Constraint-aware configurable system-on-chip design for embedded computing

Alok Prakash

2014

Alok Prakash. (2014). Constraint-aware configurable system-on-chip design for embedded computing. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/61678>

<https://doi.org/10.32657/10356/61678>

CONSTRAINT-AWARE CONFIGURABLE SYSTEM-ON-CHIP DESIGN FOR EMBEDDED COMPUTING

ALOK PRAKASH

School of Computer Engineering
Nanyang Technological University

A thesis submitted to the Nanyang Technological University
in fulfillment of the requirement for the degree of

Doctor of Philosophy

2014

Acknowledgments

I would like to take this opportunity to thank many people who have directly or indirectly helped me during the course of this research. First and foremost, I want to express my heartfelt gratitude to my supervisor and mentor Prof Thambipillai Srikanthan. His guidance during the course of this project has been invaluable. Despite being entrusted with many responsibilities, he has always found time to discuss the directions for this project with me. His patience and understanding in this regard is truly admirable.

I would like to thank Dr Christopher Clarke for helping me through a significant portion of this research. His advice and insights truly helped me at all stages of this thesis. I always learnt something important during our frequent and often long video chats. The time spent at University of Bath, under his guidance, was certainly one of the most productive and enjoyable periods in the past few years.

The discussions with Dr Siew Kei Lam were also extremely helpful throughout this research to bring about clarity and sanity.

The administrative support rendered by Jeremiah and Marilyn was the best I could have asked for. They always rendered help whenever I needed any equipment or administrative matters to be sorted out. They are truly an integral part of the research lab and their presence is absolutely necessary for the day-to-day functioning.

I would also like to thank my friends Dr Amit, Dr Bharath, Dr Ashish, Santanu, Fauzi, Neha, Dhinesh, Sharad and so many others; without them I would not have come this far. Last but certainly not the least, this thesis is dedicated to my mother who gave me life not once but twice and my father whose unwavering support is only second to his unfaltering trust in me.

Contents

Acknowledgments	i
List of Figures	vi
List of Tables	viii
Abstract	ix
1 Introduction	1
1.1 Motivation	1
1.2 Aim Of This Research	4
1.3 Main Contributions Of This Research	5
1.4 Organization Of The Thesis	7
1.5 Publications	9
2 Literature Review	11
2.1 Embedded Computing Architectures	12
2.1.1 Homogeneous Architectures	13
2.1.2 Heterogeneous Architectures	15
2.1.3 ASIC Designs	20
2.1.4 Reconfigurable Architectures	22
2.1.5 Configurable System on Chip (CSoC)	23
2.2 Configurable System-on-Chip Customization	28

2.2.1	Processor Customization	28
2.2.2	Hardware Accelerators	33
2.2.3	Memory Systems In CSoC	35
2.2.4	Methodologies For CSoC Design	37
2.3	Summary	41
3	Soft-Core Processor Customization Through Instruction Subsetting	45
3.1	Introduction	45
3.2	Related Work	46
3.3	Proposed Framework	49
3.3.1	Pre-Characterization Phase	50
3.3.2	Architecture-Independent Phase	51
3.3.3	Application-Aware Design Exploration And Customization	51
3.4	Experiments And Results	53
3.5	Summary	61
4	Architecture-aware Custom Instruction Generation	63
4.1	Introduction	63
4.2	Existing Custom Instruction Generation Methodologies	65
4.3	Proposed Architecture-Aware Custom Instruction Generation Method- ology	69
4.3.1	Fine-Grained And Coarse-Grained Custom Instructions	73
4.4	Fine-Grained Custom Instruction Enumeration	74
4.4.1	Enumeration Algorithm	77
4.5	Fine-grained Custom Instruction Selection	84
4.6	Experiments And Results	86
4.7	Coarse-Grained Custom Instructions	96

4.7.1	Motivation For Coarse-Grained Custom Instructions	97
4.8	Coarse-Grained Custom Instruction Enumeration	99
4.9	Coarse-Grained Custom Instruction Selection	103
4.10	Experiments And Results	106
4.11	Summary	113
5	Incorporating Data-Dependent Blocks Into Hardware Accelerators	118
5.1	Introduction	118
5.2	Related Work	121
5.3	FPGA Resource-Aware Enumeration Of Candidate Hardware Blocks .	123
5.3.1	Considerations During Data-Dependency Aware Selection Of Candidate Hardware Blocks	127
5.3.2	Estimating Penalty When Data Resides In Local Memory	130
5.4	Data-Dependency Aware Selection Of Hardware Accelerators	136
5.5	Implementation Of Proposed Architecture Model On Altera's NIOS II Platform	140
5.6	Summary	144
6	Memory-Aware Selection Of Hardware Accelerators	147
6.1	Introduction	147
6.2	Related work	149
6.3	Selection Heuristic	150
6.4	Experiments And Results	161
6.5	Two-Pass Based Heuristic Search	168
6.5.1	Motivation For Two-Pass Based Heuristic Search	168
6.5.2	Two-Pass Based Heuristic Search Algorithm	171
6.6	Iterative Technique For Improved Performance	177

6.6.1	Multi-thread Based Heuristic Search Algorithm	178
6.6.2	Determining Optimum Number Of Threads	184
6.7	Summary	187
7	Conclusion And Future Research	189
7.1	Conclusion	189
7.2	Recommendations for Future research	193
	References	198
	Appendix	221

List of Figures

1.1	Hardware and Software Design Gap. Source - ITRS DESIGN-2011	3
2.1	Example of hybrid-core computing	19
2.2	ASIC vs. FPGA - approximate development time	21
3.1	Design Exploration Framework.	49
3.2	Performance vs. Area Curves for Benchmark Applications.	61
4.1	A nonconvex graph with unfeasible scheduling	66
4.2	Proposed Instruction Set Extension Methodology	70
4.3	A data flow graph example	71
4.4	A pattern candidate with 3 Fine Grained Custom Instructions (FGCI)	77
4.5	Implementation of <i>add</i> operation in a logic element	78
4.6	Templates in a portion of a DFG and corresponding conflict graph	85
4.7	Reduction in number of patterns enumerated	89
4.8	Reduction in runtime of the Enumeration Phase	89
4.9	Reduction in runtime of the Selection Phase	90
4.10	Performance comparison - FGCI vs. Traditional CI	94
4.11	An example of a large pattern with low critical path.	98
4.12	Performance comparison - FGCI and CGCI vs. Traditional CI	112

4.13 Average speedup - CGCs vs. Traditional CIs	113
5.1 Identification Phase of Custom Hardware Blocks	124
5.2 Preferred code segments for hardware implementation.	128
5.3 Preferred code segments for hardware implementation.	128
5.4 Example Architecture model using Altera NIOS II.	131
5.5 Code snippet from SHA application	137
5.6 Execution time of the SHA application with varying Accelerators	143
6.1 Flowchart for the proposed selection algorithm.	155
6.2 Implementable/Unimplementable Blocks Example	156
6.3 Execution time of the SHA application with right set of Accelerators	160
6.4 Performance comparison (SHA App) - Exhaustive vs. Heuristic algorithms . . .	160
6.5 Runtime comparison (Synthetic Application Set) - First Pass	164
6.6 Performance comparison (Synthetic Application Set) - First Pass	167
6.7 Performance comparison (Sample Application) - First Pass	169
6.8 Selection thread (Sample Application) - First Pass	171
6.9 Performance comparison (Sample Application) - Second Pass	173
6.10 Performance comparison (Synthetic Application Set) - Second Pass	176
6.11 Selection threads of the sample application	179
6.12 Performance comparison (Synthetic Application Set) - Multi-thread Approach .	182
6.13 Runtime comparison (30-Blocks Application Set) - Multi-thread Approach . . .	184
6.14 Performance comparison (30-Blocks Application Set) - Multi-thread Approach	185

List of Tables

3.1	Optional Instructions' Count.	54
4.1	Enumerated Patterns using Traditional & Proposed approaches	72
4.2	Execution Time Saved By Using Proposed And Traditional Methods	96
4.3	An example for CGCI Selection Step	105
4.4	Comparing Custom Instructions generated by different methodologies	113
5.1	Frequently executed basic blocks in SHA app	137
5.2	Resource Utilization for DE2 board with various Hardware Blocks	144
6.1	Most frequently executed basic blocks in SHA application	154
6.2	Hot blocks in SHA application and their metrics	157
6.3	Selected basic blocks in each iteration of SHA application	158
6.4	Sample application sets	162
6.5	Hot blocks in a sample application and their metrics	169
6.6	Selected basic blocks in each iteration of Sample application	170
6.7	Application blocks tested per iteration (Sample Application)	172
6.8	Performance achieved after first iteration (Sample Application) - First Pass . .	179
6.9	Runtime comparison (30-Blocks Application Set) - Multi-thread Approach vs. Exhaustive Approach	186

Abstract

Field Programmable Gate Arrays (FPGAs) are rapidly becoming a popular alternative to ASICs as they continue to increase in capacity, functionality and performance. At the same time, FPGA developers are faced with the challenges of meeting increasingly aggressive design constraints such as power, delay and area costs without violating shorter Time-to-Market (TTM) pressures and lower Non-Recurring Engineering (NRE) costs for embedded systems development. In this research, efficient techniques have been proposed for processor sub-setting and customization as well as the rapid generation of application-specific hardware accelerators in order to meet the design constraints of configurable System-on-Chip (SoC) platforms.

A processor-agnostic technique has been devised for sub-setting soft-core processors by relying on LLVM compiler generated front-end application output. The proposed approach has resulted in a systematic method for the application-aware sub-setting of the micro-architecture subsystems such as hardware multipliers and floating point units of a soft-core processor. Evaluations based on widely used benchmarks show that the proposed method can be deployed to reliably subset soft core processors at high-speed without compromising compute performance.

A technique for the architecture-aware enumeration of custom instructions has been proposed next to identify area-efficient custom instructions by employing FPGA resource-aware pruning of the search space. Experimental results based on applications from widely-used benchmark suites confirm that deploying custom instructions identified in this way can improve compute performance by up to 65%. The instruction level parallelism (ILP) has also been exploited to further improve the compute performance by identifying profitable coarse-grained custom instructions. It has been demonstrated that the custom instructions using the proposed method can accelerate computations by up to 39% when compared to a base processor only implementation.

Unlike traditional custom instruction generation methods that are incapable of incorporating memory-dependent basic blocks, a novel technique for accelerating memory-dependent basic blocks has been proposed. A detailed data dependency analysis based on pre-defined memory allocation in an application has been developed to guarantee the identification of

profitable basic blocks for hardware acceleration. The profitability of a code segment for hardware acceleration is determined by using a mathematical model to represent the overheads associated with the placement of data in both the local and main memory subsystems. The proposed approach for hardware acceleration eliminates the need for Direct Memory Access (DMA) transfers or cache-coherence protocols.

A scalable technique for the automatic selection of profitable basic blocks for hardware acceleration has been devised in order to overcome the time complexity of the search space. It relies on a heuristic approach to significantly reduce the search space, thereby resulting in a high-speed technique to recommend the profitable basic blocks with mutual data-dependency for hardware acceleration. It has been shown that, while the runtime and complexity of an exhaustive selection approach increases exponentially, the proposed heuristic search-based method scales almost linearly with increasing number of candidate hardware blocks. Moreover, extensive tests confirm that the compute performance of the hardware accelerators identified with the proposed heuristic method often matches those achieved using exhaustive search results, with deviations of mere 5 to 10% at the very most. It is noteworthy that the proposed heuristic search algorithm is independent of memory subsystem configurations.

Finally, the design methodologies presented in this thesis can be integrated to realize a systematic framework for the automatic generation of area-time efficient configurable SoC by leveraging on processor sub-setting, architecture-aware custom instruction generation and data-dependency-aware hardware acceleration.

Introduction

1.1 Motivation

The growth in adoption of embedded systems today is rivaled only by its increasing complexity and performance demands. According to market research report by BCC Research in 2012 [1], the worldwide capital involved in the embedded domain increased from \$101.6 billion in 2009 to \$113 billion in 2010 and is further forecasted to grow to \$158.6 billion by 2015. We use numerous embedded devices throughout the day, often being oblivious to the underlying technology or even their presence around us. The increasing sophistication of embedded applications running on such devices can be traced directly to the increasing consumer demands and expectations, which in turn also demands better and more efficient processing units. Traditionally, Application-specific Integrated Circuit (ASIC) designs have been preferred in large volume market where the high Non-Recurring Engineering (NRE) costs can be effectively amortized over the large quantities of sold devices. However, the NRE costs of ASICs have been dramatically increasing with each process node. In less than 3 years the NRE cost has jumped from \$85 million for a 45nm design to over \$170 million for 28nm [2]. Extrapolating from these values, it has been estimated that the NRE cost for 14nm at \$340 million and 7nm at \$680 million respectively [2]. While such

high costs can be justified for ultra-high volume products such as the modern smart-phone chips, they pose a significant problem in low to mid volume products. Moreover, in these low and mid-volume products, there is a growing demand for higher performance, additional features and product differentiation. On the other hand, the extremely competitive market dictates that innovation, design, implementation, testing and mass production should be done in the least possible time. At the same time the rapid technological advancements reduces the product life cycles, thereby making a low Time-to-Market (TTM) a key priority in the embedded market [3]. This puts immense pressure on the designers to produce faster and flexible, yet high performance systems. However, while the high NRE cost and TTM pressure coupled with the market uncertainties make traditional ASIC based designs less attractive, traditional General Purpose Processors (GPPs), although extremely flexible in terms of programmability, are unable to meet the increasingly strict functional constraints (performance, area, power, etc.) of modern embedded systems.

Reconfigurable architectures such as Field Programmable Gate Arrays (FPGAs) have emerged as a viable alternative for system designers in order to fulfill the various conflicting requirements. Such architectures have evolved to become a promising candidate to bridge the gap between performance achieved by an ASIC and flexibility/programmability of a GPP. FPGAs have undergone significant growth over the last two decades, in terms of its size and capabilities. While initially, they were mostly used for simple implementations such as glue logic, their utility has gradually moved upwards from prototyping to implementing a complete System-on-Chip on modern FPGAs, generally known as Configurable System-on-Chip (CSoC). However, this comes at the cost of lower area efficiency when compared to ASICs. In order to address this weakness, system designers are motivated to generate application-specific systems on reconfigurable hardware by leveraging on the extensive customization opportunities offered by these architectures. Modern FPGAs also contain numerous

Figure 1.1: Hardware and Software Design Gaps versus Time. Source - ITRS DESIGN-2011.¹

hard blocks such as Digital Signal Processor (DSP), Block Random Access Memory (BRAM), etc. to achieve better performance area efficiency. Even so, the large design space that needs to be explored in order to arrive at the best application-specific configuration necessitates efficient design methodologies for design-space exploration and efficient customization.

One of the major challenges identified by the International Technology Roadmap for Semiconductors (ITRS) in 2011 [3] as well as in 2012 [4], was the lack of system level design and verification tools. In its 2011 report [3] on design technologies, the ITRS highlighted this design productivity gap created by the lack of appropriate design tools as shown in Figure 1.1. According to this figure the hardware design productivity is unable to match up with the technological developments made in accordance with the Moore's law. This has inspired numerous researchers in the past to create methodologies for system-level designs and exploration. Yet, as depicted in Figure 1.1, a significant amount of research still needs to be done to close this gap.

¹This figure shows the demand for software which is currently doubling every 10 months, the capability of technology which is currently doubling every 36 months, as well as the productivity of hardware and software design. Software productivity is defined as Lines of Code (LOCs) over time. The productivity for hardware-dependent software is far behind and doubles only every 5 years. The red arrow summarizes the new design gap that includes both hardware and software [3] [5].

The CSoCs based on modern FPGAs also face similar challenges due to lack of design methodologies. The need for customization in future embedded systems will continue to rise for multiple reasons such as more area-time efficient systems, product differentiation, etc. At the same time, the shrinking Time-to-Market windows will also demand high degree of flexibility while the market uncertainties will decree a certain level of future proofing. Reconfigurable architectures are well placed to address challenges such as the shrinking TTM, increasing NRE cost, etc. However, these require efficient techniques and methodologies to generate highly customized designs in order to meet stringent functional constraints such as performance, area, power, etc. This research focuses on developing such techniques and methodologies for exploring the large design space offered by CSoC designs and generate highly customized components for such a system that meets various design constraints such as performance and area.

1.2 Aim Of This Research

The main objective of this research is to develop novel techniques and design methodologies for customizing the various components of a CSoC in order to generate constraint-aware application-specific system-on-chip. The target CSoC consists of a general purpose processor, application-specific hardware accelerators and a suitable memory hierarchy. Each of these components must be customized to generate a system with high efficiency in terms of performance, area, etc. The proposed methodologies should incorporate fast and accurate estimation strategies to facilitate rapid and effective design space exploration of the system.

1.3 Main Contributions Of This Research

The major contributions of this work can be listed as follows:

- (i) *A rapid estimation methodology for processor customization through instruction subsetting:* A rapid estimation methodology has been proposed for application-specific customization of a soft core processor through Instruction Subsetting. A soft core processor can be customized by enabling or disabling optional hardware components in the base processor using software tools. A comprehensive estimation methodology was developed to rapidly analyze complex applications and evaluate the performance-area metrics when various optional hardware components are introduced, in order to identify the best configuration for the soft core processor. The proposed estimation method can be performed in the order of seconds, compared to hours as required in existing methodologies. Once a suitable configuration has been identified, Instruction Subsetting can be performed to realize a custom processor with reduced implementation area without sacrificing performance. This work has been published in [6].
- (ii) *A novel technique for architecture-aware Custom Instructions Enumeration and Selection:* Instruction Set Extension (ISE) through the addition of application-specific custom instructions provides the first level of acceleration in an application. This thesis proposes architecture-aware enumeration and selection methodologies that consider the Lookup Table (LUT) structure of the target FPGA device as additional constraints to select profitable as well as area efficient custom instructions. The proposed methodologies result in faster runtime for the ISE step while still selecting custom instructions that can be mapped efficiently on the target FPGA to achieve maximum performance in the given FPGA area. The contributions in this work have been published in [7], [8] and [9].

- (iii) *A CSoC architecture model:* A Configurable System-on-Chip architecture model has been introduced which, unlike existing models, obviates using DMA or additional cache-coherence protocols while using hardware accelerators with local memory blocks storing statically allocated variables. In order to understand the implications of introducing local memory blocks, the resulting data dependencies across the software and various candidate hardware blocks have been thoroughly investigated with the help of the proposed CSoC architecture model. A detailed case study with a real-world application has also been presented to highlight the resulting push-pull effect of using local memory blocks in hardware accelerators having data dependencies with the software sections. Parts of this work have been published in [10].
- (iv) *A performance estimation model:* In order to account for the overhead of accelerating application code segments with interleaved data-dependencies, system designers need to account for such dependencies, apart from the traditional metrics such as hardware area required to implement a code segment, time saved by acceleration, etc. during the selection step. This research derives and proposes a novel mathematical model to estimate the overall performance of candidate hardware accelerators while considering their data-dependencies using the proposed CSoC architecture model. This contribution has been published in [11].
- (v) *A rapid heuristic search algorithm to select memory-aware hardware accelerators:* In order to achieve higher performance, a designer invariably looks to accelerate larger sections of the application code in hardware. However, the interleaved data dependencies across different sections of the applications makes it difficult to rapidly and reliably select the right set of candidates for hardware acceleration. At the same time the large design space for selecting such accelerators

prohibits manual or exhaustive selection algorithms. Rapid heuristic search algorithms have been proposed in this thesis which have been proven to be rapid as well as accurate. In order to overcome local minima, which results in the selection of non-optimal solutions, various enhancements have been proposed to the original heuristic search algorithm. It has been shown that with these enhancements, the accuracy of the results improve significantly while still being orders of magnitude faster than an exhaustive search algorithm.

1.4 Organization Of The Thesis

- Chapter 2 starts with a detailed survey on various computing architectures available to embedded designers. After comparing the pros and cons of the various options available to embedded designers, the chapter focuses on CSoC and describes the existing design methodologies to generate such a computing architecture. Various research work done in the area of selecting and customizing the individual component of CSoC has also been described in this chapter. The chapter is concluded with specific objectives and motivations for the rest of this thesis.
- Chapter 3 describes an estimation methodology to customize a soft-core processor by using Instruction Subsetting in order to achieve maximum performance-area efficiency. Unlike the existing methods that are based on lengthy synthesis runs, the proposed methodology relies upon rapid estimation techniques while still being extremely reliable.
- Chapter 4 presents our novel architecture-aware custom instructions generation and selection methodologies for the next level of customization. The proposed ideas lead to significant improvement over the existing methodologies for

instruction set extension, both in terms of the performance achieved by the selected Custom Instructions as well as the tool runtime of the extension process.

- In order to achieve further customization, Chapter 5 discusses the considerations during acceleration of large blocks in an application with interleaved data dependencies. A CSoc architecture model is proposed that is capable of incorporating custom instructions as well as hardware blocks with local memory elements for storing statically allocated variables without the need for Direct Memory Access (DMA) or Cache-coherence protocols. A mathematical model has been derived to estimate the communication overhead incurred by the processor to access the variables stored in local memory blocks implemented alongside a hardware accelerator. The advantage and overheads associated with including local memory blocks in hardware accelerators have been extensively discussed in this chapter. The efficacy of the proposed architecture model is proven by implementing and testing it on a commercial FPGA platform.
- Chapter 6 utilizes the mathematical model derived in the previous chapter to account for the various communication penalties between the hardware and software code sections and select the right set of hardware accelerators in an application for maximum performance-area efficiency. A heuristic approach has been proposed that can closely match the performance of an exhaustive search algorithm, but taking only a fraction of time to select the right set of hardware accelerators while considering their memory dependencies. Additional enhancements have also been proposed to improve the performance of the heuristic search algorithm and offer a trade-off between the runtime of the selection step and the performance achieved by the recommended hardware blocks.
- Chapter 7 concludes this thesis by summarizing the important contributions made in this research followed by recommendations for future extension from this work.

1.5 Publications

The work presented in this thesis has been communicated in international journals and conferences as follows:

International Refereed Journal Papers

- [J-1] Alok Prakash, Siew-Kei Lam, Christopher T. Clarke, and Thambipillai Srikanthan. "FPGA-aware techniques for rapid generation of profitable custom instructions." *Microprocessors and Microsystems*, 37(3) 259 - 269, 2013.

International Refereed Conference Papers

- [C-1] Alok Prakash, Siew-Kei Lam, Christopher T. Clarke, and Thambipillai Srikanthan. "Modeling the Communication overhead of using local memory blocks." *IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, pages 31-34, 2013.
- [C-2] Alok Prakash, Christopher T. Clarke, and Thambipillai Srikanthan. "Custom instructions with local memory elements without expensive DMA transfers." In *22nd IEEE International Conference on Field Programmable Logic (FPL)*, pages 647 - 650, August 2012.
- [C-3] Alok Prakash, Siew-Kei Lam, Christopher T. Clarke, and Thambipillai Srikanthan. "Instruction set customization for area-constrained FPGA designs." In *IEEE International SOC Conference (SOCC)*, pages 329 - 334, September 2011.
- [C-4] Alok Prakash, Siew Kei Lam, Amit Kumar Singh and Thambipillai Srikanthan. "Architecture-Aware Custom Instruction Generation for Reconfigurable Processors", In *International Symposium of Applied Reconfigurable Computing (ARC)*, Bangkok, March 2010.

[C-5] Alok Prakash, Siew Kei Lam, Amit Kumar Singh and Thambipillai Srikanthan. "Rapid Design Exploration Framework for Application-Aware Customization of Soft Core Processors", In IEEE International Conference on Field Programmable Logic (FPL), Prague, September 2009.

Literature Review

Embedded system designers are consistently facing the ever-increasing Non-Recurring Engineering (NRE) costs and the Time-to-Market (TTM) constraints in addition to the difficulties of shorter product life cycles. At the same time the immense growth in silicon, as predicted by the Moore's law, has created a scenario where the hardware capabilities are immense yet the designers are unable to harness their full potential [3], [4]. This has led to a "Design Productivity Gap" that has proved extremely difficult to fill due to the increasing complexity of programming such devices. These contradictory set of constraints, coupled with the demand for higher performance at lower power consumption, has fueled the research for more efficient usage of the available silicon through both *novel architectures* and *design methodologies* for rapid product development and deployment.

Single chip solutions have become commonplace, enabled by more sophisticated process technologies, for tighter integration, reduced manufacturing cost, and higher power performance efficiency [12]. At the same time, while homogeneous architectures are preferred for their flexibility and ease of programmability [13], heterogeneous architectures are useful to achieve better performance and power efficiency as well as product differentiation [14], [15], [16], [17]. Similarly Application-specific Integrated Circuit (ASIC) designs have been a de facto choice for reasons such as

better performance, low power, etc. especially in large volume products. However, their increasing NRE and TTM make them less amenable due to market uncertainties and shorter product life cycles in modern embedded systems. Reconfigurable architectures are well poised to cater for some of these design requirements, such as flexibility, future proofing etc. However, they suffer from lower power-performance efficiency as opposed to ASIC designs and therefore require extensive customization to meet the design constraints. Hence, rapid and reliable tools and design methodologies are essential to generate custom designs based on reconfigurable hardware in order to meet the various design constraints.

This chapter begins with a discussion on the various computing architectures available to embedded designers, starting from homogeneous (same type of processing elements) and heterogeneous (different types of processing elements) architectures, and elaborates their advantages and shortcomings. Next, a discussion on ASIC and Reconfigurable designs is presented to discuss their appropriate use-cases. An extensive review of important research efforts made in processor customization has been presented, followed by a detailed survey of methodologies to select hardware accelerators at multiple levels of granularity. Existing design methodologies to generate a customized reconfigurable architecture have also been reviewed followed by a discussion on relevant commercial tools. This chapter is concluded with the specific shortcomings of the existing work and the motivations for this thesis.

2.1 Embedded Computing Architectures

The proliferation and competition in the embedded market has forced embedded system designers to try various computing architectures for modern embedded systems. These computing architectures range from ASICs to General Purpose Processors (GPPs), and from reconfigurable Field Programmable Gate Arrays (FPGAs) to high

performance Graphics Processing Units (GPUs). Recently, designers have also proposed homogeneous ‘sea of compute’ architectures such as the Tiler processors [13] for high performance embedded computing.

Embedded architectures can also be classified as homogeneous or heterogeneous, based on the types of computing cores in the system. A homogeneous computing architecture, as the name suggests, generally contains multiple instances of a single type of processing core. On the other hand, a heterogeneous architecture consists of varying types of cores, including a general purpose processor and other specialized components for specific tasks such as Digital Signal Processors (DSPs), FPGAs etc. There is a definite trade-off between design effort, achievable performance, power efficiency etc. provided by each of these architectures. The selection of a particular architecture in any design is based on both functional and non-functional constraints such as the system constraints, market/business requirements, designer’s expertise, etc.

2.1.1 Homogeneous Architectures

A homogeneous computing architecture, as mentioned earlier, consists of a single type of processing element which can be replicated multiple times for better performance [18], [19], [13]. They are easier to program, require moderate design effort and lend well to VLSI implementation due to ease of replication [18], [19]. Many homogeneous architectures have been proposed in the past, such as the 16-core Raw Architecture Workstation (RAW) from Massachusetts Institute of Technology [20], 167-core Asynchronous Array of Simple Processors (AsAP) by University of California at Davis [21], etc.

Hewlett-Packard announced a chip multi-processor consisting of multiple MIPS cores [22] and on-chip DRAM, especially for the embedded market. A WaveScalar

processor has been proposed by researchers at University of Washington which contains approximately 2K simple processing elements (PEs) arranged into clusters of 16 [23]. In [24], a scalable homogeneous MPSoC for next generation architectures has been proposed. This architecture is based on RISC processors and distributed memories. Such homogeneous architectures have been developed to be scalable both in their hardware as well software programming capabilities.

The homogeneous MPSoC proposed by authors in [19] contains 80 cores connected by a network-on-chip architecture where each core has two floating-point units. The network-on-chip is arranged as a '10 x 8' 2D array in a 275 mm^2 area that also contains packet-switched routers, operating at 4GHz. The MPSoC was designed on a 65 nm process technology. Intel's widely popular Core-series processors (Core i3, i5 and i7), first introduced in 2006, are a family of homogeneous multi-core processors for usage in desktops and laptops [25]. These processors are a classic example of homogeneous architectures in mainstream usage, even if they are not directly focused on embedded systems. Tiler's TilePro family of processors, on the other hand, are targeted towards the embedded market such as in next-generation of network, multimedia and cloud infrastructure [26]. They have also simplified many-core programming with their Multicore Development Environment (MDE) that features standard Linux with standard C/C++ compilers for rapid product deployment [27]. Various chip manufactures have announced their own homogeneous multi-core chips targeted towards different computing domains such as scientific, embedded and general purpose computing.

