Software Analysis for Heterogeneous Computing Architectures

A research automation framework towards more efficient HW/SW co-design

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Georgios Zacharopoulos

under the supervision of
Professor Laura Pozzi

October 2019

Dissertation Committee

Alonzo Church University of California, Los Angeles, USA
Alan M. Turing Princeton University, USA

Dissertation accepted on 21 October 2019

Research Advisor
Professor Laura Pozzi

PhD Program Director
The PhD program Director *pro tempore*

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Georgios Zacharopoulos
Lugano, 21 October 2019

To my parents Areti and Dimitris.
For always being there for me.

Men give me credit for some genius. All the genius I have lies in this; when I have a subject in hand, I study it profoundly. Day and night it is before me. My mind becomes pervaded with it. Then the effort that I have made is what people are pleased to call the fruit of genius. It is the fruit of labor and thought.

Alexander Hamilton

Abstract

NOTE: The abstract will be written in the end of the thesis.
Performance increase, in terms of faster execution and higher energy efficiency, is a never-ending research endeavor and does not come for free. The breakdown of Dennard scaling, along with the seemingly inevitable end of Moore's law economic aspect, present a new challenge to computer architects striving to achieve better performance in the modern computer systems. Heterogeneous computing is emerging as one of the solutions to overcome these limitations in order to keep the performance trend rising. This is achieved by turning the focus to specialized Hardware (HW) that can accelerate the execution of a Software (SW) application or a part of that application. The goal is to design efficient HW/SW computer architectures, where a general purpose CPU is coupled with a number of specialized HW accelerators.
The choice of which parts of an application to be accelerated, though, as well as the type of accelerators to be used, while taking into account the underlying memory system, are all non-trivial research questions and depend heavily on the SW applications characteristics that are going to be accelerated. Therefore, an in-depth SW analysis can be crucial, prior to designing a heterogeneous system, as it can provide valuable information and subsequently highly benefit performance. My initial research is revolving around various ways that SW analysis, by extending the compiler frameworks and, hence, their potential, can offer this type of information and move one step closer towards optimizing and automating the design of hybrid HW/SW systems.

Acknowledgements

This is where I acknowledge people.

Contents

Introduction

Performance increase, in terms of faster execution and higher energy efficiency, is a never-ending research domain and does not come for free. Living in an era where there is an immense amount of data, the demand for performance by modern computing systems rises even more. Technological giants, such as Google and Facebook, gather and compute a lot of data, for instance during Machine Learning related applications and lengthy simulations. This large amount of data processing requires a lot of computational power and ends up in lengthier and lengthier execution latency time.

Moore's law [55], an observation made by the co-founder of Intel Gordon Moore, predicts that the number of transistors that can be used in the same area of an integrated circuit will double roughly every 18 months. Complimentary to that, Dennard scaling [19], also known as MOSFET scaling, states that voltage and current are proportional to the size of a transistor. Therefore, as long as the same chip area is retained, power stays constant and, at the same time, more transistors of smaller size can fit onto it. Unfortunately, this is no longer the case. The transistor size has decreased over the years, but the amount of power per transistor has, recently, stopped decreasing accordingly, resulting in current leakage, a phenomenon also known as the Breakdown of Dennard scaling [21].

The breakdown of Dennard scaling [21], along with the seemingly inevitable end of Moore's law economic aspect [59], present a new challenge to computer architects striving to achieve better performance in the modern computer systems. Heterogeneous computing is emerging as one of the solutions in order to keep the performance trend rising. This is achieved by turning the focus to specialized Hardware (HW) that can accelerate the execution of a Software (SW) application or a part of that application. Specialized HW accelerators are implemented in platforms where they can be either reprogrammable, thus allowing for a large degree of flexibility as various implementations may take place utilizing the HW resources of the platform (e.g. an FPGA board), or hardwired, such as an Application-Specific Integrated Circuit (ASIC). The first type of HW implementations sacrifice part of the potential performance achieved by allowing for flexible

designs, as the same HW resources can be reprogrammed. The latter offer no flexibility but can provide better performance in comparison to FPGAs. Under the scope of this research both HW implementations were considered.

Since the performance of a general purpose CPU is becoming limited, due to physical and technological constrains, alternative computer architectures are required. Homogeneous parallel CPUs are used in order to expose parallelism of computation in SW applications, but performance is still restricted by the parts of computation that cannot be parallelized, a fact known also as Amdahl's law. Instead of a general purpose CPU – or homogeneous parallel CPUs – managing the execution of SW applications, specialized pieces of HW, namely accelerators, can be used alongside with a general purpose CPU and execute the most demanding parts of an application in terms of computation. Consequently, the need for a powerful single CPU is no more that critical, as the execution can be offloaded to other parts of HW as well. As a result, we both achieve a more balanced execution with the use of different HW resources, and we offload the execution of specific, more demanding parts of the computation to specialized HW accelerators.

One example of a widely spread heterogeneous architecture is the addition of a GPU to a CPU on the same chip, in order to exploit the parallelism and computing power that a GPU has to offer, when it comes to image processing and 3D graphics rendering. Other examples are general purpose CPUs coupled with dedicated HW that execute specific kernels or even full applications. The latter architecture could come in a number of variations, with one or more HW accelerators, and different types of coupling, tightly or loosely [17]. The design of the first option, tightly or co-processor model, is done by using the accelerator as an Instruction Set Extension in the default pipeline of the CPU. The latter implements the connection between CPU and accelerator loosely, without any knowledge of the underlying CPU micro-architecture.

The goal of the HW/SW co-design research is to design efficient heterogeneous computer architectures, so that the time latency and energy requirements are ever decreasing. The heterogeneous system that I considered during my research comprises a general purpose CPU, loosely coupled with a number of specialized HW accelerators, dedicated to the acceleration of specific parts of an application.

The choice of which parts of an application to be accelerated, though, as well as the type of accelerators to be used, while taking into account the underlying memory system, are all non-trivial research questions and depend heavily on the SW applications characteristics that are going to be accelerated. In addition to the accelerator selection problem, every HW accelerator can be synthesized with a number of optimizations embedded onto it, according to the execution

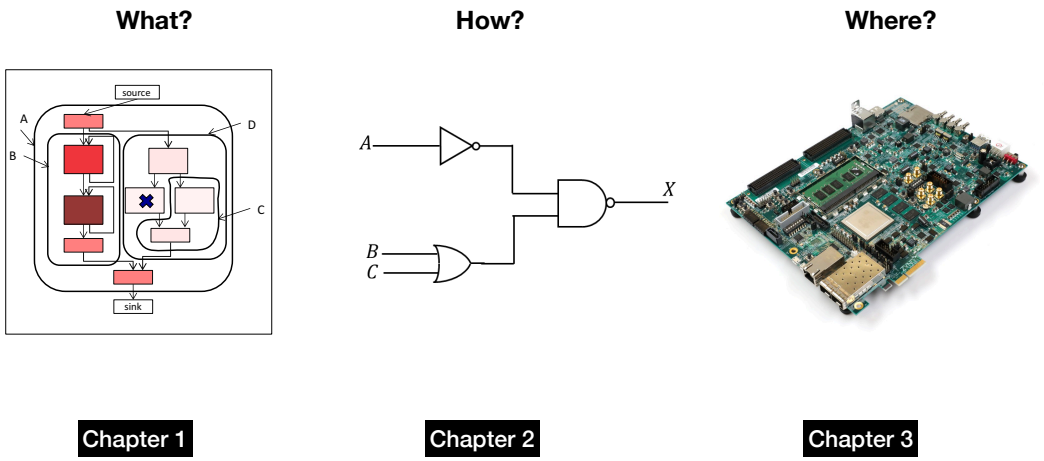

Figure 1. Overview of the research that has been conducted during my PhD and the respective chapters of the PhD thesis.

task characteristics that is targeted for acceleration. For instance, in the case that a loop is included in the execution, there could be a loop unrolling factor taken into account during the synthesis of the accelerator that may dramatically affect the execution time. Another example would be the addition of a memory buffer, e.g. a scratchpad memory, to reduce the memory latency of the execution. Furthermore, the underlying memory system, as in every computer architecture, can significantly affect the overall performance, due to communication latency, and should be taken into account during the selection of the accelerators to be implemented, along with their respective potential optimizations.

Therefore, an in-depth SW analysis can be crucial, prior to designing a heterogeneous system, as it can provide valuable information and subsequently highly benefit performance. Furthermore, such an analysis can be performed in short time (typically within a few seconds) and can be portable to other target applications or platforms. The research during my PhD has revolved around various ways that SW analysis, by extending the LLVM compiler framework [35] and, hence, its potential, can guide a HW engineer by making informed decisions early in the development cycle.

An overview of the research conducted during my PhD is depicted in Figure 1. This can be viewed as a map of this PhD thesis in order to navigate throughout my research time-line and present a high level view of how each piece is connected to each other.

Chapter 1 answers the question of *what* should be accelerated, namely which parts of computation, given a constraint on HW area resources. Under the scope

of this chapter the RegionSeeker tool-chain is presented [76]. RegionSeeker is an LLVM based framework that, given a SW application provided as input, identifies and selects, in a fully automatic fashion, HW accelerators under the constraint of an area (HW resources) budget. The granularity of the candidates for acceleration considered is that of a subgraph of the control flow graph of a function, with a single control input and a single control output. These candidates are called regions. After identification takes place, a selection algorithm solves the problem optimally of finding the subset of the initial regions list that, under a given area constraint, maximizes the collective speedup obtained. The evaluation of RegionSeeker took place by using both an industrial tool, such as Xilinx Vivado HLS [68], and a research HW accelerator simulator, such as Aladdin [57]. Experiments carried out with these tools revealed an improvement of performance compared to the state-of-the-art and a speedup gain of up to 4.6x.

In Chapter 2, the analysis that is presented attempts to answer the research question of *how* the identified and selected HW accelerators should be implemented in order to achieve improved performance. Under that scope, Data Reuse analysis, during the execution of a specific domain of applications, reveals the effectiveness of private local memory structures [74]. Furthermore, for HW accelerators that contain loops, an optimal Loop Unrolling factor can be predicted for each of the included loops [75]. The most suitable Loop Unrolling factor for each loop is defined according to the target of optimization, which can be either less use of HW resources or better speedup. With the aid of a prior LLVM based analysis of the loops and Machine Learning classification, predictions can be performed on a set of loops and the respective Loop Unrolling factors may be subsequently applied during the synthesis phase of the accelerators.

Finally, Chapter 3 tackles the research question of what should be accelerated but at the same time taking into account *where* the specialized HW is hosted. An analysis of the system at hand and its memory hierarchy can affect vastly the selection of HW accelerators and subsequently the performance achieved. Latency due to data exchange between the HW accelerators and main memory could add a significant overhead to the overall computation time. In this chapter AccelSeeker, an LLVM based tool-chain, is presented. AccelSeeker performs thorough analysis of applications and estimates memory latency along with computational latency of candidates for acceleration. The granularity of the candidates for acceleration is that of a subgraph of the entire call graph of the application. HW accelerators are selected by an algorithm so that speedup, or energy efficiency, is maximized, under a given area budget. The evaluation of AccelSeeker took place on Zynq UltraScale platform by Xilinx, considering a demanding and complex application such as H.264. With respect to methodologies based on pro-

filing information AccelSeeker attained an improved performance with an up to 2x speedup.
Automating a complex process, such as the design and implementation of heterogeneous systems, while improving performance in these systems is the broad goal of this PhD thesis. All chapters of this document attempt to provide a step closer on attaining this goal and expanding the state-of-the-art, as well as opening new paths to future work.

Chapter 1

Automatic Identification and Selection of Accelerators

Moving towards a heterogeneous era, HW accelerators, dedicated to a specific task, can improve both speedup of execution and energy efficiency in comparison to a general purpose CPU or a set of homogeneous CPUs. Nonetheless, the identification and selection of which parts of the computation are to be implemented in HW is a complex and demanding task. A thorough understanding of the application to be accelerated is necessary, the HW resources (area) budget is often tight and the granularity of the candidates for acceleration can dramatically affect the overall execution time. Furthermore, optimizations may be applied to a given, identified HW accelerator and this would produce multiple versions of equivalent computation instances that can result in various heterogeneous architectures with different characteristics and, as a result, different performance gains. In order to address all these issues I present an automated methodology that receives as input the source code of a given application and outputs a number of HW accelerators to be considered for acceleration. Among these candidates a selection takes place that maximizes collective speedup, given a HW resources (area) constraint. Finally, multiple versions of the same candidate can be considered during the selection phase as well.

1.1 Motivation

What is the rationale behind designer choices, when manually choosing application parts to be accelerated in HW, and how can those choices be replicated by an automated tool instead? Although it is possible, perhaps, that *all* of a designer's rationale cannot be replicated automatically — potentially because it requires

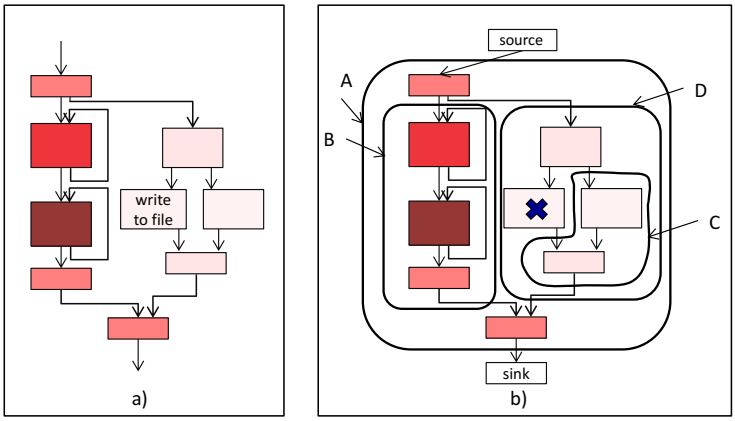

Figure 1.1. a) Example Control Flow Graph of a function, color-coded with frequency of execution (the darker the basic block, the more frequent). b) B and C are Valid Subgraphs; A and D are not Valid Subgraphs because they contain a forbidden node. B is also a CFG region, because it has a single control flow input and output.

a deep knowledge of the application at hand — it is certainly still desirable to identify at least a subset of the actions that can be automated.

