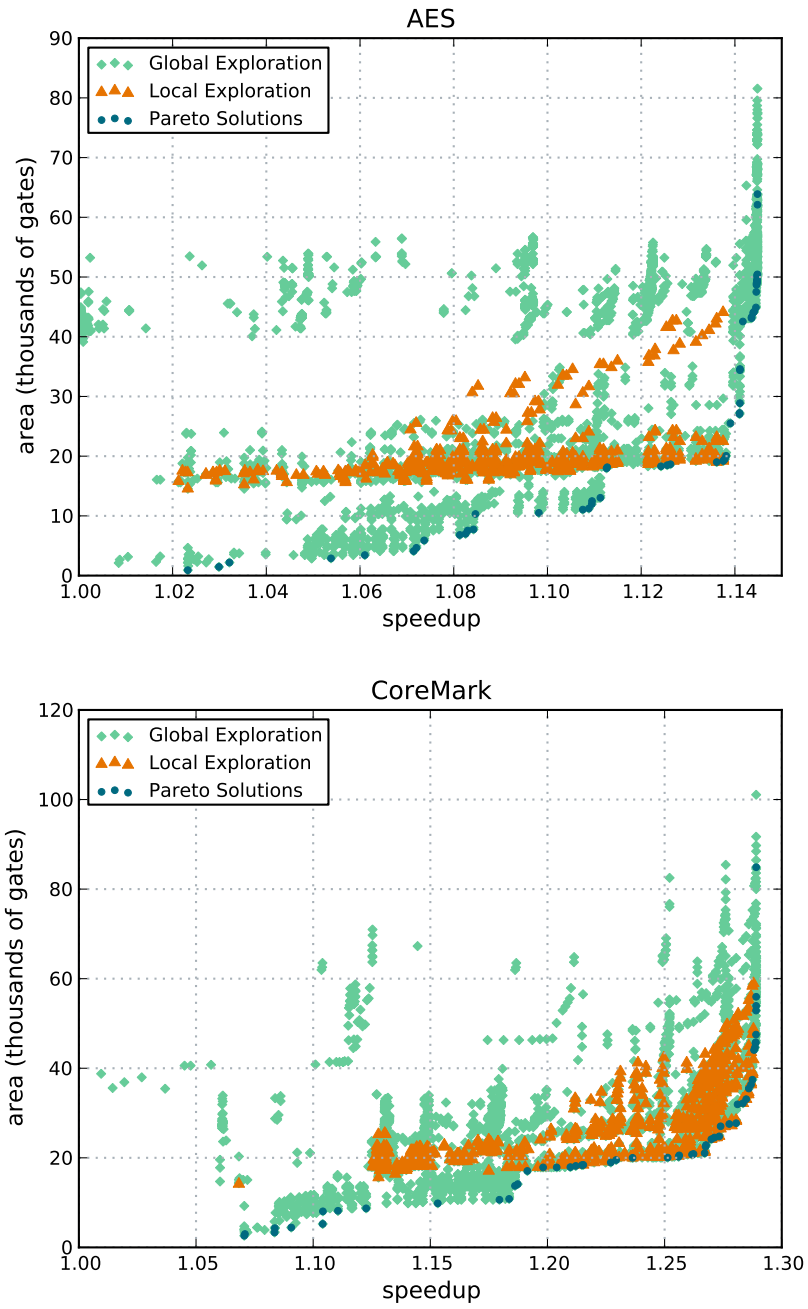

Figure 5.12: Results of globally exploring the design space of solutions derived from the ISE candidates obtained from the benchmark of the EEMBC benchmark suite and from CoreMark. This is contrasted with the results of local exploration. The Pareto solutions that can be offered to the designer are also highlighted.

5.7 Conclusions

The techniques presented in this chapter complete the hardware/software partitioning framework proposed in this thesis that, for the first time, combines the design spaces of ISE selection and resource sharing in ISE datapath synthesis. The results presented in this chapter demonstrate, that such an integration unveils new trade-offs between speedup and area that are not identified by previous selection techniques.

On the benchmarks analyzed in this chapter, the proposed heuristic finds solutions that under a fixed area constraint, achieve speedups from 8% to 238% higher than previous selection techniques.

This chapter presents an original technique to find the optimal trade-offs in the design space. This technique aims at guiding the selection process to favour ISE combinations that are likely to share resources with low speedup losses. This is achieved by using metrics that quantify the resource-sharing compatibility amongst the ISE candidates.

Thus, when there are specific area requirements for ISE implementation, a local exploration can find the solutions that achieve maximum speedup and maximum resource utilization using the available area. Additionally, a global exploration of the design space is proposed in order to generate a set of unique trade-offs between speedup and area that can be offered to the designer.

Chapter 6

Pipelining ISEs to Speedup Loops

6.1 Introduction

The ISE implementation flow proposed in Chapter 3 produces AFUs with RTL specifications that are fully compatible with hardware pipelining. This allows the processor to issue ISEs at every clock cycle regardless of their execution latency, and therefore several ISEs can be in execution at the same time. This is possible due to two properties of the generated AFUs. Firstly, ISEs that share resources form an AFU with a fixed execution latency. Therefore, within an AFU, all inputs are expected to arrive at the same time and all outputs are expected to be ready at the same time. Secondly, operations within an AFU execution cycle are independent from operations of any other execution cycle of the same AFU. This is because ISEs share resources only with other ISEs and operators that belong to the same ISE are never merged.

Under these circumstances, multi-cycle ISEs do not necessarily stop the processor from issuing another ISE at the next cycle. Instead, AFU pipeline stages can, at the same time, operate over different input data and perform the functionality of different ISEs. These pipelined AFUs can be exploited by out-of-order architectures which allow several instructions to be simultaneously in the execution stage and also allow results to be generated in a different order from the order given in the sequential instruction stream.

The scenario that motivated the creation of early out-of-order execution architectures is similar to the one represented by an in-order single-issue architecture

extended with a number of AFUs. In both scenarios, there are several functional units available in parallel. These can be pipelined, and have different instruction latencies. Thus, extended single-issue in-order architectures can be adapted to take advantage of their pipelined AFUs.

However, a desired feature in processor extensibility is to be able to leverage the design of a baseline processor core that has been optimized and verified in order to achieve customization by only plugging in additional functional units. Thus, when a typical embedded processor with single-issue in-order execution is extended with a multi-cycle AFU, the processor pipeline is stalled until the execution stage completes its execution. Therefore, only one instruction can be in the execution stage at any point in time. This constitutes the first motivation for the work presented in this chapter: most embedded processors cannot exploit the benefits of an already-present pipelined AFU.

The second motivation is the realization of the potential that pipelined AFUs have to generate significant speedups in application loops. Most programs spend the bulk of their running time in a few deeply nested loops, and this is particularly true in embedded applications. Hence, small improvements in the execution time of the instructions within a loop have a great impact in the overall execution time of the application. On the other hand, loops are able to create a stream of repeated calls to the same ISE during program execution. This represents an ideal scenario for overlapping execution of the repeated instruction calls by using hardware pipelining.

The previous chapters have approached the ISE generation problem according to the traditional methods of identifying ISEs introduced in Section 2.2 [18, 26, 29]. ISEs are identified within a basic block, thus leaving any operation that deals with the control flow of the program to the processor. Furthermore, the techniques presented so far are fully compatible with state-of-the-art ISE identification techniques.

This chapter explores a novel technique that allows a typical single-issue in-order processor to obtain the performance improvements offered by pipelined AFUs. This is achieved, with no modifications to the baseline architecture, by appropriately inserting control flow statements into ISEs. However, the proposed technique implies a major change in the way ISEs are identified in the program,

since it requires analysis of the application that goes beyond the DFGs to include control structures.

The purpose of the work presented in this chapter was to evaluate the feasibility and the potential benefits of the proposed method. Thus, the technique is independently and manually verified in a benchmark application, prior to spending effort on automation and on the integration with other tools.

This chapter is organized as follows. Section 6.2 analyzes in detail the challenges that need to be faced when a pipelined AFU is to be used to overlap the execution of consecutive loop iterations. An example extracted from a real application is used in order to show how current techniques are unable to provide this overlap. This example also shows how the introduction of a special case of ISE, referred to as *loop ISE*, removes the existing limitations by taking control of the iterations of the loop. Section 6.3 presents, through a different example application, how the proposed technique can also include loops whose bodies present control flow disruptions. In these cases, hyperblocks are formed in the loop body using compiler control flow transformations. These hyperblocks are covered by loop ISEs, thus creating ISEs with multiple exits, and therefore with multiple destination addresses. Section 6.4 shows how the functionality of loop ISEs can be implemented and generalized to support loops that contain either a single basic block or a hyperblock. Additionally, Section 6.4 provides details about how to implement AFUs that execute loop ISEs and about how these AFUs interact with the rest of the system. Section 6.5 shows, through an example, how the code of an application is transformed to include the loop ISEs. Section 6.6 explores the challenges that the construction of an automated framework for loop ISE generation presents. Additionally, an outline of the steps that can be followed are listed and analyzed. Section 6.7 introduces the experimental setup and the platform that was used in order to test the proposed techniques. Section 6.8 presents the results obtained from these experiments and compares them with state-of-the-art solutions. Finally, Section 6.9 concludes this chapter.

6.2 Pipelining ISEs

Multi-cycle ISEs can be pipelined in order to increase their throughput. Pipelining divides the circuit into several execution stages, allowing it to operate concurrently on different inputs. This permits the simultaneous exploitation of both spatial and temporal parallelism in order to increase throughput. However, to fully exploit a hardware pipeline, several ISEs must be emitted in consecutive cycles. On the other hand, application loops create repetitive streams of instructions by their nature. Furthermore, in most cases, application loops represent a large percentage of the application execution time. Nevertheless, in order to take advantage of a pipelined AFU to speedup a stream of instructions created by an application loop, an ISE must cover the entire loop body.

Prior ISE identification methods, e.g., [32], are insufficient to cover complete loop bodies as ISEs cannot include memory accesses. Loads and stores are historically only executed by the base processor. Early ISEs were designed to exchange data only with the processor's register file, hence memory operations were scheduled before and after the ISE. However, in order to create a stream of ISEs from a loop without memory operations in between, a large amount of data needs to be available. Thus, the register file is not sufficient to provide the data to a complete loop as, most commonly, entire arrays are used and transformed within loops.

[28] and [75] introduced methods that allow ISEs to exchange data with a scratchpad memory that is attached to the AFU. This local memory is called by the authors architecturally visible storage (AVS). Thus, within an ISE there might be load and store operations that access the scratchpad memory. Under this scheme, Direct Memory Access (DMA) operations, that move data between the scratchpad memory and off-chip RAM, can be scheduled before and after the execution of the loop. In [75], the authors stress the need for coherency protocols between the scratchpad memory and the cache as a requirement for correct execution. This scheme is used in the experiments presented in this chapter and is depicted in Figure 6.1(a).