Apart from being scalable, both in terms of VLSI implementation as well as design efforts, such homogeneous architectures also provide easier programmability [13]. Another approach to leverage performance with lesser design effort and high flexibility is to start with a generic homogeneous design and add heterogeneous components based on application or domain as well as market demands at a later stage

[28]. This is essentially the path taken by most FPGA vendors such as Xilinx [29] and Altera [30], where they initially released generic “sea of gates” chips and gradually incorporated heterogeneous components such as hardwired signal-processing elements, memories, processors etc. based on domain and market requirements. The inclusion of heterogeneous components is essential to achieve high computational as well as power efficiency [18], [16].

2.1.2 Heterogeneous Architectures

The increasing complexity of current embedded applications and the strict design constraints such as performance, power, etc. have contributed to increasing popularity of heterogeneous platforms [16], [17]. In the past, several authors from both academia as well as industry have proposed various heterogeneous architectures to meet design constraints in specific applications or domain. Estrin [31], in his landmark paper in 1960 suggested a new type of computing machine for scientific computation that could dynamically adapt its computational resources to different algorithms such that each algorithm could be executed very fast. He called the machine as a fixed plus variable (F+V) structure computer, which was built using a conventional micro-processor and reconfigurable hardware [31], [32]. Although it was a pioneering idea at that time, it was not part of commercial products for various reasons [33].

Lee et al. [34] described a heterogeneous system called *MorphoSys* with a MIPS-like RISC processor core (called TinyRISC), ‘8 x 8’ reconfigurable array, context memory with a Direct Memory Access (DMA) controller and Frame Buffer for high speed memory transactions. They implemented applications such as Motion estimation and Discrete Cosine Transform (DCT) algorithms of MPEG and achieved performance comparable to a few ASIC designs as well as demonstrated significantly better performance than a homogeneous general purpose Pentium MMX processor operated at comparable frequency. The *MorphoSys* architecture is one of the earliest examples

of reconfigurable heterogeneous architecture research & development and shows the advantages of such architecture over other systems.

Zhang et al. [35] used a heterogeneous architecture based on the *Pleiades* architecture [36] for baseband wireless applications to achieve high power efficiency. Their architecture consisted of an ARM8 processor core, which was manually optimized and customized for high performance and low energy operation. A '4 x 8' array of embedded FPGA formed the reconfigurable portion of their architecture, while the memory units were also customized for high efficiency. The authors used a voice compression application on their architecture to show an approximate energy improvement of eight times compared to the DSP based systems. They claimed that the class of reconfigurable processors was a viable computing platform for applications that require both flexibility and low energy consumption.

Becker et al. described different heterogeneous SoC architectures specifically tailored for mobile systems [17], [37], [38]. In [17] & [37] the authors described a configurable system-on-chip (CSoC) architecture which consisted of a SPARC compliant LEON processor core [39], a commercial coarse-grained XPP array [40] of suitable size from PACT Information Technologies and application tailored local/global memory topology with efficient Advanced Microcontroller Bus Architecture (AMBA)-based communication infrastructure. The authors showed that the adaptiveness and multi-purpose usability of CSoCs promised to be an attractive potential for embedded system industry in different application areas, e.g. (wireless) communication (multi-standard, different bandwidth and services), automotive (multipurpose architecture platforms for various kinds of control and multi-media in automobiles), etc. Moreover, the authors also claimed that the steeply rising cost of CMOS masking is favorably pushing the highly flexible and adaptive CSoC based systems to the forefront of embedded computing.

In [38] the authors showcased their heterogeneous SoC architecture in the mobile computing domain. The architecture contained an ARM7 EJS processor core, along with a coarse-grain '4 x 4' XPP array from PACT Technologies [40] and an application specific local/global memory topology with efficient AMBA based architecture similar to their previous works. The heterogeneous SoC architecture was optimized for the DCT step in the MPEG 4 video codec. The authors showed the advantage of their system in terms of increased performance and reduced power consumption. However, they did not give due consideration to the customization of individual components of the SoC, which could lead to better area-time efficiency.

Steve Knapp from Triscend Corporation [41] (now acquired by Xilinx [42]) described a configurable system bus called a Configurable System Interconnect (CSI) Bus for a CSoC architecture which was an emerging idea around 2000 [43]. The CSI Bus was specifically designed to facilitate re-use, timing reliability, increment in overall throughput, and reduction in debug time in applications that demanded short TTM as well as field adaptability. Wallner et al. in [44] proposed a Configurable System-on-Chip (CSoC) architecture that offered a programming model while providing high flexibility and adaptability by employing a micro Task Controller (mTC). This approach made the CSoC architecture forward compatible and easily adaptable to future CSoC families which differ in the number of processing clusters or reconfigurable arrays for different performance characteristics. In such cases, their micro task controller program or the mTC could be directly reused. The authors in [45] described a heterogeneous architecture for audio applications. A THUDSP2005 DSP core formed the main processing core of their system while the other peripherals could be programmed according to the application requirements.

In the recent past, the prominence of 'black' or 'dark' silicon has also become a major concern for system designers [46], [47]. This rise is primarily the artifact of aggressive process technology scaling and therefore increasing power density.

This prevents designers from utilizing the entire chip concurrently due to the fear of violating the thermal design power (TDP) constraints. One of the ways proposed in the existing literature to overcome this challenge is to use specialized cores by re-purposing the dark silicon [48], [49]. The authors proposed to instantiate special processing elements that only perform a narrow set of tasks, while keeping the rest of the chip switched off to conserve energy [49].

Recently, hardware accelerators based on the FPGAs and graphics processing units (GPUs) have emerged as two popular choices for embedded designers. Both these platforms have their pros and cons as highlighted here. In a recent work, Silva et al. [50] compared a Nvidia GPU with a FPGA for image processing applications in a traditional PC setup and concluded that both perform equally well for such applications. However, the Peripheral Component Interconnect Express (PCIe) bandwidth limited the overall performance of both devices. The authors in [51] [52] also highlighted the pros and cons of using these platforms for various scenarios. They concluded that while GPUs are more beneficial in designs with high parallelism, FPGAs are extremely useful for relatively low power designs. Another key difference between the two platforms is that while FPGAs are modular and can be used to implement almost any logic [51], GPUs have a fixed structure, thereby allowing the acceleration of only certain types of applications. However, designers need considerable experience in order to tap the full potential of FPGAs, which has triggered tremendous research interest in this field. In recent years various researchers have focused on developing design methodologies to address this issue. However, further research needs to be done to create a comprehensive design methodology to generate an application-specific system given a particular performance-area constraint [53].

A few heterogeneous platforms have become available in the last few years for commercial use. Although these platforms are not necessarily part of embedded

Figure 2.1: A hardware view of hybrid-core computing. Courtesy - Convey Computer.

systems, the concepts and methodologies involved can be useful for embedded designers. The heterogeneous systems produced by companies such as Maxeler Technologies [54] and Convey Computer [55] help customers in tapping the advantages of FPGA alongside the traditional x86_64 based processors found in numerous personal desktop computers. This allows the customers to use their familiar desktop environment, which has a high-end FPGA connected typically via the PCIe bus, to accelerate applications without a need to learn much about FPGA design. Figure 2.1 shows an example of a hybrid core computing architecture proposed by Convey Computer for their MX series architecture [56].

Recently, researchers have also focused on the area of heterogeneous multi-core platforms with diverse processing elements [57]. Such platforms typically consist of multiple general purpose processors along with reconfigurable hardware blocks, DSP blocks etc. arranged in a grid fashion. An application or domain specific design of such platforms is an open issue and researchers are generally focused on proposing optimum mapping techniques on multi-core platforms to achieve maximum performance, better fault tolerance etc. [58].

The different computing platforms have focused on using heterogeneous platforms in order to achieve higher performance while reducing power consumption in

embedded systems. While homogeneous platforms are relatively easier to design and program, the increasing performance demand as well as stringent power budget requirements necessitates the usage of heterogeneous platforms, especially in embedded systems. Different application and domain-specific heterogeneous platforms have been proposed, but the increasing complexity of embedded applications demands an automated system level design framework to generate such architectures.

2.1.3 ASIC Designs

As discussed in the earlier sections, embedded computing architectures range from application or domain specific ASIC designs to general purpose processors (GPPs) that are suitable for any application or domain. These two architectures generally form the two ends of the computing spectrum, within which various other architectures such as field programmable gate arrays (FPGAs), graphics processing unit (GPUs), application-specific instruction processor (ASIPs) etc. have been proposed as a trade-off in terms of flexibility, power-performance etc. [59], [60]. The authors in [61] demonstrated an order of magnitude difference between the performance achieved by ASIC designs and a software-only implementation on GPPs. A custom ASIC design does not require the usual instruction fetch and decode cycle during computation like general or domain specific processors [62], [63]. Therefore, the industry has traditionally relied upon ASIC designs to meet the stringent power and performance demands of embedded devices. ASIC designs are also the preferred choice for achieving low cost in very high volume products.

However, while the power and performance achieved by ASIC designs are unparalleled, their inflexibility towards design changes do not lend them well in modern embedded designs that demand shorter Time-to-Market and lower NRE cost requirements [64]. Time-to-market of an embedded device largely decides the competitiveness of a product in the market and the rapid advancement of technology has

FPGA & SoC Time to Market

ASSP / ASIC Time to Market

Figure 2.2: A comparison of approximate development timelines of ASIC and FPGA designs. Courtesy - Microsemi [66].

continuously shortened this window [65]. The consumer expectations of new and improved devices in the market today is unprecedented. Designers are consistently pushed to come up with designs with shorter development cycles and longer product life cycle. However, the turn-around-time for ASIC designs generally increases with reducing feature size, especially due to lack of suitable design methodologies. Due to the inflexibility of ASICs, even a minor change in design results in a long delay in the overall design cycle [66]. Similarly the non-recurring engineering or NRE cost of ASIC design is also increasing significantly with each update in the process technology [67] while the reducing product shelf life is making it difficult for the manufacturers to sufficiently recover their cost. Moreover, the rigidity in ASIC designs also prevents the designers to update and add new features into an existing product to increase the shelf life. At the same time, even with the flexibility and ease of designing, solutions based on GPPs are also not appropriate for embedded devices with strict power and area budget [62].

2.1.4 Reconfigurable Architectures

On the other hand a system based on reconfigurable architectures have relatively shorter time-to-market as well as NRE cost, especially for custom designs in low to mid volume products or designs requiring flexibility, future proofing etc. The authors in [66] argued that the average TTM for FPGA design is about two to three times faster than comparable ASIC designs. Figure 2.2 shows a relative comparison between TTM for FPGA and ASIC designs. In order to reduce NRE costs and meet the shrinking TTM deadlines, reconfigurable architectures have emerged as a favorable design alternative [62], even though they are less efficient in terms of power and performance when compared to an ASIC [68]. However, reconfigurable architectures provide the opportunity to be customized for specific applications or domains to achieve high performance with low power budget, while still being flexible enough for subtle and/or severe design changes at various stages of the design cycle, and even after deployment. Modern high end FPGAs also provide numerous hard IP blocks such as multipliers, dividers, DSP etc. that can be useful to reduce the power gap [68]. At the same time design reuse can also save time and effort for new products. Hence, Field-programmable gate arrays have become attractive as a readily configurable system component. A FPGA, owing to its configurable/programmable features, is flexible enough to allow the implementation of multiple types of applications, even from different domains if required, while also providing opportunities for future design changes. For such reasons, FPGAs have also been extremely popular for space as well as military applications [69], [70], [71], [72], [73]. In [72], the authors have provided a detailed survey of the various industries and successful commercial products using System-on-Programmable-Chip (SoPC) such as FPGA.

In order to tap the full potential of FPGAs, researchers and commercial entities are motivated to create a computing platform comprising of a processor for general

purpose computation and programmability, a suitable memory hierarchy, while using the FPGA fabric for acceleration, customization, etc. Such a platform is generally known as a “Configurable System-on-Chip (CSoC)” [17], [37]. A CSoC also allows for design and implementation changes for future proofing a commercial product. A large number of components can be *synthesized* on current high end FPGAs from vendors such as Xilinx, Altera etc. These benefits have been further sweetened by the decreasing costs of modern high performance FPGA architectures. This has also led to the recent development of commercial configurable heterogeneous platforms such as the Zynq platform from Xilinx [74] and SoC-FPGAs from Altera [75] which contain hard-core ARM based processors alongside the FPGA fabric.

2.1.5 Configurable System on Chip (CSoC)

A Configurable System on Chip (CSoC) can be designed on a single chip such as in a modern FPGA consisting of thousands of logic blocks and a diverse set of on-chip resources such as distributed memory, DSP blocks etc. Such heterogeneous platforms, often called CSoC in existing literature [43] [44], are becoming increasingly popular, aptly driven by fast development of VLSI technologies to enable such on-chip integration [17], [37], [72]. While these architectures provide obvious advantages in terms of cost and flexibility of implementation, such complex system design introduces a set of challenges for industrial designers and academics. Important issues to be addressed here are efficient and application-specific hardware and software architectures for better power-performance efficiency, and corresponding platform-based system level design methodologies [76], [77], [78], [79]. Various academic such as [17], [34], [35], [36] and commercial architectures such as [40], [41], [80], [81] have been developed recently. These Configurable SoCs typically consist of a Processor, Memory and on-chip reconfigurable hardware parts for customization to a particular

application [37], [72], [82]. The following briefly describes the important components of a typical CSoC:

(i) Microprocessor: The microprocessor forms the core of a CSoC [72]. It is an important component in a SoC for easier application programmability and can be generally classified into two types:

- Hard Core Processor: A hard core processor refers to the hardwired microprocessor core in a FPGA logic such as the hardwired PowerPC 405 core in earlier Xilinx FPGAs [29], or a dual-core ARM Cortex A9 core in the latest Altera [75] and Xilinx [74] FPGAs. Other examples of hard core processor based CSoCs include AT94K FPSLIC from Atmel, composed of the AVR MCU and the AT40K FPGA [81], Excalibur EPXA family from Altera, constituted by an ARM9 MCU and the APEX20KE FPGA [83]. Although these processor cores provide optimum performance with little or no effort at design time, they cannot be configured as their soft core counterpart.
- Soft Core Processor: A soft core processor is a highly configurable block which can be customized using software tools and synthesized with numerous possible configuration on almost any FPGA. Soft core processors like Microblaze from Xilinx [84], NIOSII from Altera [85] etc., provide the flexibility for highly customized processor design. The opportunities for configurability include the instantiation of optional components such as a hardware multiplier or a hardware divider block etc. along with the declaration of custom cache size, instruction set extension etc. At the same time instruction subsetting can be used to save FPGA area by disabling the support for specific un-used instructions in the application [86], [87], [88]. A soft core processor also provides a unique opportunity to cheaply explore multi-processor based design to achieve competitive performance

and energy consumption compared to existing low-power hard processor cores without the need for redesigning/re-fabricating the entire system [89]. Such processors have been popular in creating low cost custom designs (e.g. digital entertainment boxes, low-end routers and switches, industrial motor control etc. [90]) and provide easy updates without incurring additional costs to stave off obsolescence. Embedded markets with frequently changing standards such as telecom or wireless [91] have used such processors for reasons such as future proofing, while the aerospace industry [70], [92] incorporated them in Single Event Upset (SEU) resistant systems.

However, the availability of a large number of configurable options in a soft core processor makes it a challenging task to find the best possible configuration of the processor for a given application and a given performance-area constraint. Many researchers have proposed methods to find the optimum configuration of a soft core processor for a given application [86], [87], [88].

(ii) Hardware accelerators: Hardware accelerators are custom functional units designed to accelerate certain sections of the code based on performance and area constraints of the design. These custom units can be of varying granularity based on their size and implementation. The hardware accelerators can be generally classified into three categories:

- Custom Instructions - Custom Instruction are typically formed by encapsulating multiple primitive operations that constitute the critical portion of the application. Traditionally, such *micro accelerators* do not incorporate memory or control flow operations, since they augment the inbuilt Instruction Set Architecture (ISA) of a processor by direct integration into the

datapath of the processor core. Custom Instructions can enhance the performance of a system significantly [93]. However, the identification and selection of application-specific custom instructions can be a daunting task, especially in large applications [94], [95], [96], [97].

- Co-processors - When the performance demand increases, a designer needs to port more of the application code segments into hardware. This in turn calls for the need to incorporate memory and/or control operations in the hardware accelerator blocks. Such operations are generally excluded in a custom instruction in order to have deterministic latency and to avoid pipeline stalls during memory fetch [94]. On the other hand, co-processors are connected outside the processor's datapath and hence can be executed in parallel to the processor if desired [82]. Therefore they are not restricted in their operation support like a custom instruction [98]. The identification of such co-processors are usually done based on profiling results while the selection process considers the performance-area constraints in the design [82].
- IP Cores: Intellectual Property (IP) cores are dedicated hardware blocks supplied as pre-verified synthesizable software codes and can be theoretically implemented on almost any FPGA block of sufficient size. Various FPGA manufacturers like Xilinx [29] and Altera [30] provide proprietary IP cores optimized for their FPGA architectures. A few organizations also provide free and open source IP cores which can be implemented and optimized for any FPGA architecture [99]. Due to large number and variety of available IP cores in the market, it is a challenging task to automatically select suitable IP cores for a given application. Typically, designers select suitable IP Cores manually based on their expertise, licensing costs,

support and credibility of the IP provider etc. [100]. Jiao et al. [101] proposed a method based on Dynamic Data Flow graphF concept to identify a coarse grain input output relationship in a function which can be used to select a suitable IP core.

- (iii) Memory Subsystem: A suitable memory hierarchy is vital to any computing platform. The increasing complexity in embedded applications and the wide scale of usage of multimedia applications on portable devices has further aggravated the memory bottleneck created by slow off-chip memory components, limited Input/Output pins etc. [102], [98]. Therefore designers need to analyze the applications carefully to be able to decide the size and configurations of the different memory subsystems such as off-chip main memory, various levels of cache, usage of distributed blocks RAMs in FPGAs as local memories etc. This problem is more pronounced in heterogeneous platforms where the data manipulated by the hardware accelerators need to be synchronized with the data in the main memory to avoid the usage of incorrect data by the processor.

Apart from these components FPGAs also provide high level of flexibility concerning digital I/O connectivity standards. Latest FPGAs from Xilinx and Programmable System-on-Chip (PSoC) families from Cypress also provide mixed-signal programmable routing and interconnect [72], [103], [104].

However, a major hurdle in shifting towards a reconfigurable design discussed above is the lack of design methodologies and a consolidated *framework for automatic application-specific generation of such architectures*. At the same time, the rapid advances in technology has led to shorter product life cycles, which in turn has led to more “design starts”. The sheer number of configurations possible in a CSoC largely prohibits the manual selection of the optimum configuration and calls for an

automated approach. Moreover, the increasing NRE and TTM pressures due to extreme market competition etc. also demands efficient design methodologies for rapid product development. In its 2011 [3] as well as its 2012 [4] reports, ITRS repeatedly emphasized the need to refocus attention on suitable design methodologies in order to increase designers' productivity. Efficient design methodologies are paramount to aid in designing such configurable SoCs with better performance and power efficiency, before they can be more widely adopted [105], [51], [52]. The next section discusses the available toolchains and design methodologies employed in commercial as well as the academic community for generating an application-specific Configurable System-on-Chip.

2.2 Configurable System-on-Chip Customization

Important design methodologies for customizing the individual components of a CSoC will be discussed in the following subsections. Subsequently, methodologies for a complete CSoC design and relevant commercial tools will be highlighted in subsection 2.2.4.

2.2.1 Processor Customization

As discussed earlier, a Configurable System-on-Chip typically consists of a processor, hardware accelerators of multiple granularity for performance enhancements and a memory subsystem. In a CSoC, one can use a hard-core or a soft-core processor. A soft-core processor, unlike a hard processor core, offers immense flexibility and allows for further customization through micro-architectural changes (such as Instruction Subsetting) as well as instruction set extension [72]. In addition, the low cost and competitive performance of Soft core processors, especially when optimized for

a FPGA fabric, makes it an easy choice in a FPGA-based SoC [106], [86]. Commercial soft core processors such as Microblaze [84] from Xilinx, NIOS II [85] from Altera Corp, MICO 32 [107] from Lattice Semiconductor [108] etc. and open source projects such as the SPARC architecture compliant LEON processor [39] have gained popularity recently due to their low implementation cost and high flexibility [89], [109], [110], [111].

Although soft core processors generally lack in terms of power and performance as opposed to comparable hard-core processors, Lysecky et al. [89] proved the competitiveness of soft-core processors against similar low power hard-core processors by creating a customized multi-core processor configuration using Xilinx Microblaze soft core processors on a low power Spartan 3 FPGA. Their architecture, known as the “Warp Processor” provides competitive performance and power efficiency when compared to low power hard processors cores such as ARM9, ARM10, ARM11, etc. The authors proved that with extensive customization, soft-core processors can achieve performance comparable to hard-core processors, along with additional advantages of flexibility and configurability that allows for more application specific customization and future proofing.

However, the numerous configuration options available in a typical soft-core CPU lead to two important areas for investigation [86]. Firstly a designer must understand the architectural trade-offs that exist in these processors, such as pipeline depth, functional unit implementation etc. Secondly, in order to capitalize on the numerous configuration options, a system designer must be able to explore the design space rapidly and tailor the processor for a given application. This can lead to significant performance, power and area benefits [86], [87], [88]. The following sub-section presents some notable work done in the area of application specific customization of a soft core processor through manipulations in their micro-architecture using Instruction Subsetting.

2.2.1.1 Instruction Subsetting

Instruction subsetting is defined as creating an application specific instruction set processor from a more general processor, by removing the support for unused instructions [112]. Dougherty et al. [112] introduced and demonstrated the advantages of instruction subsetting as a means for reducing power consumption by manually optimizing the assembly codes of the applications. Various researchers have proposed methodologies for customizing a soft core processor for a given application using similar concepts [86], [87], [113], [106], [114].

The concept of Instruction Subsetting is well suited for soft-core processors where support for instructions such as hardware multipliers, floating point dividers etc. can be enabled or disabled before the synthesis step for power and area savings based on the requirements of the target application. Lee et al. [115] showed the advantages of reduced number of instructions for saving power. Firstly, they compiled the application using the basic instruction set. Next, they combined these smaller instructions to generate more complex instructions based on the performance gain in terms of clock cycles saved after using these complex instructions, to show the advantage of using lesser number of instructions.

Keutzer et al. in [116] reported the importance of Application Specific Instruction set processor generation in programmable logic to save power and increase performance. They claimed that a variety of factors such as increasing manufacturing cost etc. are forcing designers to move from traditional ASIC based design to a Programmable logic based design. Although the programmability provided high flexibility, it came at the cost of lower performance and higher power consumption. The authors proposed the usage of Application Specific Instruction Set Processors or ASIPs for such scenarios which can increase the performance as well as reduce power consumption of the system significantly.

Instruction Subsetting is well suited for application-specific customization of soft core processors. However, as the number of configurable parameters increases with modern soft core processors, there is a need for faster design exploration methodologies to assist designers in Instruction Subsetting [87].

Various methods have been proposed to select the right configuration of a soft core processor [86], [87], [88], [113], [106]. However, these methods mostly rely on synthesis/execution runs to arrive at the best configuration of a soft core processor. In addition, some of these methods use in-house developed tools for their work (e.g. [86], [106]), which is also not suitable for commercial use. It can be observed that the existing research on the soft-core processor customization problem has typically used an exact methodology. Such methodologies, while being reliable, typically tend to be slow and hence not desirable for a large design space.

Ray et al. [117] and [118] proposed a Low Level Virtual Machine (LLVM) compiler infrastructure [119] based methodology for estimating the performance of an embedded processor for various applications. Their method was based on a fast estimation heuristic which worked on an intermediate representation of the application obtained from the LLVM compiler. Such a methodology was not only faster than a complete compilation process, but also easier since the application was not required to be executed on the target processor.

One of the ways to efficiently use the area saved by the Instruction Subsetting step is to add application-specific accelerators (such as custom instructions) at different granularities to achieve higher system performance. Instruction Set Extension through custom instructions is clearly advantageous in FPGA-based SoCs as it lends well for such implementations.

2.2.1.2 Instruction Set Extension

Instruction set extension by using application specific custom instructions is a popular method for increasing performance in a high performance embedded system [94], [120], [121]. Custom instructions are formed by collapsing a set of basic operations from an application code into a single custom instruction. There are typically two major steps in custom instruction implementation, namely, custom instruction (pattern) *identification* and custom instruction (pattern) *selection*. A commonly adopted approach in pattern identification is pattern enumeration, which tries to identify all the *legal/valid* custom instruction patterns within an application. In the second step i.e. pattern selection, a designer selects a set of non-overlapping patterns for final implementation based on certain constraints such as area and performance [122]. The sheer number of possible patterns in an application Data Flow Graph (DFG) calls for effective pruning techniques during the enumeration phase, while still preserving the most profitable candidates for pattern selection [97]. A significant amount of research has been carried out both in the area of Enumeration as well as Selection.

Atasu et al. [94] described a branch and bound algorithm that enumerates all the legal patterns from an application DFG, while pruning off the ones, which violate the fundamental micro-architectural constraints such as number of inputs/outputs etc. In order to guarantee a deterministic latency, certain basic operations such as memory operations like LOAD/STORE and control flow are considered as invalid operations for custom instruction implementation. Various authors have used similar constraints during the enumeration phase and proposed methods to speed up this step [96], [123], [124], [125].

It can be observed that the existing work in pattern enumeration invariably takes into account the micro-architectural constraints of the target architecture. A designer takes advantage of such constraints to prune the large design space during enumer-

ation. On the other hand the authors in [122], [126] used the knowledge of the target architecture, Xilinx Virtex II FPGA in their case, to estimate the area and time of the selected custom instructions when implemented on the target FPGA. This concept could also provide additional opportunities to prune the enumeration design space further, while also selecting profitable custom instructions specific to the target architecture.

Certain basic operations such as LOAD/STORE etc. are considered as invalid operations during the custom instruction identification phase [97]. The number of registers available in the processor for data transfer between the processor and the custom instruction serves as an additional constraint for the identification phase. Custom instructions are therefore, limited in their size due to exclusion of the invalid operations and the other constraints. The general trends in existing literature are to speedup the identification phase [97] and/or improve the selection phase to implement more profitable custom instructions [127] [122].

Such instructions are useful in embedded designs with strict area constraint due to their relatively smaller size and the potential to provide high performance. However, they are limited in their maximum achievable performance due to their size, especially when the area constraint is relaxed and the designer is looking to maximize performance. In order to cater for such design requirements, a designer invariably looks towards accelerating larger portions of code using hardware accelerators that need to incorporate both memory as well as control flow operations.

2.2.2 Hardware Accelerators

Identification and implementation of application-specific hardware accelerators to achieve higher system performance has been explored extensively since the early 1990s [128] [129] and many approaches have been proposed in the past to automate this process

[128], [82], [130], [131].

The sections of the application code translated to the *hardware accelerators or co-processors*, either have direct access to the main memory through a memory controller [82] or need to use DMA for data transfers [132]. The latency of accessing the main memory from the co-processors can be high. But, it does not have adverse effects on the processor's efficiency, since unlike custom instructions, they are not part of the processor's datapath. However, the direct access of the hardware accelerator to the main memory can lead to cache-incoherence issues, requiring additional effort to maintain coherency [133], [134].

At the same time, accelerating large sections of the application code requires extensive data dependency analysis to reduce the communication time between the hardware accelerator and the relevant data memory, as well as the communication overheads of data access between hardware and software sections of the application. Various authors have proposed the usage of local memory blocks to reduce this communication time [132]. The close proximity of the memory block to the custom instructions had obvious advantages in their system such as deterministic latency, reduced memory access time and energy consumption. Additionally, this also led to lower cache-pollution since the custom instructions had direct access to the local memory elements bypassing the cache [132], [135], [136].

A few authors have also proposed using hardware accelerators at multiple levels of granularity, ranging from basic blocks to a function or process level in their design, with the data communication between the hardware and software sections being done through the usage of shared memory space [128], [137], [130], [131]. While such an approach is clearly advantageous to leverage the usefulness of each kind of accelerator, suitable data-dependency analysis needs to be performed to reduce the communication overhead of accessing appropriate data from different memory blocks. At

the same time, it should also be noted that a function has different sections executed with varying frequency during the application execution. When a complete function is accelerated in hardware, the sparsely executed code sections also need to be ported to hardware, occupying hardware area without providing sufficient performance advantage. Therefore, it might be prudent to accelerate at finer levels of granularity than at a function level to avoid accelerating non-profitable application code segments.

Moreover, due to the exponentially large design space of the partitioning step (2^N different hardware/software partitions with “N” candidate blocks) [128], an exhaustive algorithm quickly becomes intractable. This indicates the need for developing rapid heuristic algorithms for selecting the right set of application segments for hardware acceleration while considering their performance as well as their often interleaved data-dependency.

The major issue in selecting hardware accelerators is the data dependency between the different application segments which prohibits the selection of an isolated code segment for acceleration. At the same time, the large design space of selecting the hardware accelerators necessitates a rapid heuristic algorithm instead of an exhaustive approach.