Typically the designer aim will be: given an available accelerator area, extract as much as possible of the computation, under the constraint to require no more than that area, in order to maximize the resulting speedup.

Under the scope of this research I identify subgraphs of the control flow graph that have a single input control point and a single output control point, which herein will be called *regions*, as good candidates for acceleration. The rationale is that these subgraphs have a single entry point, and this corresponds to the moment of execution when the accelerator is called, and a single exit point, hence duly returning to a single location in software when the accelerator is done. Note that this type of control flow subgraph has been previously proposed and explored in compiler research — under the name of *SESE* (Single Entry Single Exit) in [2], [31], and under the name of *Simple Region* in an LLVM implementation [35] — with the aim of improving the quality of *SW code generation*, and as a scope for applying compiler optimizations and parallelization. The idea of identifying the same type of subgraph is borrowed and applied here in a novel way and to a different scenario and aim: that of automatically selecting HW accelerators.

A motivational example is provided in Figure 1.1a, which depicts the CFG of an example function, color-coded with frequency of execution (the darker the

basic block, the more frequent). A possible choice, when *manually* identifying accelerators, is to work at the granularity of functions: implement, in HW, the function most frequently executed. However, this choice might not be ideal, as the downside can be twofold: 1) a part of a function might be less frequently executed than other parts (the right side of the CFG, in the example in Figure 1.1a), therefore effectively wasting accelerator real estate. 2) a part of a function might contain non-synthesizable constructs — such as the “write to file" system call in Figure 1.1a or a function call that cannot be inlined. On the other side of the spectrum, choosing simply within the scope of single basic blocks — therefore, the body of the frequently executed loop in the picture — may not be ideal either, as the accelerator will be called once in every iteration of the loop, which may results in a large overhead. Furthermore, some speedup potential might be missed, as larger CFG regions might expose better synthesis optimizations.

CFG regions are proposed therefore as candidates for accelerators considering a granularity that can go from a single loop to an entire function, and anything in between. The main body of my research for this work is the consideration of CFG regions as candidates and a method to automatically identify and select these regions.

1.2 Related Work

Automatically identifying parts of computation to be accelerated is often called, in literature, Instruction Set Extension identification, or also HW/SW Partitioning. The distinction that is most relevant, for this research work, is the *scope* at which the suggested techniques perform identification: identifying accelerators or custom instructions at the data flow or the control flow level.

Data Flow Level. state-of-the-artmethods have been published in literature in order to automatically identify, *within a single basic block*, the subgraph of data flow according to varying architectural constraints and maximizing speedup when implemented in HW as a custom instruction. A non-extensive list include works [73], [51], [15], [53], [24] and [39], where the problem of identifying subgraphs under convexity, I/O constraint, and/or area is tackled; in [66] and [49] the I/O constraint is relaxed, to be regained via I/O serialization [52], [66], [6], [3]. In [16] the focus of the identification process is also on DFG nodes within single basic blocks, and the constraints that are taken into account are a limited number of read and write ports, and area. The methodology proposed in [23] is not limited by I/O in the selection process, but clusters MAXMISOs [4] in order to form MIMOs (Multiple Input Multiple Output instructions) that can be

executed as a single instruction.

In none of the above pieces of research, though, the inclusion of the control flow of the application is considered during the identification process. The technique proposed in Section 1.3, instead, pushes identification *beyond* the basic block level and identifies entire regions of the Control Flow Graph of the application as candidates for acceleration. Compiler transformations such as if-conversion and loop-unrolling can be, and are, used by several of the techniques mentioned above in order to enlarge the scope of within-basic-block identification, by enlarging themselves. Nevertheless, the scope remains limited to those techniques and cannot include *all* kinds of control flow.

Control Flow Level. A smaller amount of research has looked into identification within CFGs. In [77] it is proposed to implement CFG regions with multiple control exits as accelerators. However, the presence of multiple control outputs significantly complicates the processor-coprocessor interface, as opposed to a single-entry single-exit approach. Another paper proposing HW/SW partitioning [7] presents a clustering methodology that operates on a control-data network compiled from an Extended Finite State Machine (EFSM) model. While it targets control flow to a certain extent, their methodology is limited to applications that can be modeled using EFSMs, therefore considering a much more limited scope than that of generic Control Data Flow Graphs compiled from source code, as the methodology proposed in Section 1.3.

Finally, the authors of a recent work [2] consider Single Entry Single Exit regions but their target is to identify strictly parallelizable loop regions and offload them to an MPSoC target platform. This approach is limited in terms of excluding non-parallel regions from being potential candidates to be accelerated, and also in terms of not being cost-efficient, in case a designer needs to set a specific area constraint for the accelerators.

Compiler Transformations. Within compiler research, it is fairly common to identify CFG subgraphs for code optimization reasons. For example, trace scheduling, superblock and hyperblock scheduling [29], identify regions of the CFG in order to perform global code scheduling and improve code generation. *SESE* (Single Entry Single Exit) regions have been proposed in [31], and their identification was reimplemented in the LLVM framework in an analysis pass called *RegionInfo*, for the purpose of improving the performance of code generation. For my SW analysis, the idea of CFG region identification was borrowed from compiler research and was applied to automatically identify and select HW accelerators.

Application Specific Instruction set Processor (ASIP) architectures and design practices. HW Accelerators that are embedded in an Application Specific Proces-

sor can be either developed as hardwired Integrated Circuits (ICs), or mapped onto reprogrammable systems. In the first scenario, examples of Application-Specific Integrated Circuit (ASIC) platforms exist, such as the Tensilica Xtensa from Cadence [12] and the ARC processor from Synopsys [61]. These tools can be extended with accelerators and complex instructions. The CPUs can be configured during the design process to maximize performance and efficiency, without enduring the overhead of reconfiguration. An alternative, not as performing yet more flexible, is offered by FPGA-based Systems-on-Chip (SoC) examples, such as Altera (the Arria10 family [5]) and Xilinx (the Zynq SoCs [69]).

The instances mentioned above support the generation of HW circuits, but do not provide implementation paths for differentiating the execution between HW and SW. Conversely, High Level Synthesis (HLS) tools allow designers to move parts of applications, written in C or C++, between processors and accelerators. Research endeavors in this domain include LegUp [13] and ROCCC [25], while commercial applications comprise the Vivado_HLS [68] suite from Xilinx (for FPGAs) and StratusHLS [11] from Cadence (for ASIC development). However, these HLS frameworks place the responsibility of partitioning a SW application on the application developer.

1.3 The RegionSeeker Framework

The RegionSeeker framework is an automated methodology that identifies candidates for HW acceleration from application C code. An extensive SW analysis, based on the LLVM compiler infrastructure, performs, apart from the identification, an estimation of the performance gain, along with the HW resources cost, of each candidate. Subsequently given a HW resources constraint, a selection of the identified HW accelerators takes place that maximizes the cumulative performance gain. In the following subsections the methodology is presented in detail, as well as experimental results from the CHStone benchmark [30] suite.

1.3.1 Methodology

There are three parts comprising the methodology, detailed as follows. The first step is to automatically identify valid regions that are suitable candidates for HW acceleration. Secondly, an estimation of their potential merit, in terms of cycles saved (the difference between SW and HW execution cycles), is computed along with the respective cost, which is the HW resources (area) required for each region. Finally, a selection algorithm is utilized in order to optimally solve the

problem of selecting a subset of these regions that maximize the accumulated merit under a given cost, i.e., an area constraint.

Region Identification

To identify regions in both an automatic and efficient way, a *Region Identification* pass was developed under the version 3.8 of the *LLVM Compiler framework* [35]. The pass receives as input applications developed in C or C++ and performs their analysis at the Intermediate Representation (IR) level, a type of code used internally by LLVM to represent source code and allow data flow analysis and optimizations.

The pass iterates over every function of an application and, using the existing *RegionInfo* LLVM pass [64], identifies regions within every function. Subsequently, nodes that cannot be synthesized, such as system calls or calls to functions that are not inlined, are identified and labeled as forbidden. The regions containing these nodes are marked as invalid. Conversely, the valid regions are evaluated by a profiling-via-instrumentation routine. Profiling via instrumentation requires generating an instrumented version of the code, which gives more detailed results than a sampling profiler. The output of the profiling is a file that contains information regarding the execution frequency of each basic block and the total number of calls to each function, i.e., the execution frequency of each function. Using this information, the basic blocks are annotated in each function with their respective execution frequency.

Merit and Cost Estimation

The Region Identification pass, apart from the Region Identification detailed above, performs an early evaluation of the merit and cost of a region, implemented directly within the LLVM tool-chain. The evaluation relies on the LLVM intermediate representation and does not need any manual modification to perform function out-lining on the benchmark source code. The estimation of merit and the cost of a region is performed as follows.

Merit Estimation. The merit of a region is defined as the total number of cycles saved in a HW accelerator implementation compared to the respective SW implementation of the same piece of runtime of a given application. Therefore the merit of a HW accelerator is estimated as the difference between the Hardware and Software run time, across all its invocations in an application, taking into account the invocation overhead of calling a HW accelerator in a specific heterogeneous architecture. The estimation of the HW run time is computed first in the

Algorithm 1 LLVM Analysis Pass - Region Identification

Input: Application written in C/C++
Output: List of Identified and Profiled Regions

```
1: function RunOnFunction()
2:     Region_List = NULL
3:     RI = getRegionInfoAnalysis()
4:     for Region in Function do
5:         if RegionIsValid() then
6:             EvaluateRegion(Region)
7:             Region_List.Add(Region)
8:     return Region_List
9:
10: /* Estimate Merit for Region */
11: function EvaluateRegion(Region)
12:     for Basic Block in Region do
13:         getProfilingInfo(Basic Block)
```

basic block (BB) level as the the critical path of the latency (in clock cycles) of the Data Flow Graph (DFG) nodes. Runtime profiling information is used in order to determine the execution frequency of each BB. Subsequently the delay of the entire region in HW is estimated by multiplying the critical path delay of each BB with the respective execution frequency and finally summing up the products, according to the specific BBs that comprise the region. Software run-times are estimated in a similar fashion, but instead of computing critical paths at the BB level, the sum of the latency (in clock cycles) of all its constituent operations is computed, modeling that these are processed sequentially in software.

Cost Estimation. On the other side of the evaluation, the cost of a region is estimated as the area (or HW resources) required to implement its DFG nodes. The area is computed as the sum of look-up tables that is required for the DFG nodes of the respective HW accelerator. Each DFG node may take up a different amount of loop-up tables according to its complexity. The characterization for each DFG node was carried out with the aid of Vivado, a commercial High Level Synthesis tool, targeting a Virtex7 FPGA.

The final output of the analysis pass is a list of valid regions, or else accelerator candidates, each annotated with an estimated merit and cost. The region list output is in turn processed by the exact selection algorithm implemented as standalone program in C++.

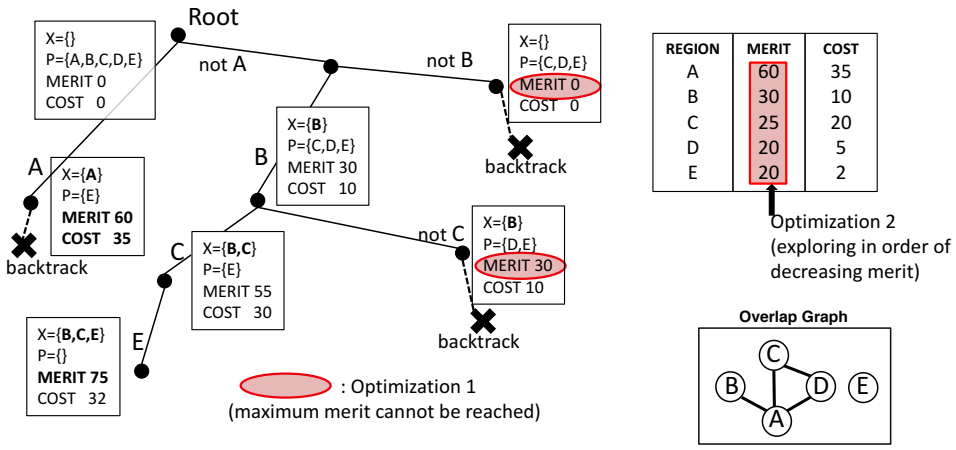

Figure 1.2. Tree exploration performed by exact, for the running example of Figure 1.1, and for a cost budget of 35.

Region Selection Algorithm

Given a merit $M()$ and cost $C()$ function for each region we can formulate the problem of selecting accelerators as follows:
Problem: Region Selection Let $\mathscr{R} = \{R_1, R_2, \ldots, R_n\}$ be a set of regions, with associated cost and merit functions C and M. For any subset $X \subseteq \{1, 2, \ldots, n\}$ of regions, we denote by $M(X) = \sum_{i \in X} M(R_i)$ the sum of the merits of its regions, and we denote by $C(X) = \sum_{i \in X} C(R_i)$ the sum of the costs of its regions.

We want to select a subset X of regions such that

1. No two regions belonging to the same CFG overlap, i.e., $V(R_i) \cap V(R_j) = \emptyset$, for all $1 \leq i, j \leq n$
2. The cost $C(X)$ is within a user-given cost budget $C_{\max}$
3. The merit $M(X)$ is maximized

This problem definition maps to what we have identified in Section 1.1 as the designer aim: given an available accelerator area, extract as much as possible of the computation, under the constraint to require no more than that area, in order to maximize the resulting speedup.

An exponential, exact branch-and-bound method based on a binary-tree search was derived in order to solve optimally the Region Selection problem. The algorithm converges to an independent set of regions that maximizes merit under a given cost. This process can be exemplified through Figure 1.2: given an initial set P, that includes all valid regions identified, and a set X, that is initially

an empty set and is going to be the subset of P that maximizes merit under a given cost. The root of the tree represents the empty set, and set P at this point contains all regions. Furthermore an overlapping graph among regions shows whether there is overlapping among valid regions contained in P, such that it poses the restriction of not allowing the selection of regions that overlap with each other, i.e., containing at least one BB that is common. The overlapping graph is seen as well in Figure 1.2. As the algorithm starts the exploration, inclusion of region A is first considered, and the set P is updated by removing all regions overlapping with A: $P = \{E\}$. According to the merit and costs of all regions in this example, shown in the table within the picture, the merit (60) and cost (35) of the solution currently explored is also updated.