Another solution that can provide a large amount of data to the AFU is presented in [76]. [76] proposes a technique, *way stealing*, that allows the AFU to exchange data directly with the data cache. Although this method requires major

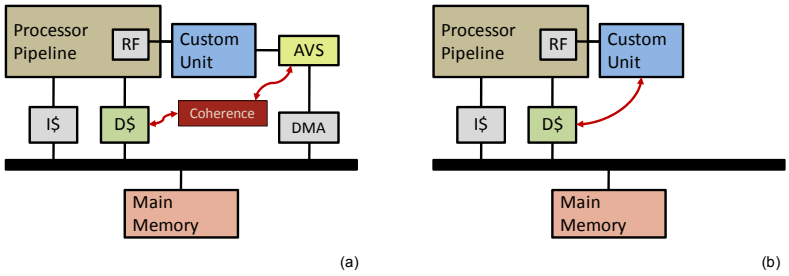

Figure 6.1: (a) Architecturally Visible Storage (AVS): local memory in the AFU [28, 75]. (b) Way stealing: the data cache can be directly accessed from the AFU [76].

architectural modifications to the processor, it does not require the implementation of coherence protocols as it does not duplicate data. Way stealing comprises techniques to preload the data that will be accessed during the loop in the data cache, before the iterations start. This scheme is depicted in Figure 6.1(b).

Motivational Example

Figure 6.2 shows an example extracted from the DCT kernel of the JPEG encoding application. The body of the main loop of the kernel is identified as an ISE. The data required for the execution of the loop can be previously loaded in a scratchpad memory attached to the AFU. Thus, when 8 inputs are available, the AFU requires 4 cycles. As there are no loop-carried dependencies, this instruction can be pipelined, yielding a speedup of 2.9× for this kernel.

Apart from the blocking condition of the multi-cycle ISE, which stalls the pipeline until the AFU completes its execution, there is another limiting factor: several operations relating to the loop itself must be done in software. In particular, the loop counter must be incremented, compared with the maximum loop count, and a conditional branch that determines whether the loop continues, must all execute in software.

This will require that the processor issues at least three instructions per iteration to facilitate the loop. Hence, instructions that control the iterations of the loop would cancel any benefits from the pipelined AFU even in the case where multi-cycle ISEs are non-blocking, i.e. the operations following them can be issued before their completion.

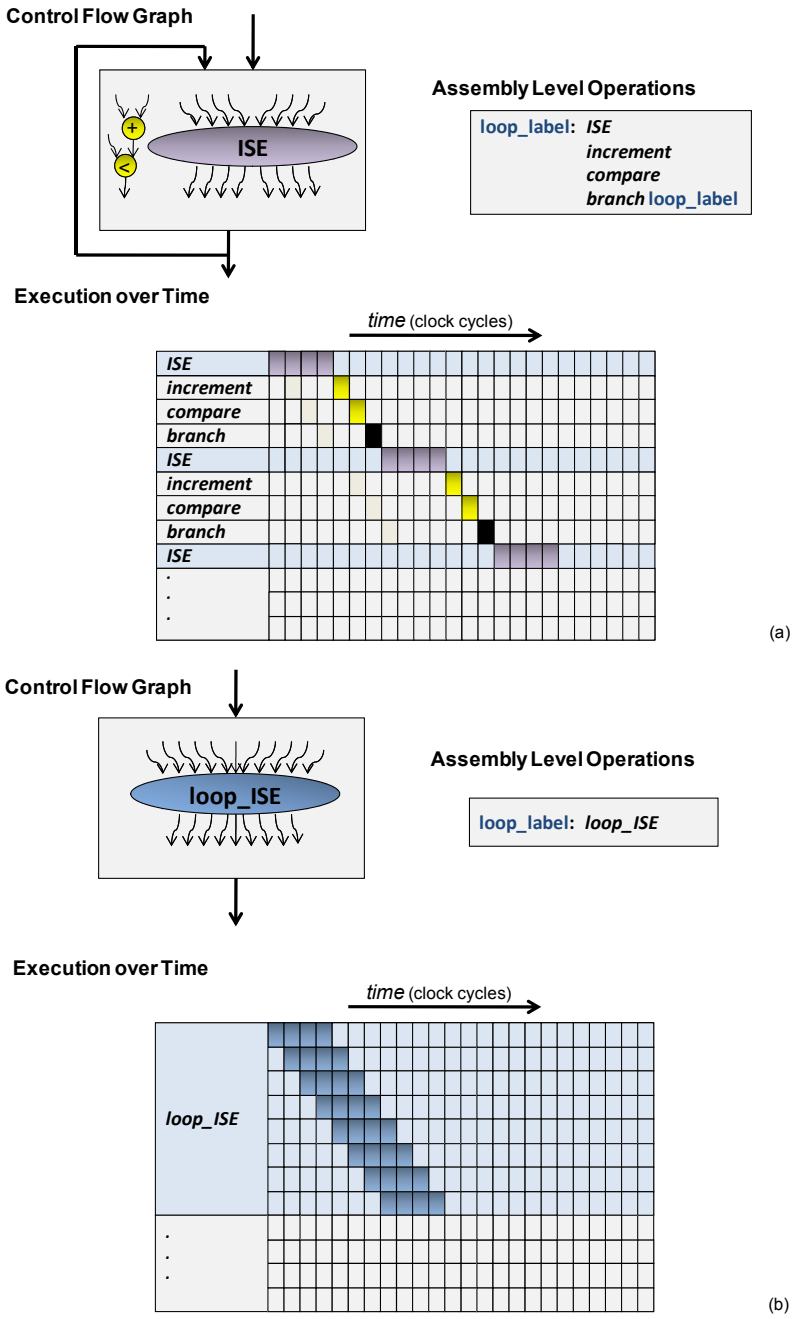

Figure 6.2: (a) A basic block extracted from the control flow graph of the DCT kernel in the JPEG application. In this basic block, a multi-cycle ISE is identified using traditional techniques. This instruction takes 4 clock-cycles to complete in the AFU. Assuming that ISEs are non-blocking instructions, or assuming that they are blocking instructions, the computations in the AFU cannot be overlapped in order to take advantage of a pipelined AFU. (b) All operations in the basic block are included in one ISE, and the AFU issues every iteration. Thus, operations in the AFU can overlap in time, yielding a speedup of at least 2.9× on the loop execution by taking advantage of a pipelined AFU.

As shown in the time table in Figure 6.2(a), the computations in the AFU cannot be overlapped in order to take advantage of a pipelined AFU. The shaded squares account for the number of cycles taken by the execution stage of each instruction. Following the ISE, the processor issues the three consecutive instructions. When the branch is taken, the processor issues another ISE. At this point, the AFU has already finished the computations of the previous instruction.

To address the mentioned limitations, the work presented in this chapter proposes to create a loop ISE that covers the loop body as well as the instructions that control the execution of a loop. A loop ISE executes the operations that correspond to the break condition of the loop in hardware, and a simple control module is added to the AFU to continuously trigger the start of a new execution of the pipelined loop body until the break condition is met. Thus, the automatic issue of ISEs every clock cycle permits the use of every stage of the pipeline during loop iterations.

This situation is shown in Figure 6.2(b), where operations in the AFU can overlap in time, considerably reducing the execution time of the loop by taking advantage of a pipelined AFU.

6.3 Allowing Loop Bodies with Multiple Exits

Significant performance improvements can be obtained by pipelining critical loops. However, loops often contain structures that cannot be included in a single ISE without introducing control dependencies. These structures include multiple control flow paths, multiple exits, inner loops and calls to functions that cannot be inlined. In these cases, unimportant paths with high resource usage can prohibit the optimization of the execution of more important paths. To mitigate this problem and further expose instruction-level parallelism, loop ISEs support loops whose bodies form hyperblocks [77].

A hyperblock is a single-entry, multiple-exit region of the control flow of a program, with no internal join points and no loops. Hyperblocks support predicated execution, which has already been considered in the field of custom instructions [78]. The use of predication increases the size of regions that correspond to a single path of control flow, which in turn increases the likelihood of finding

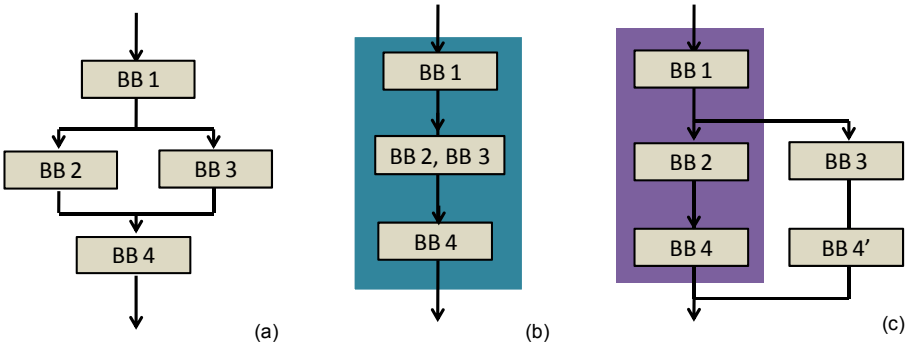

Figure 6.3: An example of control flow transformations to create single-entry regions. (a) Original control flow. (b) Predication: both branches are included in the region. (c) Tail duplication: one branch is included in the region.

instruction-level parallelism. Unfortunately, predication does not always remove all control flow disruptions, and many blocks that can be predicated are poor candidates for custom intructions. Additionally, unbalanced branches may be costly to predicate, as if-conversion always executes both sides of a branch. Moreover, one side of the branch may contain forbidden operations, such as a function call, that cannot become part of an ISE.

To construct hyperblocks, applications are profiled to identify hot loops and determine which branches are taken most frequently; heuristics are then applied to select which basic blocks are consolidated into a hyperblock. Furthermore, techniques such as loop unrolling, tail duplication, and loop peeling can assist hyperblock formation without altering the correctness of program execution.

Figure 6.3 shows an example where some of the above-mentioned compiler transformations are used in order to create a single-entry region.

Motivational Example

Figure 6.4(a) shows an example of a loop extracted from the entropy encoding kernel of the JPEG encoding application. Traditional techniques can find an ISE in the first basic block of the control flow segment. Unfortunately, there is a control flow disruption that prevents the complete loop from being converted into an ISE. Profiling indicates that the branch leading to a function call is rarely executed. Additionally, the function call cannot be executed in the AFU. Therefore, predication cannot be used to eliminate this control flow disruption.

Figure 6.4(b) shows that a compiler transformation, namely tail duplication, can be applied in order to form a multi-exit hyperblock. The hyperblock is now an independent branch of the loop, and moreover, it is the most executed path during the iterations of the loop.

Figure 6.4(c) shows how the side of the loop that forms a hyperblock can be covered by a loop ISE to be executed in custom hardware. For the common case, where the disruption does not occur, the entire loop body will benefit from the speedup achieved through the loop ISE. The loop ISE contains the operations corresponding to two conditions, one to exiting the loop, and another one to executing the branch that has been excluded.