2.2.3 Memory Systems In CSoC

The memory subsystem is an important component of any computing platform. As discussed earlier, a CSoC can either have a shared or a distributed memory for its various components [17], [37], [34].

Panda et al. in [138] proposed methods to partition the data stored in on-chip and off-chip memories. The data partitioning decision was mostly based on the access frequency of the data. Similarly Kumar et al. [139] proposed methods to partition the data on appropriate on-chip memory blocks for DSP applications. In their case, the

preferred data for on-chip storage was based on their access frequency as well as their size. The objective of this data partitioning was to reduce the power and performance of the overall memory layout, not necessarily from the hardware/software partitioning perspective. Even so, the authors provide important insights into on-chip vs. off-chip data storage and its effects on power, performance etc.

The authors in [140] have proposed a step-by-step method to construct an application-specific memory hierarchy on a FPGA-based system. Iosifidis et al. [102] describe the need for automatic parallelization as well as data allocation schemes for both static and dynamically allocated data targeted towards MPSoC platforms. They propose the use of local memories instead of large main memory for energy-efficiency and better performance. This is achieved by compile-time analysis of the applications to identify appropriate statically allocated variables and copying this data at runtime using DMA transfers. While the use of local memories is well justified, DMA transfers tend to be expensive in terms of performance and energy.

The Instruction set extension methodologies discussed earlier incorporate additional custom instructions into the base processor for higher performance. Traditionally, these custom instructions have considered memory and control as invalid operations in order to have a deterministic latency in the processor's datapath. A few authors showed that including memory and control operations as part of custom instructions can lead to significant performance boost [135], [141]. The authors proposed the use of Local memory alongside the custom hardware to alleviate the memory bandwidth bottleneck [132]. However, their architecture required additional circuitry to avoid the cache-incoherence issues while also using DMA for data transfer during application execution.

The authors in [142] describe a tool to automatically analyze and transform applications during compile time to select data-structures that can be transferred to a

scratch-pad memory to achieve both high performance and low power consumption. Such techniques can be useful to identify the relevant memory blocks for each hardware block. Cong et al. [143], [144], [145] proposed memory partitioning algorithms at a cycle accurate level to meet the hard memory bandwidth requirements.

It has been observed that most of the existing approaches use DMA for data transfer and require additional effort to avoid the cache-incoherence issues.

2.2.4 Methodologies For CSoC Design

There has been a considerable interest in the past decade to fill the "Design gap" by using efficient system-level design methodologies for generating configurable heterogeneous platforms such as a CSoC [146]. LegUp is a high level synthesis tool that compiles a C program on to a hybrid architecture consisting of a MIPS based soft processor and custom hardware accelerators [82]. The authors have demonstrated the ability of their tool to do design space exploration by varying the amount of program executed in software vs. hardware. The code sections implemented in hardware is selected and decided upon *manually* and fed into the tool as one of the inputs for design space exploration. While the tool itself is extremely useful because of its end to end design flow, memory implications during the hardware selection need to be considered for better performance.

Pilato et al. proposed the PandA framework for hardware software co-design and includes the Bambu tool for high level synthesis of memory implementation in complex applications [98], [147]. The Bambu tool can be used to automatically infer and implement suitable memory hierarchy during the high level synthesis of complex applications. This methodology is used to classify the variables according to their viable placement in internal or external memory blocks and fully supports all the constructs in the standard C language.

In [53] the authors described an iterative custom instruction generator from an application given a performance constraint. The basic blocks along the critical path were identified and fed as an input to their tool. The tool then selected a set of custom instructions from within these basic blocks to meet the timing constraint iteratively. If the performance requirement was impossible to achieve, the tool returned the best set of Custom instructions found so far. In [148] the authors describe a compiler for C-applications to be mapped on CPU-FPGA heterogeneous platforms. They proposed the usage of *many-cache* memory model in order to allow each of the FPGA-based accelerators their own cache space to avoid the cache incoherence issue. The functions to be mapped as FPGA accelerators was selected manually and achieved an average performance of 6.9X when compared to software implementation.

In [149] Wayne Luk et al. describes a high-level compilation toolchain called Harmonic targeted towards mapping application on a heterogeneous architecture consisting of general purpose processors, digital signal processors and FPGAs. The toolchain took the C application with annotations, denoting the various partitions, as the primary input and mapped it to the appropriate functional unit for efficient execution.

The hArtes “Holistic Approach to Reconfigurable real Time Embedded Systems” project is aimed at creating a complete design flow for reconfigurable heterogeneous platforms [150]. Various efforts have been made to showcase their targeted platform [151], application mapping [152], complete design flow [150] etc. In [153] the authors provided a detailed overview of the various components in the hArtes toolchain. They have shown a way to automatically or semi-automatically map the various sections of the application code on the most appropriate functional unit in the heterogeneous platform. A spin-off company called Bluebee technologies commercializes the outcomes of the hArtes project especially for Big Data processing systems. Further details about this company can be found at [154], [155], [156] and [157].

It has been observed from the existing literatures that an automated design methodology is absolutely vital in order to exploit the full features of modern heterogeneous platforms. However the existing methodologies require designer intervention and expertise during application partitioning. Such heterogeneous platforms present a rich design space for application partitioning and binding to different computational modules such as the processor and the hardware accelerators. With increasing number of accelerators used in a system for increased performance, the complexity of the design space increases manifold, thereby necessitating a rapid design methodology.

2.2.4.1 Commercial Tools

Industry leaders in Electronic Design Automation (EDA) such as Synopsys [158], Mentor Graphics [159] etc. have released their own System level design tools to leverage on the platform-based design concepts. The System Level Design solutions [160] from Synopsys includes tools for architecture design, processor designer, platform architect etc. which can be used to model and design systems at various levels of abstraction and granularity. Vista [161] is a set of similar tools from Mentor Graphics which includes solutions for architectural design exploration, virtual prototyping etc. to test different combinations of hardware and software implementation. In other words, these tools are useful to a designer during the Hardware/Software Co-design phase. The XPRES compiler [162] from Tensilica (now part of Cadence) is another example of a HLS tool specifically targeted for the Xtensa [163] processor based systems from Tensilica. However, none of them provide an automated way to do the partitioning between hardware and software implementations. Also, these tools do not necessarily focus on FPGA-based SoC design, however the concepts can be reused and extended for FPGAs. The customization and flexibility offered by FPGAs are great use-cases for such design tools in order to avoid the complex and lengthy SoC design exploration process.

The acquisition of the AutoESL behavioral synthesis tool Autopilot [164] [165] by Xilinx and its subsequent expansion and rebranding as the Vivado Design Suite [166] is one such example. Similarly, a startup company called Space CodeSign [167] provides SpaceStudio [168] tools for accelerating the HW/SW co-design process, with special libraries for aerospace and multimedia industries. The eXCite tool from “Y Explorations” [169] is yet another commercial high-level synthesis tool that can compile ISO/ANSI-C applications and synthesizes it to Verilog or VHDL RTL suitable for input to FPGA or ASIC logic synthesis tools. This tool can be used along with other partitioning tools in order to implement applications on a heterogeneous platform. Altera’s C2H compiler [170] allows a user to manually partition the functions in a C application code and mark it for hardware or software implementation.

The commercial tools described above have tried to fill the design productivity gap by automating some of the steps involved during mapping of an application written in high-level languages, such as C, to a heterogeneous platform. The automation of this process is vital in order to tap the full advantages of a configurable heterogeneous platforms. However, none of them select the right set of application code segments automatically for acceleration in order to meet the system design constraints such as performance or hardware area. Instead these tools rely on the manual intervention and expertise of the system designer for the partitioning and selection step. Additionally, some of these tools, such as Altera’s C2H [170] dictate that the acceleration be done at the function level which might include sections of the code that are seldom executed but need to be part of the function. Therefore, there is a risk of implementing less significant sections of the code which do not provide sufficient performance gain in hardware, yet use valuable hardware area.

2.3 Summary

The NRE and TTM pressures have and will continue to drive the embedded systems market in the coming years. The need to develop a product rapidly, inexpensively, with a certain level of future-proofing necessitates the use of configurable platforms. At the same time designers are also motivated to use customized heterogeneous architectures in order to meet the various functional constraints such as performance, area, power, etc. In particular we observed the trend towards developing application or domain specific CSoC platforms in order to address these requirements. The configurability in such architectures provides a certain level of future proofing with relatively low NRE and TTM, while the heterogeneity is useful in meeting the functional constraints such as performance, area, etc.

The existing work has clearly identified the major components of such architectures, namely a processor, custom hardware units with appropriate memory subsystems [37], [72], [82]. As seen earlier, a CSoC could be based on either a hard or a soft core processor. The flexibility and configurability of soft core processors allows for future proofing as well as extensive application-specific customization to achieve better performance-area efficiency. Soft core processors have therefore emerged as a popular choice, especially in low cost designs as well as in products requiring fast TTM or continuously changing standards etc. The existing literature, such as [16], [17], [38], [37], [38], etc. has also shown that heterogeneous architectures, such as a processor with application-specific hardware accelerators, leads to higher performance-area efficiency.

However, the existing methodologies for CSoC generation generally need manual expertise to select the appropriate sections of the code for hardware implementation [82], [148]. Moreover, most of them do not consider customizing the various components of a CSoC during the design process, typically ignoring the customizability

of the accompanying processor. Similarly, while some of the commercial tools discussed in the literature can be useful, all of them require significant manual expertise for application partitioning as well as overall customization of the architecture.

Hence, there is a need for developing design methodologies for effective customization, design space exploration and rapid generation of CSoC. Such methodologies not only save the NRE cost and meet TTM deadlines, but also provide more area-time efficient systems [146]. It was clearly evident from the survey of the existing work that a significant amount of research is needed in order to generate a comprehensive framework for design exploration and automatic generation of application-specific heterogeneous platforms that leverages the customizability of the individual components to meet design constraints.

The research work in the area of soft core processor customization through instruction subsetting mostly rely on the lengthy synthesis/execution runs to arrive at the best configuration of a soft core processor [86], [87], [113], [106], [114]. A rapid and scalable methodology is essential to avoid using such time-consuming synthesis/execution runs in order to determine the performance-cost benefits of various processor configurations. Also, the framework should not be restricted to a particular target processor, therefore allowing it to be employed for wider varieties of processors. Chapter 3 presents such a rapid and reliable estimation methodology for soft core processor customization.

The next level of customization can be achieved through instruction set extension of the base processor using custom instructions. Existing work in this area have used the underlying architecture information during the enumeration phase, albeit in a limited way [96], [123], [124], [125]. Specifically, the enumeration algorithms described earlier take the micro-architectural properties of the underlying hardware as a fundamental set of constraints, for example the number of input or output ports available

for custom instruction implementation. Although these constraints are mandatory to be used during enumeration, it was seen in the literature that the authors failed to consider the architecture information of the target platform during enumeration as well as selection of custom instructions [96], [123], [124], [125]. In this thesis, we will investigate the benefits of using the architecture information of the target platform as additional constraints, in order to reduce the tool runtime of the extension process while also providing most profitable candidates for instruction set extension. Chapter 4 discusses the contributions made in this direction.

The selection of hardware accelerators poses a different set of challenges than instruction set extension due to the inclusion of memory and control flow operations into the hardware accelerators. Not only does the designer need to select appropriate hardware accelerators based on their performance, they also need to ensure proper memory subsystems alongside these accelerators. As discussed earlier, although some authors have proposed ways to incorporate local memories beside the hardware accelerators, they propose to use expensive DMA protocol to transfer data between the main memory and the local memories [102], [132], [135], [141]. Moreover, they also need to use additional cache-coherence protocols to ensure data coherence along the entire memory hierarchy. These additional protocols reduce the overall effectiveness of the hardware accelerators.

In addition, the selection of hardware accelerators in current methodologies is typically based on traditional metrics such as hardware/software performance, area etc. while ignoring the memory dependency of the candidate blocks. Extensive data dependency analysis is essential to reduce: (1) the communication time between hardware accelerator and the relevant data memory, and (2) the communication overheads of data access between hardware and software sections of the application. Also, accelerating at the function level as suggested in the existing work might lead to accelerating less profitable code segments. In this research we propose to accelerate

at a finer level of granularity than function level, such as “basic block” level, to achieve maximum performance-area productivity. At the same time, the large design space of selecting the hardware accelerators also necessitates a rapid heuristic algorithm instead of an exhaustive or manual approach. These problems will be extensively discussed in chapters 5 and 6 followed by our proposed solutions.

The next chapter describes a framework to customize a soft core processor using Instruction Subsetting.

Soft-Core Processor Customization Through Instruction Subsetting

3.1 Introduction

A processor, whether a soft or a hard core, typically acts as the main computing unit in a Configurable System-on-Chip (CSoC). As discussed in chapter 2, Soft Core processors have been selected as the processor for the proposed CSoC architecture in this thesis, for its flexibility and configurability as discussed on pages 24 and 41. In order to customize a processor for a given application, various researchers have proposed methods as discussed earlier in chapter 2. Typically, processor customization can be achieved by either instruction set extension or instruction subsetting. Instruction set extension, as described later in Chapter 4, is useful to achieve higher performance than the base processor. On the other hand, Instruction Subsetting can be immensely useful to reduce the FPGA area required to implement a subsetting Soft Core processor. The parameterized nature of commercial soft core processors provides easy implementation of Instruction Subsetting leading to better area and power efficiency [112]. Instruction subsetting can be defined as a process to determine a suitable subset of instructions for a given application such that a customized processor, implementing only that subset, can be synthesized. Commercial FPGA soft-core

processors [171], [172] facilitate instruction subsetting by providing a number of configurable functional units e.g. hardware multiplier, barrel shifter, etc. that can be enabled/disabled during the processor specification phase. These configurable units incur additional FPGA area, and hence they should be used prudently. However, as discussed in Chapter 2, the existing work in this area tends to use synthesis runs and exact methods resulting in a slow algorithm for this step. With increasing sophistication of soft core processors consisting of numerous configurable options [88], a rapid design methodology is essential to complete this step.

The rest of the chapter is organized as follows. Important related works in this area have been elaborated in the next section. Section 3.3 describes the proposed design exploration framework. Section 3.4 deals with the experiments, results and discussions. This chapter is concluded in section 3.5 along with some future work.

Portions of the work discussed in this chapter have been published in [6].

3.2 Related Work

Padmanabhan et al. [88] proposed a method for application specific microarchitecture customization using the LEON [39] soft core processor. They began with the default base configuration and perturbed one parameter at a time. Each processor with a different configuration was synthesized to measure the chip's cost. In addition, the given application was executed on each processor configuration and the runtime was measured using an in-house-developed Liquid arch platform. Finally, they formulated the performance-cost measures of the various configurations into a BINLP problem and solved for optimal solution, using the commercial solver of Tomlab Mixed Integer Nonlinear Programming solver.

Sheldon et al. [87], [113] employed intelligent heuristics to minimize the number of synthesis execution runs for application specific tuning of Xilinx's Microblaze (MB)

processor. In [87], the authors proposed the use of an Application-Specific Impact Order Search Tree for determining the best processor configuration. Initially, the impact value of each MB functional unit instantiation is obtained from synthesis/execution runs. Subsequently, the functional units are chosen in order of decreasing impact. At every step a synthesis/execution run is performed and a functional unit is chosen if it leads to increased performance. In [87] and [113], the authors need to perform a few synthesis/execution runs, each of which can take tens of minutes. Although their approaches scale well with increasing number of configurable options, they still require about 3 hours to find a solution from 12 configurable options using their best method. In addition, the quality of the results with large number of configurable options is not guaranteed.

Yiannacouras et al. in [86], [106] described a SPREE based framework to quantify tradeoffs within soft processor microarchitecture and explored the impact of tuning the microarchitecture to the application. They also applied the technique of Instruction subsetting to reduce the CPU implementation area by removing the support for instructions which are not needed by the application. The microarchitecture parameters considered were Pipelining depth, Shifter implementation etc. There can be a severe penalty with such a customization when the addition of Custom Instructions is used to further customize the processor. Since a soft-core processor is implemented on a FPGA, it is an obvious choice to add custom instructions to increase its performance. In such a situation the highly customized pipeline implementation for the core ISA needs to be re-evaluated to account for the latency of the new custom instructions. Hence in our investigation, keeping in mind these future possibilities, the primary focus is on the instruction subsetting and optional resource selection rather than microarchitecture tuning. Secondly, the instruction subsetting considered in their work is not practically possible to use for commercial off the shelf soft-core processors. They have used SPREE to generate the RTL description of processors, given

the target ISA and datapath as inputs. Hence, in their work Subsetting implies removing the datapath and control logic for unused instructions before generating the processor's RTL description. This approach is not feasible for a commercial processor. The RTL description is also simulated to measure the total number of cycles for executing the application. Hardware synthesis is also performed to measure the area, clock frequency, and power of the generated soft processor.

Anderson used genetic algorithm approach for design space exploration using parameterized cores [173]. They generated Pareto-optimal points for different configurations of the NIOS processor using a genetic based SEAMO algorithm.

As discussed in the earlier, Sheldon et al. [87] and Yiannacouras et al. [86] showed with Microblaze and NIOS processors respectively that application specific customization of these soft core processors can lead to significant area savings with better performance, at times even compared to the fully blown CPU. They developed methodologies to automatically select the best configuration for a given application with restricted number of synthesis/execution runs. However, as the number of configurable parameters increase with modern soft core processors, these methodologies which necessitate time-consuming synthesis and execution runs for each configuration option become impractical. For example, an exhaustive search for a suitable Microblaze configuration can take up to 11 hours [87]. The LEON processor has approximately 90 parameters [88] to choose from and about 5×10^{24} possible configurations.

Hence there is a need to develop faster design exploration methodologies that can assist designers in choosing the best processor configuration for a given application. In this chapter a framework is proposed for design exploration of processor configurations, which does not require time consuming synthesis and execution runs for each configuration option. Instead, it relies on a single-pass compilation output of the open

Figure 3.1: Design Exploration Framework.

source LLVM compiler infrastructure to predict the performance and area trends when different functional units are selected for the soft core processor. The tool exploration time is within seconds compared to tens of minutes of synthesis runtime in previous methods. The subsequent sections explain the proposed method.

3.3 Proposed Framework

Figure 3.1 shows the proposed design exploration framework for application specific customization of soft core processors. Similar to [87], it is assumed that all the configuration parameters affect the system performance independently. This implies that the resulting gain and cost incurred by instantiating two components is the sum of the gain and cost of the individual components. This assumption does not limit the effectiveness of our estimation tool because as shown in [87], the dependence of different parameters is negligible. As shown in Figure 3.1, the framework has three major

phases: 1) Pre-characterization Phase, 2) Architecture-independent Phase, and 3) Application-aware Design Exploration and Customization.

3.3.1 Pre-Characterization Phase

The pre-characterization phase is a one-time process that needs to be performed for each target processor. In this phase, the configurable functional units of the target processor and the corresponding instructions in the processor's Instruction Set Architecture (ISA) which can benefit from these units, are identified. For example in the NIOS II processor, the hardware Integer Multiplier unit corresponds to the following instructions: *mul*, *muli*, *mulxss*, *mulxsu*, *mulxuu*. Similarly, the instructions *div* and *divu* benefit from the hardware Integer Divider. This step is independent of the applications that run on the processor and hence needs to be performed only once for a target processor.

Next, the execution time of these optional instructions in software and hardware are obtained from either the datasheet or through test applications that are subjected to the tool-chain. Software execution time is the number of clock cycles required to implement the optional instruction on the base processor when the corresponding functional unit is unavailable. Hardware execution time is the number of clock cycles required by the hardware functional unit to implement the corresponding optional instruction. In addition, the hardware area required to implement the configurable functional units is also determined.

Finally, the corresponding instructions in the LLVM virtual instruction set that implement the optional instructions are identified. Figure 3.1 shows an example of the information pertaining to the optional instructions of a particular soft-core processor that are stored in a table. It can be observed that these information include the configurable functional units, corresponding LLVM instructions, hardware area of the

configurable functional units and their software/hardware execution time. Note that different configurable functional units is associated with different hardware/software execution time. For example, the NIOS II processor executes the multiplication operation approximately 10 times faster in hardware than in software, while the floating point operations are executed approximately 12 to 15 times faster in hardware.

3.3.2 Architecture-Independent Phase

In this phase, the given application is first compiled and analyzed using the open-source LLVM compiler without any reference to the target processor. An application C Code is compiled using LLVM to produce the LLVM IR (Intermediate Representation). It should be noted that the LLVM IR uses the native LLVM virtual instruction set. A “LLVM Pass” has been written to parse the IR in order to identify the function, basic block, instructions in these basic block, their arguments and their types. Next the application is profiled using the LLVM-profiler (which uses the LLVM Just in Time Compiler) to obtain the basic block execution counts. At the end of this phase, all the optional instructions in an application along with their execution count are obtained.

3.3.3 Application-Aware Design Exploration And Customization

In this final phase, the information obtained in the pre-characterization step and the architecture-independent step is used to explore the design trade-offs of various processor configurations. In particular, equation (Eq. 3.1) is used to calculate the hardware/software execution time (in clock cycles) of each of the optional instructions identified in the architecture-independent phase in order to predict the performance gain of various processor configurations for a given application.

$$T_{fu} = [x_{fu} * T_{fu}^{hw} + \bar{x}_{fu} * T_{fu}^{sw}] * F_{fu}, \quad (\text{Eq. 3.1})$$

In equation (Eq. 3.1), T_{fu} is the total execution time for executing the particular instruction, x_{fu} is 1 if the functional unit is enabled and 0 if disabled. T_{fu}^{hw} is the time (in clock cycles) for executing the instruction if the hardware functional unit is present, whereas T_{fu}^{sw} is the time (in clock cycles) for executing the instruction using software libraries. F_{fu} is the frequency of the instruction that benefit from the corresponding functional unit.

It is noteworthy that the execution time of the instructions that cannot benefit from the configurable functional units is not estimated in this work as they do not lead to notable differences across the various configurations. It will be shown in the experimental results section that restricting the performance estimation to only the optional instructions can sufficiently enable the designers to choose the most suitable configuration for a given application. Restricting the performance estimation to only the optional instructions also results in reduced complexity in the various phases of the proposed framework.

In addition to performance estimation, each configuration is annotated with the corresponding hardware area of the functional units used. Hence, for a given application, a performance-area plot can be obtained that describes the design trade-offs for different processor configurations. This performance-area plot assists the designer in identifying the suitable application-specific processor configuration under a given performance-area constraint, without the need to undergo time-consuming synthesis and execution runs. For example, an application that does not contain significant number of floating point division instructions, will not benefit from a floating point divider unit in the processor. Thus, the support for floating point division can be removed while instantiating the processor without a significant loss in application performance. However, this results in a significant reduction in implementation area that can be utilized more efficiently for other processing units such as custom instructions or hardware accelerators.

After deciding the suitable processor configuration, the designer can then enable/disable the corresponding functional units during the processor specification phase in the target processor. This is the ISA subsetting step as shown in Fig 3.1, which enables us to effectively remove the support for instructions that are never used or do not lead to significant gains in the application. Alternatively, the performance-area plot can be incorporated in frameworks such as SPREE to rapidly explore the large design space in order to determine the best configuration before generating the RTL description of the customized processor.

Although the experimental results in this thesis is based on the NIOS II processor, the proposed framework is generic enough to be applied to any target processor. This also allows us to use the proposed framework to select the most suitable processor and the corresponding configuration for a given application from a pool of processors.

3.4 Experiments And Results

This section presents the experimental results from this work. The experiments are based on the NIOS II Processor from Altera [85] that is targeted on the STRATIX II device. The applications used in our experiments are taken mostly from two popular benchmark suite, MiBench and MediaBench. The applications from the MiBench suite (i.e. basicmath, fft and CRC32) are less compute intensive, and hence they do not need most of the configurable functional units. Therefore, Instruction Subsetting can lead to substantial area reduction for such applications. The applications from the MediaBench suite (i.e. MPEG-2 Encoder, MPEG-2 Decoder, G721 Encoder, EPIC Encoder and EPIC Decoder) are quite computationally intensive and hence they require most of the configurable functional units. In addition, a handwritten application for the raytrace algorithm was also tested with the tool.

In the NIOS II (fast core) processor, the configurable units considered in our work

Table 3.1: Optional Instructions' Count.

Application	FD	FU	ID	IM	DI
Basicmath	0	0	0	1.98E5	1.19E5
FFT	0	1.63E4	0	4.09E3	3.76E5
CRC32	0	0	0	0	0
Raytrace	0	1.03E2	0	0	248
Mpeg2 Enc	0	0	3.58E6	3.91E7	2.34E7
Mpeg2 Dec	0	0	2.48E5	1.23E6	6.34E4
G721 Enc	0	0	0	0	3.17E6
Epic Enc	2	2.75E6	8.34E4	3.84E6	2.85E6
Epic Dec	0	8.54E3	13	4.37E4	1.31E5

FD = Floating point Divide, FU = Floating point Add/Sub/Mul,
ID = Integer Div, IM = Integer Mul, DI = Double precision Instructions.

are Integer Multiplier (IM), Integer Divider (ID), single precision Floating Point Unit (FU) and Floating point divider unit (FD). At present, cache dependency has not been considered in this work. Instead, the largest possible cache was instantiated on the device. Selection of cache sizes can be considered in the future expansion of this work. Table 3.1 lists the number of optional instructions that corresponds to the configurable functional units in the benchmark applications. It is evident from the table that the applications from MiBench do not contain most of the optional instructions while MediaBench applications have a diverse combination of these instructions.

During the experiments, an important observation was made regarding the usage of the configurable functional units. Most of the soft core processors have an optional IEEE 754 compliant Floating point unit, which when instantiated, computes the *single precision* floating point operations in hardware. The double precision operations on the other hand are always executed using the software libraries. The FU is not used for such operations. It was noted that in the NIOS II processor, the double precision instructions utilizes the hardware integer multipliers (i.e. IM). This observation has been validated using hand written codes with randomly generated inputs. The execution time for the double precision instructions is approximately 3 to 5 times faster when

a hardware integer multiplier unit is present. This is an important observation, as it demonstrates that some of the configurable units can be used indirectly, and must be taken into account during the estimation process. The proposed framework can successfully capture such subtle anomalies during the pre-characterization phase. In addition, a procedure was included in the design exploration stage of the framework to estimate the number of double precision instructions.

The graphs in Figure 3.2 show the performance-area plots of different processor configurations for the applications. The Y-Axis is the execution time (number of clock cycles) and the X-Axis is the approximate hardware area incurred for different configurations. There are two lines in each graph. The dotted line is the estimated execution time of the optional instructions from our framework. The second line shows the actual performance of the application on different configuration of the processor. In order to get the actual performance, the NIOS II performance counter macros were used in the application code. This macro is minimally intrusive and thus the overhead added is extremely low compared to the application runtime. The different possible processor configurations were synthesized and the applications were executed on them to get the execution count on each of the configurations. Since there are four configurable functional units in our case (i.e. IM, ID, FU and FD), the estimation was performed for 16 configurations. However in the NIOS II architecture, it is not possible to instantiate FD without instantiating FU. This leaves us with 12 possible configurations. This does not limit the estimation framework as it can be used effectively to predict the performance of all the 16 possible configurations.

The numerical value of the points on the dotted line in the graphs is obviously much lower than the actual implementation results, as the estimation is performed on the optional instructions only, whereas the actual implementation results consist of execution time of the entire application. It should be noted that when comparing the two lines, the performance-area *trends* (rather than the absolute values) should be

observed as they provide sufficient information to enable the designer to choose the most suitable processor configuration for a given application. It can be observed from the graphs in Figure 3.2 that the proposed strategy for restricting the performance estimation to only the optional instructions is well-justified, as this can reliably predict the performance-area trends of various processor configurations.

In Figure 3.2.(a) and (b), Basicmath and FFT have a zigzag behavior. This is due to the fact that for these applications, the only significant amount of optional instructions is the integer multiplication. Hence whenever the hardware integer multiplier unit is instantiated, there is a significant dip in the execution count. Applications like CRC32, do not contain any optional instructions, hence the proposed framework outputs a flat performance-area curve. This is consistent with the actual implementation results, wherein it can be observed that the performance of the CRC32 application remains almost the same for different processor configurations. It is also noteworthy that for the actual implementation, the execution count can never be the same even in consecutive runs of the application. Hence, the minor fluctuations in the actual performance of the CRC2 application can be safely ignored.

Overall, it can be observed that the proposed framework can reliably predict the performance/area trends of different processor configurations. In addition, it can be observed that the proposed framework performs best when predicting the advantages of different configurable functional units for the MediaBench applications like EPIC Encoder and Decoder (Figure 3.2.(h) and (i)), which contain a good mixture of the optional instructions. These results demonstrate the advantages of the proposed framework as a means for rapid design exploration of processor configurations.

3.2.a: BASICMATH (MiBench)

3.2.b: FFT (MiBench)

3.2.c: CRC32 (MiBench)

3.2.d: RAYTRACE (Handwritten)

3.2.e: MPEG2 ENCODER (MediaBench)

3.2.f: MPEG2 DECODER (MediaBench)

3.2.g: G721 ENCODER (MediaBench)

3.2.h: EPIC ENOCODER (MediaBench)

3.2.i: EPIC DECODER (MediaBench)

Figure 3.2: Performance vs. Area Curves for Benchmark Applications.