At every point of the exploration, a new node u is considered for addition in the current independent set. If there is no node u satisfying condition 2 of the Region Selection Problem, the algorithm records the set X and backtracks, as X is maximal with respect to condition 2. For the running example in Figure 1.2, the cost budget $C_{\max}$ is equal to 35. Hence, exploration stops at $X = \{A\}$ because the cost budget has been reached, and backtracks. The next region chosen is B, sets X and P are again updated accordingly, to $X = \{B\}$ and $P = \{C, D, E\}$, and exploration continues until the selection algorithm converges.

Two optimizations are implemented in the exact selection algorithm in order to avoid unnecessary exploration. Optimization 1 performs a look up in order to determine whether the maximum recorded merit can be reached by the regions contained in P or not. If not the exploration stops and backtracks. Optimization 2 ranks the valid regions in terms of merit so that the first region considered for inclusion in X is the one with the maximum merit.

1.3.2 Experimental Results

The evaluation of the RegionSeeker framework took place by assuming a system constituting a single SW processor and multiple HW accelerators, exchanging shared data with private local memories. The processor activates the accelerators via a memory-mapped interface, thus requiring a transaction on the system bus. When activated, accelerators read and write data to and from the private local memories, computing their outputs, which can then be accessed by the processor. Accelerators are interfaced to private local memories with ports having a latency of one clock cycle. The control interface between the processor and the accelerators was modeled with a latency of 10 clock cycles.

The run-times of the non-accelerated SW part of the considered benchmarks were measured using the Gem5 simulator [8], modeling an ARMv8-A processor

with an issue width of 1. The processor model is atomic, with in-order execution. It is interfaced with separate instruction and data memories with an access latency of one clock cycle.

Hardware execution times were retrieved using two different HLS frameworks: the Aladdin simulator and the Xilinx Vivado_HLS commercial tool-suite. Aladdin targets ASIC implementations. It provides a fast evaluation, but does not produce a synthesizable netlist as output; nonetheless, the estimations offered by this tool are within 1% of the ones derived from an RTL implementation [57]. Hardware instances generated with Vivado_HLS are instead intended for FPGA designs. Synthesis-runs within this framework are more time-consuming, but provide exact area (HW resources in terms of Look Up Tables and Flip Flops) and latency (number of cycles) values of each accelerator, as well as a direct path to its realization. In both cases, default implementations of the accelerators were considered, i.e., no optimizations were applied to the implemented HW accelerators.

The benchmarks that were used during the experimental phase are embedded applications of varying size from the CHStone benchmark suite [30]. `adpcm` performs an encoding routine and `aes` is a symmetric-key encryption algorithm. `dfmul` and `dfsin` are small kernels that perform double-precision floating-point multiplication and sine functions employing integer arithmetics. `gsm` performs a linear predictive coding analysis, used in mobile communication. `jpeg` and `mpeg2` are larger applications, implementing JPEG and MPEG-2 compression, respectively. Finally, `sha` is a secure hash encryption algorithm, used for the generation of digital signatures and the exchange of cryptographic keys.

Figure 1.3 showcases the achieved speedup, when employing Aladdin, by the accelerators selected by RegionSeeker (labeled `regions` in the figure), with respect to the entire run-time of the applications and for different area constraints. For small-to-medium size applications such as `adpcm`, `aes`, `gsm` and `sha` speedup gains for RegionSeeker vary from 1.6x up to 3.2x. For smaller kernels, larger variations can be observed, as for `dfmul` and `dfsin` the speedup reaches 1.12x and 3.9x respectively. Finally, for larger benchmarks such as `jpeg` and `mpeg2` speedup is fairly significant: 2.5x for the former and up to 4.3x for the latter can be reached using RegionSeeker.

Similar trends are observed when Vivado_HLS is instead used for the accelerator synthesis, as reported in Figure 1.4: RegionSeeker consistently outperforms state-of-the-art approaches which target either single basic blocks or entire functions, across all benchmarks. These results highlight that the achievable speedups are highly influenced by which segments of applications are selected for accelerations, and that such choice is only marginally influenced by the adopted

merit and cost estimation tool. In fact, this was verified across the two sets of experiments, as the regions chosen were the same in 80% of the cases. As an example, out of 10 regions selected to achieve a 2.2x speedup for the `jpeg` benchmark, 8 are the same when using either Aladdin or Vivado_HLS for merit and cost estimation, and the ones that differ contribute to less than 14% of the provided gain.

Finally, in Figure 1.5 a summary of the performed experimental exploration is presented. It reports the normalized speedups obtained by RegionSeeker compared to basic block and function identification, when a fixed area budget is considered and Vivado_HLS are employed. The mean column illustrates that, on average, RegionSeeker achieves approximately 30% higher speedups with respect to the two baseline methods. Moreover, while in some cases the baselines match the performance of RegionSeeker (e.g.: `gsm` for basic blocks, `dfsin` for functions), neither of them can achieve that consistently across applications, stressing the suitability of control-flow regions as HW accelerator candidates.

The speedup that can be obtained by accelerating basic blocks is hindered by their small granularity and, consequently, the high number of HW accelerators invocations by the SW processor, i.e., the switches between software and hardware execution. Moreover, in this setting many optimization opportunities during the hardware implementation of the accelerators are missed, because they only arise when control flow is considered, as is instead the case for regions.

On the other hand, the speedup derived by selecting whole functions trails the one corresponding to regions, because of two reasons. First, function selection is limited to the ones which do not present forbidden nodes, and this may rule out promising regions within them. Second and more importantly, it is inflexible from an area viewpoint, which is especially visible when few hardware resources are available for acceleration. In those cases, the selection of functions often detects only few feasible candidates, with a small merit (e.g. in Figure 1.3: `jpeg` and `mpeg2`, for an area of less than 0.5 mm^2).

This limitation, though, is not present in regions, as simply the part referring to individual hotspots inside a function can be available for selection. Indeed, the performance of RegionSeeker stems from the high flexibility of the selection approach, as it allows the consideration of the entire spectrum of granularity ranging from whole functions to single loops, ultimately enabling a better exploitation of speedup for a given area budget.

1.4 RegionSeeker MuLTiVersioning

High Level Synthesis (HLS) tools, such as Vivado_HLS by Xilinx, may employ optimizations to HW accelerators design in order to increase performance, i.e. obtain faster execution. These HLS optimizations were not taken into account by RegionSeeker framework in the previous section. Default, non optimized versions of HW accelerators were identified and selected instead. In this section an extended RegionSeeker framework is presented, which performs the selection not only among possible CFG subgraphs, but also among different versions of each identified subgraph, namely different versions of the regions identified. This extension is referred to in the rest of the document as RegionSeeker: the MuLTiVersioning approach.

1.4.1 Methodology

The rationale, supporting the extension of RegionSeeker framework, is to achieve improved speedup that can be provided by exploiting a more varied set of HW accelerators, with different optimizations implemented onto them, to select from. This is being achieved by instantiating different versions of each HW accelerator with the same functionality, yet different speedup gains and different area (HW resources) requirements. The set of optimizations that were considered in order to design different HW implementations of the same accelerators are:

1. The Loop Unrolling (LU) factor, in accelerators that contain loops.
2. The loop pipelining option, being either on or off.
3. The array partition factor, which is the number of input and output ports of the memory buffer (scratchpad) attached to the accelerator.

Loop unrolling optimization is an HLS directive that, in the context of High Level Synthesis instantiates multiple copies of the logic implementing the functionality defined in a loop body, drastically impacting the performance of HW accelerators [34] [33]. This directive can be applied in HW accelerators containing loops whose trip count can be statically defined. It should nonetheless be applied in a careful manner, as it entails a high area cost for the duplicated logic. Furthermore, the resulting benefits can be hampered by loop-carried dependencies and frequent memory accesses.

Loop pipelining is an additional HLS directive applied in loops that allows the pipelining of the operations contained in a single body of a loop and across consecutive iterations. Restrictions regarding loop-carried dependencies across

consecutive iterations can limit the application of the loop pipelining optimization, as the result of the output of a loop iteration would be required in the following one, thus not allowing the pipelining of the loop body operations.

Given an initial set of HW accelerators, i.e., a set of regions that is derived by the RegionSeeker framework, multiple versions for each region can be generated that maintain the same functionality. Each version may employ one of the optimizations listed above, or a combination of them.

All versions of the HW accelerators were evaluated by exploiting the Aladdin HW accelerator simulator. Aladdin targets ASIC implementations. It provides a fast evaluation, but does not generate a synthesizable netlist, as opposed to Vivado_HLS. Nonetheless, the estimations provided are within 1% of the ones derived from a Register-transfer level (RTL) implementation, according to the developers of Aladdin [57]. For all simulated versions of the selected regions (or HW accelerators), the number of Cycles and number of Functional Units (FU) Area were retrieved. For the SW execution time the gem5 simulator [8] was used with two CPU settings: a) TimingCPU (a simple and slow CPU with only two pipeline stages) and b) O3CPU (a complex and fast CPU with five pipeline stages and other resources such as a branch predictor, reorder buffer etc).

The exact selection algorithm, as detailed in Subsection 1.3.1 was used subsequently to perform the optimal subset selection, given an initial set of HW accelerators along with their respective versions, as well as a specific area (HW resources) budget. An important note is that no more than one version of each candidate can be selected, as only one realization of the respective SW execution is required. To ensure that, the selection took place utilizing the overlapping graph presented in Figure 1.2 containing the basic block indexes included in each region. As a result the set of basic block indexes for multiple versions of the same region would be identical. Experiments were run in `jpeg` benchmark and four different selection approaches are presented, compared to the MuLTiVersioning approach.

1.4.2 Experimental Results

The experimental setup was the same as in the RegionSeeker framework, with a system comprising a single SW processor and multiple loosely coupled HW accelerators, exchanging shared data with private local memories. The processor invokes the accelerators via a memory-mapped interface, thus requiring a transaction on the system bus and as soon as the HW accelerators execution is complete, control returns to the SW processor.

The speedup achieved on `jpeg` benchmark, over the respective SW time of

the same set of selected regions (kernels) are showcased. The four different approaches compared are: a) the *min* where the regions with the least amount of area are included in the set and hence can be selected, b) the *base* where the regions with median values of area are selected, c) the *max* where only the maximum area regions can be selected and finally d) the MuLTiVersioning approach where any possible version of the regions can be selected.

The strength of the MuLTiVersioning approach and the benefit of having a variety of potential candidates to select from is demonstrated by the experimental outcome of the `jpeg` application (kernels run-time). In Figure 1.6 for any given area point, the speedup obtained is higher than any other methodology. For a medium area point ($200 * 10^3 uM^2$), the speedup achieved with MuLTiVersioning is 1.7x more than the second best, *base* approach. For a large area constraint ($400 * 10^3 uM^2$) the MuLTiVersioning speedup is more than 2x compared to *base* and more than 6x compared to *min*.

1.5 Conclusions

The RegionSeeker framework, along with the RegionSeeker MuLTiVersioning extension of the former, are methodologies that extend the state-of-the-art in the HW/SW co-design domain. They provide efficient solutions to the problem of automatically deciding which parts of an application should be synthesized to HW, under a given area budget. The accelerators identified by RegionSeeker consistently outperform the ones derived by data flow level algorithms and strictly function level candidates, across applications of widely different sizes and for varied area constraints. As an example, RegionSeeker offers up to 4.5x speedup for the `mpeg2` benchmark compared to as SW execution. This work was published in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) journal [76]. The MuLTiVersioning approach extends the initial selection pool of candidates and, compared to default HW accelerators configurations, offers enhanced speedup on the `jpeg` application of up to 1.7x speedup on the entire application and up to 65x speedup on the relative kernels that are synthesized into HW.

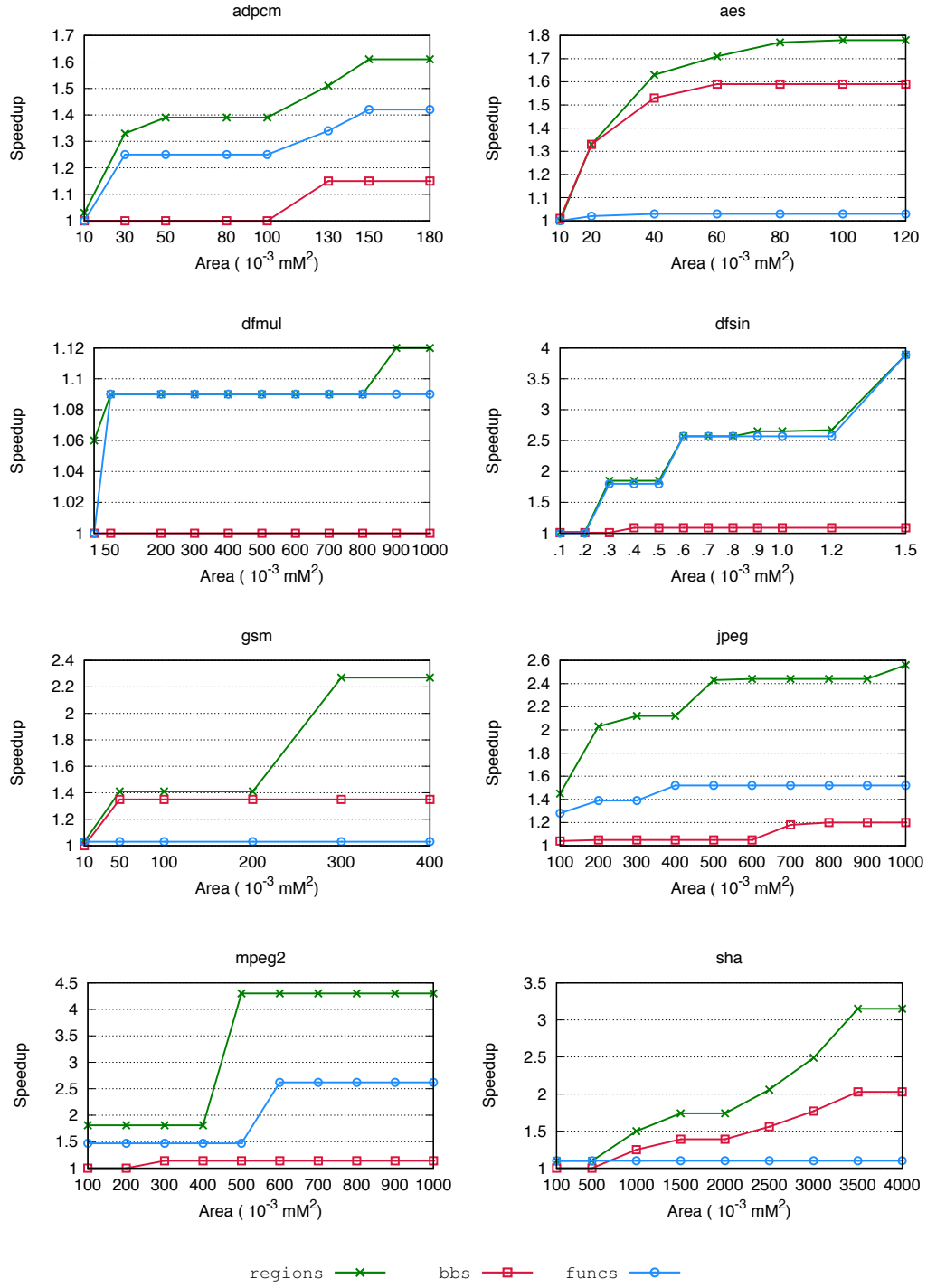

Figure 1.3. Comparison of speedups obtained on eight CHStone benchmarks by selecting regions, only basic blocks and only functions, varying the area constraint, using Aladdin and Gem5 for Speedup and Area evaluation.