6.4 AFU Implementation

Figure 6.5 shows the components that the AFU requires in order to execute loop ISEs. Additionally, the figure shows the interface between the AFU and the base processor.

The processor provides control signals to initiate the ISE, along with read-/write interfaces to its register file. In typical RISC processors, the register file can provide two inputs and read one output from the AFU at every cycle. An AFU controller, which receives the initialization signals, enables the pipelined datapath execution and activates the AFU's local storage units.

The AFU has its own architecturally visible local memory. This memory can provide inputs to the ISE and store its results after the loop executes. DMA is used to transfer data from the main memory to the local memory for ISE execution, and to copy the data back after the loop terminates, without stalling the processor. As noted in [75], this type of memory access creates coherence problems, as data that is modified in the AFU local memory and written back to the memory may be different from the values of the same data that reside in the data cache. A hardware or software coherence mechanism needs to be used to prevent the cache to AFU local memory coherence problem.

Additionally, a local register file is used to store loop-carried variables. Specialized load and store instructions are included in the set of ISEs to permit the reading and writing to this local register file.

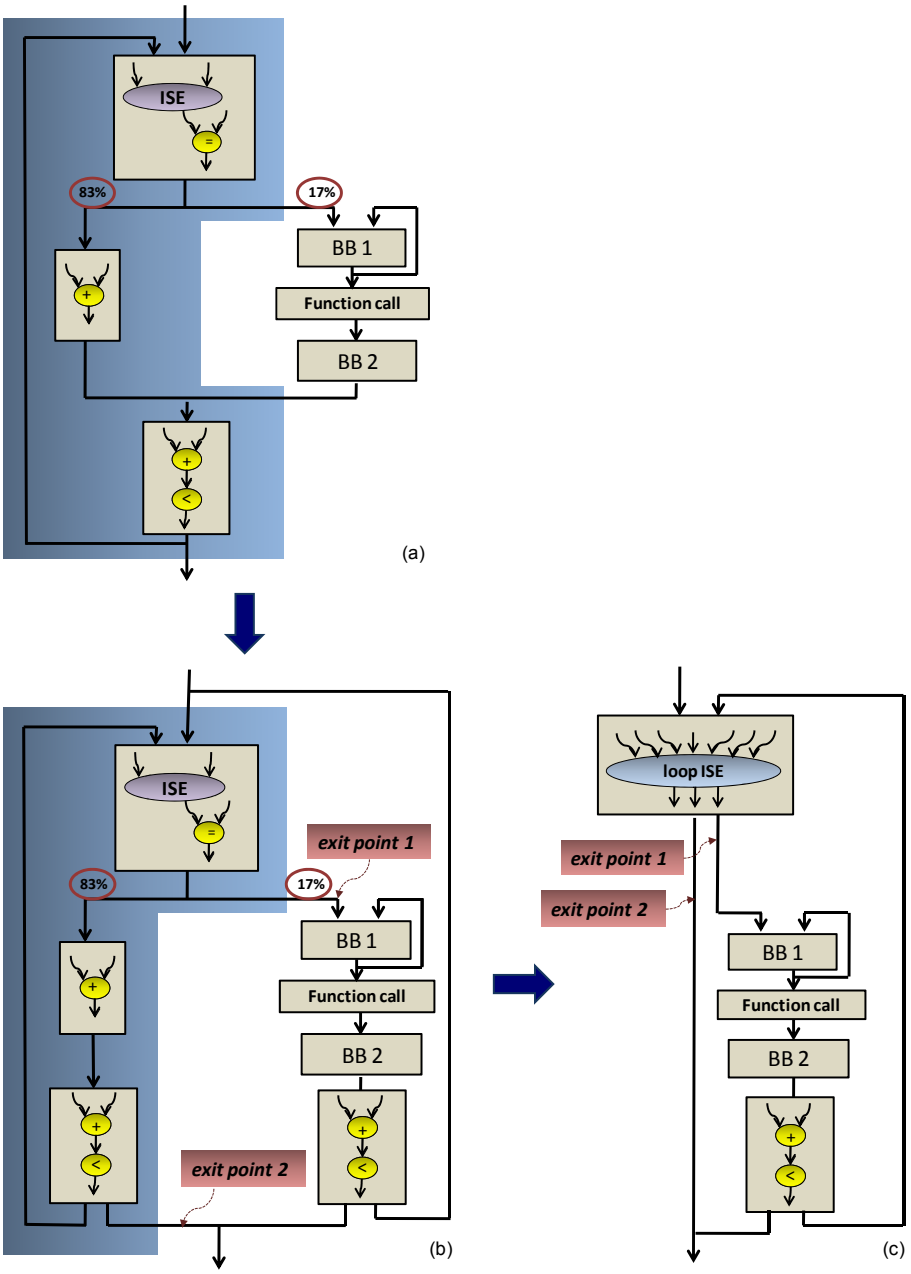

Figure 6.4: (a) Control flow segment from the entropy encoding kernel of the JPEG encoding application. (b) A hyperblock is formed by using a compiler transformation, namely tail duplication. As one of the branches is only executed 17% of the time during profiling, it is excluded. (c) The hyperblock allows the formation of a loop ISE with multiple exit points in order to optimize the execution of the most taken trace in the loop.

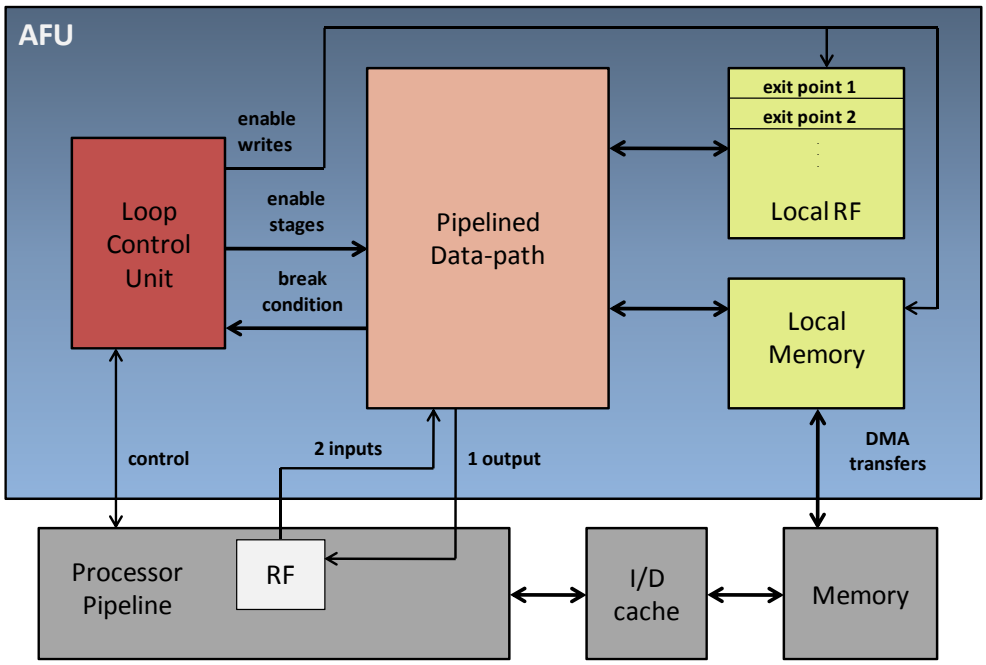

Figure 6.5: System overview of an accelerated processor. The Loop Control Unit enables the pipelined execution of the AFU. Furthermore, it detects the break condition and sends back the exit addresses (see Figure 6.4) stored in the local register file.

The AFU outputs that correspond to loop break conditions are passed to its control unit, which transfers control back to the processor.

To facilitate correct execution in the presence of multiple exit points, the AFU also returns the next instruction address associated with the met break condition to the processor. A loop ISE has itself as the default destination, which is achieved by implicitly issuing the same instruction within the AFU. The remaining destinations are the other hyperblock exits, e.g., break instructions in the original program's control flow, including the eventual completion of the loop.

The local register file also stores the next instruction addresses associated with each of the break conditions of the loop ISEs. The ISE store instruction is used to load these values at the beginning of the execution of each loop ISE. Although this data is all compile-time constant, the register file would be quite large if it must store every exit address for every loop ISE. In actuality, the number of local registers required by an ISE is the sum of the number of exit point plus the number of loop-carried variables, and the maximum number of local registers

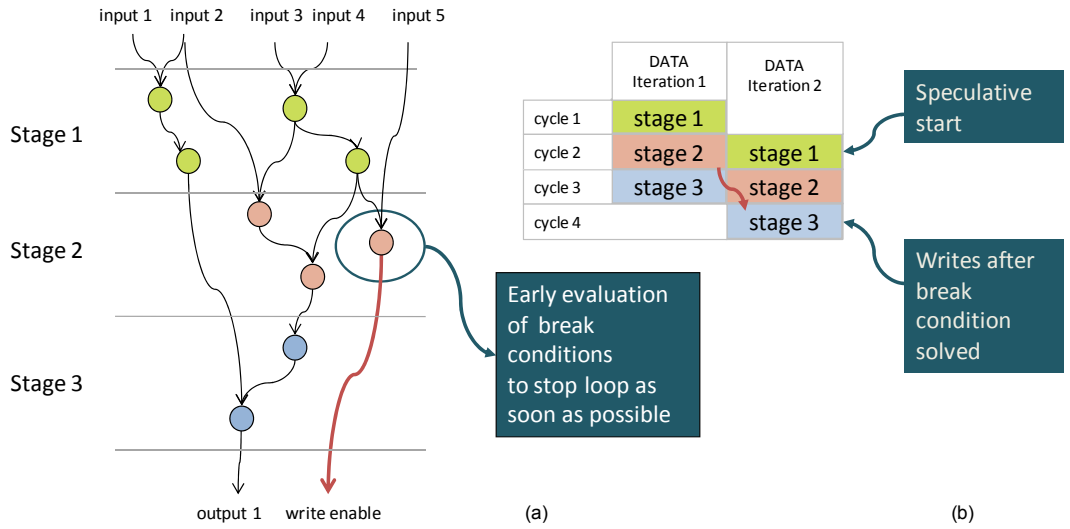

Figure 6.6: (a) ASAP scheduling is applied to the ISE so that the loop break condition is evaluated at the earliest possible point. The output of the break condition enables the data writes at the end of the iteration. (b) The second iteration of the loop and the onward iterations are started speculatively. Therefore, writes must be scheduled at least one cycle after the break condition is evaluated such when writes take place, the conditions of all of the previously issued iterations have been evaluated.

needed in the AFU corresponds to the number of local registers of the ISE that requires the most.

Due to the use of hyperblocks, loop ISEs can be viewed as a complex multi-target branch instruction. Implicitly, this suggests that the AFU would need to modify the Program Counter (PC) in the fetch stage of the base processor pipeline. Sidestepping this issue by directly providing the next instruction address to the processor is beneficial, as the architecture of the base processor can remain unchanged.