3.5 Summary

In this chapter, an efficient design exploration framework was proposed for rapid customization of soft core processors through instruction subsetting. As opposed to the existing methods, the proposed approach only relies on the front-end compiled output of an application from an open-source compiler infrastructure for compilation and analysis, it is not restricted to a particular target platform. Only a one-time pre-characterization step is required for each new target platform to capture the relevant micro-architectural information as described in this chapter. The proposed strategies in the pre-characterization step have been shown to successfully capture subtle points like nested dependence of the functional units. In addition, the runtime of the framework is only in the order of seconds as the performance-area estimation of different processor configurations can be achieved without undergoing time-consuming

synthesis and execution. The output of the proposed framework is a performance-area plot, which can assist the designer in performing instruction subsetting to save area without any compromise in performance.

The next step of customization of a processor can be done through augmenting its base instruction set architecture with application-specific custom instructions for increased performance. This step is typically referred to as Instruction Set Extension and is discussed in detail in the next chapter.

Architecture-aware Custom Instruction Generation

4.1 Introduction

In the previous chapter a method was described to customize a processor through instruction subsetting for increased area efficiency. Instruction Subsetting allows a designer to configure a processor for specific application by disabling the hardware support for un-used functional units. On the other hand Instruction Set Extension (ISE), which bolsters the base instruction set architecture of a processor with application-specific custom instructions, helps a designer to further customize a processor to achieve higher performance. In this chapter, methodologies are presented for generating efficient custom instructions (without memory or control operations).

In section 2.2.1.2 of chapter 2, a brief discussion was presented on the existing instruction set extension methodologies. These existing work have proposed methods to accelerate the identification and selection processes. During the identification process, certain target constraints are used to prune the design space. However, the implementation constraints of custom instructions on the target FPGA hardware is typically ignored during the ISE process in the existing work. In this chapter, traditional custom instruction generation methods are discussed briefly to show that dis-

regarding the characteristics of the target FPGA architecture during the initial phases of the ISE process results in custom instructions that cannot be mapped efficiently on the FPGA architecture. We then introduce the concept of FPGA-aware enumeration of custom instruction candidates wherein the pattern enumeration phase uses the target architecture information as additional pruning constraints. The enumerated custom instruction candidates are passed through the selection phase to finally select a set of high performance custom instructions for a given area constraint.

Section 4.2 elaborates some of the existing work in this area and highlights their limitations. Section 4.3 introduces the proposed instruction set extension methodology that incorporates architecture information during the enumeration stage. This is followed by a discussion of the proposed pattern enumeration and selection algorithms to determine custom instructions for FPGAs with tight area constraint. Custom instructions generated using these concepts have been termed as Fine-Grained Custom Instructions (FGCIs) in this work due to their small size. We provide the results of our findings in section 4.6 and highlight the pros and cons of the approach. Next, an extended enumeration strategy is proposed to identify profitable custom instructions for FPGAs with arbitrary area constraints. In addition to the FGCIs, the extended enumeration strategy also identifies large patterns called Coarse-Grained Custom Instructions (CGCIs) that exhibit a high degree of Instruction Level Parallelism (ILP) which can be mapped efficiently using the FPGA logic blocks. A new selection heuristic is then proposed to facilitate the selection of large custom instructions with low critical path, in order to achieve higher performance. Section 4.10 details the experiments with this set of custom instructions followed by a summary for this chapter.

The work discussed in this chapter has been published in [7], [8] and [9].

4.2 Existing Custom Instruction Generation Methodologies

As stated earlier, instruction set extension is typically performed in two steps, pattern enumeration/identification and pattern selection. Pattern Enumeration is defined as the process of identifying all the legal subgraphs within a application Data Flow Graph (DFG). These “legal” subgraphs formed by a connected set of “legal” or “valid” nodes are called “patterns”. The enumerated patterns of an application must obey a set of constraints to be compatible with the micro-architectural constraints of the underlying architecture.

Atasu et al. in [94] listed some of these constraints as follows.

- Number of inputs of the subgraph should be less than a user defined value N_{in} ,
- Number of outputs of the subgraph should be less than or equal to a user defined value N_{out}
- The subgraph must be convex.

The user defined values N_{in} and N_{out} signify the number of read and write ports available for the custom instruction in a given architecture. The convexity constraint on the other hand is needed to ensure a feasible scheduling of the identified pattern. In order for a pattern to be convex, there should not be a connection between two nodes within the pattern through a node which lies outside the pattern. Figure 4.1 shows an example of a non convex pattern where there is a connection between nodes 4 and 0, through node 3 which is not inside the pattern. Hence the selected pattern is not feasible to be scheduled.

The work in [94] described a branch and bound algorithm that enumerates all the legal patterns from an application DFG, while pruning off the ones, which violate

Figure 4.1: A nonconvex graph with unfeasible scheduling

the fundamental micro-architectural constraints. Their method produced significantly better performance when compared to the state of the art techniques at that time. However, the complexity of the branch and bound algorithm in their work increases rapidly with the size of the DFG. Apart from the micro-architectural constraints such as number of inputs/outputs etc. which need to be adhered to during pattern enumeration, certain basic operations such as memory operations like LOAD/STORE and control flow are considered as invalid operations for custom instruction implementation. The inclusion of memory operations/accesses into custom instructions generally creates two problems [132]. Firstly, the custom instruction with memory access might have a non-deterministic delay, which is undesirable since custom instructions are invariably implemented as part of the processor's datapath. Secondly, this can also create memory hazards such as Cache-incoherence and need additional logic to prevent errors. The exclusion of control flow operations such as BRANCH also ensures a deterministic runtime for these custom instructions.

Pothineni et al. described a method to identify maximal convex subgraphs from an application DFG, without any I/O constraints [96]. They claimed that such patterns, not bounded by general I/O constraints, provide upto 50% performance benefits com-

pared to the exhaustive enumeration. They also claimed that such an approach leads to reduction in the enumeration tool runtime. In [123] the authors provided a new algorithm for faster enumeration of legal subgraphs in an iterative manner. Their algorithm run time was upto two orders of magnitude faster compared to the existing work. They enumerated the patterns in the increasing order of size, while relating the patterns with $(n + 1)$ nodes to the patterns with n nodes. Atasu et al. in [124] enumerate only maximal convex patterns or subgraphs, and achieved a performance increment of an order of magnitude, while having an extremely fast tool runtime.

Chen et al. presented a novel algorithm for rapid identification of custom instructions for extensible processors [97]. Their method performed exceptionally well especially for large DFGs, taking orders of magnitude less time compared to the state-of-the-art methods. Li et al. [125] proposed further enhancements in Chen's algorithm and thus achieving up to 50% faster tool runtime for enumerating patterns with single-output constraint.

After enumerating all the possible legal patterns from the enumeration phase, the pattern selection phase strives to choose a reduced set of non-overlapping patterns for final implementation. This step uses various graph covering algorithms to optimally choose the largest set of non-overlapping patterns in order to have the maximal cover.

Jason Cong et al. described a comprehensive set of algorithms for custom instruction enumeration, selection as well as mapping in [127]. The extended instruction set identified by their algorithms enables them to achieve a maximum of 3.73X performance benefit compared to the basic instruction set. Clark et al. presented a complete design framework for automatic identification of pattern candidates within a DFG and a compiler framework to take advantage of such custom units [174], [95]. Every pattern candidate was annotated with the latency and area information obtained from a hardware library. In the selection stage, a greedy heuristic was used

to select the candidate with largest (speedup/area) ratio before selecting any other patterns. The work in [175] used a similar approach, where the area and latency of every pattern candidate was estimated by synthesizing them on a $0.13\mu m$ CMOS process.

Bonzini et al. in [93] proposed a recurrence-aware covering algorithm which achieved tangible improvement over nonrecurrence-aware covering algorithms. In their work, they tried to find patterns which were both large and recurring instead of focusing on only one kind of patterns. This is an effective approach because it often provides the best results as opposed to heuristics that only consider frequency or the size of patterns. A similar heuristic was used during the Pattern Covering phase in our work.

Guo et al. in [176] proposed a fast and effective method for the template selection step. Basically the authors created a conflict graph using all the patterns in the pattern library. A conflict graph is an undirected graph where every node in the graph corresponds to a pattern in the pattern library. All the patterns in the library are used to create this conflict graph. An edge in the conflict graph between two pattern nodes, represents an overlap between the corresponding patterns in the original DFG. The objective of creating this conflict graph is to choose the non-overlapping pattern matches for final implementation. Their method to select non-overlapping candidates provides a way to choose the maximal independent set (MIS) in an application DFG and is used extensively in our work.

Lam et al. in [122] [126] presented a custom instruction generation design flow that incorporates a novel way for estimating the area of custom instructions in the final hardware by *clustering* the candidate patterns [122]. Their approach partitions every pattern into subgraphs so that each subgraph could be completely mapped onto a set of FPGA logic elements with the same hardware configuration. This approach helped

them to estimate the number of logic elements required to implement every pattern in the target FPGA. This work served as a motivating example for our novel architecture-aware enumeration algorithms which will be discussed later in this chapter.

The proposed pattern enumeration technique in this work extends the existing state-of-the-art methods proposed by authors in [97] and [125], which have been shown to be orders of magnitude faster than other existing algorithms for large DFGs. In particular, additional constraints have been incorporated, which ensures that the enumerated patterns can be mapped onto the target FPGA in a predictable manner. The next section presents the proposed instruction set extension methodology that incorporates the target architecture information during the enumeration phase to identify a set of architecture-aware custom instructions for the selection phase.

4.3 Proposed Architecture-Aware Custom Instruction Generation Methodology

Our instruction set extension methodology, as shown in Figure 4.2, incorporates the architecture information at the enumeration phase.

In particular, our enumeration algorithm identifies only patterns that can be fully mapped onto a single logic block of the target FPGA architecture. This strategy leads to the following advantages:

- The extra set of pruning constraints results in the identification of fewer patterns, thus reducing the enumeration runtime.
- The smaller set of enumerated patterns also reduces the problem size and runtime of pattern selection phase.
- As the hardware area-time of the custom instruction patterns from the enumeration step are implied, effective pattern selection can be performed to choose

Figure 4.2: Proposed Instruction Set Extension Methodology

a set of custom instructions that maximizes the utilization of the FPGA space. When compared to traditional design approaches, which typically focus on selecting large and recurring custom instructions, our experimental results show that the proposed approach leads to significant performance improvement over existing approaches especially for FPGA designs with tight area constraints.

The rest of this section highlights the advantages of incorporating the target architecture information in the enumeration step using the DFG example shown in Figure 4.3.

The patterns identified from the example DFG in Figure 4.3 are shown in Table 4.1. In this example, it is assumed that the I/O (Input/Output) constraints is 4/1, and trivial patterns with single nodes have been omitted. The second column lists

Figure 4.3: A data flow graph example

the seven patterns enumerated using a traditional enumeration method. The fourth column shows the three patterns obtained using the proposed architecture-aware enumeration technique. It can be observed that fewer patterns are enumerated with the proposed technique as it only identifies patterns that can be mapped entirely onto the FPGA logic block. The aim of this example is to show that the reduced set of patterns identified using the proposed method can still lead to selection of the most profitable custom instructions. As explained in [8], the performance (in terms of cycle savings) of the selected custom instructions can be calculated using the formula in equation Eq. 4.1.

$$cycle\ savings = \left[\sum_1^n NN * DO - \sum_1^n CLP * DO \right] \quad (Eq. 4.1)$$

where NN denotes number of nodes (or operations) in a custom instruction, DO is the number of occurrences of the custom instruction, n is the total number of selected custom instructions and CLP is the critical path of the custom instruction. The

Table 4.1: Enumerated Patterns using Traditional & Proposed approaches

Number	Traditional Enumeration		Proposed Enumeration	
	Patterns	Size*Occurrence	Patterns	Cycle Savings/Area (Logic Blocks)
1	2,4 {add, add}	$2*2 = 4$		$0/2 = 0$
2	2,4,7 {add, add, add}	$3*1 = 3$		$0/3 = 0$
3	4,7 {add, add}	$2*2 = 4$		$0/2 = 0$
4	5,6 {and, or}	$2*1 = 2$	5,6	$1/1 = 1$
5	5,6,7 {and, or, add}	$3*1 = 3$	5,6,7	$2/1 = 2$
6	6,7 {or, add}	$2*1 = 2$	6,7	$1/1 = 1$
7	2,4,5,6,7 {add, add, and, or, add}	$5*1 = 5$		$2/3 = 0.66$

first term in the above equation represents the software execution time of a custom instruction whereas the second term depicts the number of clock cycles required to execute the custom instruction in the hardware. Based on this equation, the performance achieved per unit area (in terms of FPGA logic blocks) is calculated for every pattern as shown in the last column of Table 4.1.

A commonly-used selection heuristic which is based on the size and occurrences of the patterns is shown in column 3 of Table 4.1. Based on this heuristic, an algorithm which selects the enumerated patterns from the traditional approach will have a high preference for pattern 7. Next, assume that the proposed architecture-aware method enumerates the pattern as shown in column 4 of Table 4.1 (Section 4.4 discusses how these patterns are enumerated and the rest are pruned by the proposed approach). Using the same selection heuristic as before (column 3 of Table 4.1), an algorithm which maximizes the heuristic function will have a high preference for selecting pattern 5. Column 5 shows the cycle savings per unit area for all these patterns. It can be observed that when compared to the traditional approach, the proposed method can lead to the selection of custom instructions that achieves sig-

nificantly better performance per unit area (i.e. pattern 5 has a performance-area ratio of 2, while pattern 7 has a performance-area ratio of only 0.66). In section 4.6 it will be demonstrated, with experimental results, that the proposed strategy performs exceptionally well in resource constrained designs as it is capable of maximizing the utilization of the FPGA space.

It is also possible to enumerate all the patterns using an existing pattern identification method and then obtain the hardware area-time measures of each pattern (either through hardware estimation or synthesis) in order to perform hardware-cognizant pattern selection. However, the disadvantage of such approach is that the enumeration process would be more time consuming as many redundant patterns are identified. The proposed method incorporates efficient hardware estimation rule-sets as part of the pruning constraints in the enumeration phase, thus reducing the enumeration runtime as well as the problem size for pattern selection. Using the concepts of architecture-aware enumeration, two types of custom instructions are identified as discussed in the next subsection.

4.3.1 Fine-Grained And Coarse-Grained Custom Instructions

As described above, the information regarding the structure of the Lookup Table (LUT) in the target FPGA device is used during enumeration to identify patterns that can be mapped most effectively on the FPGA lookup tables. In this work a *logic block* is defined as a set of 32 FPGA basic logic elements with identical hardware configuration that implements a 32-bit wide custom instruction. In the first part of this work, only patterns that can fit into exactly one logic block of the target FPGA device were identified. Such custom instructions have been termed as Fine-Grained Custom Instructions (FGCI), due to their relatively smaller size so as to be able to fit into one logic block. FGCI help to maximize performance in designs with strict area constraints.

Subsequently, larger patterns with high instruction level parallelism but smaller critical path, were also identified. These patterns are useful for designs with relaxed area constraint. Such custom instructions have been termed as Coarse-Grained Custom Instructions (CGCI) in the remainder of this work.

The following sections present the steps involved in the proposed architecture-aware pattern enumeration and selection methods for both types of custom instructions.

4.4 Fine-Grained Custom Instruction Enumeration

As stated earlier most of the major works in the area of pattern enumeration have considered, though in a limited way, the architecture information during the enumeration process. In all these work, the primary constraints used for selecting a legal sub-graph or pattern is the number of input and output ports, convexity of the pattern, and legal operations in the pattern. This work introduces additional architecture specific constraints during pattern enumeration to limit the number of enumerated patterns to those that would lead to efficient mapping on the target FPGA. It will be shown that even with a reduced set of enumerated pattern candidates, profitable patterns can be selected later in the pattern selection phase.

Similar to [122], Trimaran [177] is used to compile the C Code of the benchmark application to an intermediate level. This intermediate output of the Trimaran compiler is called a “Rebel” file which is used as the input to our enumeration stage. The Trimaran compiler is also used to profile the application in order to identify the most frequently executed basic blocks of the application. The “Rebel” file from Trimaran provides the application representation that consists of the basic operations of the Trimaran instruction set architecture. The basic operations of Trimaran can be classified into three main groups of instructions:

- logical operations e.g. and, or, xor etc.;
- shift operations e.g. shl, shr etc.;
- arithmetic operations e.g. add, sub, mult, div etc.

Lam et al. in [122] showed a novel method of estimating the area of a given pattern. They applied various architecture specific constraints to ensure that a candidate pattern can be fully implemented within a logic group (a set of FPGA logic elements with the same hardware configuration) on the Xilinx Virtex 4 FPGA. In particular, after the pattern selection phase, they partitioned each of the selected patterns into (clusters), wherein each cluster could be implemented in one logic group. This enabled them to estimate the total number of logic groups required to implement the selected patterns on the target architecture. Lam et al. used two sets of rules to partition a pattern into clusters. They can be listed as follows:

- The rules listed in the first set are used to decide if a graph node can be incorporated inside a pattern in the identification step. A graph node can be incorporated in a pattern if:
 - (i) It is a logical or shift operation and the cluster does not consist of any arithmetic operations (e.g. Cluster 1 in Figure 4.4). The programmable K-LUTs (K-input LUTs) can implement non-arithmetic functions of up to K inputs.
 - (ii) It is a logical operation that is executed before a single add/sub operation in the cluster (e.g. Cluster 2 in Figure 4.4). The add/sub operation can be mapped onto the LUT-based logic elements to exploit the fast carry chains as shown in Figure 4.5. Since the output of the LUT is the partial sum ($A \oplus B$), all logical operations must be executed first to generate the required operands (i.e. A and B) for the add operation.

- (iii) It is a shift operation and the cluster contains a single add/sub operation (e.g. Cluster 3 in Figure 4.4). The shifted value of an addition/subtraction result by a constant can be realized by configuring the routing architecture to feed the suitable data range as inputs to another logic element.
 - (iv) It is an add/sub operation and the cluster does not contain any other arithmetic operations. A LUT-based logic element can only implement a single add/sub operation effectively by exploiting the fast carry chain. Other arithmetic operations (e.g. multiplication, division, etc.) are mapped to the CGAUs.
- The second set of rules is used to determine the actual number of inputs and outputs of a candidate pattern when it is implemented in the logic elements:
 - (i) The total number of cluster inputs is at most K with only one output. The maximum number of input ports and output ports for a LUT is K and 1, respectively.
 - (ii) One of the inputs to the add/sub operation must be directly connected to an external input of the cluster (e.g. Cluster 2 in Figure 4.4). As shown in Figure 4.5, the carry-out (i.e. C_{out}) is selected from either the carry-in (i.e. C_{in} or an input operand (i.e. A) that is also fed directly to the input pin of the LUT.

A more detailed description of these set of constraints can be found in [122]. These rules were incorporated along with the previously mentioned micro-architecture constraints such as number of input/output constraints, convexity, etc. during our enumeration phase in order to enumerate only “clusters” or “Fine Grained Custom Instructions (FGCI)” as they are termed in this work. This ensures that all the enumerated patterns could be implemented in the logic blocks of the FPGA. It is noteworthy

Figure 4.4: A pattern candidate with 3 Fine Grained Custom Instructions (FGCI)

that although the experiments were performed based on a Virtex 4 FPGA, the idea is generic enough to be used with other FPGA families.

The next subsection briefly describes the enumeration algorithm used in [125] and explain the modifications made in order to incorporate the target architecture information.

4.4.1 Enumeration Algorithm

It was mentioned earlier that the enumeration algorithm used in this work is based on the algorithms proposed in [97] and [125]. This enumeration algorithm works in the following manner: It searches for valid patterns, starting with an empty pattern P , the convex DFG G of a basic block and a redundancy guarding node referred to as r_g . The algorithm recursively invokes the enumeration procedure, which consists of three functions, *select_node*, *unite* and *split*. Function *select_node* returns a selected

Figure 4.5: Implementation of *add* operation in a logic element

node from the remaining node set ($G-P$). The function *unite* handles the condition of addition of valid nodes into the pattern to create a new pattern, while the *split* function handles the situation where the selected node cannot be added to the pattern. The function *unite* merges multiple nodes if the node selected by the function *select_node* is of high quality, which accelerates the search process. It is in this *unite* step, wherein the proposed additional constraints can be checked during the merging of a selected node with the existing pattern. Meanwhile, the function *split* decomposes the current DFG $G(V,E)$ into one or two sub-graphs by splitting them when necessary, thus reducing the depth of recursive search. Using this methodology the output of the *unite* phase always produces a valid pattern which is ready for the selection stage. The details of the algorithm can be found in [97] and [125].

In order to enumerate patterns in an FPGA-aware manner, further constraints have been added to the *select_node* and *unite* functions of [125]. The modified *select_node* function will utilize the types of nodes that are already present in the existing pattern to identify another node that can be merged, while the new *unite* function

checks for additional constraints while merging the selected node (or the pattern consisting of the selected node) with the existing pattern. For example, since the Virtex-4 FPGA cannot implement more than one ADD/SUB operation in a single logic block, a valid pattern can only contain at most one ADD/SUB operation. To ensure this, the modified `select_node` function checks the existing pattern for ADD/SUB operations and will not return a new node which is an ADD/SUB operation, if the existing pattern already has such an operation. Similar changes were made to check for the other rules as well. The modified `select_node` function is called `mod_select_node` and the pseudocode is shown in Algorithm 1.

The first step in `mod_select_node` function checks the nodes in the existing pattern “P” and classify them into specific categories e.g. arithmetic, logic operations, etc. The remaining `mod_select_node` function works similar to the way described in [125], except that it now incorporates additional checks to return suitable nodes to the new unite function for valid pattern (FGCI) generation. In particular, it checks the existing pattern for I/O violations. If the I/O constraints are violated, the function selects a suitable node from the successor or predecessor set (for output and input violation respectively), in order to resolve these violations. The `get_output_resolve_nodes` and `get_input_resolve_nodes` functions return suitable nodes to resolve the output and input violations in S_O and S_I respectively. These selected nodes are verified against the FGCI generation rule-set before being returned as valid nodes. If no such nodes can be found, then the function returns a NOT_FOUND value and the search branch is terminated. On the other hand, if there is no I/O violation in the existing pattern, `mod_select_node` function selects a suitable node from the disconnected set keeping in mind the hardware estimation rule-sets. If a suitable disconnected node can be found, it is selected and returned. Otherwise, the function selects a suitable node from the connected set and returns it if the node conforms to the FGCI generation rule-set.

Algorithm 1 Modified Select Node

```
1: mod_select_node(P, G,  $r_g$ ) {
2:
3:   Classify the nodes of the pattern “P” based on node operations
4:    $flag := 0$ ; // flag used to avoid re-calculating the resolvable nodes
5:
6:   if ( $|OUT(P)| > OUT\_LIMIT$ ) && ( $|IN(P)| > IN\_LIMIT$ ) then {           //
   denotes both IO violation
7:      $S_O := get\_output\_resolve\_nodes()$ 
8:      $S_I := get\_input\_resolve\_nodes()$ 
9:     if ( $S_O == \{\}$ ) || ( $S_I == \{\}$ ) then
10:      return NOT_FOUND
11:    end if
12:     $flag := 1$ ;
13:  end if
14: }
15: if ( $|OUT(P)| > OUT\_LIMIT$ ) then {           // denotes output violation in P
16:   if ( $flag == 0$ ) then
17:      $S_O := get\_output\_resolve\_nodes()$ 
18:     if ( $S_O \neq \{\}$ ) then
19:       return node in  $S_I$  or an invalid node in  $Pred(G,P)^*$  that does not violate
   the cluster rules;
20:     else
21:       return NOT_FOUND
22:     end if
23:   end if
24: end if
25: }
26: if ( $|IN(P)| > IN\_LIMIT$ ) then {           // denotes input violation in P
27:   if ( $flag == 0$ ) then
28:      $S_I := get\_input\_resolve\_nodes()$ 
29:     if ( $S_I \neq \{\}$ ) then
30:       return node in  $S_I$  or an invalid node in  $Pred(G,P)^*$  which does not
   violate the cluster rules;
31:     else
32:       return NOT_FOUND
33:     end if
34:   end if
35: end if
36: }
```

The new *unite* function also checks the existing pattern before merging the pattern (consisting of the selected node) with the existing pattern. Using the same example

```

37: if ( $Disc(G, P)^* \neq \{\}$ ) then           // denotes that there is no I/O violation in P
38:     return node in  $Disc(G, P)^*$  which does not violate the cluster rules;
39: end if
40: if ( $Pred(G, P) \cup Succ(G, P)^* \neq \{\}$ ) then
41:     return node in  $(Pred(G, P) \cup Succ(G, P))^*$  which does not violate the cluster
        rules;
42: else
43:     return NOT_FOUND
44: end if

```

*Note: if ($r_g \neq NOT_A_NODE$), each of these sets is intersected with $Pred(G_0, r_g)$ where G_0 is the DFG containing all the nodes of the basic block.

as before, if the existing pattern already contains an ADD/SUB operation, the *unite* function will not merge the existing pattern with a pattern that contains an ADD/SUB operation. This limits the number of patterns that can be enumerated. The other rules were similarly incorporated in the new *unite* function. We call this modified unite function as *fgci_enum* and the pseudo code is shown in Algorithm 2. The *fgci_enum* function deviates from the algorithm described in [125] by incorporating an additional check for valid FGCI generation.

This additional check is achieved by a new function *FGCI_evaluation*, which is described in Algorithm 3. This pseudo code, along with the required I/O constraints validation, can be used to test all the node constraints mentioned in [122] for a given pattern. It works by first arranging all nodes in a given pattern in a topological order and then traversing the pattern in that order, checking for various rules needed to ensure that all the operations of the pattern can fit into a single logic block. For example, since a logic block in Virtex 2/4 (even the newer FPGA such as Virtex 5/6/7) has single carry chain, it does not allow patterns with two arithmetic operations. Similarly, since the last stage of the LUT (Figure 4.5) is the partial sum ($A \mathbf{xor} B \mathbf{xor} C$), all *logical* or *shift* operations must be executed first to generate the required operands (i.e. A and B) for the *add* operation. Therefore, the algorithm does not allow a pattern with *logical* or *shift* operation after an arithmetic operation to pass as a valid FGCI. The variables

Algorithm 2 Algorithm for FGCI Enumeration

```
1: fgci_enum(P, G,  $r_g$ ) {
2:
   
$$P' = \begin{cases} P \cup u \cup (Succ(G, u) \cap Pred(G, P)) & \text{if } "u" \in Pred(G, P) \\ P \cup u \cup (Pred(G, u) \cap Succ(G, P)) & \text{if } "u" \in Succ(G, P) \\ P \cup u, & \text{if } "u" \in Disc(G, P) \end{cases}$$

3: if (( $u \in Disc(G, P)$ ) && ( $Succ(G, u) \cap Succ(G, P) == \{\}$ ) && ( $OUT\_LIMIT == 1$ )) then
4:   return;
5: end if
6:
7:  $r_g = NaN$ ;
8: if (( $fgci\_evaluation(P') == 1$ ) &&  $in\_check(P')$  &&  $out\_check(P')$ ) { then //
   function to check FGCI node rules and IO constraints
9:   patterns_add (P');
10:  enumerate (P', G,  $r_g$ ) ;
11: }
12: else
13:   return
14: end if
```

such as `has_arith_op` etc. are used to denote if a pattern has an arithmetic operation etc. In the final step, the patterns enumerated from the `fgci_enum` function are tested for I/O constraints conformity. This step is especially useful for patterns containing "Shift Operations" since the inputs to a Shift Operation are handled differently in a FPGA [178].

It should be noted that the approaches proposed in this chapter is targeted towards a low end device with only 4-inputs LUT with a single carry chain such as the ones in Xilinx Virtex 2 or 4. The ideas presented in this paper can be easily extended to current and future generation of devices such as Xilinx Virtex 5/6/7 that contain 6-input LUTs with similar carry chain connections [179] that can either implement a single 6-input function or dual 5-input functions. The larger LUTs in newer FPGA devices typically enable more functionality in a single Logic block, which potentially leads to better performance-area ratio. Therefore, in order to target a different FPGA

Algorithm 3 FGCI Evaluation

```
1: fgci_evaluation(pattern) {
2:   Sort the nodes in the pattern in topological order.
3:   bool has_arith_op, has_logical_op, has_shft_op = FALSE;           // variables
   has_arith_op, has_logical_op, has_shft_op show whether or not a pattern has
   arithmetic, logical or shift operations respectively
4:
5:   for every node in the pattern do
6:     if node == arith_op then
7:       if has_arith_op == TRUE then    // any prior node in the pattern is an
       arithmetic operation
8:         return 0;    // FGCI not possible; FGCI cannot contain two arith ops.
9:       else
10:        has_arith_op = TRUE;
11:      end if
12:    else if node == logical_op then
13:      if has_arith_op == TRUE then    // any prior node in the pattern is an
      arithmetic operation
14:        return 0; // FGCI not possible; a logical operation cannot appear after
        an arith op
15:      else
16:        has_log_op = TRUE;
17:      end if
18:    else if node == shift_op then
19:      has_shft_op = TRUE;
20:    end if
21:  end for
22:  return 1;                                     // pattern is a valid FGCI
23: }
```

device, the number of Input/output ports allowed in a single cluster or FGCI need to be changed, thereby allowing the concept itself to be applied to any FPGA family across different vendors.