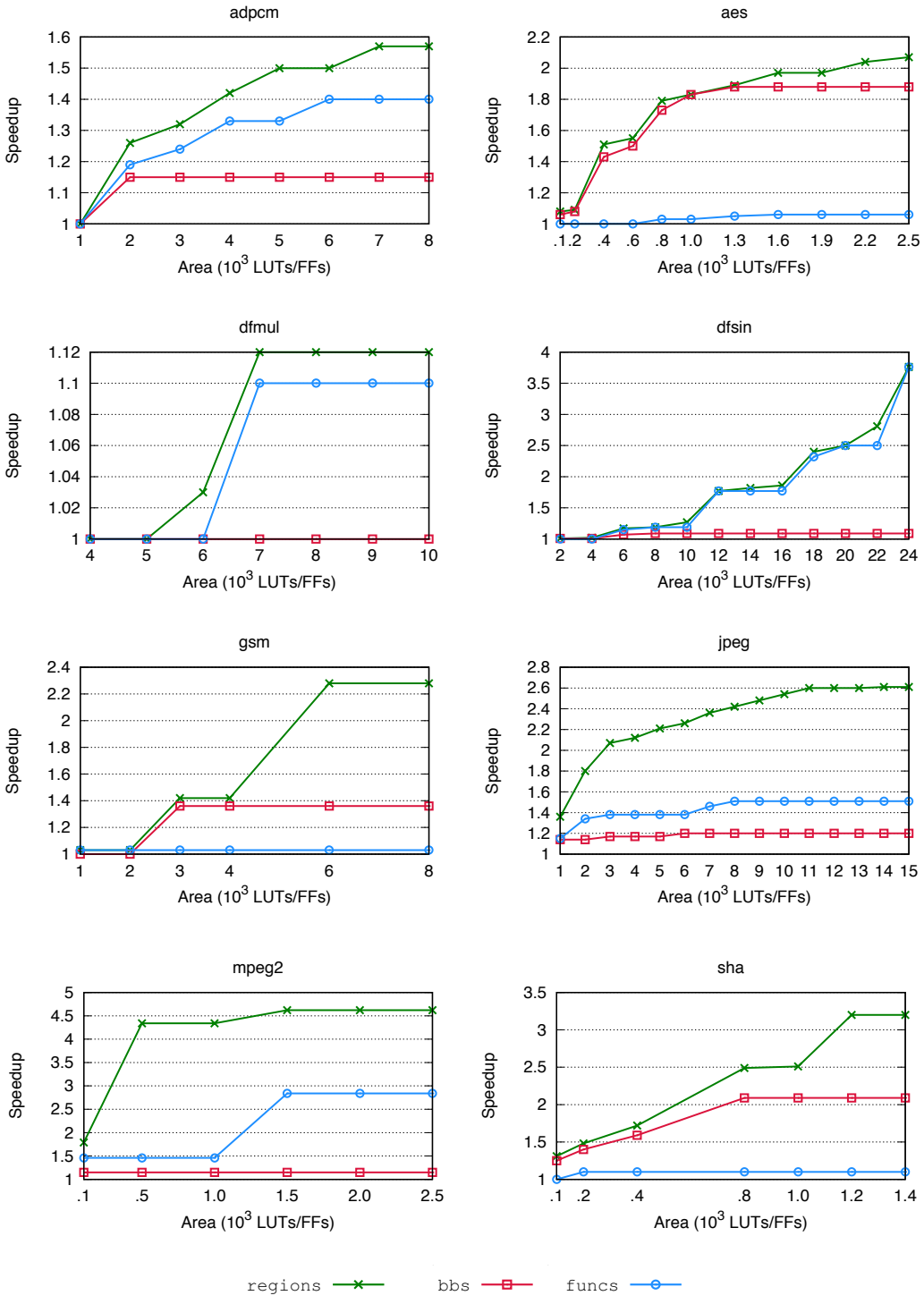

Figure 1.4. Comparison of speedups obtained on eight CHStone benchmarks by selecting regions, only basic blocks and only functions, varying the area constraint, using Vivado_HLS and Gem5 for Speedup and Area evaluation.

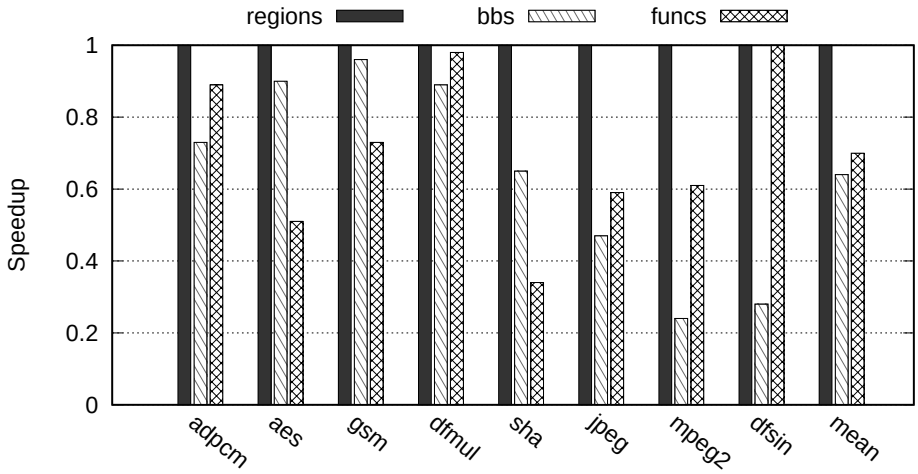

Figure 1.5. Normalized Speedup of RegionSeeker with respect to function and basic block selection, considering, for each benchmark, a fixed area constraint. Synthesis performed with Vivado_HLS.

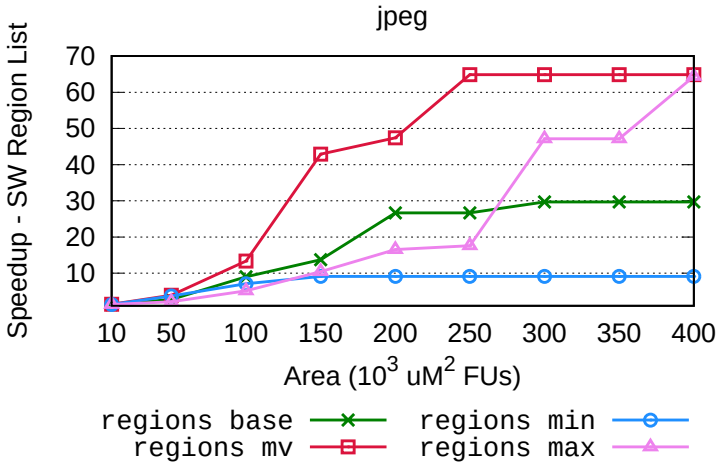

Figure 1.6. Comparison of speedup obtained on jpeg benchmark, over the SW time of the equivalent kernels (regions), varying the area constraint, using Aladdin and gem5 for Speedup and Area evaluation. Four approaches are compared: The min where the regions with least amount of area are selected, the base where the regions with median values of area are selected, the max where only the maximum area regions can be selected and finally the MuLTi-Versioning approach where any version of the regions can be selected.

Chapter 2

Automatic Optimizations for Accelerators

Identifying good candidates for HW acceleration is the first step to realize heterogeneous computing system designs that offer increased performance compared to a homogeneous system restricted to general purpose SW CPU(s). However, a set of optimizations applied on HW accelerators can decrease even more the computational times, thus leading to an improved performance compared to default non-optimized HW accelerator implementations. Modern High Level Synthesis (HLS) tools can apply such optimizations to HW accelerators and increase the performance of their implementations, as well as the overall performance of the entire heterogeneous system. HLS tools such as Vivado_HLS [68], however efficient, though, are far from optimal since they require a lot of manual decisions from the programmer's part when it comes to the choice of *how* these accelerators can be synthesized. Furthermore, the resolution of which optimizations may be applied to which HW accelerators can be a complex problem, as it depends heavily on each HW accelerator characteristics.

In order to bring automation one step forward in HW/SW co-design, and under the scope of this part of my research, I tackled the problem of automating the decision making process of which optimizations should be applied to candidates for HW acceleration within a certain context. These optimizations include memory management of the data consumed and produced by the HW accelerators, a set of optimizations targeted to loops (e.g. loop pipelining, loop unrolling, loop flattening etc), pipelining consecutive pieces of computation such as subsequent function calls or loop bodies of consecutive iterations and array optimizations, such as array partitioning in blocks of the same size. Among the various optimizations available, I have focused on two major categories: a) Data Reuse analysis

and b) Loop Unrolling factor prediction. Both of these instances are explained in more detail in the following two sections.

2.1 Data reuse Analysis

2.1.1 Motivation

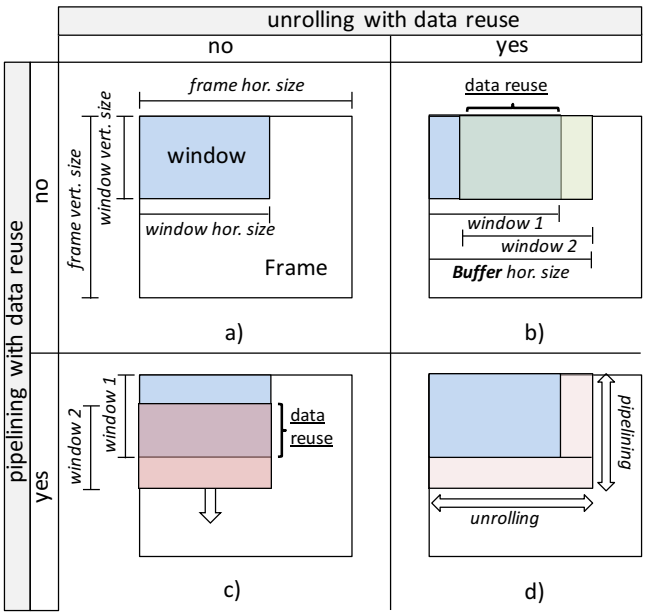

Figure 2.1. At each iteration, sliding window applications process a subset of the input data (a). The managed set of subsequent iterations present a high degree of overlap, both in the horizontal (b) and vertical (c) dimensions. The framework in 2.1.3 automatically leverages both, maximizing data reuse (d).

Loops are ideal candidates for acceleration. In almost every application, there is a number of them that contain a large number of iterations and there is a sufficient amount of computation taking place in their bodies. In addition to that, there are nested loops which commonly show a high level of data reuse. An effective exploitation of data reuse across consecutive iterations of loops can significantly lower the required amount of data exchange between HW accelerators and the main memory, thus reducing the respective bandwidth to and from accelerators, and increasing their performance.

An example of such high data reuse can be observed in sliding window applications, where there is typically a window of accesses scanning a wider domain, such as a two-dimensional array. Given that the level and pattern of data reuse

is known a priori, it is feasible to design specific memory structures, also known as memory buffers, attached to the HW accelerators. These memory buffers can exploit data reuse by keeping data locally and, hence, minimize the memory latency due to communication with the main memory.

Data reuse exploitation in High Level Synthesis (HLS) is still in premature stage. state-of-the-art methods [65] either rely on manually rewriting the source code, preceding HLS, or on source-to-source translation [50] [56], and are therefore poorly integrated in HLS tool-chains.

The methodology detailed in Subsection 2.1.3 attempts to bridge this gap. It presents a compiler-driven framework, based on the LLVM Polly [64] library, able to identify automatically data reuse potential in computational kernels in order to guide the synthesis of complex HW accelerators. As seen in Figure 2.1 these accelerators exploit timely unrolling and pipelining, by embedding a local storage holding elements which are re-used across iterations.

Sliding window applications, common in the image processing field, are targeted for acceleration. In such domain, a transformation is applied to each element of a large two-dimensional input array (a frame) according to the values in a smaller domain of accesses (a window). Large, yet constrained, input/output links are considered as the main architectural constraint in the design of this type of HW accelerators. Such arrangement is usually supported by commercial application specific platforms, such as the Tensilica Xtensa processor [63].

2.1.2 Related Work

In the domain of identifying automatically accelerators, research has so far focused mostly on accelerating data-flow [24] [27], not taking into equal account the potential for optimization by memory accesses. Exceptions are provided by papers [9] [28] where the authors support the claim that accelerators with custom storage can provide better speedup compared to the ones that accelerate data-flow only. However, these papers focus on the identification of the accelerators, and do not present a methodology to automatically identify the optimization potential, as well as synthesize them accordingly.