In a sense, the formation of a hyperblock can be viewed as a type of profile-guided static branch prediction. Consequently, branches that have been absorbed into the hyperblock, including hyperblock exits, are removed from the program and are no longer issued to the branch predictor; conflicts involving these branches are eliminated as a consequence.

Initiation Constraints

The *Initiation Interval (II)*, in the context of pipelining, is the number of cycles that the AFU must wait before issuing the next iteration of the loop. To initiate a new ISE at every clock cycle, the aggregate I/O bandwidth required for its computations cannot exceed the number of available memory and register file read and write ports. If these conditions are not met, then the input and output operations must be scheduled accordingly. This can be taken as the resource constraints of the initiation interval of the loop, and can be formalized as follows:

$$ResII = \max\left(\left\lceil \frac{numberOfInputs}{numberOfInputPorts} \right\rceil, \left\lceil \frac{numberOfOutputs}{numberOfOutputPorts} \right\rceil\right) \quad (6.1)$$

In other words, *ResII* is the initiation interval, as determined solely by resource constraints.

The greatest dependency distance, in clock cycles, found between iterations is another constraining factor of the initiation interval. This is known as recurrence constraint *RecII*.

The maximum instruction throughput is the inverse of stage delay or initiation interval of the AFU. It specifies the number of clock cycles between the initiations of sequential instructions into the AFU. It is calculated as follows:

$$II = \max\left(ResII, RecII\right) \quad (6.2)$$

Pipeline Scheduling

As-Soon-As-Possible (ASAP) scheduling [69] is applied to the ISE so that the loop break conditions are evaluated at the earliest possible point. However, the pipelined execution of several loop iterations at once, technically, is speculative. Therefore, write operations must not be scheduled before the break conditions of the previous iterations are resolved. An example of this is given in Figure 6.6. Inputs and outputs are then scheduled along different pipeline stages making sure that the number of read and write accesses does not exceed the number of ports at any time.

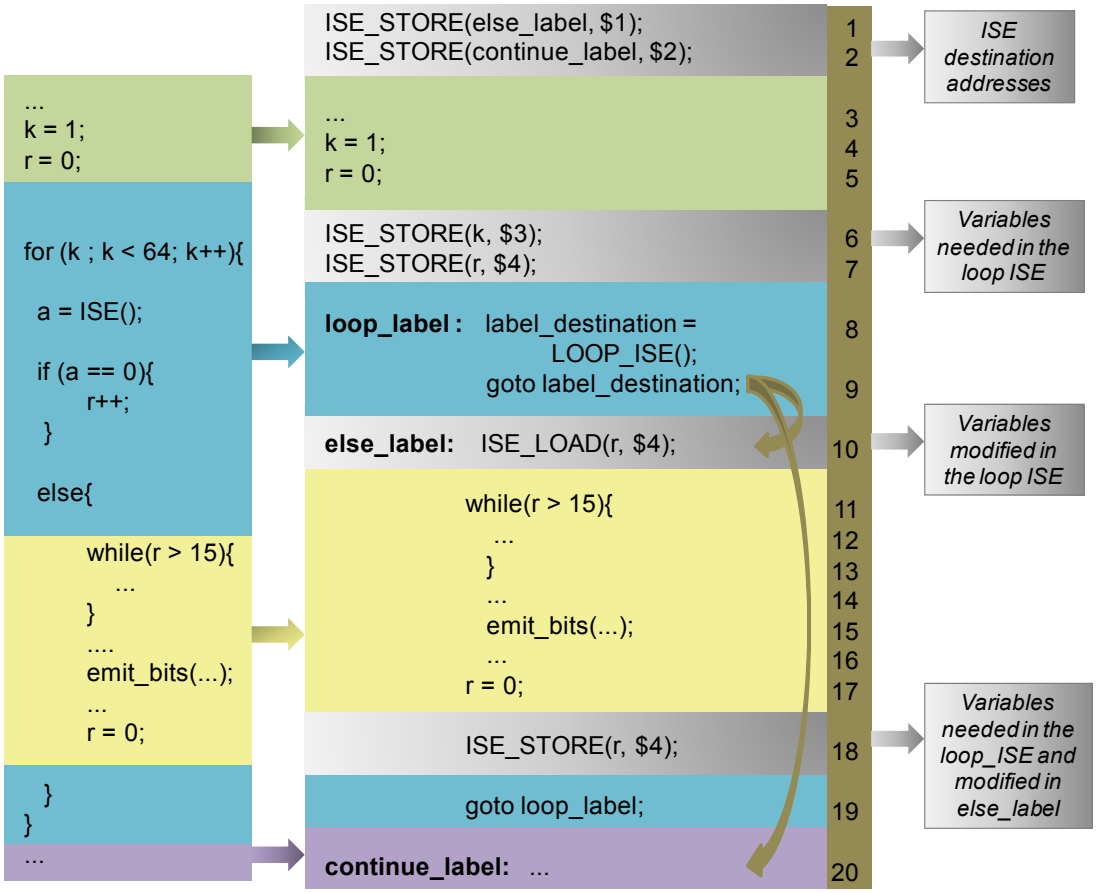

Figure 6.7: Code abstraction, taken from the JPEG entropy encoding kernel, showing the transformations needed in order to use a loop ISE.

6.5 Code Example

Figure 6.7 shows the high-level code transformations that are required to facilitate the use of a loop ISE for the JPEG entropy encoding kernel shown in Figure 6.4.

Before the loop, the addresses of the exit points are stored, as discussed in the previous section, using a dedicated instruction, `ISE_STORE`. In these addresses, the next instruction that correspond to the exit point that took place can be found. One exit point takes place when the condition to execute the excluded branch of the loop is met. Label `else_label` represents this address and its value is stored in position 1 of the local register file, as indicated in line 1 of the code in the figure. The second exit point correspond to the case where the loop finishes its execution. In this case, label `continue_label` represents the address where the following instructions are found, and its value is stored in position 2 of the

local register file, as indicated in line 2 of the code in the figure. In addition to the next instruction address corresponding to the exit points of the loop ISE, two local variables that are accessed during the execution of the loop need to be stored in the register file. This corresponds to lines 6 and 7 of the code. `k` and `r` are stored in positions 3 and 4 of the register file, respectively.

The `for` loop is replaced with a `LOOP_ISE` in line 8 of the code. This instruction returns the next instruction address corresponding to the exit point that internally took place. This is, either `else_label` or `continue_label`. A jump instruction following the loop ISE, in line 9 of the code, then transfers control to the appropriate point of continuation.

In the case where `else_label` is the following instruction address, the operations that where excluded of the loop are executed in software. This operations correspond to lines 11 to 17 of the code in the figure.

Data that was modified in the AFU and is used in the excluded branch must be moved from the AFU back into the processor in advance. This is accomplished with the `ISE_LOAD` instruction, which moves variable `r` back into the register file as indicated in line 10 of the code. If data in the AFU local memory is required, then a DMA transfer may be initiated as well.

Afterwards, any values modified by the software code that may be needed by future iterations of the loop are written back to the AFU. In this case, `r` is re-initialized to 0 and sent back to the AFU using an `ISE_STORE` instruction. This is indicated in line 18 of the code. Then, the execution re-enters the loop ISE for the next iteration as indicated in line 19 of the code.

If the loop terminates in the AFU, the next instruction address provided by the ISE corresponds to the `continue_label` in line 20. In this case, there is no return to the ISE, so normal software execution continues after the loop.

6.6 Loop ISE Generation Framework

The experiments that were performed in the work presented in this chapter manually recognized the hyperblocks to form and the loop ISEs to implement. However, as it has been proved that loops can be considerably sped up with this technique, further efforts can be made in the future in order to automate its design flow.

The identification of loop ISEs presents different challenges from those found in identification of typical ISEs. In loop ISE identification, the search is simplified as it focuses on loops. The main challenges are now centered on developing heuristics to form hyperblocks and on evaluating the feasibility and gain of their hardware implementation.

This section gives an overview of the steps that can be taken to generate loop ISEs from applications and the aspects that need to be considered in the process.

1. **Profiling:**

   The application is executed with representative input sets in order to recognize the most executed paths of the code.

2. **Loop identification and hyperblock formation:**

   As the proposed technique focuses on loops, these are first identified using compiler intermediate representations. Loops that do not contain control flow disruptions are listed as loop ISE candidates as long as all of the operations on its loop body can be implemented in the AFU. These operations are arithmetic operations and memory operations that can be transformed later into reads/writes from/to the AFU local memory.

   Loops that contain control flow disruptions are further analyzed in order to create hyperblocks that cover the most commonly taken path of loop iterations.

   Hyperblock formation is a common compiler technique that is guided by profile information. It is typically used to create a single manageable block, free of control dependencies, in which optimizations and scheduling strategies can be freely applied.

   Two steps take place during hyperblock formation:

   a) The basic blocks to be part of the hyperblock are selected.

   b) Compiler transformations are applied in order to create a region that satisfies the conditions of valid hyperblocks. These conditions are:

      - There exists an entry basic block which is the only block of the selected blocks that can have incoming control flow arcs.

- No nested inner loops exist inside the selected blocks.

Heuristic functions are used in the process of forming hyperblocks, and these are primarily guided by the execution frequency of the basic blocks found during the profiling phase.

As hyperblocks are normally used to generate code that targets multiple-issue architectures, heuristic functions also consider characteristics of the target processor such as issue rate. Thus, as the purpose of forming hyperblocks in this case is different, new heuristics are needed.

Operations in the hyperblock must be executable in the AFU, blocks that are selected must comprise only arithmetic operations and memory operations that can be transformed later into reads/writes from/to the AFU local memory.

As the blocks included in the hyperblock are to be implemented entirely in hardware, the area required in order to implement the functionality of the basic blocks must be one of the considerations.

Another interesting consideration that would strengthen the heuristic functions is the cost of the data transfers that are required in order to execute the excluded blocks in the base processor when these exchange data with the blocks that are to be executed in the AFU.

Thus, loops in which valid hyperblocks can be formed are considered as loop ISE candidates.

3. **Memory requirement check:**

   Data structures accessed in the basic block or hyperblock are identified using existing disambiguation techniques. Load/store instructions to data structures that cannot be disambiguated cannot be included.

   On the other hand, if the data structures that are used or transformed in the loop ISE do not fit in the area available for the local memories of the AFU, the loop ISE candidate cannot be implemented.

   Thus, loop ISE candidates that are formed by basic blocks or hyperblocks that fail the memory requirement check are discarded.