After enumerating all the valid FGCI, the selection phase begins. In this phase, the most profitable patterns are selected to be implemented as custom instructions.

The next section explains the pattern selection step in detail.

4.5 Fine-grained Custom Instruction Selection

Pattern Selection step is the second important phase in the instruction set extension process. After enumerating all the possible legal subgraphs from the enumeration phase, the Selection Process aims at finding a reduced set of non-overlapping patterns for final implementation. Similar to the work in [122] the selection phase is divided into smaller steps.

After enumerating all the possible legal patterns from the enumeration phase, a graph-subgraph isomorphism using “VFLib”[180] is performed to find the similar patterns and group them together. Each pattern group is defined as a “template”, whereby each template corresponds to a set of similar patterns. In the next step, pattern selection is performed to find a set of non-overlapping patterns for final implementation.

The pattern selection is based on the approach described in [176]. A conflict graph is created based on the enumerated patterns. A conflict graph is an undirected graph where every node in the graph corresponds to a pattern. An edge in the conflict graph between two nodes, represents an overlap between the corresponding patterns in the DFG. The objective of creating this conflict graph is to facilitate the selection of a set of non-overlapping patterns for final implementation. The following example explains this process.

Figure 4.6.a shows the different templates and corresponding patterns in a DFG. In particular, the DFG has three distinct templates, each of which consists of a set of similar patterns in the DFG. The corresponding conflict graph is shown in figure 4.6.b. In the conflict graph, the larger blocks show a template, whereas the internal smaller boxes depict the corresponding patterns. The strong black lines in the conflict graph show the overlapping of the patterns in the DFG.

4.6.a: A part of application DFG

4.6.b: Conflict Graph for the previous example DFG

Figure 4.6: Templates in a portion of a DFG and corresponding conflict graph

Once the conflict graph is created, the steps described in [122] are used to compute the local Maximum Independent Set (MIS) in order to obtain a set of non-overlapping patterns within each template. This aims to find the largest set of vertices in every template of the conflict graph that are mutually non-adjacent. In the existing state-of-the-art selection algorithms, various heuristics are used to compute the Local MIS. The three such popular heuristics are listed below:

- (i) Number of nodes in a template. In this heuristic, templates with largest number of operations are chosen iteratively in that order.
- (ii) Number of corresponding non-overlapping patterns of a template. This heuristic favors the templates which has the maximum number of non-overlapping patterns in the application DFG.
- (iii) Product of number of nodes in a template and the number of its corresponding non-overlapping patterns in the application DFG. This approach ensures that

both the size of the template as well as its frequency are given due consideration during the selection process. This heuristic generally gives the best results in terms of performance achieved [176] [93]. Proper weights need to be applied to the the product terms for best results. [176] showed that a power of 1.2 applied on number of nodes term yields the best results. In this work, a different value of this power term is used, which is determined experimentally.

Due to its obvious advantage over the other heuristics, the third heuristic is used for local MIS computation with a slight variation in the absolute value of the power term. After the local MIS is calculated using one of the previously mentioned heuristics, the templates are arranged in decreasing order of their MIS weight. In order to do a constraint-aware pattern selection, the template with the largest MIS weight is evaluated first. It is selected for implementation if the area constraint is not violated. If this template violates the area constraint, the template with the next largest MIS weight is evaluated and so on, until a suitable template is found and selected. Once a template is selected, the corresponding patterns of the template are implemented as custom instructions. These selected patterns are then removed from the conflict graph, and the remaining area for custom instruction implementation is updated. The algorithm repeats to find a new set of local MIS for each template. The algorithm terminates when either the conflict graph becomes empty or the area constraint is violated. Algorithm 4 shows the proposed pattern selection algorithm.

4.6 Experiments And Results

In this section results obtained using the proposed architecture-aware custom instruction design flow were compared to a traditional approach that aims to select large and recurring custom instructions without considering how they can be mapped onto the target FPGA. In the traditional approach, legal patterns are enumerated in the

Algorithm 4 Pattern Selection

```
1: pattern_selection{
2:
3:   remaining_area=area_constraint;
4:
5:   Compute local MIS of patterns with area less than the area_constraint;
6:
7:   Compute weight of these patterns based on a heuristic (no.of nodes*Number of
   local MIS);
8:
9:   Sort all the templates in decreasing order of MIS weight;
10:
11:  Find the template with the largest MIS weight;
12:
13:  if Area of this template is less than the remaining_area then
14:    Choose this template for implementation;
15:    remaining_area=(remaining_area)-(area of selected template);
16:    Remove the overlapping patterns from the conflict graph;
17:    Restore the patterns temporarily removed previously from the Conflict graph;
18:    Recompute the local MIS of the patterns left in the conflict graph and Proceed
    to Step 4;
19:
20:  else if Area of this template is greater than the remaining_area then
21:    Choose the template with the next largest MIS weight;
22:    if Area of this template is less than the remaining_area then
23:      Proceed with Step 8;
24:    else
25:      if No templates are left then
26:        exit;
27:      calculate Performance with the selected templates and matches;
28:    else
29:      Proceed with Step 14;
30:    end if
31:  end if
32: end if
33: }
```

first stage, and this is followed by pattern selection that attempts to select large and recurring custom instructions. The enumeration algorithm used in the traditional approach was taken from [97] and [125], without any extra hardware constraints during enumeration. The pattern selection step in the traditional approach uses the greedy

selection methodology described in [122] and [176]. A design exploration step is also implemented in the traditional approach that relies on solving the knapsack problem [127], [95] in order to identify the most profitable set of custom instructions (from the selected patterns) that meet the given area constraint.

The sample applications were taken from two widely used Embedded Benchmark suites. They are representative applications from various domains; SHA, Blowfish, Rijndael are from the Security suite of MiBench[181]; SUSAN and BASICMATH from the Automotive; PATRICIA from Networking; and lastly CJPEG, and EPIC are multimedia application from the MediaBench 2 [182] benchmark suite. Figure 4.7 compares the number of patterns enumerated by the traditional method with the proposed method (wherein only FGCI's are enumerated) discussed in Section 4.4 on page 74. Due to the inclusion of additional constraints, the number of patterns enumerated is reduced by approximately 39% on average. This also led to the reduction in the runtime of the enumeration phase by about 30% on average as shown in Figure 4.8. The runtime in this work has been measured using the 'gettimeofday' function in C. It is noteworthy that although the time saved by enumerating lesser number of patterns is partially offset by the need to evaluate additional architecture-aware constraints, there is still a significant saving in the overall runtime. A few applications such as RIJNDAEL do not show significant reduction in the number of enumerated patterns because these applications mostly consist of smaller patterns which can fit into one logic block of the target FPGA. This is an application characteristic and not the limitation of the proposed methodology. On the other hand, applications such as SUSAN or SHA achieve tremendous reduction in the number of enumerated patterns (88.08% for SUSAN and 64.28% for SHA), which aptly shows the advantage of the proposed strategy.

The percentage reduction in the runtime of pattern selection step using the method discussed in Section 4.5 is shown in Figure 4.9. In particular, when compared to the

Figure 4.7: Reduction in number of patterns enumerated

Figure 4.8: Reduction in runtime of the Enumeration Phase

Figure 4.9: Reduction in runtime of the Selection Phase

traditional approach, the proposed method achieves an average runtime reduction of about 41% and a maximum runtime reduction of about 88%. This reduction in runtime is due to the significant reduction in the problem size of the pattern selection step, as shown in Figure 4.7.

Next, the performance gain (in terms of cycle savings) obtained from the proposed design flow is compared to the traditional approach. The cycle savings after using the extended instruction set has been calculated using equation Eq. 4.1 [122]. The heuristics described previously are used to compare the existing custom instructions and our architecture aware FGCI.

The number of logic blocks was limited to 15 as it was experimentally found out that most of the applications do not exhibit significant performance increment beyond that. This is also evident by the trend lines that tend to plateau as the area is increased to 15 logic blocks. This in turn means that the maximum number of custom instructions implemented is 15 in any case. It should be remembered that in this work a logic block is defined as a set of 32 FPGA basic logic elements with identical hardware configuration that implements a 32-bit wide custom instruction.

The graphs in Figure 4.10 clearly shows the advantage of the proposed architecture-

4.10.a: Sha (MiBench)

4.10.b: Blowfish (MiBench)

4.10.c: Rijndael (MiBench)

4.10.d: Susan (MiBench)

4.10.e: Basicmath (MiBench)

4.10.f: Patricia (MiBench)

4.10.g: Cjpeg (Mediabench)

4.10.h: Epic (Mediabench)

Figure 4.10: Performance vs. Area Curves for Applications using “large and frequently occurring” Selection Heuristic.

aware custom instruction generation process. In particular, as the hardware area-time of the candidates are available after the enumeration process, the selection process can effectively choose a set of custom instructions that lead to high performance at low cost. As can be observed from the graphs, FGCI significantly out-perform the existing custom instructions, especially in designs with strict area constraints. In order to show the performance benefit of the proposed FGCI, the average clock cycles saved by using the custom instructions from the traditional method as well as the custom instructions obtained from the proposed approach, is also presented. The average clock cycles saved for both the methods were obtained by averaging the clock cycles saved over varying area constraint starting from 1 to a maximum of 15 Logic Blocks. Hence the clock cycles saved show the average performance per unit area for both the methods. It can be observed from Table 4.2 that the proposed method has significant performance benefits, in terms of clock cycles saved, when compared to the traditional approach especially for cases where the area constraints are extremely tight. On average this method achieves more than 65% higher performance compared to the traditional method. As discussed earlier, this result demonstrates that when the available area is restricted, it is sufficient to enumerate patterns that are small and occur frequently, instead of an exhaustive pattern enumeration from the DFG.

It can be noticed that in some benchmarks such as SHA and Patricia, that as the area constraints are relaxed thereby allowing more and more FPGA area, the performance benefit of our approach begins to reduce as opposed to the traditional Custom Instructions. The reason for such a behavior can be explained considering the larger custom instructions. Relaxing the area constraints allows the usage of larger templates which are essentially unavailable in the FGCI enumeration, since these larger templates could not be implemented in one LUT. The performance of the larger templates will begin to outweigh the FGCI as they become available for implementation

Table 4.2: Execution time (in terms of clock cycles) saved by using Proposed and Traditional Methods

APPLICATION	Traditional CI	FGCI	Percentage Saving over Traditional CI
SHA	26590008	30433318	14.45
BLOWFISH	6248900	7178469	14.88
RIJNDAEL	7717907	11789239	52.75
SUSAN	58789	202145	243.85
BMLARGE	17550394	18521604	5.53
PATRICIA	330780	388859	17.56
CJPEG	220715	377541	71.05
EPIC	96899	199686	106.08

with relaxed area constraints. To cope with such a performance bottleneck of FGCI for relaxed area constraints, the enumeration algorithm is modified to identify certain larger templates based on a specific rule. This new approach is discussed in the subsequent sections.

4.7 Coarse-Grained Custom Instructions

The previous sections showed the advantage of incorporating the LUT architecture information of the target FPGA device during the enumeration phase of custom instruction generation. The custom instructions thus enumerated were termed as Fine Grained Custom Instructions, or FGCI in short, since they were constrained to fit into a single logic block of the target FPGA. FGCI were shown to be capable of providing an average of over 65% extra performance when compared to traditional methods in strict area constrained designs.

It was also shown that while FGCI performed very well in strict area-constrained designs, they did not scale well with increasing available FPGA space. This was attributed to the fact that the FGCI approach focused on enumerating smaller and frequently occurring patterns that can efficiently utilize the FPGA space. Such patterns

were extremely useful in designs with strict area constraints as they provide more performance gain per unit area when compared to the larger and less frequently occurring patterns. On the other hand when the area constraint is relaxed, it is also necessary to leverage on larger patterns to achieve higher performance.

In this section, a new enumeration strategy is proposed to identify profitable custom instructions for FPGAs with arbitrary area constraints. In addition to the FGCI, large patterns that exhibit a high degree of instruction level parallelism are also identified which can be mapped efficiently using the FPGA logic blocks. A new selection heuristic is also proposed in Section 4.9 to facilitate the selection of large custom instructions with low critical path for increased performance.

4.7.1 Motivation For Coarse-Grained Custom Instructions

This section discusses the limitation of our FGCI approach proposed in the earlier sections using an example DFG shown in Figure 4.11. As mentioned earlier, the enumeration of large patterns was intentionally omitted during the enumeration process of FGCI. Although these large patterns generally occur less frequently than the smaller ones, they can still contribute to notable performance gain, provided there is sufficient FPGA space to implement such large patterns. The enumeration technique proposed in this section (denoted as Coarse-Grained Custom Instructions (CGCI) approach) not only enumerates the smaller patterns as the FGCI approach, but also identifies large profitable patterns that exhibit a high FPGA performance-area ratio. An example of such a pattern is shown in Figure 4.11. This example is used to highlight the limitation of the previous approach. It can be observed that the pattern consists of four FGCI in accordance with the concepts explained in section 4.4. These four FGCI are then implemented as four individual custom instructions, each with a critical path of 1 FGCI. Equation Eq. 4.1 on page 71 is used to calculate the

Figure 4.11: An example of a large pattern with low critical path.

number of clock cycles saved using these extended instructions [8]. Using that equation and assuming that the dynamic occurrence of the pattern is 10, the number of clock cycles saved is:

$$\sum_{1}^{4} (10 * 3) - \sum_{1}^{4} (10 * 1) = 80 \text{ clock cycles.}$$

However, it can be observed from Figure 4.11 that three of the FGCI's can be implemented in parallel, and hence the critical path of the entire pattern is only 2 logic blocks. If the pattern in Figure 4.11 can be identified as a single custom instruction (as opposed to four smaller custom instructions), a higher performance gain can be achieved, i.e. from equation Eq. 4.1 on page 71, the saving will be:

$$(10 * 12) - (10 * 2) = 100 \text{ clock cycles}$$

Hence, it can be observed that while it is prudent to enumerate small and frequently occurring patterns for area constrained designs, the concept does not hold true when the area constraint is relaxed. In such cases, larger patterns with high Instruction level parallelism also need to be enumerated to achieve high performance. In this work, Coarse-grained Custom Instruction or CGCI is defined as a combination of FGCI, where more than one of the constituent FGCI's can be implemented in

parallel. This in turn means that the critical path of CGCIs is always smaller than the number of FGCI. Since every FGCI can be implemented in 1 logic block, the number of FGCI in a CGCI is equivalent to the number of Logic blocks required to implement it.

In the next section, an extended enumeration algorithm that overcomes the limitation of the FGCI approach is presented. A new Selection heuristic that selects suitable custom instructions for arbitrary area constraints is also proposed. The experimental setup and results are discussed in Section 4.10 followed by some concluding remarks in Section 4.11.

4.8 Coarse-Grained Custom Instruction Enumeration

The enumeration algorithm discussed in section 4.4 was modified to include patterns that exhibit a high degree of instruction parallelism and at the same time, can be efficiently mapped onto the FPGA architecture. While this algorithm looks similar to the one proposed in FGCI, the critical differences are highlighted below. In particular, apart from enumerating FGCI, the new algorithm also identifies patterns in which the number of FGCI is greater than the critical path of the pattern. These patterns are annotated with the hardware area and critical path (both in terms of number of FGCI) during the enumeration phase. Again, it should be noted that the area of a CGCI (in number of logic blocks) is basically the number of FGCI in it, whereas its critical path is obtained by determining the maximum schedule length of the custom instruction, e.g. $(2*t)$ in Figure 4.11, where “t” is the latency of a logic block.

Algorithm 5 describes the new enumeration algorithm that incorporates the necessary steps to evaluate the number of FGCI and the critical path of every new pattern after merging the selected node with the existing pattern (along with the other nodes needed to meet the convexity constraint [94]). Just as it was discussed in Section

Algorithm 5 Algorithm for CGCI Enumeration

```
1:
   
$$P' = \begin{cases} P \cup u \cup (Succ(G, u) \cap Pred(G, P)) & \text{if } "u" \in Pred(G, P) \\ P \cup u \cup (Pred(G, u) \cap Succ(G, P)) & \text{if } "u" \in Succ(G, P) \\ P \cup u, & \text{if } "u" \in Disc(G, P) \end{cases}$$

2: if  $((u \in Disc(G, P)) \& \& (Succ(G, u) \cap Succ(G, P) == \{\}) \& \& (OUT\_LIMIT == 1))$  then
3:   return;
4: end if
5:
6:  $r_g = NaN$ ;
7: Count the number of FGCI in the pattern P' using count_fgci function;
8: Evaluate the critical path of the pattern P' using based on scheduling of FGCI;
9: if  $((number\_of\_FGCI)_{patternP'} == 1) \& \& in\_check(P') \& \& out\_check(P')$  {
   then // function to check FGCI node rules and IO constraints
10:   patterns_add (P');
11:   enumerate (P', G,  $r_g$ ) ;
12: }
13: else if  $((number\_of\_FGCI)_{patternP'} > (critical\_path)_{patternP'})$  then
14:   patterns_add (P');
15:   enumerate (P', G,  $r_g$ );
16: }
17: else
18:   return
19: end if
```

Algorithm 6 Algorithm for counting number of FGCI in a pattern

```
1: count_fgci (Pattern P) {
2:   Enumerate every sub-pattern in Pattern P.
3:   Evaluate every sub-pattern for valid FGCI using fgci_evaluation.
4:   Find the set of FGCI that completely "covers" the pattern P using conflict graph
   and MIS methods.
5:   number_of_FGCI = number of sub-patterns required to cover the pattern P.
6:   return number_of_FGCI.
7: }
```

4.4, a new pattern P' is formed by adding a new node 'u' to the existing pattern P as well as the other nodes needed to ensure the convexity of the generated pattern. The extra nodes needed for convexity depends upon the set to which the selected node belongs to. For example, if 'u' is the element of Pred(G, P), then all the nodes in the

intersection of set $\text{Succ}(G, u)$ and the set $\text{Pred}(G, P)$ must also be added and so on. In the current state-of-the-art methods any new pattern formed by this step needs to be verified against the traditional micro-architectural constraints such as the number of Input/Output nodes etc. However, in our case, it is also checked against the additional rules (Lines 9 and 13 in Algorithm 5) to ensure the enumeration of only the most profitable custom instruction candidates. These additional checks rely on two values obtained previously in Lines 7 and 8 in Algorithm 5, to ensure that only profitable custom instruction candidates that are large and exhibit high degree of parallelism are enumerated. These two values are as follows:

- (i) The number of FGCI in the pattern (Line 5 in Algorithm 5 and Algorithm 6). In order to obtain this, all the sub-patterns of a pattern are enumerated in the first step. Each sub-pattern is then evaluated using the *fgci_evaluation* algorithm to obtain a set of valid FGCI. The FGCI adhere to the rules in *FGCI_evaluation* that was proposed on page 83, which ensures that a pattern implemented in a single logic block of the target FPGA is a valid FGCI. Next, a set of non-overlapping FGCI are selected that “cover” the entire pattern using the conflict graph and Maximum Independent Set (MIS) methods proposed in [176]. These concepts have been explained in detail later in Section 4.9 for the selection of Coarse-Grained Custom Instructions (CGCI). The number of valid FGCI required to cover the pattern gives the area measure of the pattern.
- (ii) The critical path of the pattern in terms of number of FGCI (Line 8 in Algorithm 5). Since the FGCI covering the pattern P' are known from Step “1”, it is easy to calculate the critical path of P' based on the schedule time step of FGCI in pattern P' , based on the level of parallelism present in the pattern P' . As an example, consider the pattern in Figure 4.11. The three FGCI at the first scheduling level can be implemented in parallel and therefore the time required

to execute them is equivalent to the time for a single logic block. The last FGCI in Figure 4.11 needs the output of the first three FGCI and hence it is executed in the second time step. Therefore the critical path of the entire pattern is 2 logic blocks.

As explained earlier, the FGCI_evaluation algorithm basically ensures that the pattern conforms to the rules needed to implement a pattern in a single logic block of the target FPGA fabric.

Once the number of FGCI and the critical path of a pattern are defined, the remaining steps in Algorithm 5 are performed. A pattern consisting of a single FGCI, is identified as a valid pattern if it satisfies the I/O constraints (specific to the target FPGA device). On the other hand, a pattern with multiple FGCI is chosen if and only if the number of FGCI in the pattern is greater than its critical path i.e. patterns similar to the example pattern in Figure 34.11, discussed before. This strategy leads to the enumeration of the most profitable set of patterns for the given application on a particular architecture. It is noteworthy that the proposed enumeration algorithm identifies custom instructions wherein their FPGA area-time measures can be inferred without the need for time-consuming hardware implementation. The area-time in our case was calculated using FGCI_evaluation function at its core as described above.

It should also be remembered that the CGCI enumeration algorithm 5 produces a set of patterns which contain both FGCI as well as the large profitable custom instructions such as 4.11. In the next subsection, a selection heuristic is proposed that favors these large patterns when the FPGA area is unconstrained, while still capable of selecting smaller patterns for area-constrained design.

4.9 Coarse-Grained Custom Instruction Selection

Using the method proposed in the last section, the most profitable set of patterns are obtained for a given application. This set basically comprises of two types of patterns, each of which are specifically useful for different area constraint scenarios. The smaller patterns known as FGCI, with critical path of 1 LUT perform very well when the area is highly constrained. As was shown in the earlier sections on FGCI, when the area constrained is high, it is sufficient to identify such small patterns, which usually occur very frequently. On the other hand, the second type of patterns enumerated using the proposed method are the ones with critical path more than one, but which exhibit high instruction level parallelism. These patterns known as CGCI help us to achieve high performance in designs with relaxed area constraints.

Although the set of patterns identified contain all the profitable patterns in an application, a suitable pattern selection algorithm is a must to choose the right set of patterns for different area constraint. In order to achieve this, a new selection heuristic is proposed that promotes small patterns when the area constraint is high, but allows large patterns for implementation in designs with sufficient FPGA area. As discussed during the FGCI selection step, the enumerated patterns were grouped together using “VFLib” [180] to find isomorphic patterns. A group of isomorphic patterns was then represented as a corresponding template. Our selection algorithm follows a similar basic step described in [176] where a conflict graph is used to select a set of non-overlapping patterns. Algorithm shows the proposed pattern selection algorithm. Once the conflict graph is created, steps similar to the one described in Section 4.5 were used to compute the local Maximum Independent Set (MIS) to find a set of vertices in every template of the conflict graph that are mutually non-adjacent.

Here the new heuristic is defined in equation Eq. 4.2 to compute the local MIS weight.

$$MIS_Wt. = Frequency * \left[\frac{No.ofNodes}{Critical_path} \right]^3 \quad (Eq. 4.2)$$

The new heuristic has two important differences from the heuristics used in [93] [176] [122] as well as our previous work in [7] and [8]. Firstly, the existing heuristic (which favors large and frequently occurring patterns) is divided by the critical path of the pattern. This enables a selection preference for large patterns with a high degree of parallelism when the area constraint of the design is relaxed. At the same time, since the critical path of a FGCI, by our definition is 1, the new heuristic also caters to the selection of smaller and frequently occurring patterns that can efficiently utilize the FPGA space when the area constraint of the design is tight. The second difference is the extra power of 3 to the second term in equation Eq. 4.2. This was found empirically to reinforce the selection of larger patterns with smaller critical path. It should, however, be noted that the power value need not be fixed to 3. For example, the optimal power value can be determined empirically for domain-specific applications that share common computation patterns to promote the selection of larger patterns with smaller critical path in those applications.

After calculating the local Maximum Independent Set, all the templates are sorted according to their MIS weight. Next, the algorithm starts with the template with the largest MIS weight to ensure constraint-aware pattern selection. This template is selected if and only if it does not violate the current area constraint. In case the available area is not sufficient to implement this pattern, it moves on to the template with second largest MIS weight and continues till it finds a suitable template and chooses it. After selecting a particular template for implementation, all the patterns in the maximum independent set of the selected template are implemented as custom instructions and are removed from the conflict graph. The remaining area constraint

was updated and the MIS weight was recalculated for the remaining templates in the conflict graph. The algorithm is reiterated till either the area constraint is violated or the templates are exhausted in the conflict graph.

This can be explained further by using an example shown in Table 4.3 which shows a set of patterns enumerated using the CGCI enumeration method described in Section 4.8.

Table 4.3: An example for CGCI Selection Step

Template Number	Serial	Number of Nodes	Critical LUT Path	MIS Weight	Clusters
1		3	1	1	1
2		9	3	3	5
3		5	2	2.5	2
4		8	2	4	4
5		3	1	3	2
6		2	1	2	1

As the area constraint is relaxed starting from 1 LUT, the templates with area less than or equal to 1 LUT are considered for implementation first. These are small templates and in general occur more frequently in the application compared to the larger templates. After a few iteration when the area constraint is sufficiently relaxed, the larger templates such as Template 2 & 4 in the table 4.3 start to appear in the selection list because their MIS weight is larger than that of the clusters. Hence, the heuristic is able to tap in the advantages of larger templates in the relaxed constraint scenario. This behavior can be clearly observed in the following section 4.10 showing the advantages of CGCI enumeration over our previous FGCI enumeration as well as the traditional custom instruction approach.

4.10 Experiments And Results

In this section, the results obtained using the proposed FPGA-aware custom instruction generation approach are compared to the FGCI approach discussed in the earlier sections as well as the existing state-of-the-art custom instruction generation approaches from [93] [176] [97] [122]. The existing state-of-the-art approaches are denoted as the traditional method. In the traditional method, legal patterns are first enumerated, followed by a pattern selection stage that attempts to select large and recurring custom instructions. The enumeration algorithm used in the traditional approach is from [97]. The pattern selection step uses the greedy selection approach described in [93] and [176]. A design exploration step was also implemented for the traditional approach that relies on solving the knapsack problem [95] [127] to identify the most profitable set of custom instructions (from the selected patterns) that meet the given area constraint. It should be noted that the three methods produce different sets of custom instructions based on what is deemed profitable by the corresponding heuristics.

An average reduction of about 10% was observed in the number of patterns enumerated using the proposed CGCI approach when compared to the traditional method. The reduction in the number of patterns enumerated by the proposed method also translates to the reduction in the enumeration runtime, though it is not very significant since our algorithm has to check for additional constraints during enumeration.

Next, the cycle savings achieved by using the proposed CGCI approach is compared to the traditional approach, as well as our previously proposed FGCI method. The performance achieved by using each of the three extended instruction sets was estimated using equation Eq. 4.1 described on page 71, which gives a fairly accurate estimation of the performance gained by using these extended instructions [122]. As mentioned in Section 4.8, our design is targeted for the Xilinx Virtex 2 FPGA, but the

concepts and algorithms can be easily retargeted to any FPGA family. It should be noted that even a low end FPGA such as the Virtex 2 FPGA shows the advantage of our methodology to create designs that are more efficient. Better and newer FPGA families can achieve even higher performance. For example, the newer FPGAs like Xilinx Virtex 5/6/7 or the Altera Stratix IV/V have adaptive-LUT structures with more number of inputs, I/O etc. and therefore can fit in larger patterns with more I/O, thus providing better performance per unit area. The frequency of operation of the processor after the addition of the extended instructions generated by any methodology is likely to fall compared to the base processor, especially in the case of instructions with larger critical path. Although the results in this work do not consider this effect of reduced processor frequency, it is still a fair comparison since the frequency would reduce by implementing any large custom instructions from any enumeration methodology.

It should be noted that the addition of custom instructions using any methodology is normally done under the presumption that the advantage of adding the custom instructions offsets the penalty paid because of reduced frequency. Moreover, the larger custom instructions can be implemented as a multi-cycle custom instruction in the newer FPGAs as part of the processor pipeline, thereby eliminating the penalty. In addition, our method is particularly efficient in this regard compared to the traditional method, since the extended instructions were selected with low critical path as much as possible.

Figure 4.12 shows performance achieved in terms of clock cycles saved, after using the custom instructions obtained by the various methods, for eight benchmark applications considered earlier in section 4.6. Figures 4.12 show three trend lines, one for each of the three methods, Traditional in green, FGCI in blue and CGCI in red. The X-axis represents the FPGA area in terms of logic blocks. The Y-axis shows the estimated clock cycles saved as calculated by equation Eq. 4.1.

It can be observed that the CGCIs proposed in this work consistently perform better than the traditional method. It also overcomes the limitation of FGCI in applications such as SHA, BASICMATH and PATRICIA, where the FGCI could not outperform the traditional method during relaxed area constraints. Evidently, the proposed method performs significantly better than the traditional method with an average improvement of over 67%. In applications such as Susan, the CGCI method under performs slightly when compared to FGCI. This is attributed to the fact that these applications predominantly consist of smaller and frequently occurring patterns where the FGCI method has been shown to perform exceptionally well.

