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	PhD Program Director
Professor Laura Pozzi	**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 [36], 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 [15], 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 [17].

The breakdown of Dennard scaling [17], along with the seemingly inevitable end of Moore's law economic aspect [38], 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 [14]. 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

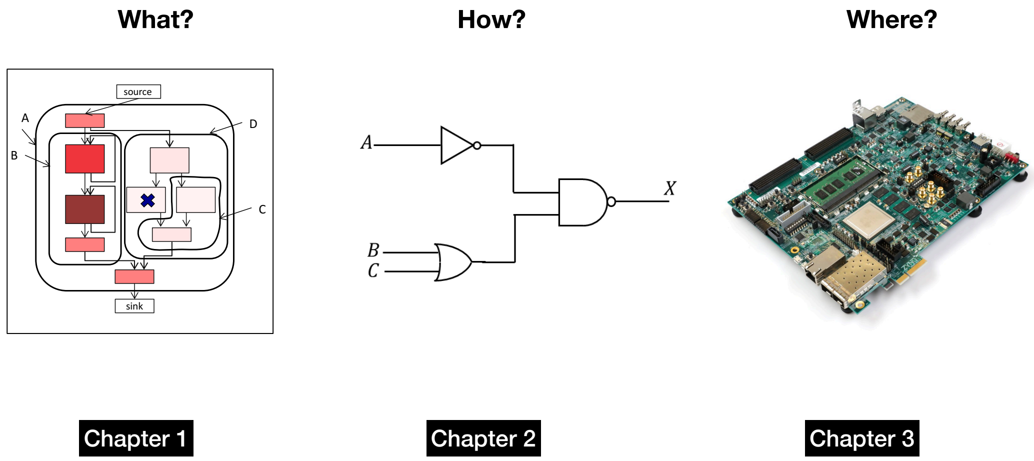

Figure 1. Overview of the research that has been conducted during my PhD and the respective chapters of the PhD thesis.

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 [29] 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 [49]. 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 [43], and a research HW accelerator simulator, such as Aladdin [37]. 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 [47]. Furthermore, for HW accelerators that contain loops, an optimal Loop Unrolling factor can be predicted for each of the included loops [48]. 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 profiling 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 the performance trend 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 HW 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

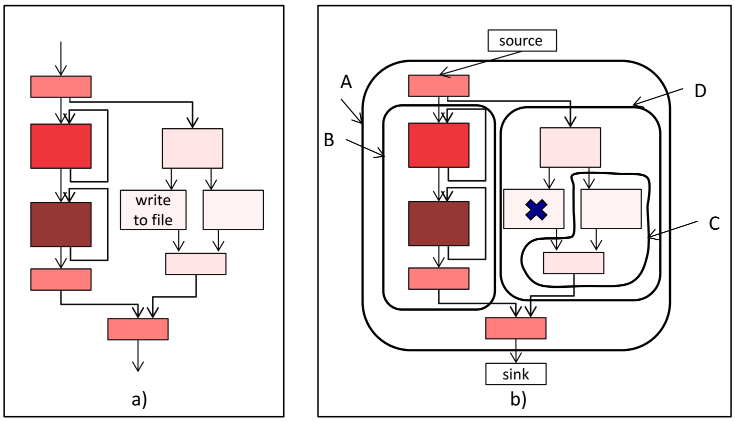

Figure 1.1. a) Example Control Flow Graph of a function, colour-coded with frequency of execution (the darker the basic block, the more frequent). b) B and C are Valid Subgraphs; A and D are not Valid Subgraphs because they contain a forbidden node. B is also a CFG region, because it has a single control 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 [1], [26], and under the name of *Simple Region* in an LLVM implementation [29] — 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, colour-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 [46], [33], [12], [35], [19] and [30], where the problem of identifying subgraphs under convexity, I/O constraint, and/or area is tackled; in [41] and [32] the I/O constraint is relaxed, to be regained via I/O serialization [34], [41], [5], [2]. In [13] 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 [18] is not limited by I/O in the selection process, but clusters MAXMISOs [3] 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 [50] 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 [6] 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 C code, as the methodology proposed in Section 1.3.