4. **Gain and cost estimation:**

   The loop ISE candidates are analyzed independently in order to estimate their gain and the area cost that they imply. The gain represents the speedup that the application can obtain from the implementation of a candidate. This can be derived from the execution frequencies of the basic blocks of the loops obtained from profiling. However, other factors need to be considered. Firstly, the execution overlap of loop iterations that can be obtained during uninterrupted execution of the loop ISE. This factor depends on loop carry dependencies, and on input and outputs requirements in contrast with input and output ports of the register file and of the local memory. Secondly, the gain estimation must also consider the additional instructions that need to be inserted in the code that account for data transfers of data between the AFU local storage units and the main memory or the processor register file. On the other hand, area cost can be estimated from the datapath of the loop body, from the requirements of the control unit and from the local memory space that is needed.

5. **Standard ISE identification:**

   Traditional ISE identification can be carried out in the remaining basic blocks in the application. This process lists a set of ISE candidates.

6. **Selection and AFU implementation**:

   Given a set of loop ISE and ISE candidates, this phase makes use of the estimations of gain and cost of the candidates in order to select a subset that maximizes speedups in the available area. The techniques described in Chapters 3 to 5 can be adapted to support loop ISE candidates in addition to standard ISE candidates. The datapath of the loop body of loop ISEs can share resources with the datapath of other loop ISEs or standard ISEs.

7. **Code transformation:**

   Instances of the selected ISEs and loop ISEs are inserted in the program at the appropriate locations. DMA transfers and additional load and store operations are placed in the most profitable positions.

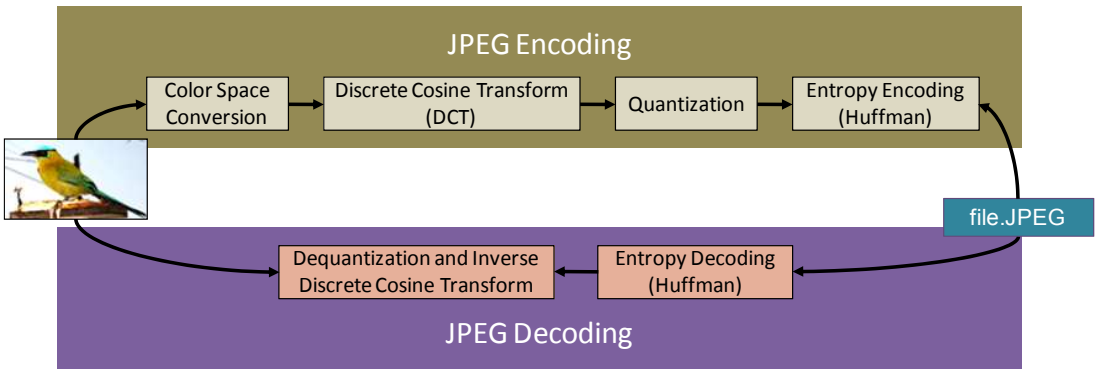

Figure 6.8: Kernels of the JPEG encoding/decoding chain.

6.7 Experimental Setup

In order to evaluate the loop ISEs, the JPEG encoding/decoding chain was taken from the EEMBC benchmark suite [72]. As shown in Figure 6.8, JPEG compression is comprised of three kernels: Discrete Cosine Transformation (DCT), quantization, and entropy encoding; likewise, JPEG decompression is composed of entropy decoding, de-quantization, and Inverse DCT (IDCT). For all of the experiments, a 24-bit RGB-encoded picture with a resolution of 1024x768 pixels was used. This resolution is comparable to the image resolution found in current web-cams and mobile phone cameras. JPEG was chosen because many of the smaller kernels in EEMBC contain straightforward loops with no internal control flow, and therefore do not require hyperblocks.

The evaluation platform is an OpenRISC processor, with an interface for custom instructions that is similar in principle to Altera's Nios II soft processor.

For the purpose of comparison, prior methods for ISE generation that do not encompass full loops were implemented.

Loop ISEs are compared with ISEs identified by the techniques presented in [32] which cannot access data in memory. These type of ISEs can only interface with the processor's register file; the register file of the target processor has two read ports and one write port, so ISE identification uses these constraints. For this approach, instruction identification, C code generation and ISE implementation in VHDL have been done by automated tools.

Loop ISEs are also compared with ISEs identified by techniques presented

in [28], which can access architecturally visible local memory. The latter was reimplementated including speculative DMA transfer techniques to ensure coherence between the processor's data cache and the AFU's local memory, as described in [75]. For this approach, instruction identification, C code generation and ISE implementation in VHDL were done by hand.

The proposed approached can also access architecturally visible local memory and was also implemented including speculative DMA transfer techniques as described in [75]. Instruction identification, C code generation and ISE implementation in VHDL was by hand. Nevertheless, in order to make a fair comparison with other methods, efforts were made to keep generality in the manipulations in order to enable future automation.

For ISEs enhanced with architecturally visible storage, I/O constraints of 8 reads and 8 writes per cycle were assumed. Each macroblock in JPEG is an 8×8 array of 16-bit integers, and can be placed into a single local memory that has 8 independent read ports and 8 independent write ports. This local memory was implemented as a 64-entry register multi-ported register file. This was an application-specific decision that was only feasible because of the small memory size; a 1 kB memory with 8 read and 8 write ports would be prohibitive in terms of both delay and area.

The modified C programs are cross-compiled using gcc 3.4.4 based on newlib for the OpenRISC. The OpenRISC, including AFUs, is synthesized on a Xilinx Virtex II FPGA with 32 MB of external SDRAM. The performance numbers reported here are taken from the system running on the FPGA. For the experiments a 8 kB 2-way set associative instruction cache and a 8 kB 4-way set associative data cache were used. Both caches use the LRU replacement policy and coherence between the AFU local memory and data cache is maintained by a MESI Level 1 protocol.

6.8 Results

The complete JPEG application was run on four configurations of the soft processor platform. The baseline uses the processor with no AFUs. Then, the three ISE-based strategies outlined in the preceding section were used for comparison.

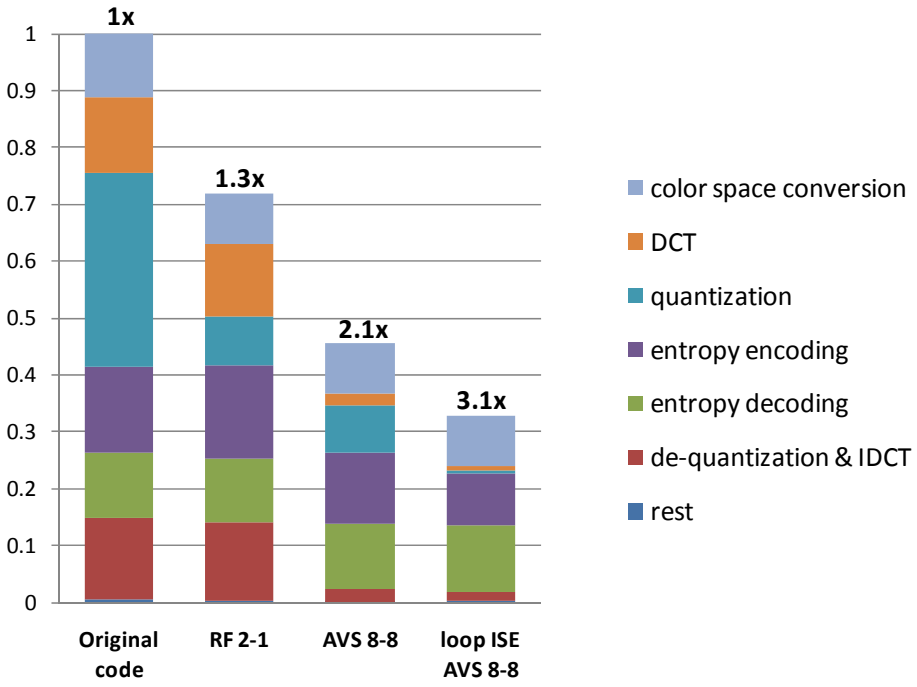

Figure 6.9: Relative execution time and speedups obtained by the different approaches in comparison with the original execution time. RF 2-1: AFU with a maximum of 2-1 inputs-output from the RF. AVS 8-8: AFU with a maximum of 8-8 inputs-outputs from local memories. loop ISE AVS 8-8: AFU with loop ISE capabilities and a maximum of 8-8 inputs-outputs from local memories. Over the complete JPEG compression and decompression algorithms a speed up of 3.1× is achieved by the implementation that includes loop ISEs, compared to a 2.1× speed up achieved by the state-of-the-art.

RF 2-1 refers to the strategy where ISEs are identified as described in [32] and read their data from the processor's register file, which has 2 read ports and 1 write port. AVS 8-8 refers to the strategy where the ISEs may use architecturally visible storage in the form of local memories, as described in [28] and a coherence protocol as described in [75]. loop ISE AVS 8-8 refers to the strategy proposed in this paper, which uses loop ISEs with the same architecturally visible storage organization as AVS 8-8.

Figure 6.9 shows the relative execution times of the three strategies listed above, normalized to software execution without ISEs. Additionally, this figure decomposes the execution time of the complete application into the execution time of each individual kernel. RF 2-1 could only speed up quantization. The OpenRISC processor does not include a hardware divider, so the baseline imple-

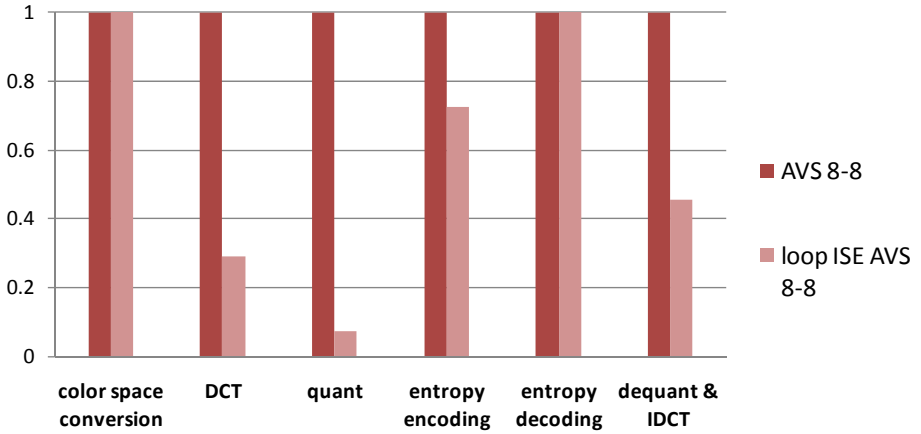

Figure 6.10: Relative execution time of each of the kernels obtained by the presented method, loop ISE AVS 8-8, in comparison with the state-of-the-art, AVS 8-8. For most of the kernels, loop ISE AVS 8-8 achieves higher speedups.

mentation performs division in software. The ISE found by RF 2-1 is a hardware divider. This yields a speedup of 1.4×. In addition to the hardware divider, AVS 8-8 finds multi-cycle ISEs that speed up the DCT and IDCT kernels, yielding an overall speedup of 2.1×. loop ISE AVS 8-8 finds speedups in DCT, IDCT, quantization, and entropy encoding. The ISEs found in each kernel are different from RF 2-1 and AVS 8-8, because they are loop bodies, which include some control flow operations. Additionally, it is important to observe that RF 2-1 and AVS 8-8 achieve the same execution time for the quantization kernel, i.e., they both find the same hardware divider. loop ISE AVS 8-8 is able to find an ISE that includes local memories in addition to the divider. Although AVS 8-8 can find these types of ISEs in general, the ISE identification method estimated that control flow in the loop would lead to excessive DMA transfers, which would eliminate much of the speedup achieved by the ISE. By converting the loop to a hyperblock, loop ISE AVS 8-8 is able to find an ISE that includes local memories. Overall, the speedup achieved by loop ISE AVS 8-8 is 3.1× compared to the baseline.