As discussed in the case of FGCI, the maximum number of custom instructions was 15 in all cases. Table 4.4 shows the number of custom instructions implemented by the various methodologies. The numbers in the bracket show the area in logic blocks of these custom instructions. A large number of custom instructions may lead to reduced clock frequency due to delay in the arbitrator logic between the soft-core processor and custom hardware. However, as shown in Table 4.4, the maximum number of custom instructions implemented for an application is only 15. Based on earlier investigation in the influence of the number of custom instructions to the delay of arbitrator logic that was reported in [183], it was found that the number of selected custom instructions in the proposed method will not have any notable impact on the clock frequency. In addition, the decrease in clock frequency due to the arbitrator logic was not considered even in the traditional method, therefore providing a fair comparison in all cases.

Figure 4.13 shows the absolute speedup achieved when the proposed CGCIs and the traditional CIs were used as extended instructions when compared to the base processor implementation. The relatively low speedup in a few applications is attributed to the fact that it was assumed that every instruction in the base processor takes exactly one clock cycle giving the base processor an unrealistic advantage.

4.12.a: Sha (MiBench)

4.12.b: Blowfish (MiBench)

4.12.c: Rijndael (MiBench)

4.12.d: Susan (MiBench)

4.12.e: Basicmath (MiBench)

4.12.f: Patricia (MiBench)

4.12.g: Cjpeg (Mediabench)

4.12.h: Epic (Mediabench)

Figure 4.12: Performance achieved by custom instructions generated using Traditional, FGCI and CGCI methods.

Table 4.4: Number of Custom Instructions (and their area) implemented using the various methodologies.

Application	Traditional CI(Area)	FGCI(Area)	CGCI(Area)
SHA	13(13)	9(9)	10(12)
BLOWFISH	11(15)	15(15)	11(15)
RIJNDAEL	14(15)	15(15)	14(15)
SUSAN	2(3)	2(2)	2(3)
BASICMATH	9(12)	8(8)	9(12)
PATRICIA	8(15)	8(8)	8(12)
CJPEG	9(14)	15(15)	10(15)
EPIC	10(13)	15(15)	10(14)

Figure 4.13: Average speedup achieved by using the proposed CGCIs against Traditional CIs, as extended instructions in the Applications

In reality, the speedup is expected to be higher. As the objective of this work is to demonstrate that the proposed method can lead to better solutions than the existing methods, a detailed cycle level simulations to show a more realistic speedup over the base processor is not performed at the moment.

4.11 Summary

Instruction set extension, while being very useful for high performance embedded systems, needs careful selection of the extended instructions for achieving high per-

formance per unit area. In this chapter, an automated design flow for custom instruction generation was proposed that takes into account the underlying target FPGA architecture during the custom instruction enumeration phase. In particular, the proposed enumeration approach incorporates a set of constraints, so that it only identifies custom instruction patterns which can be completely implemented on one logic element of the FPGA architecture. This poses additional constraints in the enumeration phase which helps in reducing the problem size of the selection process without sacrificing on the quality of the final solution.

At the same time, since the hardware area-time of the candidates are available after the enumeration process, the selection process can effectively choose a set of custom instructions that lead to high performance at low cost. Experimental results based on eight applications from two widely used benchmark suites, MiBench [181] and MediaBench [182], show an average performance improvement of 65% over the existing approach.

Next, it was shown that while the proposed heuristic performed very well in area-constrained designs, they did not scale well when the area constraint is relaxed. This is due the fact that the heuristic focuses on maximizing the resource utilization of FPGA space with limited resources which leads to the pruning of larger patterns from the design space that have relatively lower performance gain per unit area. Hence, this strategy may not lead to the best results when the hardware area constraint is relaxed and the application contains large patterns.

In order to generate profitable custom instructions for FPGAs with arbitrary area constraints, the proposed enumeration methodology was extended and a new selection heuristic was also proposed to identify the most profitable custom instruction candidates for designs with arbitrary area. The extended enumeration approach identifies large patterns that have relatively smaller critical path. Such patterns have

been termed as CGCI and shown to achieve higher performance in designs with relaxed area constraints when compared to the existing work. The proposed template selection heuristic enables a selection preference for large patterns with a high degree of parallelism when the area constraint of the design is relaxed. Based on the results of our experiments on benchmark application from the MiBench and MediaBench Benchmark suites, it was shown that the proposed method achieves significant performance improvement of over 67% on average over the current state-of-the-art approach. Our simulation results show that the proposed method can achieve a maximum speedup of 39.27% with an average speedup of 19% when compared to the base processor implementation.

It is noteworthy that the FGCI approach is still relevant for many embedded designs that typically have very strict area constraints. The CGCI approach however, can be used in a wider range of designs especially given that FPGAs are becoming commonplace in high performance embedded systems. It should also be noted that while techniques for Pattern Enumeration have received considerable attention in the past, to the best of our knowledge, the existing Pattern Enumeration techniques, aside from our own work published in [7] and [8], have never taken the implementation constraints of FPGA architecture into account. The novelty of our work lies in the fact that these constraints are used during enumeration so as to make a more informed-enumeration and thereby achieving much higher performance from the selected custom instructions with less FPGA area utilization. The experiments undertaken in this work relies on a low-end Xilinx Virtex II FPGA in which the basic logic element is a 4-input LUT. The results obtained for this FPGA showed that high performance custom instructions can still be realized using a low-end FPGA device when architecture aware methods are employed to generate custom instructions. The performance achieved on modern FPGAs (with more efficient LUT architecture) is expected to be higher with the proposed methods.

This chapter started with relatively smaller custom instructions, defined as Fine-grained Custom Instructions or FGCI that could fit into exactly one logic block of the target FPGA architecture. Next, Coarse-grained Custom Instructions or CGCIs were enumerated and selected, that need more than one logic block for implementation but are still limited to the basic block boundaries in an application. In general, with the increase in size and density of modern FPGAs, it has become easier to implement in more of the application blocks into hardware to achieve higher performance. This in turn means that it is possible to move a complete basic block with all its instructions into hardware or even entire functions with multiple basic blocks. However in order to implement an entire basic block in hardware, one needs to incorporate all the operations of the basic block into the corresponding hardware block. In section 4.2 the micro-architectural constraints used to prune the search space during enumeration of custom instruction were discussed. It was mentioned that certain operations such as memory and control flow are considered as *invalid* nodes for custom instruction. This is to ensure deterministic latency in the processor's datapath where the custom instructions are typically connected.

On the other hand, a hardware accelerator is generally connected as a co-processor to the base processor. Therefore it is neither required nor preferred to avoid memory or control operations during hardware translation of a basic block. The subsequent chapters will elaborate the implications of incorporating memory and control flow operations into a hardware accelerator block. The need for a selection heuristic to select the most suitable set of memory blocks in order to achieve the best performance from the translated hardware blocks will also be discussed in detail.

In this chapter, the Trimaran [177] compiler was extensively used to translate the application C Code to an intermediate level for further analysis. In the rest of this work, another open source compiler infrastructure called LLVM [119] is used for application analysis. The specific differences between the two compilers have been

briefly highlighted in the Appendix section on page 221.

Incorporating Data-Dependent Blocks Into Hardware Accelerators

5.1 Introduction

In the previous chapters, two methods were proposed to customize a processor, namely *Instruction Subsetting* and *Instruction Set Extension*. Instruction Subsetting by disabling the support for unused instructions in an application allowed us to achieve better area efficiency. On the other hand, Instruction Set Extension, by introducing architecture-aware custom instructions into the base Instruction Set Architecture (ISA) of the processor, was useful in increasing the performance of the processor. It was demonstrated that the fine-grained custom instructions are suitable for highly area-constrained designs, while coarse-grained custom instructions are useful for designs with slightly relaxed area-constraint. However, when sufficient area is available and the primary goal of the designer is to achieve higher performance, the need for hardware acceleration of larger application code segments arises. As discussed in section 2.2.2 of chapter 2, hardware accelerators are larger blocks of application code, generally comprising of one or more basic blocks. Hence, these accelerators also contain memory and control flow operations apart from the basic operations in a custom instruction. The usage of memory blocks in hardware accel-

erators necessitates data sharing and communication as well as syncing between the hardware and software components of the application. This results in an overhead while implementing hardware accelerators with local memory due to the need for additional data transfer as well as protocols to ensure coherence between the different parts in the memory hierarchy. In the past, some architectures have been proposed to facilitate the memory sharing between hardware and software sections by using expensive Direct Memory Access (DMA) protocol for data transfer. Data coherence has been handled by using cache-coherence protocols that require additional hardware or software intervention.

In this work, a novel architecture is proposed, which can implement traditional custom instructions, such as the ones proposed in chapter 4, as well as hardware accelerators with local memory. Unlike previously reported architectures, the proposed architecture does not require expensive DMA transfers or cache-coherence protocols. The proposed architecture is focused mainly on accelerating code segments with statically allocated variables (both scalars and arrays) which are widely prevalent in embedded applications that are written with due considerations to constrained resources. Such variables can be initialized in the local memory address space before the program execution to avoid data transfer during execution and kept in the un-cached space to avoid the need for cache-coherence protocols. It is noteworthy that the proposed concepts in this chapter can be complementary to other solutions proposed in existing work, such as using DMA and/or cache-coherence protocols, for a wider range of memory variables. In this work, *an architecture model is proposed that is capable of implementing hardware accelerators with local memory for statically allocated variables that does not suffer from expensive DMA transfers and overheads expended to ensure cache coherence.*

Prior to the implementation process, potential candidate blocks from the application need to be identified or enumerated to assess them later for hardware acceler-

ation. Although the need for cache-coherence strategies are avoided by placing the local memory blocks in un-cached address space, this also results in a penalty for the processor to access the data stored in the local memory without utilizing the data cache. In order to consider and minimize this overhead during the selection process, a model is proposed to estimate the penalty incurred by the processor in accessing local memory blocks instantiated alongside other hardware accelerators without using the data cache.

Next, the need for a suitable selection algorithm to select the right set of candidate code blocks for hardware acceleration is discussed. The selection algorithm for the candidate code blocks is based on their memory access pattern, apart from the traditional metrics such as difference in software and hardware performance etc. The typically complex memory dependency across different sections of an application creates a situation where merely selecting more parts of the application for hardware acceleration does not necessarily guarantee increased performance. Instead, a more holistic analysis must be performed in order to select the right *set* of candidates for hardware selection. Using the proposed penalty model, the selection algorithm and the proposed architecture model paves the way for accelerating large sections of an application while being mindful of the intricate data dependency across the various sections of the application.

The proposed architecture model was implemented and verified on the Altera DE2 Development board using a case study of the Secure Hash Algorithm (SHA) application from the popular MiBench Benchmark suite. In the next section, we first elaborate and discuss the pros and cons of the existing state-of-the-art work in using local memory blocks alongside hardware accelerators, specifically for statically allocated variables.

Portions of the work described in this chapter have been published in [10] and [11].

5.2 Related Work

The inclusion of memory operations into hardware blocks necessitates a proper channel for these hardware blocks to access the memory. The authors in [132], [136], [134] proposed transferring the relevant data from the main memory to the local memory of the FPGA (i.e. close to the custom hardware block) by a DMA transfer in a basic block before the custom hardware block executes. After the hardware block executes on the data in its local memory, the data is transferred back to main memory in a suitable basic block using another DMA transfer cycle. The authors demonstrated the advantage of their work by considering vectors and loop-carried scalars, without pointer access, for storage in local memory. It should be noted that a significant portion of time savings achieved by executing the code segment in hardware, is eventually lost due to the DMA transfers [133], [136]. The authors proposed techniques to minimize the effect of these multiple DMA transfers but with limited success.

A second problem with such designs is the need for extra protocols to maintain cache coherence. A cache coherence problem occurs when both the custom instruction and the CPU try to access and modify the same data set. When the custom instruction unit operates on the data set, the data is accessed and stored in the local memory itself. If the same data was present in the cache its contents remain unchanged and the cache contains polluted data. This is called the cache incoherence problem. Kluter et al. in [133] and [134] suggested modification to the cache controller to avoid using a rather costly hardware coherence protocol for solving the cache incoherence problem. Ensuring cache-coherence is becoming increasingly important, especially due to the growing popularity of multi-core systems. Therefore, companies such as ARM [184] and Synopsys [158] provide hardware IPs [185] [186] for easy integration while providing better performance and power efficiency. In a recent work Kluter et al. [187] proposed to integrate the local memory for custom

instructions into the data cache in order to obviate the need of cache-coherence protocols. Moreover, they placed an additional constraint on the local memory blocks to be single-ported so as to successfully “merge” it with the data cache. Unlike their previous work using DMA, in [187] they used prefetch instructions to transfer data between the main memory and one of the ways of the data cache where the local memory for custom instructions reside. Although they avoided using DMA for data transfer, their communication cost is still nonzero [187]. However, our work strives to avoid using any data transfers as well as a need for cache-coherence protocols. It should also be noted that similar to our work, the concepts proposed in the existing work discussed here consider local memory blocks for statically allocated variables that can be disambiguated at compile-time.

In [188] the authors have proposed a new architecture to allow memory operations in custom instructions. In their design, custom instructions have direct access to the level one data-cache in the system and therefore able to initiate a memory transfer from the entire processor address space. In order to ensure the validity of the data after the custom instruction execution, all the write backs to the memory only happens after the execution of the custom instruction. On the other hand in case of a Translation Lookaside Buffer(TLB) fault, the compiler introduces additional instructions to ensure the validity of the data. The authors have provided limited information of their work for proper evaluation. However, it should be noted that one of the prime reasons for not allowing memory operations inside a custom instruction is to ensure deterministic latency of the data [94]. Since the custom instruction is part of the processor’s pipeline, a delay in fetching data stalls the entire pipeline thereby resulting in penalty for the processor. In [189], the authors have presented a profile driven data-dependency analysis to enable the selection of hardware accelerators that encapsulate loop carried dependencies. This ensures that a code section with high intra-dependency, such as loop carried dependencies, is executed on a single

processing module to control the inter-process communication. The motivation for our work is also similar, i.e. to minimize the overhead of communication between hardware and software sections, however, we propose to analyze and incorporate various hardware accelerators with data-dependencies with other sections of the application, provided there is a net performance gain.

The following section briefly highlights the various metrics considered during the enumeration/identification of hardware accelerators in order to account for the advantages and overheads of partitioning. The issue of complex data-dependency in an application that need to be considered during the selection of hardware accelerators with local memory blocks is also discussed.

5.3 FPGA Resource-Aware Enumeration Of Candidate Hardware Blocks

Unlike the custom instructions described in the previous chapter, hardware accelerators are larger blocks of application code, such as one or more basic blocks or even functions, as discussed in 2.2.2 of chapter 2. Also, as discussed on page 43 in chapter 2, *the accelerators in our work are explored at the basic block level of granularity to maximize the performance-area gain*. These accelerators also contain memory and control flow operations apart from the basic operations in a custom instruction. As highlighted in the previous sections, the usage of memory blocks in hardware accelerators necessitates data sharing and communication between the hardware and software components of the application. Therefore, apart from the traditional metrics such as the area required to implement a code section in hardware, time taken to execute in hardware and software, etc., the selection process for hardware accelerators must also consider the memory dependency between the various candidate hardware blocks. This must be done to reduce data transfer between hardware and

software sections of the application after acceleration.

Figure 5.1: Identification Phase of Custom Hardware Blocks

As a first step of selecting suitable code blocks for hardware acceleration, a designer generally performs an enumeration or identification step to list down the possible candidates for implementation. Figure 5.1 shows the flow chart of this step in which the LLVM compiler infrastructure [119] was used to do a front end compilation of the application code and obtain a target-independent Intermediate representation (IR). This IR was profiled using the same compiler toolchain to obtain the execution frequency of all the basic blocks in the application. The most frequently executed code sections or the so called “hot blocks” are usually the most preferred candidates for hardware acceleration. Therefore, the basic blocks were arranged in decreasing order of execution frequency and shortlisted to choose the top most frequently exe-

cuted basic blocks for further consideration. The number of blocks selected at this stage can be varied according to the user input, but generally, depending on the size of embedded applications, only the top 20 - 30 amongst all the basic blocks in an application have any significant execution frequency. This observation also follows from the well-known Pareto principle or 90/10 rule of thumb [190], stating that 90% of the execution time is spent in 10% of the code.

After shortlisting these candidate basic blocks, a data-dependency analysis is performed to determine which of these basic blocks can potentially be executed in hardware. This decision can be based on certain design constraints which might decree that certain portion of the code must be executed in software. At the same time, currently in this work, basic blocks with any dynamically allocated memory are not considered for implementation in hardware. Based on such constraints the shortlisted candidates are divided into two categories: *Implementable* and *Un-implementable* Basic Blocks. The importance of un-implementable blocks will be discussed later in section 5.4, while describing the selection heuristic. For the Implementable blocks, various metrics need to be obtained that help us to identify the most suitable hardware candidates for varying performance requirement. These metrics are inputs to the algorithm and as such their method of estimation does not impact the performance of our algorithm. Here, these metrics are listed and a way to obtain them using the methods proposed in the existing work is also presented. The accurate estimation of these metrics is a challenging problem by itself, however, it is not a focus of this research.

- (i) **Hardware Area:** The amount of FPGA area required to implement a hardware block is an important criterion for selecting a particular hardware block over another. A rapid and reliable estimate of this metric is paramount to any hardware selection algorithm. The techniques proposed in [122] can be effectively used

to estimate the FPGA area required to implement the hardware blocks. The authors have shown the technique to be rapid and reliable [122]. Again, it should be noted that the proposed search heuristic in this thesis does not depend on the specific method used to estimate the hardware area required to implement the candidate block.

- (ii) **Hardware Execution Time:** The time taken to execute a block of code in the FPGA hardware is one of the most important criteria for hardware selection. This factor along with the software execution time tells us the amount of time that can be saved by executing a piece of code in hardware instead of software. Again the methods described in [122] can be used to estimate the execution time of a code block in hardware.
- (iii) **Frequency of Basic Blocks:** The frequency of basic blocks is the estimated dynamic frequency of a block during runtime and is obtained by profiled application code. It is number of times the basic block is executed during run-time.
- (iv) **Software Execution Time:** The software execution time gives an estimated time required to execute the block of code in software. The software time can be estimated as the number of basic operations in the block of code. In a realistic scenario, each of the basic operations such as arithmetic, logical etc take a single clock cycle for execution whereas the memory operations generally need longer time to execute if the relevant data is not found in the cache. Even so, with larger available cache in modern FPGA designs this way of estimating the software execution time is not far from the real execution time.
- (v) **Memory size estimation:** Since the total hardware area required to implement a code segment in hardware also includes the size of memory required to store the relevant variables, this size must also be estimated. In order to estimate the size of these memory blocks, the Intermediate output from the LLVM compiler

was parsed to obtain this information. All the arrays and variables accessed in the high frequency blocks, implementable as well as un-implementable, were identified. Additionally, the number of times each of these arrays/variable are accessed in each iteration of the corresponding basic block was also obtained. This gives the number of times these memory locations are accessed by each candidate block. After selecting the high frequency blocks as mentioned earlier, an in-house developed tool was used to extract the names, type and size of each variable manipulated in these basic blocks. The intermediate output obtained from LLVM lends well to such analysis due to its human readable format.

5.3.1 Considerations During Data-Dependency Aware Selection Of Candidate Hardware Blocks

In an application, instructions typically depend on data set manipulated by other instructions within the program. This creates a dependency of data between the various code segments analogous to a consumer-producer scenario. After the selection of hardware accelerators with memory, these dependencies result in data transfer between the software and hardware code segments that depend on the same data set. In order to account for such overheads, designers are motivated to consider them during the selection phase and ensure that the advantage of accelerating a code segments overrides the potential overhead. This is explained by using the following example:

In Figure 5.2 the big rectangle represents a schematic for an application code, whereas the smaller shaded block shows a segment of potential Hardware Block (HB). The HB accesses variables stored in the memory location shown as Local Memory (LM) a total of “G” times. The same variables can be accessed from various parts of the rest of the application a total of “F+H” times. Given this information, a

Figure 5.2: Preferred code segments for hardware implementation.

Figure 5.3: Preferred code segments for hardware implementation.

potential HB is preferred for selection if:

$$F + H \ll G \quad (\text{Eq. 5.1})$$

which means that the rest of the application code rarely accesses the Local Memory LM. This in turn ensures that the LM is rarely accessed by the processor (executing the rest of the application not executed by the custom hardware block), but accessed very frequently by the selected custom hardware block HB. Therefore, even though there is an overhead for the processor to access the data from the local memory instead of a potential cache-hit, there is an overall time saving by accelerating the selected hardware block with local memory.

This example shows the selection of one hardware block. In practice there are

more than one potential hardware block, such as shown in Figure 5.3, depending on the available area and performance constraints. In such circumstances, the general strategy is to select various code segments accessing an overlapping memory area to be executed as custom hardware blocks simultaneously. However, since the multiple code sections can access the same memory location as shown in Figure 5.3, one needs to evaluate *all* possible combinations of blocks for hardware acceleration in order to arrive at the best combination for hardware acceleration. At this point it can also be observed that in such a scenario, just implementing more code sections into hardware does not guarantee increased performance due to the overhead from other parts of the application. Therefore system designers are motivated to consider such dependencies during the selection of hardware accelerators.

It has been observed that the usage of local memories along side hardware accelerators is becoming increasingly popular, especially to reduce communication time between the accelerator and local memory as well as the wide availability of distributed Block Random Access Memorys (BRAMs) in FPGA. On the other hand, this also results in overheads such as the need for additional cache-coherence logic in the existing work. In this work, while the need for cache-coherence protocols was avoided, it however resulted in another penalty due to the cache-bypassed implementation of the local memory blocks as discussed in the next section. It is important to note however that even though this new penalty is incurred, this is a fast access to the local memory blocks located on the on-chip memory space. Additionally, unlike certain cache-coherence protocols, a separate hardware circuit is not required to achieve this. Even so, models and algorithms have been proposed in this work to minimize this penalty and provide an opportunity to reduce this overhead to close to zero by using the right selection of hardware accelerators. The penalty estimation model is presented in the next section.

5.3.2 Estimating Penalty When Data Resides In Local Memory

In order to minimize the overhead incurred by the processor to access the data stored in the local memory blocks alongside other hardware accelerators while bypassing the cache, a mathematical expression is derived to model this penalty. This section also describes the proposed architecture model for an application-specific Configurable System-on-Chip (CSoC), capable of implementing traditional custom instructions as well as hardware accelerators with local memory.

Figure 5.4 shows the proposed architecture model using the Altera's NIOS II soft core processor. This architecture model is used to explain the various penalties and advantages considered in our selection algorithm. There are two important aspects to be noted in this architecture. Firstly, as is shown in the figure, a memory mapped slave interface has been created that connects the local memory directly to the main system bus. This ensures that the local memory shares the memory address map with the main memory of the processor. This design allows the processor to access the data from the local memory as it would from the main memory. As stated earlier, the permitted variables stored in the local memory are constrained to statically allocated variables. Therefore, the proposed model does not require data to be transferred from the main memory to the local memory, thereby avoiding the cost and energy spent in DMA data transfers used in the existing solutions. Instead the relevant variables are located in the local memory address space during design time. This is achieved by identifying the arrays of data that are needed by the custom hardware block and storing them in the local memory during design time. When the processor needs to operate on this data set, it can easily access the local memory via the system interconnect fabric.

Secondly, the local memory is implemented in the un-cached pages of the system memory hierarchy. This is to ensure that processor always bypasses the cache sys-

hardware accelerators with memory blocks can be implemented in a similar fashion and henceforth not differentiated for the implementation purposes. This is one of the major features of our research, whereby hardware accelerators of varying granularity can be selected using a common selection heuristic described later in chapter 6 and implemented on the proposed architecture model. As shown by authors in [128], such a flexibility can be immensely useful to exploit the full potential of hardware acceleration.

In order to estimate the penalty, consider a block of code “X” being executed in the processor that needs to process an array variable “A”. In a traditional memory hierarchy, the array would be stored in the main memory and will be brought into the processor’s registers through the cache hierarchy. In an ideal situation the array “A” might already be in the cache and therefore the access latency of this array is minimal. In a different setup where the system has a hardware accelerator “Y” and the array “A” needs to be processed by the hardware accelerator, it is useful to store the array “A” alongside the hardware accelerator in the local memories. As mentioned earlier, this is useful for two reasons. Firstly, the access latency from the accelerator to the local memory is minimal and secondly this latency is deterministic unlike the access from the main memory. Now, if the code block “X” also needs to access this array “A”, it must access it from the local memory via the system bus instead of the data cache. Obviously this requires a longer access time compared to the access from the cache. The difference in latency of access from “X”, to the local memory instead of the data cache is referred to as the “access penalty”. In order to quantify this access penalty, the time taken for “X” to access the data from both the local memory as well as the data cache was modeled. A few terms are defined below, before formalizing an equation to calculate this penalty.

- (i) **Access time from Processor to Main Memory:** In order to access a data stored in the main memory the processor needs to communicate through the

main system bus, such as the Avalon System Interconnect Fabric in case of Altera NIOS II. The access latency depends on the type of Memory as well as its location. For on-chip memories the access time is obviously faster than off-chip memory modules. Similarly, depending on the speed grade of the memory module, the access time changes accordingly. The time taken for the processor to access any memory connected via the System bus, but not directly to the processor, is denoted as T_{PM} where,

T_{PM} - Time taken from Processor to Memory.

Since the local memory in our architecture model is tied to the same System Interconnect as the main memory, the time taken by the processor to access the local memory is also T_{PM} .

- (ii) **Access time from Processor to Cache:** Based on the application requirements, most modern processor have a cache hierarchy to store the most frequently used Instruction or Data for faster access. The effectiveness of cache is dependent on several factors such as its size, the loop or data size of the critical code section in the application, etc. The time taken for the processor to access the data from its cache memory is termed as T_{PC} where,

T_{PC} - Time taken from Processor to Cache.

T_{PC} can be significantly different for different processor architectures and different code sections, varying from the best case scenario of 1 clock cycle to multiple clock cycles in the worst case (due to the need of cache flushing etc.).

- (iii) **Access time from Hardware accelerator to the Local Memory:** The local memory is implemented using the widely available distributed block RAMs in the modern FPGAs. The time taken by the accelerator to access any data stored in these Block RAMs is deterministic based on the implementation. One can safely assume that the data can be accessed every clock cycle with proper implementation and this time is denoted by T_{HM} where,

T_{HM} - Time taken from Hardware Accelerator to Local Memory.

Based on the code section for hardware acceleration one can design a multi-ported memory, thereby minimizing T_{HM} as much as possible through parallel data access.

Now, let us calculate the number of clock cycles saved through implementing a code section “X” in hardware.

$$\text{Cycles saved} = \underbrace{(SW_{Time} - HW_{Time}) * Freq_X}_{\text{1st term}} + \underbrace{(T_{PC} - T_{HM}) * Freq_X}_{\text{2nd term}} - \underbrace{\sum_{(i=1)}^{(i=X-1)} (T_{PM} - T_{PC}) * Freq_i}_{\text{3rd term}} \quad (\text{Eq. 5.2})$$

The first term in Equation Eq. 5.2, is the general term used to calculate the number of cycles saved by implementing any software code into hardware. The second term calculates the difference in time of accessing a variable from the cache in case of software implementation and from the local Block RAMS in case of hardware implementation. This difference can be maximized by reducing T_{HM} as mentioned above. It should also be noted that the best case scenario is assumed for the software implementation by assuming that the variable is found in the cache. As mentioned earlier, in a more realistic scenario cache performance is dependent on various factors such as the cache size, loop size etc. In future, such factors can be considered to further increase the accuracy of this estimation model.

The third term is the penalty associated with the processor accessing a block of memory implemented in hardware alongside another code block. This penalty is analogous to the situation where the processor needs to access the data from the main memory instead of the cache hierarchy. The summation in this term indicates

that one needs to factor in the penalty from all the basic blocks currently being executed in the software, but needing to access the data stored in the FPGA BRAMs. *Therefore the main objective of our selection algorithm, as discussed in the later sections, is to minimize this penalty as much as possible by selecting the best set of candidates for hardware implementation. Ideally, all the basic blocks having common data-dependencies should be ported to hardware at the same time to have zero overall penalty.*

This penalty is also dependent on the latency of the processor to access any memory blocks via the system bus which itself can vary with different FPGA devices and families. In our experiments, the Altera NIOS II processor is used on Altera DE2 board containing a Cyclone II FPGA. Based on the NIOS II Processor data sheets [191] the latency of the LOAD/STORE operations depend on the choice of the specific NIOS II Core; economy, standard or fast. The worst case latency value was used as stated for the NIOS II economy core which is 6 clock cycles for testing the proposed heuristic. The processor to cache latency is stated as one clock cycle per read in the same datasheet. Using these numbers, for a single basic block “i” with “m” LOAD/STORE operations to a BRAM based memory, the third term can be written as follows:

$$\begin{aligned}
 & (T_{PM} - T_{PC}) * Freq_i \\
 \text{or,} \quad & (6 - 1) * m_i * Freq_i \\
 \text{or,} \quad & 5 * m_i * Freq_i \tag{Eq. 5.3}
 \end{aligned}$$

Extending equation Eq. 5.3 for all the basic blocks in software trying to access the BRAM memory and generalizing the constant to “ α ” the total penalty is:

$$\sum_{(i=1)}^{(i=n, i!=X)} \alpha * m_i * Freq_i \tag{Eq. 5.4}$$

The upper limit of “i” in Equation Eq. 5.4 shows that the block “X” is assumed to be in hardware and hence its penalty need not to be considered. This equation was verified by performing experiments on the Altera DE2 board using different NIOS II configurations (economy/standard/fast). The weight value “ α ” is dependent on various properties of the FPGA device, the type of processor, the efficiency of its memory management unit etc. In the current SoC model using NIOS II processor, “ $\alpha = 5$ ” as derived in equation Eq. 5.3. In newer FPGA devices or memory technologies the latency (T_{PM}) might reduce, thereby reducing the overall penalty.