In sliding window applications, there are research endeavors both by academia and industry to exploit data reuse. The smart buffers [26] generated by the ROCCC compiler [67] allow for automatic detection of data reuse opportunities, but cannot be interfaced with interconnects of varying width. The methodology described in [40] employs reuse buffers spanning multiple frame columns, which pose a significant area overhead. Both [26] and [40] are not able to combine Hardware unrolling and pipelining, which are instead jointly supported the

methodology detailed in 2.1.3. An alternative approach, described in [20], requires a great deal of Hardware resources as well, as it requires the storage of large parts of a frame being processed inside the custom hardware. In [72], the authors propose an analytical method to gather microarchitectural parameters for sliding-window applications on FPGAs. Their design however ultimately needs to be manually implemented and hence the work neglects high level synthesis aspects.

The commercial Vivado_HLS tool requires extensive manual rewrite of the source code, in order to instantiate a reuse memory buffer. On the other hand, the approach presented here relies on automated code analysis to derive the characteristics of the target application.

2.1.3 Methodology

In order to generate the custom-storage HW accelerators, a two-steps methodology is carried out. First the data reuse analysis of the application takes place and then the synthesis of the part of computation to be implemented in HW. The first is performed with the utilization of compiler static source code analysis while in the latter details regarding the Hardware implementations are provided. The phases of analysis and synthesis lead to the design and implementation of custom-storage accelerators that manage to minimize the latency due to data transfer between main memory and the HW accelerators.

Data Reuse Analysis

In order to identify the level of data reuse that takes place throughout the execution of every window application, there are three pieces of information that are vital: a) the size of the window, b) the stride and c) the frame size. An example of data reuse, accounting for roughly 66% the size of the window between consecutive iterations, can be seen in Figure 2.2. The window size is the access pattern within the innermost body of the loop. The innermost and outermost loop stride is the value of the induction variable increase for the innermost and the outermost loop respectively. Finally, the frame size is the iteration space within which the sliding window is moving. To extract this information I have developed an analysis pass, based on the LLVM Polly framework [64]. Application-specific parameters are then considered in conjunction with architecture constraints (input/output width) to automate the synthesis of efficient HW accelerators.

The analysis pass, as seen in Algorithm 2, iterates over regions of the application functions and identifies Static Control Parts (SCoPs). The SCoPs are sub-

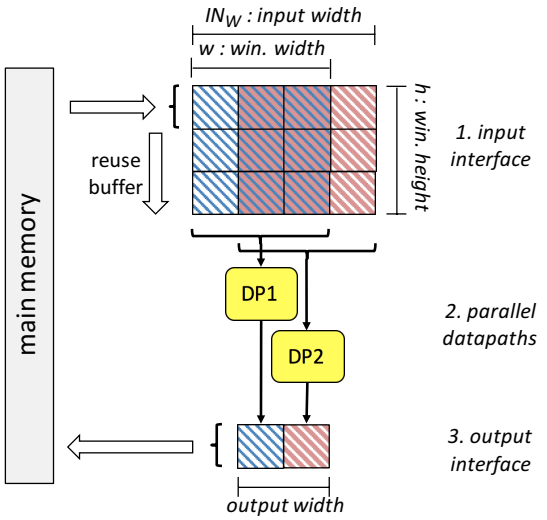

Figure 2.2. Data Reuse of memory accesses (66.6%) across consecutive iterations of a loop, in a sliding window application.

graphs of the control flow graph of a function where the flow of control is known statically. For each SCoP, loop and scalar evolution information is collected from the body of the loop.

Loop information supports methods that can provide the loop depth, so as to identify the innermost loop. Scalar evolution information can be used to extract the loop trip count, which is the iteration space of each loop, and thus compute the frame size. The stride value, both vertical and horizontal, is obtained by a function that was developed based on existing methods of the Loop analysis LLVM pass. Lastly, the read memory accesses of the innermost body of the loop are identified by using `isl` functions, which compute the distance (or delta) of each of these read accesses with respect to the first one. Given the access pattern, the window size is computed as the minimum enclosing rectangle. After having identified the necessary information, the implementation of a local buffer that fits these needs can be carried out.

Hardware Implementation

The parameters retrieved with the analysis pass (frame size, stride, horizontal and vertical window size) and the characteristics of the interconnect (input and output width) are employed to derive efficient HW accelerators implementation with local storage and data reuse.

As showcased in Figure 2.2, accelerators implementations embed multiple

Algorithm 2 LLVM Analysis Pass - SCoP Identification and Data Reuse Analysis

Input: Application written in C/C++

Output: List of Identified SCoPs and their respective Frame, Window, Stride sizes/values.

```
1:  function RunOnRegion()
2:      getAnalysis(ScopInfo)
3:      scop = getScop()
4:      RunOnScop(scop)
5:
6:  function RunOnScop(scop)
7:      LI = getLoopInfo()
8:      SE = getSE()
9:      if L == OutermostLoop then
10:         getTripCountFortLoop()
11:         getStrideForLoop()
12:     else if L == InnerMostLoop then
13:         getReadMemoryAccesses()
14:         ComputeDistancesForReadAccesses()
15:         ComputeWindowSize()
```

combinatorial datapaths, each executing one iteration of the loop body of the target application. The input interface embeds a local storage, whose horizontal size corresponds to the available input data width of IN_W data elements, while its vertical size is equal to the vertical size of the application window h. It is implemented as a $IN_W * h$ shift register, operating in the vertical (top-down) direction. During execution, the first row of the shift register is filled with input data in each clock cycle. A subset of the elements stored in the shift register is connected to each of the different datapaths according to their managed sets, e.g.: the first one having inputs corresponding to the buffer columns ranging from 0 to $w-1$ (the horizontal size of the window) and the second one corresponding to the buffer columns ranging from 1 to w. Figure 2.2 illustrates such scheme for the simple case of $IN_W = w+1$.

At the beginning of the execution, h rows are stored in the shift register before activating the datapaths logic. Afterwards, this activation is performed for each new row, discarding the last (topmost) line and storing a new one in the first (bottom) position of the shift register. At the completion of a vertical slide of the window through the frame, a new one is started, increasing the horizontal

displacement of the buffer by $IN_W - w + 1$ elements.

Finally, since no reuse opportunities are present for outputs, the output interface simply concatenates the values generated by the datapaths, and transfers them as a single and wide memory access.

2.1.4 Experimental Results

The evaluation of this approach is carried out in three benchmarks of varying window sizes. `Sobel` is an edge detection algorithm with an access pattern of a 3x3 window. `BlockSAD` is a kernel in H.264and is used to detect the similarity among 4x4 blocks. Finally, `Maximum Filter` computes the brightest pixel among neighbors in 8x8 blocks. Three different configurations were considered, spanning from a singe datapath and minimum input width (Conf.1) to multiple datapaths and increased input width (Conf.2 and Conf.3) as seen in Figure 2.2. Multiple datapaths translate to more parallel windows executing and, hence, increased demand in area resources.

The comparison of the approach introduced above is carried out against two state-of-the-art HLS tools: ROCCC and Vivado_HLS. Vivado_HLS is compared in two modes, one being the default and the other one after extensive manual rewrite of the source code in order to obtain increased data reuse.

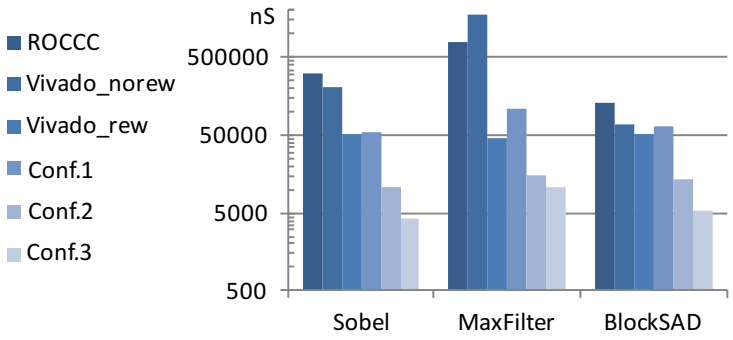

Figure 2.3. Execution time to process a 100x100 frame.

Execution time, as seen in Figure 2.3, is extracted from a targeted Xilinx Virtex7 FPGA platform. It can be observed that ROCCC systems have similar performance with respect to Vivado_norew ones. Conf.1 accelerators — even though they do not require code modifications — are as efficient as Vivado_rew ones. Conf.2 and Conf. 3 that are supported only by the framework presented in 2.1.3, dramatically decrease run-times, with an order-of-magnitude speed up on average between Conf.1 and Conf.3. The other state of the art tools *fail to provide an*

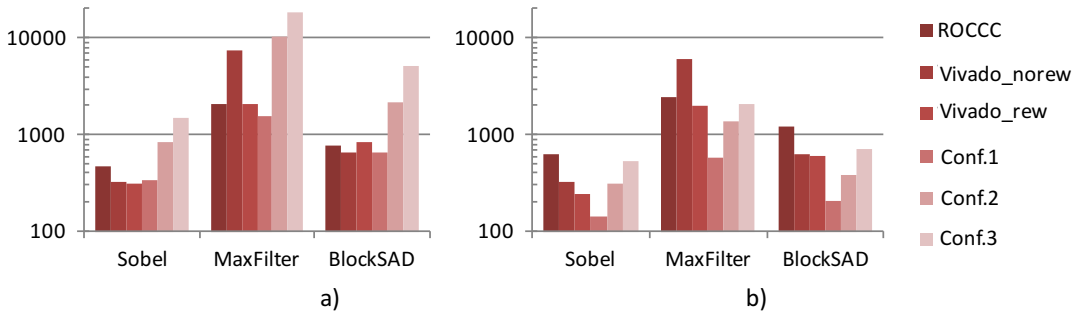

Figure 2.4. Comparison of required resources for our generated systems and for baseline approaches: LUTs (a) and Flip-Flops (b).

equivalent solution with such low execution time. Figure 2.4 reports the amount of area resources required by ROCCC, Vivado_HLS and our own generated accelerators. Unsurprisingly, accelerators featuring a high number of datapaths (Conf.3) require more resources than single-datapaths approaches (Conf.1, Vivado). Nevertheless, the area increase in terms of Flip-Flops is comparable to the other to state-of-the-art tools, as the size of the buffer only is increased slightly to support a high degree of parallelism. On the other hand, the results highlight that complex accelerators require an increased amount of combinatorial logic (LUTs), with respect to ROCCC and Vivado_HLS.

2.1.5 Conclusions

It has been demonstrated that static source code analysis can be crucial, when it comes to automatically optimizing the synthesis of accelerators that are dedicated to sliding window applications. My SW analysis identifies data reuse, as well as data locality, and subsequently allows to exploit these characteristics by making use of appropriate memory buffers. The experimental results reveal an order-of-magnitude performance improvement with respect to state-of-the-art methodologies. This work was published in HiPEAC IMPACT 2017 Seventh International Workshop on Polyhedral Compilation Techniques [74].

2.2 Machine Learning Approach for Loop Unrolling Factor Prediction

2.2.1 Motivation

High Level Synthesis tools, utilized to synthesize accelerators, require manual decisions to be made, so as to build efficient accelerators. These decisions regard the choice of high-level optimizations and transformations to be applied, therefore a good understanding of the SW parts to be accelerated is essential. Optimizations applied during synthesis can highly improve the performance achieved, as well as allocate less HW resources for a given computer architecture. Nonetheless, the selection of optimizations is a challenging task due to two main reasons. First, hardware synthesis is a time-consuming process, limiting in practice the amount of possible implementations that can be evaluated. Second, the effect of assigning different values to directives is difficult to foresee, due to low-level application characteristics.

Simulation tools such as Aladdin [57] have been developed in order to rapidly estimate the performance and cost (area) of HLS-defined designs. Nonetheless, even when employing estimation tools, an exhaustive evaluation of all directives settings for each candidate accelerator in a heterogeneous system is still unfeasible beyond simple cases. Addressing this challenge, a machine learning framework is proposed that is able to infer the proper implementation of an HLS design based on its characteristics, automatically derived from a source code analysis pass, based within the LLVM compiler framework [35].

Within the scope of this piece of research, the focus is placed on *loop unrolling*, an already well known optimization from the compiler domain, as well as the HW domain. The loop unrolling optimization replicates the body of a loop a given number of times in order to expose parallelism, which especially in a HW implementation can lead to substantial speedup gains [34]. This directive should nonetheless be applied sensibly, because it entails a high area cost for the duplicated logic; in addition, its ensuing benefits can be hampered by loop-carried dependencies and frequent memory accesses.

It is thus clear that there is a trade-off between execution time and the area budget that a computer architect has at hand, as well as the level of complexity and more potential side effects that need to be taken into consideration. Since HW realizations are targeted, the goal of this work is to reach a *sweet spot* between performance and HW resources.

Within the sphere of this research work, the following contributions are made.

First, a novel *Machine Learning* approach is introduced, based on Random Forest classification, instead of estimation-based models, to predict accurately the optimal loop unrolling factor of loops in applications to be synthesized in HW. The use of this methodology can provide results with better prediction score and in much less time, compared to the state-of-the-art. Second, the whole process is fully automated, from the analysis of the input applications, using the LLVM compiler infrastructure [35], up to the training of the Random Forest Classifier. Finally, the trained Random Forest classifier can be used to generate accurate loop unrolling predictions for any given application – a piece of information that can directly be used by an HLS tool such as Vivado_HLS, in order to synthesize parts of these applications to HW.

2.2.2 Related Work

Research papers have explored the applicability of machine learning to apply compiler optimizations. In Software compilers, it has been employed by Agakov et al. [1] to speed up iterative compilation, by Monsifrot et al. [42] to produce compiler heuristics and by Kulkarni et al. [33] to select the order in which optimization passes should be performed. Stephenson et al. [60] have made use of Supervised Classification, such as near neighbor (NN) classification and Support Vector Machines (SVM) methods, to produce accurate predictions in optimal unrolling factors. In all above-mentioned research works the authors targeted Software compilation; in Subsection 2.2.4, a comparative performance evaluation of the framework presented in Subsection 2.2.3 to the methodology proposed by Stephenson et al. is carried out, showcasing the benefit of the choice of loop features and classification strategy in the HLS scenario.