Figure 6.10 compares the execution time of each kernel using AVS 8-8 and loop ISE AVS 8-8; the results are normalized to the former. These execution times only account for the computation time of the kernels and do not account for the overhead of DMA transfers, which occur prior to the kernel invocation; moreover, some of the DMA transfer activity may overlap with software execution

Strategy	N. of flip flops	N. of LUTs
base processor	10,924	21,879
AVS 8-8	11,124	29,196
loop ISE AVS 8-8	12,006	32,988

Table 6.1: FPGA usage of the base processor and of the extended soft processor with the proposed approached and with the state-of-the-art.

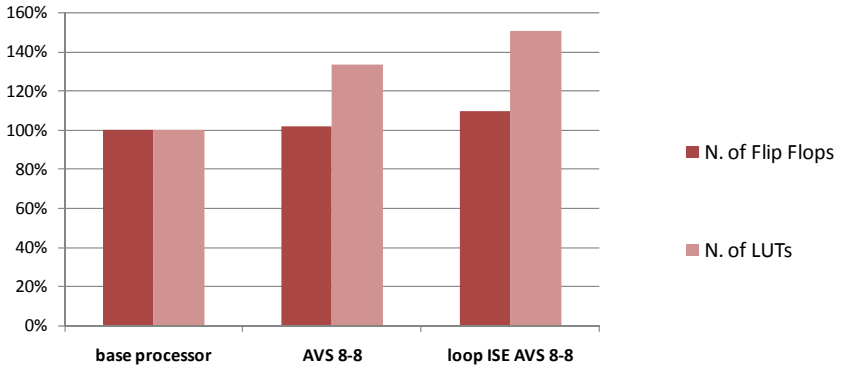

Figure 6.11: FPGA usage of the extended soft processor with the proposed approached and with the state-of-the-art in comparison with the usage of the base processor.

of earlier parts of the application.

Table 6.1 shows the number of flip flops and the number of Look-Up Tables (LUTs) used by the base processor and the strategies AVS 8-8 and loop ISE AVS 8-8. Similarly, Figure 6.11 graphically shows the percentage of the base processor that is increased with the solutions AVS 8-8 and loop ISE AVS 8-8.

The base processor used 10,924 flip flops and 21,879 LUTs, while the processor extended with strategy AVS 8-8 used 11,124 flip flops and 29,196 LUTs, and the processor extended with strategy loop ISE AVS 8-8 used 12,006 flip flops and 32,988 LUTs. This shows that the area required to facilitate the AFU's control logic does not represent an important overhead in the design.

6.9 Conclusions

This chapter has demonstrated a method to create ISEs that cover the execution of loops with exits, with the main purpose of supporting pipelined functional units. This approach broadens the scope of instruction-level parallelism for ISEs

and obtains higher speedups compared to traditional methods, primarily through pipelining, the exploitation of spatial parallelism, and reducing the overhead of control flow statements and branches.

Specific examples have been analyzed in order to show the benefits of the approach. Furthermore, performance improvements have been shown in the context of the JPEG application, which contains a wide variety of kernels where the technique has been applied and tested in an FPGA emulation platform. This method permits to measure, rather than estimate, the performance gain of the proposed approach in comparison to existing techniques that employ smaller ISEs that cannot affect control flow.

A detailed case study of the JPEG application shows that the proposed method achieves a speedup of 3.1× over pure software execution; in contrast, the most sophisticated ISEs that exist prior to loop ISEs, which can access local memories but without pipelining or support for hyperblocks, achieve a speedup of 2.1× over software.

The main goal of the work presented in this chapter was to prove that including loop ISEs in the application code has great potential to speedup loops, mainly due to the overlapping of loop iterations that can be obtained in the AFU pipeline. Although this chapter has demonstrated the effectiveness and feasibility of loop ISEs through non-automated tools. The procedures and challenges for future integration and automation of the loop ISE generation tools were explored. Moreover, manipulations in the experiments were performed making sure that they could be reproduced by automated tools.

Chapter 7

Conclusions

Although driven by the same physical laws, every application domain shapes and sizes its computing systems under different constraints, goals and demands. Moore's law is a fundamental influence on all computing systems: transistor density of semiconductor chips doubles roughly every 18 months. However, power consumption seems to follow the same trend, and as transistors reach deep-nanometer scales, manufacturing cost rises, variability increases and reliability decreases [79]. Thus, before physical limits stop the transistor density trend, other challenges have appeared. Therefore, every application domain has to adjust their designs in order to improve the computing power of every new generation of products while meeting particular constraints.

Embedded systems require small processors with low power consumption, as they are very often battery powered. Meeting performance goals under power constraints is more challenging than ever, as users demand an increasing number of features in a single device. Personal and high performance computing, although less constrained by area and power, have already reached the limits of obtaining performance gains by increasing complexity and clock rates of a single processor.

In this scenario, customization and parallelization represent the alternatives to be able to scale in performance while keeping power consumption to reasonable levels. Customization leads to more efficient designs, as resources are spent to meet the exact requirements of the application. On the other hand, parallelization distributes the computational load amongst several existing resources, thus allowing to scale performance at the cost of increasing transistor count.

ISEs represent an alternative to customize a processor by providing the additional resources that exploit the exact level of instruction-level parallelism that a particular application offers. ISEs have been used in embedded applications because they allow systems to exploit parallelization and reduce power consumption within a reasonable area overhead, as only execution units are replicated. Therefore, the rest of Central Processing Unit (CPU) resources can be shared, and there are no synchronization costs.

Software applications express several other levels of parallelism, such as thread, data and task parallelism that can be exploited by systems with large computational resources such as multi-processor systems. Although these levels of parallelism are more costly to exploit, computing systems with looser area and power consumption requirements have used it as the only alternative to scale in computing power at the expense of resources.

The potential of multi-processor systems is not yet fully exploited by all domains, as most applications are not designed to take advantage of these platforms. However, the limit is imposed by the applications and the amount of parallelism that they present. Therefore, customization and instruction-level parallelism remain important as they complement the parallelism offered by multi-processor systems. Moreover, heterogeneous multi-processor systems, with each core customized to a different application domain, have shown to be more efficient than multi-processor systems composed of general purpose cores [80].

Thus, ISEs play an important role in the design of high-performance, energy-efficient computing systems, not only for today's embedded platforms but also for tomorrow's heterogeneous multi-processor architectures. However, designers still regard ISEs as an expensive design decision, as design cycles are long, manufacturing costs are high, and the flexibility is limited. In order to reduce design time and effort, automation techniques need to advance. The trade-offs involved in ISE synthesis are complex and techniques to effectively explore the design space are required. On the other hand, flexibility is important as the possibility to reuse the design on more than one application reduces significantly the design costs. Thus, reconfigurability is a feature that must be looked at.

This thesis presented a collection of novel techniques that join different stages that take place in the process of hardware/software partitioning applications

through ISEs. These techniques advance the state-of-the-art in automation and reconfigurability while generating ISEs that maximize the perfomance gained as a function of the additional commited resources.

Most previous works on ISEs solve separate stages of the design: identification, selection, and implementation [18, 33, 8]. However, the interactions between these stages also hold important design trade-offs. In particular, this thesis has addressed the lack of interaction between the hardware implementation stage and the two previous stages. Interaction with the implementation stage has been mostly limited to accurately measuring the area and timing requirements of the implementation of each ISE candidate as a separate hardware module. However, the need to independently generate a hardware datapath for each ISE limits the flexibility of the design and the performance gains. Hence, resource sharing is essential in order to create a customized unit with multi-function capabilities.

Previously proposed resource-sharing techniques aggressively share resources amongst the ISEs, thus minimizing the area of the solution at any cost. However, it is shown that aggressively sharing resources leads to large ISE datapath latency. Thus, this thesis presented an original heuristic that can be parameterized in order to control the degree of resource sharing amongst a given set of ISEs, thereby permitting the exploration of the existing implementation trade-offs between instruction latency and area savings.

This thesis has also presented an innovative predictive model that is able to quickly expose the optimal trade-off solutions between instruction latency and area savings in the resource-sharing design-space of a given set of ISEs. Predictions are generated by capturing patterns from previously explored design spaces. Compared to an exhaustive exploration of the design space, the predictive model is shown to reduce by two orders of magnitude the number of executions of the resource-sharing algorithm that are required in order to find the optimal trade-offs.

Additionally, the scope of the proposed resource-sharing heuristic, together with the predictive model to effectively explore the created design space, goes beyond the field of ISE synthesis. The problem that has been solved can appear in other application domains in which RTL synthesis is performed. Moreover, this work presents a step towards more intelligent logic synthesis tools, that are

able to offer optimization alternatives such as resource sharing amongst hardware modules, and that in addition, are able to provide the user the optimal trade-offs of the design space.

Thus, a thorough study of resource sharing as an implementation alternative for ISEs, demonstrated that the area requirements of a set of ISEs is not just the sum of the individual area requirements, and that the instruction latency might change in hardware implementation. Moreover, it has been shown that there is an important design space that must be explored and considered at higher levels of the design. Therefore, an interaction with the ISE selection stage is required, and it presents a highly complex combinatorial problem space, in which exhaustive exploration is infeasible.

The techniques presented in this thesis are the first ones to combine the design spaces of ISE selection and resource sharing in ISE datapath synthesis, in order to offer the designer solutions that achieve maximum speedup and maximum resource utilization using the available area. The results presented demonstrate that such an integration unveils new trade-offs between speedup and area that are not identified by previous selection techniques. Optimal trade-offs in the design space are found by guiding the selection process to favour ISE combinations that are likely to share resources with low speedup losses. This is achieved by using metrics that quantify the resource-sharing compatibility amongst the ISE candidates. On the benchmarks analyzed, the proposed heuristic finds solutions that under a fixed area constraint, achieve speedups from 8% to 238% higher than previous selection techniques.