Finally, the authors of a recent work [1] 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 [24], 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 [26], 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 Processor 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 [10] and the ARC processor from Synopsys [39]. 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 [4]) and Xilinx (the Zynq SoCs [44]).

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 [11] and ROCCC [20], while commercial applications comprise the Vivado_HLS [43] suite from Xilinx (for FPGAs) and StratusHLS [9] 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 source 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 [25] 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 and Toolchain* [29]. 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 [40], 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 toolchain. 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

```
function RunOnFunction()
    Region_List = NULL
    RI = getRegionInfoAnalysis()
    for Region in Function do
        if RegionIsValid() then
            EvaluateRegion(Region)
            Region_List.Add(Region)
    return Region_List

/*Estimate Merit for Region*/
function EvaluateRegion(Region)
    for Basic Block in Region do
        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++.

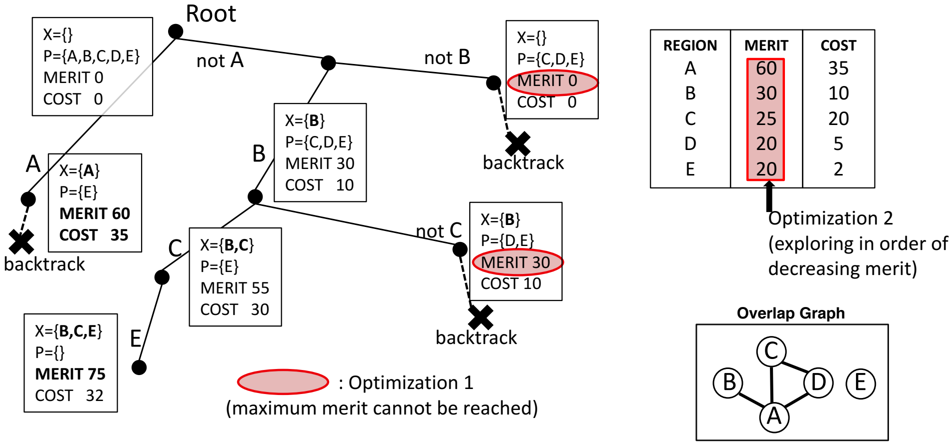

Figure 1.2. Tree exploration performed by exact, for the running example of Figure 1.1, and for a cost budget of 35.

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 [7], 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 [37]. 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 [25]. `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 [28] [27]. 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 [37]. 