Liu et al. [36] used a Random Forest classification model in the context of HLS, extending the Iterative Refinement framework proposed in [38] [46] [71] and [78]. They address a different problem with respect to the one tackled in this section: that of retrieving the set of Pareto-optimal implementations of a given design by navigating its configuration space. A similar stance, addressing system-level design, is illustrated by Ozisikyilmaz et al. [45]. As opposed to these works, my aim is to perform a predictive assignment of synthesis directives, based on a training performed on an independent input set. This problem was also investigated by Kurra et al. [34]. Contrary to their methodology, my methodology does not depend on a detailed estimation delay model of the loop body so as to predict loop unrolling factors in HLS instances.

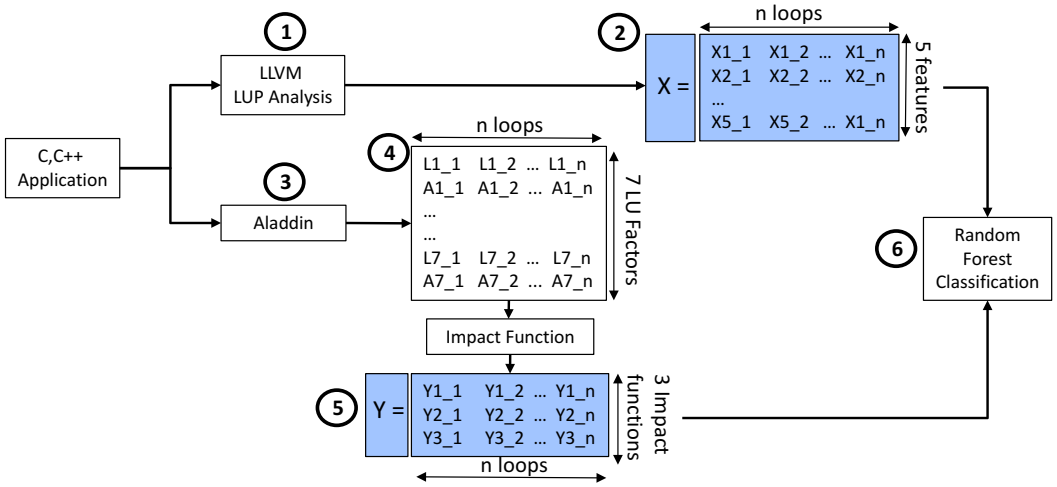

Figure 2.5. Overview of the Loop Unrolling Prediction methodology.

2.2.3 Methodology

In this section, first the employed objective function that determines the optimal loop unrolling factor is presented. Then, the LLVM analysis pass that was developed in order to automatically extract relevant loop features and the approach followed to retrieve the area and run-time performance of HLS designs is detailed. Lastly, the supervised learning classifier method is demonstrated, which, during the training phase, gathers the data from the previous steps to produce a loop unrolling predictor, and, during the test phases, assigns loop unrolling factors based on loop features.

Optimal Loop Unrolling Factor – Objective Function

The *optimal loop unrolling factor* is defined as follows. Given a defined set of unrolling factors S: <1, 2, 4, 8, 16, 32, 64>, there is one for every loop that maximizes the **Impact (I)**, given by the following formula:

$$I(L,A) = \alpha \cdot \frac{(L_1 - L)}{L_1} + \beta \cdot \frac{(A_1 - A)}{A_1},\ \alpha + \beta = 1$$

Where L_1 is the latency of the of the function containing a loop and being synthesized as HW accelerator, for Loop Unrolling Factor (LUF) that is equal to one, i.e., a fully rolled loop. L is the latency of the HW accelerator for any possible LUF from the defined set. Respectively, A_1 is the area resources requirements of the HW accelerator with LUF equal to one and A the area for any possible LUF

from the defined set.

Subsequently, the optimal LUF is defined as the one that maximizes the Impact function above. Note that, when $LUF = 1$, then $I(L,A) = 0$ which corresponds to a baseline implementation. $I(L,A)$ may also be negative for suboptimal LUF choices (where unrolling might increase area without decreasing latency), but will always be ≥ 0 for optimal unrolling factors.

For the evaluation presented in Subsection 2.2.4 three different strategies were considered: a) Optimize for both latency and area ($\alpha = \beta = 0.5$). In this configuration a balance is maintained between decreasing the number of execution cycles and keeping low the usage of HW area resources in a given implementation. b) Optimize for latency ($\alpha = 0.7, \beta = 0.3$). Minimizing latency is favored by this approach, thus focusing on increasing the speedup of an application, and finally c) Optimize for area ($\alpha = 0.3, \beta = 0.7$). This approach aims at decreasing the area budget of the implementation, therefore achieving an average speedup, but maintain low usage of HW area resources. All three configurations fulfill different architectural needs and explore realistic alternative scenarios.

LLVM Analysis Pass – Loop Features Extraction

Loop features are automatically identified by an analysis pass (depicted as point 1 in Figure 2.5) that was developed within the LLVM compiler infrastructure [35]. Features are retrieved starting from applications written in C or C++, operating on their Intermediate Representation, provided by the LLVM front-end passes.

My *LLVM Loop Unrolling Prediction Analysis Pass* iterates over functions of the applications and identifies loops. On each of them, it performs loop, scalar evolution and dependence analysis to extract their features, summarized in Table 2.1: the critical path, the trip count, the presence of loop carried dependencies and the required memory accesses (load and stores).

The choice of features is based on the factors that influence the cost and the achievable speedup of Hardware unrolled loops: a loop with a long critical path may be expensive to duplicate, while loop carried dependencies and memory accesses may force a serialization of execution irrespectively of the degree of unrolling. These considerations lead us to consider a markedly different feature list with respect to works focusing on software targets, such as the one of Stephenson et al. (Table 2.2).

Features - X Vector
Critical Path
Loop Trip Count
Has Loop Carried Dependencies
Load Instructions
Store Instructions

Table 2.1. Features extracted by LLVM LU Analysis Pass.

Features - X Vector 1	**Features - X Vector 2**
# Operands	*# Floating Point Operations*
Range Size	*Loop Nest Level*
Critical Path	*# Operands*
# Operations	*# Branches*
Loop Trip Count	*# Memory Operations*

Table 2.2. Feature vectors selected by Stephenson et al. [60].

Latency and Area Estimation

To establish a link between LUFs and performance/cost of implementations, latency and area values must be extracted both for the loops in the training set (in order to optimize the classifier) and the ones in the test set (to measure its accuracy).Aladdin [57] (point 3 in Figure 2.5), a pre-RTL power-performance simulator for Hardware accelerators was utilized in order to retrieve latency and area information. All functions in the considered benchmarks were simulated by employing each feasible unrolling factor in the S set defined above on every loop contained. Latency is reported by Aladdin in clock cycles, while area is expressed in μm^2 in a $45nm$ technology. The result is shown as point 4 in Figure 2.5.

The Impact (I) was computed afterwards for the different α and β values, to retrieve the optimal loop unrolling factor for every loop of a function, which is the index of the LUF that maximizes I. The result is three vectors $\{Y1, Y2, Y3\}$ (point 5 in Figure 2.5) that contain the target values for the classification algorithm. The $Y1$ vector includes the optimal loop unrolling factor that balances the Hardware implementation of the accelerators in terms of low latency and low area. Values in the $Y2$ vector favors low-latency implementations, applying more aggressive loop unrolling, whereas $Y3$ favors low-area ones.

Algorithm 3 LLVM Analysis Pass - Loop Unrolling Prediction Analysis

Input: Application written in C, C++
Output: X (Feature Vector)

```
1: function RunOnFunction( )
2:     for BB in Function  do
3:         if L=getLoopForBB()  then
4:             LoopUnrollingPredictionAnalysis(BB,L)
5:
6: function LoopUnrollingPredictionAnalysis(Basic Block BB, Loop L)
7:      LI=getLoopInfoAnalysis()
8:     SE=getScalarEvolutionAnalysis()
9:     DA=getDependenceAnalysis()
10:    /* Gather Features for X Vector */
11:    x1=getCriticalPath(BB)
12:    x2=getTripCountForLoop(L)
13:    x3=getLoopCarriedDependencies(BB)
14:    x4=getNumberOfLoadInstructions(BB)
15:    x5=getNumberOfStoreInstructions(BB)
```

Random Forest Classification

This information (X and Y vectors) extracted as described above are used as input to a Random Forest (RF) classifier (point 6 in Figure 2.5). Supervised learning is performed by detecting the correlation between the input, the compiler extracted information, used as the X feature vector, and the output, which is the optimal loop unrolling factor for each loop.

Random Forest was used as the supervised learning model, which has been shown by Liu et al. [36] to outperform alternatives such as Multilayer Neural Networks and Support Vector Machines classification in the context of HLS design space exploration. Random Forest algorithms follow a decision tree methodology, combining many weak classifiers to derive a strong one, allowing the generation of low-complexity and robust classifiers.

The algorithm employed, as presented in Algorithm 4, follows an approach similar to a *k-fold cross validation* strategy. The whole data set (X and Y vectors, see points 2 and 5 in Figure 2.5) is divided randomly between a training set and a test set, where the training set is equal to 80% of the whole data set and the remaining 20% is the test set. Then, the Random Forest model is used for the

Algorithm 4 Random Forest Classification - Training and Test

Input: X and Y Vectors
Output: Trained Random Forest Classifier

```
1: for i in NumberOfTrainingSessions do
2:     X_train, X_test, Y_train, Y_test=train_test_split(X,Y)
3:     /* Training Phase */
4:     M=RandomForestLearningModel
5:     M.train(X_train, Y_train)
6:     /* Evaluation Phase */
7:     Pred=M.predict(X_test)
8:     Error=abs(Pred-Y_test)
9:     Score=M.score(X_test-Y_test)
```

training process on the training set and out-of-sample predictions are carried out for each element of the test set. After all predictions on the test set have been computed, the prediction score and the average error (as defined in Subsection 2.2.4) are computed for the current training session.

2.2.4 Experimental Results

To evaluate the classification performance of a trained classifier, two different metrics were adopted. The *Prediction Score* states the percentage of optimal (according to $I(L,A)$) LUFs that were correctly identified on the out-of-sample test set. The *Average Error* instead measures the average distance between the indexes in S of the correct and the predicted LUF.

In order to comparatively evaluate the proposed methodology, combining Random Forest classification and LLVM-based loop features extraction, benchmarks of different complexity were considered. Small and medium-sized ones are `adpcm`, an audio encoding kernel, `stencil`, an implementation of an iterative algorithm that updates array elements according to a given pattern, and `sha`, a secure hash encryption method used in the information security domain. `jpeg` and `mpeg2` are instead larger benchmarks, which perform image and video compression, respectively. Applications were drawn from the CHStone [30] and the Scalable Heterogeneous Computing (SHOC) benchmark suites [18]. In total, they comprise 87 different loops.

Random Forest classification was implemented using the Scikit-learn suite [47], that includes state-of-the-art implementations of Machine Learning models

in python. Scikit-learn was also employed to re-implement the two methods proposed by Stephenson et al. [60], that are consider as baselines.

Giving an initial proof of concept for the strategy proposed, Figure 2.6 reports the difference between the indexes of the predicted optimal (according to impact value) loop unrolling factors and the ones retrieved with an exhaustive exploration, considering 18.000 out-of-sample predictions on all the benchmark loops. Results are highly concentrated on zero, indicating a high rate of correct predictions.

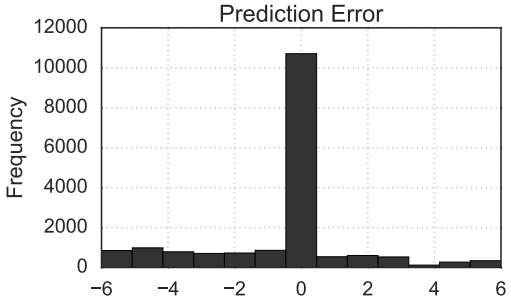

Figure 2.6. Distribution of Loop Unrolling Factor Prediction Errors over 18.000 out-of-sample predictions.

Classification Models and Features

The evaluation of the choice of features (X vector in Table 2.1) and training model Random Forest (RF), takes place against two state-of-the-art methodologies proposed by Stephenson et al. [60]. The latter present different classification strategies: Support Vector Machines (SVM) and Near Neighbor (NN) and a different choice of investigated loop features, reported in Table 2.2. Figure 2.7 shows, for a choice of $\alpha = 0.5$, the prediction score and the average error of the nine strategies resulting from different feature vectors and classification strategies. Experimental results highlight that the presented framework (X feature vector and RF classification) outperform other choices, reaching a prediction score above 60% and an average error of less than 1.4. Similar results were obtained for impact functions favoring area or latency ($Y2$ and $Y3$ vectors).

Iterative Refinement

In the second round of experiments, the comparison of the proposed method was carried out against an Iterative Refinement approach, used in [38] [46] [71] [78].

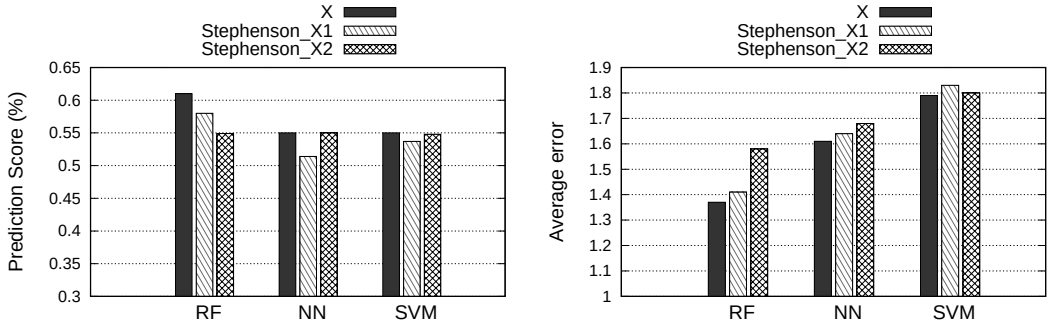

Figure 2.7. Left: Comparison of the Prediction Score across Random Forest, Nearest Neighbor, Support Vector Machines models and the respective feature selection: X vector, Stephenson et al. X1 and X2 vectors [60]. Right: Features extracted by my LLVM Loop Unrolling Analysis Pass.