Another observation made at the implementation level is that the datapath of multi-cycle ISEs can be pipelined in order to increase their throughput. An interesting case of this potential is that of application loops, where the loop body is identified as an ISE. In this case, the loop would create a stream of instructions that could feed the pipelined datapath, thus overlapping the execution of consecutive loop iterations. However, it has been shown that traditional ISE identification techniques do not allow this optimization due to control flow overhead. In order to obtain the benefits of overlapping loop executions, this thesis has proposed to carefully insert loop control flow statements into the ISEs, thus allowing the ISE to control the iterations of the loop. The proposed technique

broadens the scope of ISE identification by providing the methods to go beyond a basic block in order to obtain greater impact in performance. Thus, performance gains exceed those of traditional ISEs by effectively exploiting hardware pipelining of ISE computations and by reducing the overhead of control flow statements and branches. A detailed case study of a real application shows that the proposed method achieves 91% higher speedups than the state-of-the-art, with an area overhead of less than 8% in hardware implementation. These results demonstrated that the proposed technique is able to exploit an already present hardware pipeline, thus permitting further speedups at the cost of a small area overhead.

In summary, this thesis has presented a deep analysis of the ISE implementation stage, and propagated its findings back through the selection and identification stages, in order to enable the design of more efficient processors through an intelligent exploration of the available design space. This approach has exposed new trade-offs that were not previously understood or exploited, and has revitalized the area of processor customization with new synthesis techniques and analytical tools.

7.1 Critical Analysis

This section provides a critical review of the methodology and the results that are presented in this thesis.

Chapter 3 has presented a technique to explore the design space of implementation solutions using resource sharing. During the execution of the resource sharing process, datapath area and delay estimations are required. These estimations do not consider that the circuit might be re-timed in order to match the clock frequency of the processor. This decision was made for two reasons. Firstly, as these metrics are calculated several times during the algorithm, an accurate estimation would greatly increase the running time of the algorithm. Secondly, the simplified estimation that was adopted allows for an architecture-independent comparison between the solutions, as the clock cycle of the processor can be varied in other stages of the design. Moreover, it has been shown that relative positions in the design space are preserved when re-timing is performed during datapath synthesis.

Chapter 5 has used application speedup as a metric in order to compare different selection and implementation solutions. Although this metric considers the effects on area and delay of the instructions due to re-timing, the estimation of the total running time of the benchmarks is calculated based only on the execution time of arithmetic, logic and memory operations. Thus, overhead due to control flow statements and cache misses are dismissed. This can be alleviated by performing simulation of the application with every solution that is considered in the design space. However, the running time of the heuristic would be prohibitive if simulation was performed for every design point. Nevertheless, it would be interesting to see the results on simulation of the Pareto optimal solutions, in order to demonstrate that the same proportions are maintained when the precise execution time of the application is considered.

Chapter 6 uses only two benchmarks, namely JPEG encoding and JPEG decoding, in order to demonstrate the benefits of the technique. However, as the technique was applied manually, and these two chained applications comprise a wide variety of kernels, the experiments performed were considered sufficient. Moreover, the kernels present in the JPEG encoding-decoding chain are similar to other kernels present in other streaming applications such as MPEG.

7.2 Future Work

There are many interesting extensions that can be made to the resource-sharing heuristic proposed in this thesis. Further optimizations can be performed in hardware synthesis if the bit-width requirements of each operation are taken into account. This would require a more detailed representation of the graphs and more information to be extracted from the compiler intermediate representation of the application. Also, when some of the inputs of the ISEs are constant, other hardware optimizations can be found. This information can also be extracted from the compiler intermediate representation of the application. For example, shift operations where the number of shifts is constant can be replaced by simple wires.

In order to improve flexibility, the reconfigurability of the generated AFUs could be further exploited by reusing the existing multi-functional datapaths with

different multiplexer configurations. Thus, ISEs that were not considered in the design might be mapped in the existing datapath, by creating new microcodes to drive the execution. Transformation techniques could be used to break some of the limitations of the existing AFUs. The identity of the operators can be used in order to bypass an operation that is not required in the function to map in the existing datapath. For example, an adder can be bypassed by setting one of its inputs to zero.

On the other hand, a thorough study of the impact of the different resource sharing trade-offs in power consumption could open up new research directions. Although there is not a direct relationship between the level of merging and the power consumption of the resulting datapath, observations made upon experimentation could be used to derive conclusions and to extract patterns. From this study, energy models could be derived, and potentially, estimations can be used in the execution of the resource-sharing heuristics in order to guide the merging process towards more energy-efficient solutions.

This work has demonstrated, with a case study, that the proposed loop ISEs have great potential to leverage existing pipelined ISE datapaths, in order to increase performance gains. Thus, the identification of this special type of ISE as an automated process has been left for future work. However, this thesis has outlined the challenges that the construction of an automated framework for loop ISE generation presents, as a starting point for this future work. Additionally, the resource sharing and selection heuristics developed in this thesis can be augmented to process loop ISEs in addition to traditional ISEs.

Finally, another possible research direction is the customization of multiprocessor systems through ISEs. Given a collection of applications, AFUs can be generated to extend each of the cores in the system. For example, two ISEs that are likely to be in the same program thread should be in the same core. On the other hand, two ISEs that are likely to be in parallel threads should be mapped to different cores. This scenario creates a design space of different trade-offs and imposes new challenges for its exploration. ISEs should be mapped and merged while taking into account resource-sharing compatibility and contention in order to maximize the utilization of the ISEs. Thus, the heuristics presented in this thesis can be extended to address this new design space.

Bibliography

[1] ARC Tools and Processor Cores. `http://www.arc.com/documentation/productbriefs.html`, 2010.

[2] R. E. Gonzalez. Xtensa: A Configurable and Extensible Processor. *IEEE Micro*, 20(2):60–70, 2000.

[3] Tensilica XPRES Compiler. `http://www.tensilica.com/uploads/pdf/XPRES%201205.pdf`, 2010.

[4] Altera Nios II Processor. `http://www.altera.com/products/ip/processors/nios2/ni2-index.html`, 2010.

[5] Xilinx MicroBlaze Processor. `http://www.xilinx.com/tools/microblaze.htm`, 2010.

[6] O. Almer, R. Bennett, I. Bohm, A. Murray, X. Qu, M. Zuluaga, B Franke, and N. Topham. An End-to-End Design Flow for Automated Instruction Set Extension and Complex Instruction Selection Based on GCC. In *GROW'09: Proceedings of the 1st International Workshop on GCC Research Opportunities*, 2009.

[7] H. Bunke, G. Guidobaldi, and M. Vento. Weighted Minimum Common Supergraph for Cluster Representation. In *ICIP'03: Proceedings of the International Conference on Image Processing*, volume 2, pages II–25–8 vol.3, 14-17 Sept. 2003.

[8] P. Brisk, A. Kaplan, and M. Sarrafzadeh. Area-Efficient Instruction Set Synthesis for Reconfigurable System-on-Chip Designs. In *DAC'04: Proceedings*

of the 41st Annual Conference on Design Automation, pages 395–400, New York, NY, USA, 2004. ACM Press.

[9] N. Moreano, E.and Cid de Souza Borin, and G. Araujo. Efficient Datapath Merging for Partially Reconfigurable Architectures. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 24:969 – 980, Jul. 2005.

[10] N. Pothineni, P. Brisk, P. Ienne, A. Kumar, and K. Paul. A High-Level Synthesis Flow for Custom Instruction-Set Extensions for Application-Specific Processors. In *ASP-DAC'10: Proceedings of the 15th Conference on Asia South Pacific Design Automation*, 2010.

[11] Q. Dinh, D. Chen, and M.D.F Wong. Efficient ASIP Design for Configurable Processors with Fine-Grained Resource Sharing. In *FPGA'08: Proceedings of the ACM/SIGDA 16th International Symposium on Field Programmable Gate Arrays*, pages 99–106, Monterey, CA, USA, Feb. 2008.

[12] D.C. Zaretsky, G. Mittal, R.P. Dick, and P. Banerjee. Dynamic Template Generation for Resource Sharing in Control and Data Flow Graphs. In *Proceedings of the 19th International Conference on VLSI Design, 2006*, 3-7 Jan. 2006.

[13] T. Akutsu and M. M. Halldórsson. On the Approximation of Largest Common Subtrees and Largest Common Point Sets. In *Theoretical Comp. Science*, pages 405–413. Springer-Verlag, 1994.

[14] S. Khanna, R. Motwani, and F. F. Yao. Approximation Algorithms for the Largest Common Subtree Problem. Technical report, 1995.

[15] C. Alippi, W. Fornaciari, L. Pozzi, and M. Sami. A DAG-Based Design Approach for Reconfigurable VLIW Processors. In *DATE'99: Proceedings of the Conference on Design, Automation and Test in Europe*, page 57, New York, NY, USA, 1999. ACM.

[16] J.A Fisher, F. Homewood, G. Brown, G. Desoli, and P. Faraboschi. Lx: A Technology Platform for Customizable VLIW Embedded Processing. In

ISCA'00: Proceedings of the 34th Annual International Symposium on Computer Architecture, volume 0, page 203, Los Alamitos, CA, USA, 2000. IEEE Computer Society.

[17] P. Yu and T. Mitra. Scalable Custom Instructions Identification for Instruction-Set Extensible Processors. In *CASES'04: Proceedings of the 2004 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 69–78, New York, NY, USA, 2004. ACM.

[18] K. Atasu, L. Pozzi, and P. Ienne. Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints. In *DAC'03: Proceedings of the 40th Conference on Design Automation*, pages 256–261, New York, NY, USA, 2003. ACM Press.

[19] H. Choi, I.C. Park, S.H. Hwang, and C.M. Kyung. Synthesis of Application Specific Instructions for Embedded DSP Software. *IEEE Transactions on Computers*, 48:603–614, 1999.

[20] M. Arnold and H. Corporaal. Designing Domain-Specific Processors. In *CODES'01: Proceedings of the 9th International Symposium on Hardware/-Software Codesign*, pages 61–66, New York, NY, USA, 2001. ACM.

[21] R. Kastner, A. Kaplan, S. Ogrenci, and E. Bozorgzadeh. Instruction Generation for Hybrid Reconfigurable Systems. *ACM Transactions on Design Automation of Electronic Systems*, 7:605–627, 2001.

[22] D. Goodwin and D. Petkov. Automatic Generation of Application-Specific Processors. In *CASES'03: Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 137–147, New York, NY, USA, 2003. ACM.

[23] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction Generation and Regularity Extraction for Reconfigurable Processors. In *CASES'02: Proceedings of the 2002 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 262–269, 2002.

[24] K. Atasu, G. Dündar, and C. Özturan. An Integer Linear Programming Approach for Identifying Instruction-Set Extensions. In *CODES+ISSS'05: Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 172–177, New York, NY, USA, 2005. ACM.

[25] P. Bonzini and L. Pozzi. Polynomial-Time Subgraph Enumeration for Automated Instruction Set Extension. In *DATE'07: In Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 1331–1336. ACM Press, 2007.

[26] P. Biswas, S. Banerjee, N.D. Dutt, L. Pozzi, and P. Ienne. ISEGEN: an Iterative Improvement-Based ISE Generation Technique for Fast Customization of Processors. *IEEE Trans. VLSI Syst.*, 14(7), 2006.