In the next section, these expressions are used to estimate the clock cycles saved by hardware acceleration while accounting for the data-dependencies, to discuss the selection step using a sample real world application, SHA from the popular MiBench [181] embedded benchmark suite.

5.4 Data-Dependency Aware Selection Of Hardware Accelerators

After identifying the potential blocks for hardware acceleration, the next step is to select them for final implementation while considering their data-dependency with the other code segments. This is achieved by using expression Eq. 5.2 and the penalty model in expression Eq. 5.4 to guide the selection process.

Figure 5.5 shows a code snippet from the sha_transform function in the SHA application C code from the popular MiBench embedded benchmark suite [181]. In this code snippet, the local variables “A” through “E” and array “W” can be easily implemented as part of Local Memory since they are copied to the stack before getting utilized. On the other hand, the structure sha_info cannot be implemented since it is a pointer to a dynamically allocated memory space. The arrays and elements

accessed by the memory operations inside the candidate basic blocks in SHA application are shown in Table 5.1 along with signs for implementable/non-implementable basic blocks. This example clearly shows the complex memory dependency across the various high frequency blocks in the application. Hence there is a need for a suitable selection algorithm that considers these data-dependencies in order to achieve maximum performance from hardware acceleration.

```
void sha_transform(SHA_INFO *sha_info)
{
    int i;
    LONG temp, A, B, C, D, E, W[80];
    .
    .
    .
}
```

Figure 5.5: A code snippet from the sha_transform function in the SHA C code.

Table 5.1: Most frequently executed basic blocks in the SHA application and corresponding arrays/elements accessed.

Name of Basic Blocks	Percentage of total runtime	Variables Accessed			Implem- entable
		Structure Sha_info	Array "W"	Scalars "A-E"	
sha_transform_entry	0.6	✓	✓	×	×
sha_transform_bb3	38.7	×	✓	×	✓
sha_transform_bb5	0.6	✓	×	✓	×
sha_transform_bb6	12.1	×	✓	✓	✓
sha_transform_bb10	12.1	×	✓	✓	✓
sha_transform_bb13	12.1	×	✓	✓	✓
sha_transform_bb16	12.1	×	✓	✓	✓
sha_transform_bb18	0.6	✓	×	✓	×

Therefore, during the selection step, it is ensured that the hardware blocks accessing the same memory location are selected as a *collection* for hardware implementation. This ensures that the penalty, estimated by the expression Eq. 5.4, of accessing the Local Memory from the Processor side is minimized, i.e. the ratio $[G/(F+H)]$ in figure 5.2 is maximized. In order to achieve this, the list of arrays and elements accessed by the candidate basic blocks obtained from the identification phase is used along with the traditional metrics for hardware acceleration (described earlier in section 5.3) to evaluate the net advantage of accelerating these blocks in hardware. The net performance by considering the advantage of hardware acceleration as well as the penalty, can be obtained by using equations Eq. 5.2 and the penalty model in expression Eq. 5.4. It is noteworthy that since entire basic blocks can be implemented in this work as HBs, inclusion of control operations is a given for our method.

On the other hand, it can be observed that blocks such as *sha_transform_entry* and *sha_transform_bb5* are un-implementable blocks for hardware acceleration in our current setup. However, since these blocks access overlapping data with the other implementable blocks, the penalty in this case can not be completely avoided. This provides an upper limit to acceleration and the proposed methodology helps in exploring the design space within this range. The selection step will be further elaborated in chapter 6. This chapter focuses on the proposed architecture to include such hardware accelerators with or without memory or control flow operations.

In order to prove the efficacy of the proposed architecture model, different applications were analyzed from the widely used MiBench [181] benchmark suite. Applications such as Patricia from the networking application set, has 3 of the top 11 blocks without any memory operations, therefore these blocks could be easily implemented as accelerators by encompassing their control operations at the end of the block in the proposed architecture. The rest of the high frequency blocks in this application have memory operations to a dynamically allocated memory space which is

not targeted by the proposed model and hence not implemented as a complete block. Even so, smaller custom instructions as proposed in chapter 4 can be selected and implemented from these blocks in the proposed model.

Similarly, for the FFT application from the same benchmark suite, the size is an argument supplied at runtime, therefore most of the variables become dynamically allocated variables in this application. Still, the top basic block with over 37% of the total execution time is completely independent of any memory and hence a perfect candidate for hardware acceleration. On the other hand, for a fixed size FFT, as used in many signal processing applications, the proposed architecture provides even better acceleration by accelerating more sections of the applications accessing fixed size arrays and variables. [135] showed the benefits of using local memory in a fixed point FFT application, however, as stated earlier they used DMA to move the data from the main memory to local memory.

The authors in [132] described the motivation of using local memory by using a security benchmark, Advanced Encryption Standard (AES) application, with a small fixed table called *Sbox* with 256 elements. The authors showed the advantage of transferring this fixed table to the local memory alongside the hardware accelerator processing it. Similarly, the SHA application from the security suite from MiBench, presents a good combination of blocks where some of the blocks access statically allocated memory while other memory accesses are made to dynamically allocated blocks. Therefore, this application will be used in the next section to illustrate the concepts proposed in this thesis.

5.5 Implementation Of Proposed Architecture Model On Altera's NIOS II Platform

This section provides a detailed description of our setup and experiments. The proposed architecture model was implemented and verified on the Altera DE2 development board containing a Cyclone 2 FPGA device. Altera provides provision for incorporating various types of custom instructions into their NIOS II soft processor including support for multi-cycle custom instructions. This is a useful feature since it allows the addition of arbitrary size custom instruction without significantly affecting the processor's operating frequency. Other processors that lack such options, can also take advantage of such algorithms by accessing a pre-defined memory space to trigger the accelerator execution. On starting execution, the accelerator can block the bus access for other operations by asserting a wait state on the bus.

As explained in the previous section, this work proposes to select custom hardware blocks based on the relative isolation of their corresponding memory locations being accessed. In order to achieve this, the arrays and variables accessed by every basic block of interest need to be identified in the first step. Only the top most frequently executed basic blocks are analyzed for this purpose. The LLVM-IR was profiled using the profiler framework in LLVM to obtain the most frequently basic blocks in the application. Table 5.1 lists the top most frequently executed basic blocks in the SHA application as obtained from LLVM along with the name of the arrays and elements accessed from these basic blocks. The array "W" in this case has 80 integer elements whereas elements "A" through "E" act as registers for storing 5 intermediate integer values. During pure software execution these arrays and elements are stored in the memory and the processor needs to access them from the main memory through the cache hierarchy. Hence, storing these variables in the local memory alongside the custom hardware block has obvious advantage of quicker access time

when the custom hardware block is used for execution, as it allows the custom instruction to include memory access. In our design, the array “W” and variables “A-E” are initialized as a pointer to a hardware location in the address space of the local memory. This location is declared as an un-cached space, forcing the CPU to always access this data directly from the local memory instead of the data cache.

In an attempt to accelerate specific portions of an application code, corresponding Verilog/VHDL codes need to be generated. Next, the corresponding “C” lines have to be replaced with a *call* to this hardware block, usually done through a “*#pragma*” declaration that runs a specific assembler instruction. An in-house tool called “IR2Verilog” was developed to convert the LLVM-IR of selected application basic blocks into the corresponding Verilog implementation. The relevant lines in the “C” code were replaced manually by inserting *#pragma* calls to the corresponding hardware block. The details of the IR2Verilog tool is explained in the appendix section on page 222. In order to reduce the cycle count of the custom instructions, a multiple port memory can be implemented to permit more than one array access per cycle. Some care would be needed to ensure the correct operation in the case of reads and write to potentially the same memory location during the same state. However, in our current implementation, the architecture has not been tested for a multi-ported memory.

In order to verify the architecture model and the proposed concepts, the LLVM IR of specific blocks from the SHA application were translated into Verilog using the aforementioned IR2Verilog tool. In our current implementation, these blocks were then incorporated with a NIOS II system as multi-cycle custom instructions. It should be noted that while the hardware blocks obtained from the basic blocks of the SHA application are connected to the NIOS II processor using the NIOS II custom instruction interface, they are not traditional custom instructions. Unlike the traditional custom instructions, our hardware blocks incorporate control and certain memory operations as discussed earlier. The NIOS II custom instruction interface allows us to easily

connect traditional custom instructions without memory or control operations. It also permits connecting larger hardware blocks with control and certain types of memory operations in order to verify the proposed architecture model and interfaces. This was done purely for the ease of implementation and verification without any loss of generality. In a different soft core processor, the selected hardware accelerators can be easily connected as co-processors based on the interface design, since the data communication in the proposed architecture model is always performed through the main system bus.

These hardware blocks are called from the SHA application C code in a series of experiments with different combinations of these blocks. A minimally intrusive, “Performance counter peripheral” was instantiated in the NIOS II system to measure the application runtime in number of clock cycles. The NIOS II system was running at 50MHz clock speed. Since the NIOS II system allows a multi-cycle custom instruction, the maximum clock speed of the entire design does not suffer significantly even with larger hardware blocks.

Figure 5.6 shows the total execution runtime of the SHA application with various combinations of the hardware blocks. The first column shows the software only execution time without any hardware instruction. Except for the first column, the combinations in each column represent the set achieving best performance with the “column-1” number of blocks. For example, the third column shows that the combination “BB3+BB6” achieves the best performance amongst all the combinations of 2 candidate blocks.

As can be seen from the graph, the application runtime generally decreases with addition of more custom hardware blocks. A jump is seen in the application runtime from “BB3” to “BB3+BB6” because from Table 5.1, basic block need to access the elements “A-E” in addition to the array “W”. Again, from the Table 5.1, it is seen that

Figure 5.6: Total Execution time (number of clock cycles) of the SHA application with various combinations of custom hardware blocks. Runtime reduces drastically with the right set of HBs.

the basic blocks “BB10, BB13, BB16” also need to operate on the variables “A-E”. When the basic block “BB6” is executed in hardware, the corresponding variables “A-E” are also stored in the local memory for quicker access. However, the processor needs to access these variables while executing basic blocks “BB10, BB13, BB16”. This is slower than accessing the main memory (via the cache) from the processor and hence the advantage of executing “BB6” as a custom hardware block is offset due to this slow access in “BB10, BB13 and BB16”. In effect, the “BB3+BB6” option increases “G”(due to use of array “W” in BB6), but also increases “H” more due to the new penalties from blocks “BB10, BB13 and BB16” accessing elements “A-E” from the un-cached memory space. This confirms that porting more blocks to hardware without analyzing their memory dependency does not always guarantee increased performance. Indeed, as shown in this example, the performance degrades by accelerating “BB6” along with “BB3”. In fact, any combination of two blocks in this example achieves worse performance than accelerating “BB3” alone due to the excessive data dependency with the blocks implemented in software.

However, as observed in the fourth column, basic block “BB6” is essential to

Table 5.2: Resource Utilization for DE2 board with various Hardware Blocks

	Logic Elements	Dedicated Logic Registers	Memory bits
All Custom Hardware Blocks(BB3-BB16) + Avalon Slave Interface + Local Memory	2662	1260	4096
Rest of the Design(NIOS II CPU, on-chip memory, performance counter peripheral, jtag uart etc.)	4893	3125	321664

achieve better performance when accelerated together with the rest of the blocks. Therefore, at each stage all possible combinations of blocks must be examined to calculate their overall performance while considering the penalty from the un-accelerated parts of the application.

It is evident that as more basic blocks “BB10” through “BB16” are ported into hardware, the application runtime reduces drastically, with the runtime of the last column being about half of the software only execution time. Table 5.2 shows the resource utilization of the hardware blocks in comparison to the rest of the design elements as reported by the Altera Quartus 11.0 tool. The total HW usage on the DE2 is less than 23% and 13% for the logic elements and registers respectively.

5.6 Summary

This chapter highlighted the data-dependency issues that need to be considered during the selection of hardware accelerators with local memory blocks along with the traditional metrics such as hardware area, time etc. In order to account for the overhead resulting from such data dependencies between the hardware and software sections of the application, a mathematical model was derived to be used during the selection step. An architecture model for an application-specific CSoC was proposed to realize traditional custom instructions as well as hardware accelerators with memory

and control operations. Unlike existing methodologies that use local memory alongside hardware accelerators, the proposed method avoids expensive DMA for data transfer between main memory and local memory for statically allocated variables. Instead the data was initialized in the local memory during the design phase and kept in un-cached pages. This also eliminates the need for additional cache-coherence sub-systems.

It was also shown that the selection of the right set of hardware accelerators from an application depends heavily on the data dependencies across the various candidate blocks. While the local memory blocks provide fast and deterministic access to the hardware accelerators, the penalty incurred by the processor while accessing the data stored in the local memory blocks must be considered during the hardware-software partitioning step in order to minimize the overall penalty. The proposed architecture model and concepts have been verified on a NIOS II soft-core processor by instantiating appropriate custom hardware blocks for the SHA application from the MiBench Benchmark suite. Additionally, it was also shown that by selecting the right set of basic blocks for hardware implementation based on their data-dependencies, the SHA application can be executed upto 47% faster when compared to the pure software implementation.

The number of candidate blocks considered in the SHA application is limited to 6, based on their execution frequency. The selection process needs to consider their memory dependencies so as to reduce the communication overhead between the blocks executed in software and hardware. In order to account for such dependencies, especially in applications with high data dependencies, manual selection of the right set of candidates for hardware acceleration is highly limiting. With the proliferation of multimedia applications, such cases will be widely prevalent. Similarly even an automated exhaustive search is not feasible, especially in larger applications with multiple candidates blocks and interleaved memory dependency. In order to cater

for such wide variety of applications, a rapid heuristic based selection method is proposed in the next chapter.

Memory-Aware Selection Of Hardware Accelerators

6.1 Introduction

In a CSOC, such as the one discussed in this research, application specific customization by adding suitable hardware accelerators is one of the popular ways of generating high performance systems with strict power and area budgets. The inclusion of hardware accelerators, however, needs a careful analysis of the application in order to identify the best segments of the application code for hardware implementation. Unlike the custom instructions discussed in chapter 4, which have traditionally avoided memory and control flow operations, hardware accelerators typically include these operations. Data manipulated by one section of the application, e.g. the hardware section, must be updated and/or synced in the appropriate location so as to ensure that the next section, e.g. the software section, uses the updated data value. This results in an overhead for the partitioning step and must be accounted for during the selection of hardware accelerators with memory. Therefore, a more complex selection criteria must be employed in order to consider the complex data dependencies between the various hardware and software sections of the application code. At the same time the large design space of the partitioning step (2^N different hard-

ware/software partitions with “N” candidate blocks) [128], largely prohibits the usage of an exhaustive selection algorithm.

In the previous chapter, a novel architecture model was proposed that was shown to be capable of including hardware accelerators with local memory elements without the need for DMA or cache-coherence systems. A mathematical model to estimate the penalty incurred by an un-accelerated code section in accessing a block of memory instantiated in the FPGA within an accelerator for a different code section was also presented and verified using a case study on a real world application. In this chapter, an automated selection framework is presented that uses this model to account for the memory dependencies of different hardware accelerators. Combined with the traditional selection metrics such as performance gain, required hardware area etc., the proposed framework automatically recommends the most suitable set of hardware blocks for implementation with varying area constraint. It is also shown that while the runtime and complexity of an exhaustive selection approach increases exponentially, our method scales almost linearly with increasing number of candidate hardware blocks. At the same time the performance achieved by the candidates recommended by the proposed framework is also shown to be typically within 5 to 10% of the exhaustive search results.

The rest of the chapter is organized as follows. A few related work in this area is reviewed in the next section followed by a discussion of the proposed selection heuristic in Section 6.3 with experimental results in Section 6.4. Our methodology is then extended in sections 6.5 and 6.6 for improved accuracy and reliability of the selection heuristics. Section 6.7 closes this chapter with a few concluding remarks.

6.2 Related work

The authors in [128], [130] and [131] have proposed automatic partitioning strategies using hardware accelerators at varying levels of granularity. They used multi-grained accelerators, where the granularity of the accelerators ranged from basic blocks to a function or process level in their design, and the data communication between the hardware and software sections was done through the usage of shared memory space. In [128], they proposed the copying of relevant data in the shared memory space each time, prior to executing the software or hardware components of the application.

In [130] the authors proposed an architecture, they termed as KAHRISMA, where hardware accelerators of varying granularity could be added to the processor at runtime in order to leverage dynamic reconfiguration of modern high-end FPGAs and realize different processors. They proposed the addition of custom instructions in the base processor ISA at runtime using dynamic reconfiguration to suit different applications. Similarly in [131], the authors proposed a run time system to incorporate different fine or coarse grained custom instructions into the extensible processor according to application requirements. It should be noted here that, while the authors mentioned the usage of hardware accelerators at varying levels of granularity, they did not include the “traditional custom instructions” which are smaller than basic blocks and have the added advantage of recurrence at multiple places in the application [93]. Additionally based on their works discussed in [128], the data communication between the hardware and software components of the application requires data movement during application execution which incurs additional cost in terms of performance and power. Also, the selection of hardware accelerators was mainly based upon software and hardware performance of the code sections. Another key difference in our design is that the maximum range of accelerator granularity is a basic block instead

of a function. This gives us greater control on the partitioned code segments and avoids accelerating certain sections of a function that are not profitable, yet need to be implemented if the entire function is accelerated.

Sun et al. in [137] also showed the benefits of using hardware accelerators of varying granularity including “traditional custom instructions” on Xtensa processors [163], a commercial configurable processor from Tensilica. However, their memory architecture was also based on shared-memory configuration and required data transfer between the different memory blocks during execution.

LegUp is a high level synthesis tool proposed by the authors in [82] that compiles a “C” program on to a hybrid architecture consisting of a MIPS based soft processor and custom hardware accelerators. The authors also demonstrated the ability of their tool to do design space exploration by varying the amount of program run in software vs. hardware. The code sections implemented in hardware were selected manually and fed into the tool as one of the inputs for design space exploration. While the tool itself is extremely useful because of its end to end design flow, memory implications during the hardware selection should be considered for better performance. It is noteworthy that the algorithms proposed in this thesis can be merged directly into such high level synthesis tools to guide the tool for automated hardware-software co-design and design space exploration.

6.3 Selection Heuristic

After the enumeration step and the derivations discussed in the previous chapter 5, the access penalty for a code section to access a data stored in the local memory is given by the equation Eq. 5.3

$$Penalty_i = \alpha * m_i * Freq_i \quad (Eq. 6.1)$$

where m_i is the number of LOAD/STORE operation in a software code block “i” with execution frequency $Freq_i$ accessing data from a local memory of hardware block “X”. The constant term “ α ” has been estimated to be 5 for the proposed CSOC model using NIOS II processor as discussed in the previous chapter. Only those software blocks which need to access the data stored in the local memory incur this penalty, since “ m_i ” is zero for the other blocks. Hardware blocks such as “X” which are assumed to be in hardware do not incur any penalty. Ideally, the partitioning algorithm should attempt to minimize this penalty by selecting the right set of hardware accelerators. This can be achieved by promoting the selection of implementable blocks with high data dependency simultaneously for hardware acceleration. The partitioning algorithm proposed in this work strives to search such closely related blocks.

Before describing our heuristic search algorithm, a few terms are defined that are used during the selection process.

(i) Max_Penalty(P_{max})

This term has to be calculated only for the implementable blocks, however the penalty must be included from both the *implementable* as well as *unimplementable* blocks currently being executed in the processor/software accessing a particular memory block. Equation Eq. 6.1 derived in the previous chapter estimates the penalty for a single block to access the uncached memory. In order to calculate the Max_Penalty for Block X, one needs to add the penalties from the blocks currently executed in software having data dependency with X. Hence,

$$P_{max} = \sum_{(i=1) \atop (i=Num_Blks, i \neq X)} \alpha * m_i * Freq_i \quad (\text{Eq. 6.2})$$

where, Num_Blks is all the blocks, *implementable* as well as *un-implementable* currently being executed in the processor/software, having data dependency with Block X. As discussed earlier for equation Eq. 6.1, m_i is the number of LOAD/STORE operation in a software code block “i” with execution frequency $Freq_i$ accessing data from a local memory of candidate hardware block “X”. This value must be updated after every iteration if a block is ported to hardware, effectively reducing its penalty to zero.

(ii) Min_Penalty(P_{min})

The minimum penalty or (P_{min}) needs to be calculated only for the implementable blocks, however the penalty is *only* included from the un-implementable blocks. This term signifies the minimum overhead of porting to a block to hardware with data dependencies with un-implementable blocks. As stated earlier, the un-implementable blocks cannot be implemented in hardware due to either design constraints or hardware limitations. The current hardware model does not implement dynamically allocated memory blocks. This value needs to be calculated only once for every candidate block using equation Eq. 6.1. Similar to calculating the max penalty in the previous term, P_{min} can be calculated using the following equation:

$$P_{min} = \sum_{(i=1)}^{(i=Num_Blks', i!=X)} \alpha * m_i * Freq_i \quad (\text{Eq. 6.3})$$

where, Num_Blks’ signifies the *un-implementable* blocks which need to be always executed in the processor/software, but having data dependency with Block X. Again, as discussed earlier for equation Eq. 6.1, m_i is the number of LOAD/STORE operation in a software code block “i” with execution frequency $Freq_i$ accessing data from a local memory of candidate hardware block “X”.

(iii) Block Advantage(*Block_Adv*)

This is the general advantage of implementing any section of code in hardware and is calculated as the difference of its software and hardware runtimes multiplied by its execution frequency i.e.

$$(SW_{time} - HW_{time}) * Freq \quad (Eq. 6.4)$$

(iv) Memory Access Advantage (*Mem_Access_Adv*)

It quantifies the memory dependency and is the advantage of executing a code section in hardware if it accesses a memory block which has already been ported to hardware alongside another accelerated block. This is again calculated using expression Eq. 6.1 each time when a block is considered for hardware execution and its relevant memory blocks have *already* been ported to hardware.

(v) Guaranteed Advantage (*Guaranteed_Adv*)

This term calculates the advantage of implementing a block of code into hardware taking into consideration all the penalties and advantages associated with its corresponding memory as well as all the penalties incurred by other blocks which overlap with its memory access. This can be obtained by the following expression:

$$Guaranteed_Adv = Block_Adv + Mem_Access_Adv - P_{max} \quad (Eq. 6.5)$$

(vi) Potential Advantage (*Potential_Adv*) This is the maximum advantage that can be obtained from implementing a block in hardware. This assumes that all the blocks that have overlapping memory dependency are also implemented in hardware except the un-implementable blocks. This is calculated using the

following expression:

$$Potential_Adv = Block_Adv - P_{min} \quad (\text{Eq. 6.6})$$

Based on the terms defined above, the heuristic selection algorithm will be described with the help of a sample application, the Secure Hash Algorithm (SHA) from the MiBench benchmark suite [181]. LLVM is used to compile and profile the SHA application to identify the most frequently executed basic blocks in the application. Table 6.1 lists the 6 most frequently executed basic blocks in the SHA application. These 6 blocks constitute the 87.85% of the total runtime of this code.

Table 6.1: Most frequently executed basic blocks in SHA application

Name of Block CI	Dynamic Frequency
sha_tranform_bb3	311872
sha_tranform_bb6	97460
sha_tranform_bb10	97460
sha_tranform_bb13	97460
sha_tranform_bb16	97460
sha_transform_bb5	4873

Figure 6.1 shows the flowchart for the proposed algorithm. After identifying the high frequency blocks in the SHA application, the Implementable and Un-Implementable blocks need to be identified and separated. For the SHA application while the first 5 blocks contain only statically allocated arrays and the last block, sha_transform_bb5 needs data to be loaded from dynamically resolved locations. Figures 6.2.a and 6.2.b show excerpts from the two basic blocks to highlight this difference. Figure 6.2.a contains the part of the sha_transform_bb3 basic block showing the access from a statically defined array “W” with 80 32-bit integer values. This is considered as an Implementable block. On the other hand, statements in figure 6.2.b are from sha_transform_bb5 in which the structure sha_info points to a dynamically allocated memory space. Therefore sha_transform_bb5 is considered as an Un-implementable block.

Figure 6.1: Flowchart for the proposed selection algorithm.

The identification of un-implementable blocks is important for the calculation of $Min_Penalty$ (P_{min}). For example consider the basic block sha_transform_bb6, part of which is shown in figure 6.2.c. Even though basic block sha_transform_bb5 cannot be implemented in hardware, it loads data to a variable [%8] used by basic block sha_transform_bb6. In the specific case of the SHA example, basic blocks sha_transform_bb6,10,13 and 16 manipulate 5 integer values A, B, C, D and E which is loaded by the sha_transform_bb5. Therefore one must consider the fixed penalty from the sha_transform_bb5 (an un-implementable block) before

Figure 6.2: Excerpt from SHA application

deciding to implement the other blocks in hardware. In other words, basic blocks sha_transform_bb6,10,13 and 16 should only be implemented in hardware if their advantage supersedes this fixed penalty.

In the next step after separating the implementable and un-implementable blocks, the various metrics of these basic blocks listed in Section 5.3 of the previous chapter

need to be estimated/obtained. The specific method used to obtain these values is not critical to the selection algorithm and therefore, not a focus of this thesis. Section 5.3 suggested a way to rapidly estimate these values using the methods described by other authors such as [122]. For the purpose of this case study, these values were estimated manually for the SHA application from the LLVM IR of the individual blocks.

Table 6.2: Hot blocks in SHA application and their metrics

Information about the Hardware Block: Name, Frequency, SW/HW time, HW Size etc					Memories Accessed with their size					
Name	Frequency	HW Time	SW Time	HW Size (Logic Blocks)	W	A	B	C	D	E
sha_tranform_bb3	311872	12	21	10	5	0	0	0	0	0
sha_tranform_bb6	97460	6	25	6	1	1	1	1	1	1
sha_tranform_bb10	97460	6	25	6	1	1	1	1	1	1
sha_tranform_bb13	97460	6	25	6	1	1	1	1	1	1
sha_tranform_bb16	97460	6	25	6	1	1	1	1	1	1
sha_tranform_bb5	4873	0	0	0	0	1	1	1	1	1
Mem_Size	0	0	0	0	320	32	32	32	32	32

Table 6.2 shows the SHA application with its estimated metrics, various arrays/variables, their size and the static access frequencies. After obtaining these values, the selection process is initiated. The selection process tries to select at max one basic block in every iteration, provided the overall advantage of selecting this block is more than the penalties from the other blocks that are not implemented yet, and therefore currently being executed in software. As mentioned earlier, this is called the “Guaranteed advantage” of this block. This guaranteed advantage is calculated for all the blocks that have not been moved into hardware yet. On the other hand, if the penalties are greater than the overall advantage, then the block with its highest potential advantage is kept as a potential candidate for the next iteration. In the subsequent iterations, this potential block is considered to have been ported in hardware

Table 6.3: Selected basic blocks in each iteration of SHA application

Iteration number	Guaranteed Basic Blocks selected in this iteration	Potential Blocks	All selected blocks
1	Block 3	NILL	Block 3
2	NILL	Blocks 16	Block 3
3	NILL	Blocks 13 & 16	Block 3
4	Blocks 13, 16 & 10	NILL	Blocks 3, 16, 13 & 10
5	Block 6	NILL	Blocks 3, 16, 13, 10 & 6

and other suitable blocks are searched to go along with it in hardware. Eventually, when this set is found, the entire set is chosen to be ported to hardware.

At the end of each iteration the overall performance is calculated, whether a block is selected or not. If no block could be selected at any iteration, the performance remains equal to the last iteration. The process continues till the set of implementable blocks is exhausted. At the end of this selection process, a performance area curve is obtained that can help a designer to select the application portions to accelerate in hardware given a particular area constraint. This can also be useful to determine the maximum performance that can be achieved by hardware acceleration for any application.