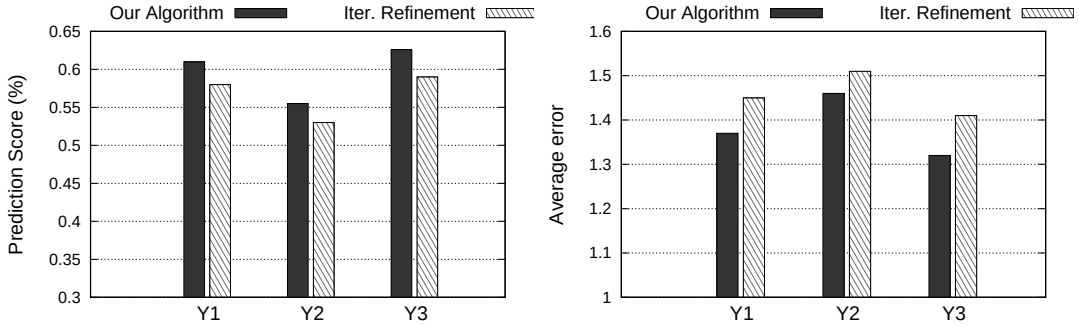

Figure 2.8. Left: Comparison of the Prediction Score across Random Forest, Nearest Neighbor, Support Vector Machines models and the respective feature selection: X vector, Stephenson et al. X1 and X2 vectors [60]. Right: Features extracted by my LLVM Loop Unrolling Analysis Pass.

Iterative Refinement uses part of the training data set to obtain a first version of the classifier, whose performance is then improved by using a second, disjoint set of input and outputs.

For this evaluation, three different settings of Y target vectors $\{Y1, Y2, Y3\}$ were considered, as described in Subsection 2.2.3. The employed data, the features (X vector) and the training model (Random Forest) were the same both for Algorithm 4 and the one using Iterative Refinement. For Iterative Refinement, 75% of the training set is allocated for the initial training phase, and the remaining 25% for the refinement phase.

The prediction score, as seen in Figure 2.8, ranges from 53% to 63% across the three Y vectors. Nevertheless, our methodology consistently outperforms

the Iterative Refinement approach, while achieving the highest prediction score (63%) for the setting that favors low area resources ($Y3$). A similar observation can be made for the average error values, where the suggested approach keeps a lower average error for all predictions, across all vectors, with the one related to the $Y3$ vector being the lowest (1.32).

2.2.5 Conclusions

A novel methodology based on LLVM analysis and Random Forest classification that performs loop unrolling factor prediction for HLS designs was performed. This approach achieves better prediction score in comparison to state-of-the-art Machine Learning methods. Experimental evidence showcases that, by carrying out accurate predictions of loop unrolling factors, high performance accelerator implementations can be realized, while avoiding time-consuming exhaustive explorations. This work was published in the 2018 International Conference on High Performance Computing & Simulation (HPCS) [75].

Chapter 3

System Aware Accelerators Identification

In the final Chapter of the dissertation the notion of automatic selection of HW accelerators is extended by taking into account the underlying system where the specialized accelerators are hosted. The knowledge of the characteristics of the system that is targeted can be critical for the choice of the HW accelerators to be implemented. The memory system of a platform for instance can vastly affect the latency due to data exchange between the main memory of the system and the HW accelerators. AccelSeeker, an LLVM-based tool-chain, is presented as a framework that performs automatic identification and selection of HW accelerators targeted to a specific System-on-Chip (SoC). AccelSeeker performs thorough analysis of applications source code and estimates memory latency along with computational latency of candidates for acceleration. The granularity of the candidates for acceleration is expanded to that of a subgraph of the entire call graph of an application to accommodate for the communication overhead between main memory and the HW accelerators. **NOTE:** Write for EnergySeeker.

3.1 AccelSeeker: Accelerators for Speedup

3.1.1 Motivation

System-level design, and heterogeneous computing as a whole, is witnessing a breakthrough. Emerging best practices based on High Level Synthesis (HLS) allow unprecedented productivity levels. HLS dramatically shortens development cycles by employing C/C++ descriptions as entry points for the development of both Software (SW) and Hardware (HW) greatly facilitating the task of migrating

functionalities between the two.

However, the design of heterogeneous systems comprising SW processors and HW accelerators is still a demanding endeavor, during which key decisions are left solely to manual effort and designer expertise [10] [43]. Furthermore, the long time required for HW synthesis, coupled with the huge space of alternative implementations exposed by real-world applications, limits in practice the number of accelerator choices that can be considered manually by a designer before HW/SW partitioning is settled.

In order to limit the entailed design effort, it is therefore crucial to identify the set of viable acceleration options quickly, and also early in the design process, before performing later and more detailed estimations. This key step is currently poorly supported by design automation tools. Indeed, state-of-the-art early partitioning strategies are solely based on profiling information [70] [35] which, as was also shown by the authors of [62], may often be misleading.

To fill this gap and offer an efficient solution to the problem stated above, AccelSeekeris presented. AccelSeeker offers a methodology to identify and select the suitable acceleration candidates in an application, from its software source code. Being implemented within the LLVM [35] compiler infrastructure, first models an estimation of the cost (required resources) and merit (potential speedup) of all candidate accelerators in a single, quick pass, and then selects the set that maximizes the estimated speedup, while not exceeding a resource target. The use of AccelSeeker can therefore guide Integrated Circuit architects in the early design phases, highlighting which segments of a computing flow should be targeted with HLS, and hence where to focus the process of applying optimizations to.

On the other hand, it indicates which parts are not likely to yield tangible benefits if realized in HW — either because they present a low computational footprint, or because their characteristics hamper their potential for HW acceleration, e.g. they require an excessive amount of data transfers while performing limited computations.

The approach of AccelSeeker is markedly different from that of performance estimators, as the most promising candidates are identified in a single, high-level exploration, reducing the scope of further, and more detailed, estimations. It also differs from approaches based solely on profiling information, as profilers do not offer a measure of costs and run-times of HW implementations. Furthermore, they do not account for invocation overheads – potentially leading to the selection of frequently called, but small, candidates. Finally, data transfers is an important factor that state-of-the-art profilers do not take into account – hence potentially suggesting candidates requiring an excessive amount of communi-

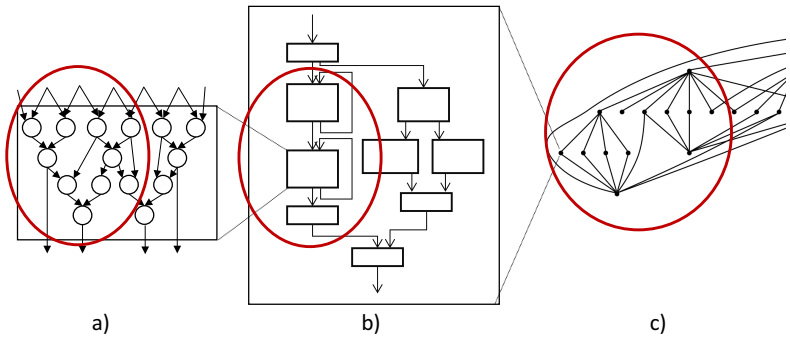

Figure 3.1. Evolution of the SoA in automatic selection of custom instructions/accelerators: (a) from data-flow level [51] [24], (b) to control-flow level [76] [44], (c) to function-call graph level (this work).

cation, that in turn can significantly weaken any potential performance gained. AccelSeeker, instead, takes into account both the communication overhead and all the characteristics of the platform that are required in order to generate an estimation model which leads to high-performance HW accelerators choices.

3.1.2 Related Work

High Level Synthesis tools have considerably matured in recent years [41]. Nowadays, available commercial tools (e.g.: Xilinx Vivado HLS [68], Cadence Stratus HLS [11]), as well as academic alternatives (e.g.: Legup [13], Bambu [48]) support the design of very large accelerators from C/C++ code. They reach performance levels comparable to hand-crafted implementations specified in low-level Hardware Description Languages (HDL) such as VHDL or Verilog [37].

Nonetheless, the automated selection of the application parts most amenable to HW acceleration is still an open research topic. Selection approaches based on synthesis results [14] scale poorly to complex applications, as these are only available late in the design process. Estimation frameworks offer a detailed analysis on the performance and resource requirements of a HW-accelerated system while avoiding full synthesis runs, either by interfacing software and HW simulators (e.g., gem5 [8] and Aladdin [57] in [58]), or by adopting a hybrid stance, in which HW execution times are estimated, while software ones are measured on a target platform (as in Xilinx SdSoC [32]). However, in both cases, estimations are performed *after* the partitioning of HW and software, which is left to a trial-and-error basis. A methodical solution for partitioning is instead proposed

in this chapter.

The downside of poor partitioning choices, and consequently the importance of automated tools such AccelSeeker that guide the selection of high-quality accelerator sets, is even more prominent when considering the high effort required to optimize the implementation of HLS-defined accelerators. Design optimization entails the specification of multiple directives to steer designs towards the desired area-performance trade-off. The link between directives and the performance of implementations is not straightforward, hence requiring the evaluation of multiple alternatives to reach the intended results, as exemplified in [54] [78] [22] [36]. It is therefore key to focus up-front this optimization effort only on those candidate accelerators which can lead, from an application perspective, to tangible speedups.

To this end, the approach introduced here is inspired by previous works on automatic identification of instruction set extensions. Most techniques in this field target customizable processors augmented with application-specific functional units, within the processor pipelines. Hence, these techniques usually constrain their search to the scope of single basic blocks [51] [24], as depicted in Figure 3.1a. Recently, the authors of [76] and [44] have instead targeted the identification of larger code segments, including control-flow structures belonging to single functions (depicted in Figure 3.1b). However, such scope still falls short of the one employed in HLS tools, which are devoted to the implementation of dedicated accelerators interfaced on a system bus [17]. In this setting, the cost of data movement becomes so prominent that even control-flow structures inside functions fail to deliver performance. Suitable accelerator candidates must then encompass entire functions, including in turn all functions called within their call tree. AccelSeeker considers this same granularity (Figure 3.1c), advancing the state of the art in automatic accelerators identification.

3.1.3 Methodology

The methodology embedded in AccelSeeker, whose high-level scheme is depicted in Figure 3.2, is detailed. First, the definition of a candidate for acceleration is provided, and then how the selection of a subset of potential candidates to be implemented in hardware is being performed (A and C in the figure). Subsequently, the approach employed to estimate the candidates performance and resource requirements (B in the figure) is presented.

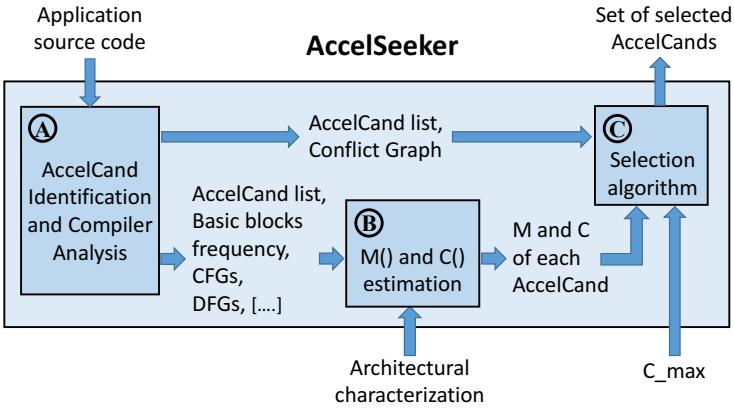

Figure 3.2. The phases of the AccelSeeker approach. A) Candidates for acceleration are identified, from source code analysis. B) Given an estimation of merit and cost for each candidate, and C) given a maximum available cost, AccelSeeker performs selection of a subset of such candidates.

Candidate Identification

In order to discover which parts of an application can be most profitably accelerated in hardware, the investigation of its function-call graph takes place, i.e., a Directed Acyclic Graph $G(N, E)$, where every node $n \in N$ corresponds to a function and every edge $e = (u, v) \in E$ corresponds to a call from function u to function v. A *root* is a node that reaches all other nodes of the graph, i.e., for every other node $n \in N$, there exists a path from the root to it. The function-call graph G has a root, which represents the top-level function of the application. Figure 3.3a shows an example of a call graph (note that edge directions are not shown here, for picture clarity; they are, however, intended from top to bottom).

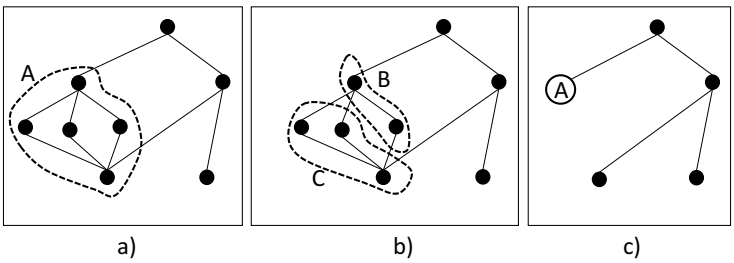

Figure 3.3. a) An example call graph. Black nodes are the functions present in a SW application, and edges represent function calls. Subgraph A is an AccelCand. b) Subgraphs B and C are not AccelCands (B has outgoing edges, C has no root). c) Call graph resulting from selection of A as accelerator.

A candidate accelerator is defined, and called **AccelCand**, as a subgraph $S(N_s, E_s)$ of graph G, exhibiting the following two characteristics: the subgraph has a root; the subgraph has zero outgoing edges. The former means that the subgraph has a node that reaches all other nodes in the subgraph; the latter means that, for every node $n_s \in N_s$, no edge (n_s, m_s) exists in G such that $m_s \notin N_s$.

Figure 3.3a and 3.3b show three example subgraphs, labeled A, B, C. While subgraph A is an AccelCand, subgraph B is not, because it does not have zero outgoing edges, and subgraph C also is not an AccelCand, because it does not have a root. The call graph resulting from selection of AccelCand A as accelerator is shown in Figure 3.3c: the whole subgraph is subsumed to a single (accelerator) call.