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 [7] 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 [49]. 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.

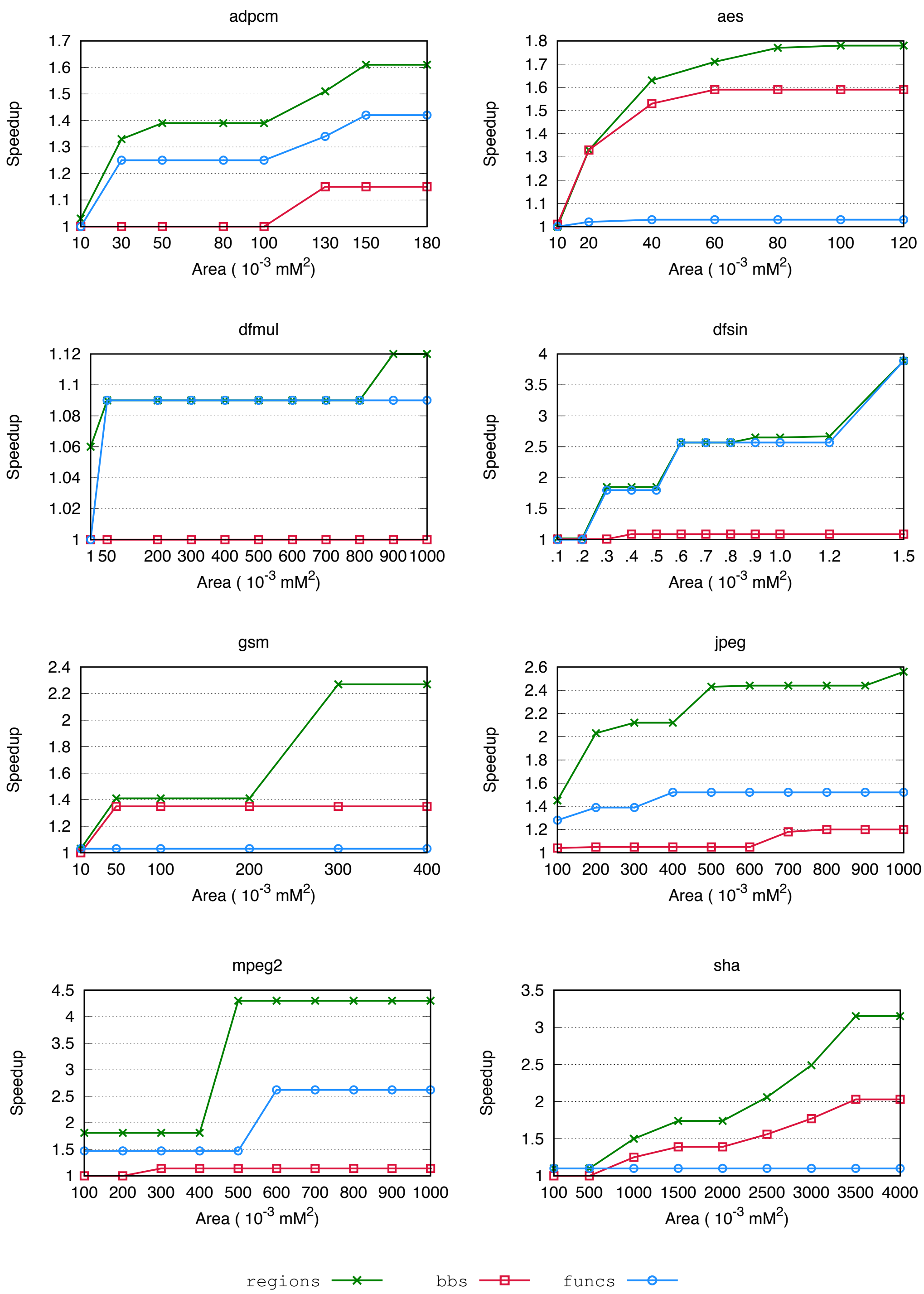

Figure 1.3. Comparison of speedups obtained on eight CHStone benchmarks by selecting regions, only basic blocks and only functions, varying the area constraint, using Aladdin and Gem5 for Speedup and Area evaluation.

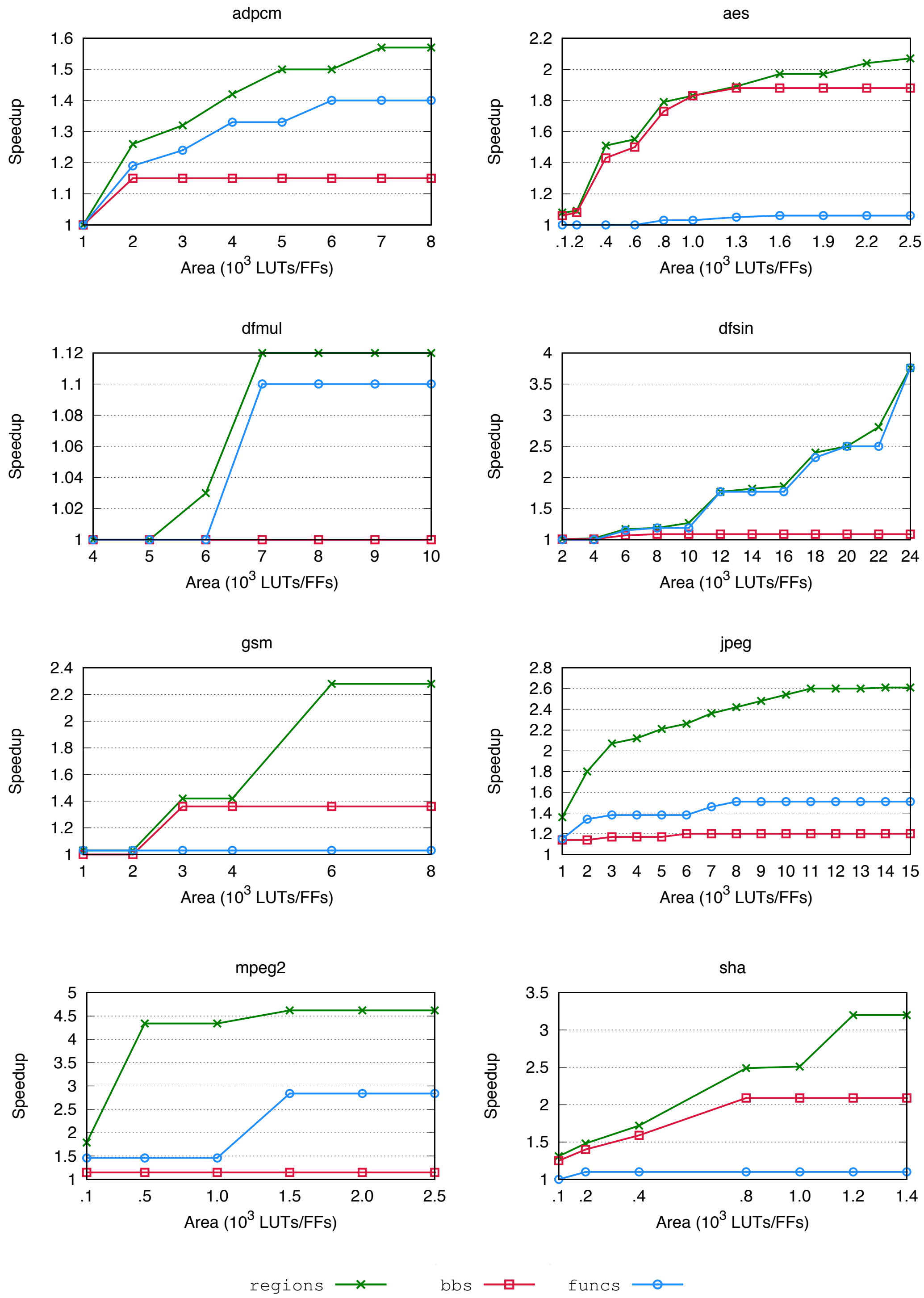

Figure 1.4. Comparison of speedups obtained on eight CHStone benchmarks by selecting regions, only basic blocks and only functions, varying the area constraint, using Vivado_HLS and Gem5 for Speedup and Area evaluation.

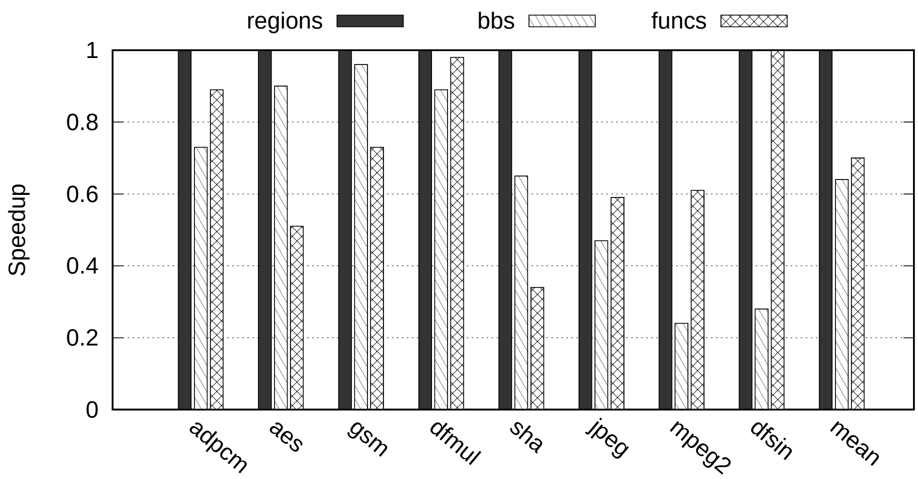

Figure 1.5. Normalized Speedup of RegionSeeker with respect to function and basic block selection, considering, for each benchmark, a fixed area constraint. Synthesis performed with Vivado_HLS.