[27] L. Pozzi and P. Ienne. Exploiting Pipelining to Relax Register-File Port Constraints of Instruction-Set Extensions. In *CASES'05: Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 2–10, New York, NY, USA, 2005. ACM.

[28] P. Biswas, S. Dutt, L. Pozzi, and P. Ienne. Introduction of Architecturally Visible Storage in Instruction Set Extensions. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 26(3):435–446, 2007.

[29] A.K. Verma, P. Brisk, and P. Ienne. Rethinking Custom ISE Identification: a New Processor-Agnostic Method. In *CASES'07: Proceedings of the 2007 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 125–134, New York, NY, USA, 2007. ACM.

[30] K. Atasu, O. Mencer, W. Luk, C. Ozturan, and G. Dundar. Fast Custom Instruction Identification by Convex Subgraph Enumeration. In *ASAP'08: Proceedings of the 2008 International Conference on Application-Specific Systems, Architectures and Processors*, pages 1–6, Washington, DC, USA, 2008. IEEE Computer Society.

[31] T. Li, Z. Sun, W. Jigang, and X. Lu. Fast Enumeration of Maximal Valid Subgraphs for Custom-Instruction Identification. In *CASES'09: Proceed-*

ings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, pages 29–36, New York, NY, USA, 2009. ACM.

[32] L. Pozzi, K. Atasu, and P. Ienne. Exact and Approximate Algorithms for the Extension of Embedded Processor Instruction Sets. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 25(7):1209–1229, July 2006.

[33] P. Yu and T. Mitra. Characterizing Embedded Applications for Instruction-Set Extensible Processors. In *DAC'04: Proceedings of the 41st annual Design Automation Conference*, pages 723–728, New York, NY, USA, 2004. ACM.

[34] C. Galuzzi, E.M Panainte, Y. Yankova, K. Bertels, and S. Vassiliadis. Automatic Selection of Application-Specific Instruction-Set Extensions. In *CODES+ISSS'06: Proceedings of the 4rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 160–165, 2006.

[35] K. Atasu, R. Dimond, Mencer O., W. Luk, C. Özturan, and G. Dündar. Optimizing Instruction-Set Extensible Processors under Data Bandwidth Constraints. In *DATE'07: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 588–593, 2007.

[36] T. Li, W. Jigang, S. K. Lam, T. Srikanthan, and X. Lu. Efficient Heuristic Algorithm for Rapid Custom-Instruction Selection. In *ICIS'09: Proceedings of the 8th IEEE/ACIS International Conference on Computer and Information Science*, pages 266–270, Washington, DC, USA, 2009. IEEE Computer Society.

[37] C. Galuzzi, D Theodoropoulos, R. Meeuws, and K. Bertels. Algorithms for the Automatic Extension of an Instruction-Set. In *DATE'09: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 548–553, 2009.

[38] F. Sun, S. Ravi, N. Raghunathan, and N. K. Jha. Synthesis of Custom Processors Based on Extensible Platforms. In *ICCAD'02: Proceedings of*

the 2004 IEEE/ACM International Conference on Computer-Aided Design, pages 641–648, 2002.

[39] Clark N., H. Zhong, and Mahlke S. Processor Acceleration Through Automated Instruction Set Customization. In *MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 129, Washington, DC, USA, 2003. IEEE Computer Society.

[40] P. Bonzini and L. Pozzi. A Retargetable Framework for Automated Discovery of Custom Instructions. In *ASAP'07: Proceedings of the IEEE 18th International Conference on Application-specific Systems, Architectures and Processors*, pages 334–341, 2007.

[41] T.J. Callahan and J. Wawrzynek. Instruction-Level Parallelism for Reconfigurable Computing. In *FPL'98: Proceedings of the International Workshop on Field Programmable Logic*, pages 248–257. Springer-Verlag, 1998.

[42] J. W. Sias, H. C. Hunter, and W. W. Hwu. Enhancing Loop Buffering of Media and Telecommunications Applications Using Low-Overhead Predication. In *MICRO 34: Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 262–273, Washington, DC, USA, 2001. IEEE Computer Society.

[43] R. Karri and A. Orailoglu. ALPS: An Algorithm for Pipeline Data Path Synthesis. In *MICRO 24: Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 124–132, 1991.

[44] F. Sun, S. Ravi, A. Raghunathan, and N.K. Jha. A Synthesis Methodology for Hybrid Custom Instruction and Coprocessor Generation for Extensible Processors. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 26:2035 – 2045, Nov. 2007.

[45] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist, and M. Sivaraman. PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators. *J. VLSI Signal Process. Syst.*, 31(2):127–142, 2002.

[46] K. Fan, M. Kudlur, H. Park, and S. Mahlke. Cost Sensitive Modulo Scheduling in a Loop Accelerator Synthesis System. In *MICRO 38: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 219–232, Washington, DC, USA, 2005. IEEE Computer Society.

[47] M. Kudlur, K. Fan, and S. Mahlke. Streamroller: Automatic Synthesis of Prescribed Throughput Accelerator Pipelines. In *CODES+ISSS'06: Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, pages 270–275, New York, NY, USA, 2006. ACM.

[48] K. Fan, M. Kudlur, H. Park, and S. Mahlke. Increasing Hardware Efficiency with Multifunction Loop Accelerators. In *CODES+ISSS'06: Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, pages 276–281, New York, NY, USA, 2006. ACM.

[49] S. Yehia, S. Girbal, H. Berry, and O. Temam. Reconciling Specialization and Flexibility Through Compound Circuits. In *HPCA'09: Proceedings of the 15th International Symposium on High Performance Computer Architecture*, pages 277–288. IEEE Computer Society, Feb. 2009.

[50] J. M. P. Cardoso. Dynamic Loop Pipelining in Data-Driven Architectures. In *CF'05: Proceedings of the 2nd Conference on Computing Frontiers*, pages 106–115, New York, NY, USA, 2005. ACM.

[51] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner. An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors. In *ISCA'05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 272–283, Washington, DC, USA, 2005. IEEE Computer Society.

[52] K. Fan, H. Park, M. Kudlur, and S. Mahlke. Modulo Scheduling for Highly Customized Datapaths to Increase Hardware Reusability. In *CGO'08: Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 124–133, New York, NY, USA, 2008. ACM.

[53] N. Clark, A. Hormati, and S. Mahlke. VEAL: Virtualized Execution Accelerator for Loops. In *ISCA'08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 389–400, Washington, DC, USA, 2008. IEEE Computer Society.

[54] K. Fan, M. Kudlur, G. Dasika, and S. Mahlke. Bridging the Computation Gap Between Programmable Processors and Hardwired Accelerators. In *HPCA'09: Proceedings of the 15th International Symposium on High Performance Computer Architecture*, pages 313–322, 2009.

[55] G. Ansaloni, P. Bonzini, and Laura P. Heterogeneous Coarse-Grained Processing Elements: A Template Architecture for Embedded Processing Acceleration. In *DATE'09: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 542–547, 2009.

[56] S. L. Shee, S. Parameswaran, and N. Cheung. Novel Architecture for Loop Acceleration: a Case Study. In *CODES+ISSS'05: Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 297–302, New York, NY, USA, 2005. ACM.

[57] J. Jeon and K. Choi. Loop Pipelining in Hardware-Software Partitioning. In *ASP-DAC'98: Proceedings of the 1998 Conference on Asia South Pacific Design Automation*, pages 361–366, 1998.

[58] M. Weinhardt and W. Luk. Pipeline Vectorization. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 20(2):234–248, 2001.

[59] J. Liao, W. Wong, and T. Mitra. A Model for Hardware Realization of Kernel Loops. In *FPL'03: Proceedings of the 13th International Conference on Field-Programmable Logic and Applications*, pages 334–344, 2003.

[60] P. Tirumalai, M. Lee, and M. Schlansker. Parallelization of Loops with Exits on Pipelined Architectures. In *Supercomputing'90: Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, pages 200–212, Washington, DC, USA, 1990. IEEE Computer Society.

[61] D. M. Lavery and W. W. Hwu. Modulo Scheduling of Loops in Control-Intensive Non-Numeric Programs. In *MICRO 29: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 126–137, Washington, DC, USA, 1996. IEEE Computer Society.

[62] G. McLachlan and T. Krishnan. The EM Algorithm and Extensions. *Wiley Series in Probability and Statistics*, 1997.

[63] C. M. Bishop. *Pattern Recognition and Machine Learning.* Chapter 12. Springer, 2006.

[64] R. Kohavi. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In *IJCAI'95: Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1137–1143, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[65] S. Yehia, N. Clark, S. Mahlke, and K. Flautner. Exploring the Design Space of LUT-Based Transparent Accelerators. In *CASES'05: Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 11–21, New York, NY, USA, 2005. ACM.

[66] B.M. Pangrle. On the Complexity of Connectivity Binding. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 10(11):1460–1465, Nov. 1991.

[67] Synopsys DesignWare Library. `http://www.synopsys.com/dw/buildingblock.php`, 2010.

[68] S. Ghiasi, E. Bozorgzadeh, S. Choudhuri, and M. Sarrafzadeh. A Unified Theory of Timing Budget Management. In *ICCAD'04: Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design*, pages 653–659, Washington, DC, USA, 2004. IEEE Computer Society.

[69] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill Higher Education, 1994.

[70] SNU-RT real time benchmarks. `http://archi.snu.ac.kr/realtime/benchmark`,2010.

[71] C.G. Lee and M. Stoodley. UTDSP benchmark suite, 1992. `http://www.eecg.toronto.edu/corinna/DSP/infrastructure.html`, 2010.

[72] T. R. Halfhill. EEMBC Releases First Benchmarks, 2000.

[73] CoreMark. `http://www.coremark.org/home.php`, 2010.

[74] G. Brassard and P. Bratley. *Fundamentals of Algorithmics*. Prentice-Hall Inc., 1996.

[75] K. Theo, P. Brisk, P. Ienne, and E. Charbon. Speculative DMA for Architecturally Visible Storage in Instruction Set Extensions. In *CODES+ISSS'08: Proceedings of the 6rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*.

[76] T. Kluter, P. Brisk, P. Ienne, and E. Charbon. Way Stealing: Cache-Assisted Automatic Instruction-Set Extensions. In *DAC*, pages 31–36, 2009.

[77] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. *SIGMICRO Newsl.*, 23(1-2):45–54, 1992.

[78] P. Bonzini and L. Pozzi. Code Transformation Strategies for Extensible Embedded Processors. In *CASES'06: Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 242–252, New York, NY, USA, 2006. ACM.

[79] W. Feng. Making a Case for Efficient Supercomputing. *Queue*, 1(7):54–64, 2003.

[80] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core Architecture Optimization for Heterogeneous Chip Multiprocessors. In *PACT'06: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 23–32, New York, NY, USA, 2006. ACM.