The methodology is limited to considering call graphs that are acyclic, and hence constructs such as recursion cannot be dealt with. This is in line with the limitations of HLS tools.

Cost and Merit Estimation

Herein, we detail how the abstract cost and merit are automatically computed from source code (Figure 3.2, B). As the goal of the presented framework is to select the most performing candidates *in advance of their optimization*, AccelSeeker considers their default implementations, e.g., ones where no function is inlined and no loop is unrolled. High-performance implementations will likely have greater resource requirements, in turn potentially requiring to discard some of the selected AccelCands. Nonetheless, these additional design decisions will be performed within the limited scope of the candidate set retrieved by our tool (as opposed to the whole design) thereby easing the ensuing effort.

Selection Algorithm

3.1.4 Experimental Results

3.2 EnergySeeker: Accelerators for Energy Efficiency

Conclusions

Bibliography

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, J. Thomson, M. Toussaint, and C. K. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the 4th International Symposium on Code Generation and Optimization*, pages 295–305. IEEE Computer Society, 2006.

[2] M. A. Aguilar, R. Leupers, G. Ascheid, and L. G. Murillo. Automatic parallelization and accelerator offloading for embedded applications on heterogeneous mpsocs. In *Proceedings of the 53rd Design Automation Conference*, pages 49:1–49:6. ACM, June 2016.

[3] J. Ahn and K. Choi. Isomorphism-aware identification of custom instructions with I/O serialization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(1):34–46, Jan. 2013.

[4] C. Alippi, W. Fornaciari, L. Pozzi, and M. Sami. A DAG based design approach for reconfigurable VLIW processors. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 778–79, Mar. 1999.

[5] Altera. Arria 10 SoCs: Highest system level integration SoC in production. `www.altera.com/products/soc/portfolio/arria-10-soc/overview.html`, Nov. 2016.

[6] K. Atasu, R. G. Dimond, O. Mencer, W. Luk, C. C. Özturan, and G. Dündar. Optimizing instruction-set extensible processors under data bandwidth constraints. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 588–593, Nice, France, Feb. 2007.

[7] M. Baleani, F. Gennari, Y. Jiang, Y. Patel, R. K. Brayton, and A. Sangiovanni-Vincentelli. HW/SW partitioning and code generation of embedded control applications on a reconfigurable architecture platform. In *Proceedings of the*

10th International Workshop on Hardware/Software Codesign, pages 151–56, Estes Park, CO, May 2002.

[8] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, Feb. 2011.

[9] P. Biswas, N. Dutt, P. Ienne, and L. Pozzi. Automatic identification of application-specific functional units with architecturally visible storage. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 212–217, Mar. 2006.

[10] M. Cacciotti, V. Camus, J. Schlachter, A. Pezzotta, and C. Enz. Hardware acceleration of HDR-image tone mapping on an FPGA-CPU platform through high-level synthesis. In *International System-on-Chip Conference*, pages 158–162. IEEE, Sept. 2018.

[11] Cadence. Stratus high-level synthesis. `www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html`, Apr. 2016.

[12] Cadence. Tensilica customizable processors. `https://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable`, Mar. 2017.

[13] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2):1–27, Sept. 2013.

[14] A. Canis, J. Choi, B. Fort, R. Lian, Q. Huang, N. Calagar, M. Gort, J. J. Qin, M. Aldham, T. Czajkowski, et al. From software to accelerators with LegUp high-level synthesis. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, page 18. IEEE, Sept. 2013.

[15] X. Chen, D. L. Maskell, and Y. Sun. Fast identification of custom instructions for extensible processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):359–68, Feb. 2007.

[16] J. Cong, Y. Fan, G. Han, and Z. Zhang. Application-specific instruction generation for configurable processor architectures. In *Proceedings of the 2004*

ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays, pages 183–89, Monterey, CA, Feb. 2004.

[17] E. G. Cota, P. Mantovani, G. Di Guglielmo, and L. P. Carloni. An analysis of accelerator coupling in heterogeneous architectures. In *Proceedings of the 52nd Design Automation Conference*, pages 1–6. ACM, June 2015.

[18] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[19] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[20] Y. Dong, Y. Dou, and J. Zhou. Optimized generation of memory structure in compiling window operations onto reconfigurable hardware. In *Reconfigurable Computing: Architectures, Tools and Applications, ARC*, pages 110–121, 2007.

[21] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 365–376, 2011.

[22] L. Ferretti, G. Ansaloni, and L. Pozzi. Cluster-based heuristic for high level synthesis design space exploration. *IEEE Transactions on Emerging Topics in Computing*, (99):1–9, Jan 2018.

[23] C. Galuzzi, E. M. Panainte, Y. Yankova, K. Bertels, and S. Vassiliadis. Automatic selection of application-specific instruction-set extensions. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 160–165, Oct. 2006.

[24] E. Giaquinta, A. Mishra, and L. Pozzi. Maximum convex subgraphs under I/O constraint for automatic identification of custom instructions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(3):483–494, 2015.

[25] Z. Guo, B. Buyukkurt, W. Najjar, and K. Vissers. Optimized generation of data-path from C codes for FPGAs. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 112–117, Mar. 2005.

[26] Z. Guo, B. Buyukkurt, and W. A. Najjar. Input data reuse in compiling window operations onto reconfigurable hardware. In *Proceedings of the 2004 ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 249–256, 2004.

[27] G. Gutin, A. Johnstone, J. Reddington, E. Scott, and A. Yeo. An algorithm for finding input-output constrained convex sets in an acyclic digraph. *J. Discrete Algorithms*, 13:47–58, 2012.

[28] M. Haaß, L. Bauer, and J. Henkel. Automatic custom instruction identification in memory streaming algorithms. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 1–9, Oct. 2014.

[29] R. Hank, S. Mahlke, R. Bringmann, J. Gyllenhall, and W. Hwu. Superblock formation using static program analysis. In *MICRO 26: Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 247–255, Sept. 1993.

[30] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii. CHStone: A Benchmark Program Suite for Practical C-Based High-Level Synthesis. In *Proceedings of the 2008 IEEE International Symposium on Circuits and Systems*, pages 1192–1195. IEEE, 2008.

[31] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *ACM SIGPLAN Notices*, volume 29, pages 171–185. ACM, June 1994.

[32] V. Kathail, J. Hwang, W. Sun, Y. Chobe, T. Shui, and J. Carrillo. SDSoC: A higher-level programming environment for Zynq SoC and Ultrascale+ MPSoC. In *Proceedings of the 2016 ACM/SIGDA 24th International Symposium on Field Programmable Gate Arrays*, pages 4–4, Feb. 2016.

[33] S. Kulkarni and J. Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. *ACM SIGPLAN Notices*, 47(10):147–162, 2012.

[34] S. Kurra, N. K. Singh, and P. R. Panda. The impact of loop unrolling on controller delay in high level synthesis. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 391–396. EDA Consortium, 2007.

[35] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization*, pages 75–88, Mar. 2004.

[36] H.-Y. Liu and L. P. Carloni. On learning-based methods for design-space exploration with high-level synthesis. In *Proceedings of the 50th Design Automation Conference*, pages 1–6. IEEE, June 2013.

[37] X. Liu, Y. Chen, T. Nguyen, S. Gurumani, K. Rupnow, and D. Chen. High level synthesis of complex applications: An h. 264 video decoder. In *Proceedings of the 2016 ACM/SIGDA 24th International Symposium on Field Programmable Gate Arrays*, pages 224–233, Feb. 2016.

[38] G. Mariani, G. Palermo, V. Zaccaria, and C. Silvano. Oscar: An optimization methodology exploiting spatial correlation in multicore design spaces. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(5):740–753, 2012.

[39] K. Martin, C. Wolinski, K. Kuchcinski, A. Floch, and F. Charot. Constraint programming approach to reconfigurable processor extension generation and application compilation. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 5(2):10, 2012.

[40] W. Meeus and D. Stroobandt. Automating data reuse in high-level synthesis. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 1–4, Mar. 2014.

[41] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt. An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems*, 16(3):31–51, Sept. 2012.

[42] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *AIMSA*, volume 2, pages 41–50. Springer, 2002.

[43] S. Nouri, J. Rettkowski, D. Göhringer, and J. Nurmi. HW/SW co-design of an IEEE 802.11 a/g receiver on Xilinx Zynq SoC using high-level synthesis. In *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, pages 1–6. ACM, June 2017.

[44] J. Oppermann and A. Koch. Detecting kernels suitable for C-based high-level hardware synthesis. In *Smart World Congress*, pages 1157–1164. IEEE, July 2016.

[45] B. Ozisikyilmaz, G. Memik, and A. Choudhary. Efficient System Design Space Exploration Using Machine Learning Techniques. In *Proceedings of the 45th Design Automation Conference*, pages 966–969. ACM, June 2008.

[46] G. Palermo, C. Silvano, and V. Zaccaria. ReSPIR: a Response Surface-Based Pareto Iterative Refinement for Application-Specific Design Space Exploration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(12):1816–1829, Nov 2009.

[47] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[48] C. Pilato and F. Ferrandi. Bambu: A free framework for the high level synthesis of complex applications, Mar. 2012.

[49] N. Pothineni, A. Kumar, and K. Paul. Application specific datapath extension with distributed I/O functional units. In *Proceedings of the 20th International Conference on VLSI Design*, pages 551–558, Bangalore, India, Jan. 2007.

[50] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the 2013 ACM/SIGDA 21st International Symposium on Field Programmable Gate Arrays*, pages 29–38, Feb. 2013.

[51] L. Pozzi, K. Atasu, and P. Ienne. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(7):1209–29, July 2006.

[52] L. Pozzi and P. Ienne. Exploiting pipelining to relax register-file port constraints of instruction-set extensions. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 2–10, San Francisco, CA, Sept. 2005.

[53] J. Reddington, G. Gutin, A. Johnstone, E. Scott, and A. Yeo. Better than optimal: Fast identification of custom instruction candidates. In *Proceedings of the 12th IEEE International Conference on Computational Science and Engineering*, pages 17–24, 2009.

[54] B. C. Schafer and K. Wakabayashi. Divide and conquer high-level synthesis design space exploration. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 17(3):29:1–29:19, June 2012.

[55] R. R. Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59, 1997.

[56] M. Schmid, O. Reiche, F. Hannig, and J. Teich. Loop coarsening in C-based high-level synthesis. In *Proceedings of the 26th International Conference on Application-specific Systems, Architectures and Processors*, pages 166–173, July 2015.

[57] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proceedings of the 41st Annual International Symposium on Computer Architecture*, pages 97–108. IEEE, July 2014.

[58] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks. Co-designing accelerators and SoC interfaces using gem5-aladdin. In *MICRO 49: Proceedings of the 46st Annual International Symposium on Microarchitecture*, pages 1–12, Oct. 2016.

[59] T. Simonite. Moore's law is dead. Now what? *MIT Technology Review, May*, 13:40–41, 2016.

[60] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, pages 123–134. IEEE, 2005.

[61] Synopsys. ARC processor cores. `www.synopsys.com/designware-ip/processor-solutions/arc-processors.html`, Dec. 2016.

[62] B. A. Syrowik, B. Fort, and S. D. Brown. Use of CPU performance counters for accelerator selection in HLS-generated CPU-accelerator systems. In *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, pages 1–6, June 2018.

[63] Xtensa customizable processors: http://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable.

[64] C. L. Tobias Grosser, Armin Groesslinger. Polly - Performing polyhedral optimizations on a low-level intermediate representation. In *Parallel Processing Letters*, Apr 2012.

[65] F. Valina. Implementing memory structures for video processing in the Vivado HLS tool. In *Xilinx*, Sept. 2012.

[66] A. K. Verma, P. Brisk, and P. Ienne. Rethinking custom ISE identification: A new processor-agnostic method. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 125–134, Salzburg, Austria, Oct. 2007.

[67] J. R. Villarreal, A. Park, W. A. Najjar, and R. Halstead. Designing modular hardware accelerators in C with ROCCC 2.0. In *Proceedings of the 18th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 127–134, 2010.

[68] Xilinx. Vivado high-level synthesis. `www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`, Mar. 2017.

[69] Xilinx. Xilinx all programmable SoC portfolio. `www.xilinx.com/products/silicon-devices/soc.html`, Mar. 2017.

[70] Xilinx embedded system tools reference manual, 2018.

[71] S. Xydis, G. Palermo, V. Zaccaria, and C. Silvano. A meta-model assisted coprocessor synthesis framework for compiler/architecture parameters customization. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 659–664, 2013.

[72] H. Yu and M. Leeser. Automatic sliding window operation optimisation for FPGA-based computing boards. In *Proceedings of the 14th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 76–88, April 2006.

[73] P. Yu and T. Mitra. Scalable custom instructions identification for instruction set extensible processors. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 69–78, Washington, DC, Sept. 2004.

[74] G. Zacharopoulos, G. Ansaloni, and L. Pozzi. Data reuse analysis for automated synthesis of custom instructions in sliding window applications. *HiPEAC IMPACT 2017 Seventh International Workshop on Polyhedral Compilation Techniques*, Jan. 2017.

[75] G. Zacharopoulos, A. Barbon, G. Ansaloni, and L. Pozzi. Machine learning approach for loop unrolling factor prediction in high level synthesis. *2018 IEEE International Conference on High Performance Computing & Simulation (HPCS)*, pages 91–97, 2018.

[76] G. Zacharopoulos, L. Ferretti, E. Giaquinta, G. Ansaloni, and L. Pozzi. RegionSeeker: Automatically identifying and selecting accelerators from application source code. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(4):741–754, Apr. 2019.

[77] M. Zuluaga, T. Kluter, P. Brisk, N. P. Topham, and P. Ienne. Introducing control-flow inclusion to support pipelining in custom instruction set extensions. In *Proceedings of the 7th Symposium on Application Specific Processors*, pages 114–121, 2009.

[78] M. Zuluaga, A. Krause, P. Milder, and M. Püschel. Smart design space sampling to predict Pareto-optimal solutions. In *ACM SIGPLAN Notices*, volume 47, pages 119–128, 2012.