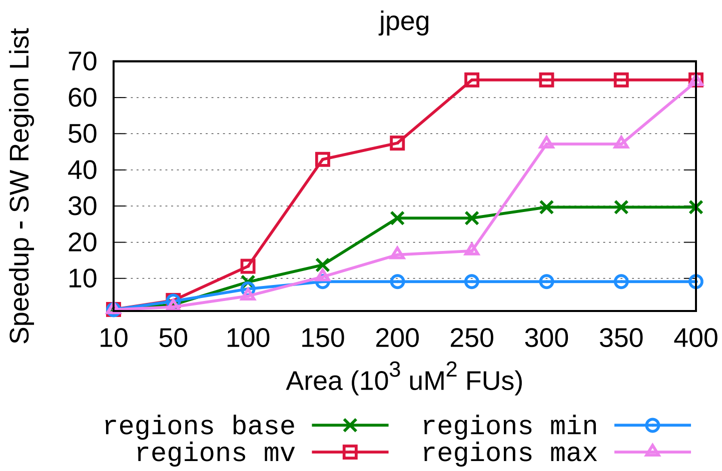

Figure 1.6. Comparison of speedup obtained on jpeg benchmark, over the SW time of the equivalent kernels (regions), varying the area constraint, using Aladdin and gem5 for Speedup and Area evaluation. Four approaches are compared: The min where the regions with least amount of area are selected, the 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

Optimizations applied to HW 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 [43], 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

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 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 would be 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.

2.1.2 Related Work

In the domain of identifying automatically accelerators, research has so far focused mostly on accelerating data-flow [19] [22], not taking into equal account the potential for optimization by memory accesses. Exceptions are provided by papers [8] [23] 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 [21] generated by the ROCCC compiler [42] allow for automatic detection of data reuse opportunities, but cannot be interfaced with interconnects of varying width. The methodology described in [31] employs reuse buffers spanning multiple frame columns, which pose a significant area overhead. Both [21] and [31] 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 [16], requires a great deal of Hardware resources as well, as it requires the storage of

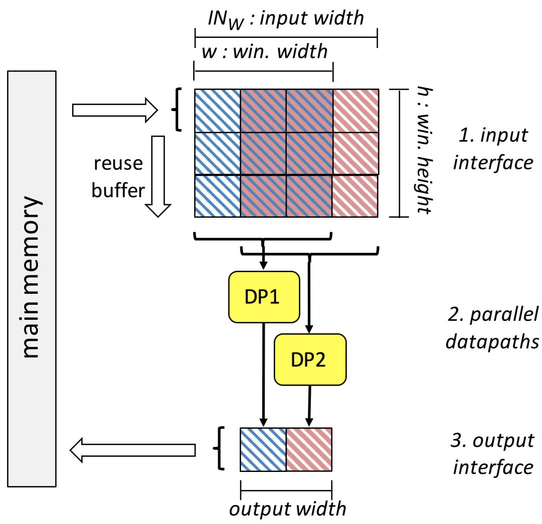

Figure 2.1. Data Reuse of memory accesses (66.6%) across consecutive iterations of a loop, in a sliding window application.

large parts of a frame being processed inside the custom hardware. In [45], 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 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 2.1. 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 [40].

The analysis pass iterates over regions of the application functions and identifies Static Control Parts (SCoPs). The SCoPs are subgraphs 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.

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 2.1. Multiple datapaths translate to more parallel windows executing and, hence, increased demand in area resources.

The comparison of our approach 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.

Execution time, as seen in Figure **??**, 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 our framework, 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 equivalent solution with such low execution time*. Figure **??** 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 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 [47].

2.2 Machine Learning Approach for Loop Unrolling Prediction

Chapter 3

System Aware Accelerators Identification — Speedup gain and Energy Saving

3.1 AccelSeeker

3.2 EnergySeeker

Conclusions

Bibliography

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

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

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

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

[41] 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.

[42] 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.

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

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

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.