The selection process for the SHA application in every iteration is shown in Table 6.3. Based on the memory dependency across basic blocks as shown in Table 6.2, Block 3 is selected in the first step since it has the most isolated memory dependency compared to any other implementable block. In the second step however no block can be selected, since there is overwhelming penalty from other blocks sharing exactly the same memory locations. Block 16 is hence selected as a potential candidate. As described earlier, in the subsequent iterations, Block 16 is considered as an already implemented block and hence no penalty is considered from this block. The objective is to find a set of blocks which can ported to hardware in one go to achieve maximum performance while incurring minimum penalty.

In the third iteration even after considering two of the remaining blocks, such as, Block 13 and 16 the penalty is still more than the overall advantage and therefore still no additional blocks are selected for implementation. Blocks 13 and 16 are now treated as potential blocks together. In the fourth iteration however, blocks 16, 13 and 10 can be selected together since their combined advantage exceeds the penalty from the other blocks. The last remaining block, Block 6, is selected in the 5th and last iteration. The corresponding performance achieved and its comparison with an exhaustive search methodology for every iteration is presented in Figure 6.4.

As shown in the previous chapter, this behavior was also verified by selectively implementing these basic blocks of the SHA application in FPGA while the rest of the application was executed on the soft-core NIOS II processor running on Altera DE2 FPGA board. Figure 6.3, shows the total execution time of the SHA application with selective combinations of its hot blocks running in the FPGA logic blocks as hardware accelerators. As is evident by the figure and the discussions above, selecting the right set of candidates for hardware acceleration is paramount to any high performance system design.

This work also provides a way to do this selection using a greedy exhaustive approach. In the exhaustive approach all the possible combinations of blocks are tried. For example in the first iteration of the exhaustive search, the performance achievable by each block is analyzed and the one with the best positive performance is selected. Again it should be noted that when a particular block is being considered during an iteration, it is assumed that the rest of the blocks are in software and thereby incurring penalty. Only when the advantage of implementing this block in hardware is greater than the penalties from all the other blocks, that this block is selected for hardware implementation. The exhaustive search proceeds by looking at the exact number of candidates in the combination as the iteration number. For example, all

Figure 6.3: Total Execution time of the SHA application with various combinations of custom hardware blocks. Runtime reduces drastically with the right set of Hardware blocks.

possible combinations of 2 blocks in the second iteration, 3 blocks in third iteration and so on, until all the blocks are considered.

Figure 6.4: Performance achieved by Exhaustive and Heuristic Search algorithms.

Figure 6.4 shows the performance curve for the SHA application using the pro-

posed heuristic search as well as an exhaustive search algorithm. In iterations 2 to 3 there was no new block selected because of greater penalties than advantage. Overall the heuristic search tracks the exhaustive results with great accuracy, while being significantly faster especially for larger applications as shown in the subsequent section.

6.4 Experiments And Results

This section discusses the methods used to evaluate the proposed heuristics. The experiments started with the SHA application to run it through the phases as described in the earlier sections. The SHA application was chosen as existing work has shown that it significantly benefits from hardware acceleration [192]. This work has refrained from evaluating the proposed method with other applications from benchmark suites as that would require manual estimation of the metrics such as hardware area, software/hardware performance etc. for every candidate block. Section 5.3 discussed about estimating these values. This is time consuming, especially when high estimation accuracy is desired and is not the focus of this work. Here, it is assumed that the designer can obtain these values before using the proposed partitioning algorithm. Therefore, instead of using a few applications from benchmark suites such as MiBench [181] or CHStone [193], the experiments were performed on synthetically generated applications. The idea was to test the proposed heuristic algorithm with thousands of applications covering varying application size, number of memory blocks, memory dependency etc. In order to create this comprehensive application set, observations were made on the regular embedded applications specifically from the widely used MiBench and CHStone benchmarks suites. These applications exhibited the following general characteristics which forms the basis of our synthetic application generator.

- (i) Number of basic blocks with significant execution frequency varies from about 10 in the smaller applications such as SHA, FFT etc to approximately 30-40 in larger applications such as MPEG IV, JPEG etc.
- (ii) The number of basic blocks amongst this high frequency group that can be implemented in hardware in its entirety is determined by design constraints and the type of memory allocation. As discussed earlier, currently only statically allocated memory blocks are considered for our hardware model. The number of implementable and un-implementable blocks is application specific characteristics and therefore this ratio is varied to simulate different types of applications.
- (iii) Total number of memory locations (that can be implemented in our hardware model) in the application typically varies from 5 to 15.
- (iv) The memory access pattern of the different basic blocks to the various memory locations is totally an application specific characteristic and therefore a specific trend in this was not observed. Hence, a larger number of dependency scenarios were simulated. This motivated us to test the proposed heuristic on thousands of applications instead of a few from benchmark suites.

The memory dependency and the ratio of implementable and un-implementable blocks is kept random in all the applications. Based on these constraints, the following application sets were generated as shown in Table 6.4.

Table 6.4: Sample application sets

Sets	No. of Apps	No. of Basic Blocks	No. of Mem. Blocks
SetA	1000	8	6
SetB	1000	15	5
SetC	1000	20	10
SetD	100	25	8
SetE	14	30	15

As can be seen from the data set, a diverse mix of applications have been generated with varying number of implementable/un-implementable blocks, memory blocks and random memory dependencies. The size of the data set is kept large to ensure that it captures the typical application behavior during the testing of the proposed heuristic. In order to compare the results from the proposed heuristic, an exhaustive selection is also performed for every application. In the exhaustive selection, the same metrics are used to calculate the advantages and penalties of each block as described in section 5.3 (page 123), 5.3.2 (page 130) and 6.3 (page 150). However during the selection process every possible combination of the basic blocks is tried for hardware implementation. The performance obtained from these combinations of basic blocks is compared with the performance obtained from the combination provided by the proposed heuristic search. The average difference between the two methods is calculated to measure the accuracy of our heuristic. Since the exhaustive selection method tries all possible combination of the basic blocks to find the best solution (combination of basic blocks), the complexity of the method rises exponentially with increasing number of blocks. Therefore, while our method scales almost linearly with the increasing number of blocks, it was increasingly difficult to perform the exhaustive method on larger applications. Due to this reason, our data set of applications with 30 blocks is much smaller than the other sets. The experiments were performed on a Linux machine with an Intel Core 2 CPU running at 3 GHz and 8 GBs of RAM.

Figure 6.5 shows the comparison between the time taken by the exhaustive and the proposed methods. As can be seen clearly from the graph and the data underneath, the time taken by the proposed heuristic search is negligible compared to the exhaustive search especially for larger applications. The average runtime of Zero for a few cases indicate that the runtime is negligibly small.

Next let us compare the performance achieved by the blocks recommended by both methods. Figure 6.6 show the histogram plot of the average difference for the

Figure 6.5: Runtime comparison between Exhaustive and Heuristic Search algorithms.

various data sets. A larger number of applications fall in the bin with Zero difference between the performance achieved by the proposed and exhaustive search methods. There are a few outliers in the bin with large average difference e.g. in Sets C and D, but most of the applications fall within the 15-20% performance difference. Upon closer inspection of the applications with large performance difference, it was observed that in applications with extreme memory dependencies i.e. almost all the blocks are accessing all the memory blocks with similar frequency, an error in selecting the block in the first few iterations has a cumulative effect. It should be noted that the algorithms were tested on a very large data set (total 3115 applications), and therefore the percentage of applications within the difference bound is a more realistic metric than the absolute number of applications in any bin. The results clearly show the effectiveness of our heuristic search algorithm while being orders of magnitude faster than an exhaustive search.

Next, efforts were made to improve the accuracy of selection heuristic. In the subsequent sections, two extensions to our heuristic will be introduced to improve the accuracy of the selection heuristic, while also being rapid and reliable.

6.6.a: 8 Blocks

6.6.b: 15 Blocks

6.6.c: 20 Blocks

6.6.d: 25 Blocks

6.6.e: 30 Blocks

Figure 6.6: Performance achieved by Exhaustive and Heuristic Search algorithms.

6.5 Two-Pass Based Heuristic Search

The previous proposed a selection heuristic to rapidly and reliably recommend a set of hardware blocks by considering their often interleaved memory dependencies. While the selection heuristic ran orders of magnitude faster than an exhaustive search method, there were a few outliers with large difference in performance achieved by the blocks recommended by the heuristic and exhaustive algorithms. This section proposes an extension to the original heuristic to improve the accuracy of the proposed search algorithm.

In the next subsection, the motivation for a Two-Pass based heuristic search algorithm is described by highlighting the limitation of the previous heuristic search using a synthetically generated sample application. This will be followed by a description of our Two-Pass based heuristic algorithm in the subsequent section.

6.5.1 Motivation For Two-Pass Based Heuristic Search

Table 6.5 shows the hot blocks of a sample application. It has 6 implementable blocks and 1 un-implementable block. There are a total of 6 memory variables that these blocks access with the frequency as shown in the table. This application represents a very realistic scenario where several blocks tend to access the same variables with different frequency. As discussed earlier, this poses a difficult challenge to the designer while selecting limited number of blocks for acceleration.

Using the concepts discussed earlier, Figure 6.7 shows the performance comparison of blocks selected using the proposed heuristic search and the exhaustive search. As seen from the figure after the second iteration, the performance achieved by the blocks recommended by the proposed heuristic does not match with the performance achieved by the blocks recommended by the exhaustive search. In other words, the

Table 6.5: Hot blocks in a sample application and their metrics

Information about the Hardware Block: Name, Frequency, SW/HW time, HW Size etc					Memories Accessed with their size					
Name	Frequency	HW Time	SW Time	HW Size (Logic Blocks)	A	B	C	D	E	F
Block 0	311872	12	21	320	5	0	0	0	0	0
Block 1	207460	6	25	192	1	1	0	0	1	0
Block 2	157460	6	25	192	0	1	1	0	0	0
Block 3	97460	6	25	192	0	0	1	1	0	0
Block 4	97460	6	25	192	2	2	0	0	1	1
Block 5	37460	6	25	192	10	0	0	0	0	5
Block 6	4873	0	0	0	0	1	1	1	1	1
Mem_Size					320	32	32	32	32	32

heuristic search does not recommend the correct set of blocks for acceleration.

In order to explain this behavior let us consider the blocks selected by the heuristic and exhaustive approaches in each iteration. Table 6.6 shows the blocks selected in each iteration by using heuristic and exhaustive search algorithms in the sample

Figure 6.7: Performance achieved by Sample application using Exhaustive and Heuristic Search algorithms in First Pass.

Table 6.6: Selected basic blocks in each iteration of Sample application

Iteration number	Heuristic Search Results			Exhaustive Search Results
	Guaranteed Basic Blocks selected in this iteration	Potential Blocks	All selected blocks using Heuristic search	All selected blocks using Exhaustive Search
1	Block 3	NILL	Block 3	Block 3
2	Block 2	NILL	Blocks 3,2	Blocks 3,2
3	Block 0	NILL	Blocks 3,2,0	Blocks 0,1,4
4	Block 1	NILL	Blocks 3, 2, 0 & 1	Blocks 0, 1, 4 & 5
5	Block 4	NILL	Blocks 3, 2, 0, 1 & 4	Blocks 0, 1, 2, 4 & 5
6	Block 5	NILL	Blocks 3, 2, 0, 1, 4 & 5	Blocks 3, 2, 0, 1, 4 & 5

application.

Using the heuristic search described in Section 6.3, the selection starts with Block 3 in the first iteration based on various advantages/penalties discussed earlier. It then selects blocks 2, 0 and so on, in subsequent iterations based on the earlier blocks that are already selected i.e. the search continues to look for a new block to match with the previously selected blocks. From Table 6.6, it can be seen that in the third iteration, the heuristic search selects Blocks 0, 2 and 3, while the exhaustive search selects Blocks 0, 1 and 4. This means that the exhaustive search is able to select a set with different blocks leading to better performance as compared to the heuristic search. The heuristic search prunes this best set (Blocks 0,1 & 4) when starting with Block 3 in the first iteration and selecting new blocks based entirely on the earlier selections, thereby leading to a local maxima in terms of performance. A wrong selection in the beginning creates cumulative errors in the subsequent iterations. On the other hand, the exhaustive algorithm searches through all combinations and can select a totally different set of blocks which achieves the best performance at each stage.

In order to avoid getting stuck in such local maxima points, a second pass is performed in our search algorithm which treats the results from the previous heuristic search as a starting point. The next subsection discusses this method and explains

Block 3 $\Rightarrow$ Block 2 $\Rightarrow$ Block 0 $\Rightarrow$ Block 1 $\Rightarrow$ Block 4 $\Rightarrow$ Block 5

Figure 6.8: First pass selection thread of the sample application

a way to overcome the limitation of our proposed algorithm.

6.5.2 Two-Pass Based Heuristic Search Algorithm

In this work, the blocks of an application are considered as a “node” in a “thread”, whereby a thread is formed by connecting or “selecting” one or more blocks in every iteration. The concepts of “node” and “thread” in this context is explained using the sample example discussed in Section 6.5.1. As discussed earlier, the hot blocks in the sample application consist of 6 *implementable* blocks as shown in table 6.5 that could be implemented in hardware. Table 6.6 shows the basic blocks selected in each iteration of the proposed selection algorithm. During the generation of application thread, even the blocks selected as potential candidates are considered as a valid node. This follows from the fact that a block selected as a potential candidate is a better match with the already implemented blocks than any other block in the unselected pool. Based on this observation and using the table 6.6, the thread for the sample application was created as shown in Figure 6.8.

From this thread, one can conclude, for example that the Blocks 2 has a greater affinity with block 3 than any other block. Therefore the nodes in closer proximity along the thread have greater match in terms of their memory dependencies than the blocks farther apart. Based on this observation the “two pass method” can be described as follows.

In the first pass the thread is created as in the previous approach. In the second pass however a “sliding window” is moved along this thread created from the first pass and find suitable combinations of blocks of different numbers such as combinations

Table 6.7: Combinations of blocks tested in each iteration of Sample application in the second pass

Iteration number	Heuristic Search Results		Exhaustive Search Results
	Combinations of Basic Blocks tested in this iteration	All selected blocks using Heuristic search	All selected blocks using Exhaustive Search
1	(3); (2); (0); (1); (4); (5)	Block 3	Block 3
2	(3,2); (2,0); (0,1); (1,4); (4,5)	Blocks 3,2	Blocks 3,2
3	(3,2,0); (2,0,1); (0,1,4); (1,4,5)	Blocks 0,1,4	Blocks 0,1,4
4	(3,2,0,1); (2,0,1,4); (0,1,4,5)	Blocks 0, 1, 4 & 5	Blocks 0, 1, 4 & 5
5	(3,2,0,1,4); (2,0,1,4,5)	Blocks 0, 1, 2, 4 & 5	Blocks 0, 1, 2, 4 & 5
6	(3,2,0,1,4,5)	Blocks 3, 2, 0, 1, 4 & 5	Blocks 3, 2, 0, 1, 4 & 5

of 2 blocks, 3 blocks, 4 blocks and so on and so forth. The size of the window is kept equal to the number of blocks that need to be selected in the particular iteration. For example, while selecting 3 blocks at a time, a sliding window of length equal to three nodes is moved along the thread and so on. Since the first pass aligns the basic blocks based on their memory dependency, the second pass has a much greater probability of selecting the best set of candidates for any combinations.

Using this concept on the sample application of Table 6.5, the combinations tested in the second pass are shown in Table 6.7. The combinations are selected from nodes that lie *adjacent* along the thread 6.8 generated from the first pass.

As can be seen from the Table 6.7 now the blocks selected from the heuristic search algorithm matches with the blocks recommended from the exhaustive search. This is also confirmed by the performance comparison between the blocks from heuristic and exhaustive searches as shown in Figure 6.9.

Using this two pass heuristic approach on our sample data set consisting of 3115 applications as described in section 6.4 the average error reduces significantly. Figure

Figure 6.9: Performance achieved by Sample application using Exhaustive and Heuristic Search algorithms after Second Pass.

6.10 shows the histogram plot of the average performance difference for the various data sets after applying the second pass method. The reduction in error is especially pronounced in applications with 20, 25 and 30 blocks as compared to Figure 6.6. The extra time taken to evaluate the additional combinations is still negligible compared to the exhaustive search since a large number of combinations are still pruned in this heuristic.

It is evident from Figure 6.10 that the proposed heuristic algorithms can achieve very similar performance when compared to an exhaustive approach while being orders of magnitude faster. While the error bounds after applying the second pass heuristic falls significantly when compared to a single pass heuristic, there are still applications, such as in Figures 6.10 (c) and (d), with large performance difference between the blocks recommended by heuristic and exhaustive search. Similar to the applications causing large error in heuristic search using the single pass heuristic, these applications were found to have blocks with extreme memory dependencies i.e. almost all the blocks are accessing all the memory blocks with similar frequency.

6.10.a: 8 Blocks

6.10.b: 15 Blocks

6.10.c: 20 Blocks

6.10.d: 25 Blocks

6.10.e: 30 Blocks

Figure 6.10: Performance comparison between Exhaustive and Heuristic Search algorithms after second pass.

Hence, an erroneous selection of the block in the first few iterations has a cumulative effect. The next section discusses yet another improvement to the proposed algorithms in order to increase their reliability, while still being tremendously fast when compared to the exhaustive algorithm.

6.6 Iterative Technique For Improved Performance

In the previous section, a technique was proposed to improve the accuracy of the originally proposed heuristic search for selecting suitable basic blocks from the application for hardware acceleration, while considering their memory dependencies. The originally proposed heuristic, while being orders of magnitude faster than an exhaustive search approach, did not achieve good estimation accuracy for a few sample applications. The first enhancement to this heuristic discussed in section 6.5 achieved significantly better results, but still had a few applications with large difference in performance achieved by the blocks recommended by heuristic and exhaustive algorithms. In this section, a second enhancement to the heuristic is presented to ensure that almost all applications lie within the 15 to 20% performance difference, while most of them being within 5 to 10% difference.

The heuristic search algorithm introduced in section 6.3 starts with selecting a single block with the maximum positive guaranteed advantage in the first iteration. This is repeated in every iteration to ensure that the selection of a new block for acceleration does not incur more penalties from other blocks(due to memory overlap) than the performance gained by hardware implementation of the block. This helped us in selecting appropriate blocks in every iteration with increasing performance gain. However, as discussed in section 6.5.1 this could also lead to getting stuck in a local maxima since the blocks selected in the previous iteration solely dictate the selection of subsequent blocks.

Section 6.5.2 discussed the use of blocks selected in the first pass as a “node” in a “thread”, which helps in aligning the blocks based on their memory dependencies. In the second pass, a sliding window is moved along this “thread” to select the best combination in every iteration. Since the blocks are aligned according to their memory dependencies in the first pass, the second pass helps in selecting the best set without a need to solely rely on earlier selections. This, as was shown in last section, improves the estimation accuracy significantly. However, there are still a few applications, such as in Figures 6.10 (c) and (d), with large performance difference. This is due to the fact that, while the Two pass heuristic does not rely solely on the blocks selected previously, it is still dependent on the thread created in the first pass. If the thread created in the first pass itself is grossly erroneous, then the second pass might not be able to find the best solutions. In order to overcome this limitation, a new approach is proposed that uses multiple threads, iteratively, from the first pass and using the two pass method on these threads to identify the best candidates for acceleration.

The details of this algorithm is explained in the next section and also introduce a way to find the optimum number of threads to consider from the first pass.

6.6.1 Multi-thread Based Heuristic Search Algorithm

In order to generate multiple threads from the first pass to be considered in the second pass, the search began by selecting different candidates in the first iteration of the first pass. Let us consider the sample application described on page 169 in section 6.5.1. The thread obtained for this application from the first pass was shown in 6.8. In order to generate this thread, the blocks were treated as node, where the placement of the node in the thread depends on the iteration in which it was selected. The blocks themselves were selected based on the largest positive guaranteed advantage at every iteration.

Table 6.8: Guaranteed advantage of all blocks in first iteration (first pass) of Sample application

Name	Freq.	HW Time	SW Time	HW Size (Logic Blocks)	Block Adv.	Max. Penalty	Guaranteed Advantage	Min. Penalty	Potential Advantage
Block 0	311872	12	21	320	2806848	3884900	-1078052	0	2806848
Block 1	207460	6	25	192	3941740	12942330	-9000590	48730	3893010
Block 2	157460	6	25	192	2991740	2547930	443810	48730	2943010
Block 3	97460	6	25	192	1851740	836030	1015710	48730	1803010
Block 4	97460	6	25	192	1851740	14578595	-12726855	73095	1778645
Block 5	37460	6	25	192	711740	10320365	-9608625	24365	687375
Block 6	4873	0	0	0	0	1	1	1	1
Mem_Size					320	32	32	32	32

Thread 1	Block 3 ⇒ Block 2 ⇒ Block 0 ⇒ Block 1 ⇒ Block 4 ⇒ Block 5
Thread 2	Block 2 ⇒ Block 3 ⇒ Block 0 ⇒ Block 1 ⇒ Block 4 ⇒ Block 5
Thread 3	Block 0 ⇒ Block 1 ⇒ Block 4 ⇒ Block 5 ⇒ Block 2 ⇒ Block 3
Thread 4	Block 1 ⇒ Block 4 ⇒ Block 0 ⇒ Block 5 ⇒ Block 2 ⇒ Block 3
Thread 5	Block 5 ⇒ Block 0 ⇒ Block 1 ⇒ Block 4 ⇒ Block 2 ⇒ Block 3
Thread 6	Block 4 ⇒ Block 1 ⇒ Block 0 ⇒ Block 5 ⇒ Block 2 ⇒ Block 3

Figure 6.11: Selection threads of the sample application

In order to create multiple threads, all the blocks were arranged in decreasing order of their guaranteed advantage in the first iteration of the first pass. Table 6.8 shows the guaranteed advantage of the various blocks in the first iteration. The different threads were generated by using the blocks one at a time as the starting point of the thread. For this sample application, the various threads have been shown in figure 6.11. The total number of threads considered at this point is equal to the number of implementable blocks in the application.

6.12.a: 8 Blocks

6.12.b: 15 Blocks

6.12.c: 20 Blocks

6.12.d: 25 Blocks

6.12.e: 30 Blocks

Figure 6.12: Performance comparison between Exhaustive and Heuristic Search algorithms using multiple threads (Number of threads is equal to the number of implementable blocks).

It should be noted that only the first node in the thread was selected this way, the rest of the nodes were selected based on their guaranteed advantage, memory dependencies, etc. as in the original algorithm. Once these threads are generated in the first pass, they are fed into the second pass algorithm to further improve the accuracy. A final selection of blocks is made based on the combinations obtained after the second pass using all the threads.

Figure 6.12 shows the results after using the multi-threaded approach for the data set consisting of 3115 applications as described in section 6.4. As can be seen from the figures, the difference in performance achieved by the blocks recommended by the heuristic and exhaustive methods has improved significantly, better than the Two Pass heuristic approach described in section 6.5. The maximum number of threads from the first pass was used for each application to obtain the best set of blocks for hardware acceleration. As described above, the maximum number of threads in this case is equal to the number of implementable blocks in the individual applications.

In order to show the timing comparison between the exhaustive search and using multiple threads in the heuristic search, let us focus on larger applications. The run time for heuristic search for smaller applications tend to be too fast as shown in figure 6.5 to show the difference meaningfully. Therefore, the application set with largest applications was considered i.e. set with 30 blocks. Figure 6.13 shows the runtime comparison with varying number of threads for the 14 applications from 'SetE' as described in Table 6.4. The number of threads were varied from 1 to maximum number of implementable blocks, generally at steps of 5. The X-Axis shows the various applications, while the Y-Axis represents the run time in seconds for using different number of threads in each application. The last column in each application shows the run time of the heuristic search algorithm when maximum number of threads are used. The maximum number of threads is different for various applications since each application can have different number of implementable blocks.

Figure 6.13: Runtime comparison with varying number of threads in multi-thread approach.

As expected, the figure shows a linear relationship between the execution time and number of threads in each application. However, as observed in Figure 6.14, the improvement in the accuracy of the heuristic search plateaus after a fewer number of threads. This suggests that a fewer number of threads might suffice to achieve the best results within least amount of time. It should be noted that the accuracy of the heuristic search is very high, even achieving perfect results in many cases. This proves the reliability of the heuristic algorithms proposed in this research. The next section presents a way to find an optimum number of threads to use for any application.

6.6.2 Determining Optimum Number Of Threads

The previous subsection described the use of multiple threads from the first pass to increase the accuracy of our heuristic search algorithm. Obviously, increasing the

Figure 6.14: Performance comparison with varying number of threads in multi-thread approach.

number of threads used for this purpose also results in longer execution time for the search algorithm. In effect, the heuristic algorithm moves closer to the results obtained by the exhaustive search but consequently needs longer time than a single thread heuristic search proposed earlier. Hence, there is a need to decide upon an optimum number of threads required to get the estimation accuracy within a fixed value, while also being fast. This subsection proposes a way to obtain the optimum number of threads.

In order to calculate the optimum number of threads, the performance achieved by the hardware blocks selected using “n” number of threads is compared, where “n” varies from 1 to the “maximum number of implementable blocks” in the application. The performance is calculated based on the guaranteed advantages and penalties as discussed in earlier sections. When n is equal to 1, it essentially means that a single thread is used with two passes as proposed in section 6.5. Whenever an extra thread is added to the framework, the overall performance is calculated again and compared

Table 6.9: Runtime Comparison With Varying Number Of Threads vs. Exhaustive Search

Appln. #	Runtime in seconds							
	1 Thread	5 Threads	10 Threads	15 Threads	20 Threads	25 Threads	Max Threads	Exhaustive Algorithm
1	0.03	0.13	0.26	0.39	0.52	0.65	0.65	1583.34
2	0.02	0.10	0.20	0.31	0.41	0.47	0.47	354.32
3	0.02	0.09	0.18	0.27	0.36	0.40	0.40	167.14
4	0.02	0.08	0.16	0.24	0.32	0.33	0.33	78.53
5	0.02	0.12	0.23	0.35	0.46	0.56	0.56	736.73
6	0.03	0.13	0.26	0.39	0.52	0.64	0.64	1580.16
7	0.03	0.15	0.30	0.45	0.60	0.74	0.77	10800.00
8	0.02	0.08	0.16	0.23	0.31	0.33	0.33	77.60
9	0.03	0.15	0.29	0.44	0.59	0.73	0.76	10900.00
10	0.02	0.08	0.16	0.24	0.31	0.33	0.33	78.08
11	0.03	0.13	0.26	0.39	0.52	0.65	0.65	1590.42
12	0.03	0.17	0.33	0.50	0.66	0.83	0.90	2400.00
13	0.03	0.15	0.29	0.44	0.59	0.73	0.76	1800.00
14	0.03	0.13	0.26	0.38	0.51	0.64	0.64	1572.86

with the previous performance value. The algorithm terminates when the consecutive performance values do not increase beyond a pre-specified threshold (usually within 1 to 2 %), i.e. adding any extra thread might not result in performance gain. As can be seen from Figure 6.14, for the application set with 30 blocks, the optimum number of threads generally lie between 5 to 10, whereas the maximum number of implementable blocks could be upto 30 or even more for larger applications.

It should be noted that an exhaustive search essentially evaluates every possible combination. The heuristic search tries to minimize the number of combinations that need to be evaluated, while striving not to prune away the best set of candidates. The number of sets evaluated in the proposed three algorithms namely, One Pass, Two pass and multiple thread approach is still much lesser than the exhaustive approach. This is also evident by the runtime of the different algorithms as shown in Figure 6.13 and Table 6.9 depicting the run-time of heuristic algorithm using multiple threads in the application set with 30 blocks and exhaustive algorithm.

This proves the effectiveness of the proposed heuristic to select hardware ac-

celerators while considering their complicated memory dependencies. The CSoc architecture model proposed in chapter 5 allows us to implement specific types of variables, such as statically allocated variables etc., as part of the local memory implementation. Although the scope of accelerating an application with only these types of variables is limiting, it is not severely detrimental in embedded applications which are written for a limited memory space. Moreover, the proposed constraints can be easily incorporated by a software programmer during application development in order to take advantage of our hardware model, with minimal effort.

6.7 Summary

In this chapter, the mathematical model derived in the previous chapter was used to propose a rapid heuristic search algorithm to automatically and reliably select the suitable code blocks in an application for hardware implementation while considering their often convoluted memory dependencies. It has been shown that the heuristic based approach is extremely fast and achieves comparable performance to an exhaustive approach. It is also noteworthy, that even though the penalty model proposed in the previous chapter was used to account for the overheads of complex memory dependencies in an application, the proposed search techniques can also be used with other penalty models. In other words, the heuristic search algorithm is independent of the technique used to estimate the overhead, therefore lending itself to a wider variety of architecture models such as the ones proposed in existing work discussed in chapter 5.

Two extended methodologies were also proposed for better accuracy and reliability of the solutions provided by our heuristic search algorithm. From the experiments performed on synthetically generated benchmarks, it was seen that the accuracy of the selection algorithm improved significantly with the extensions, often achieving

perfect results and generally achieving over 90% of performance from an exhaustive approach. With increasing complexity of embedded applications, such a framework can prove extremely useful since an exhaustive selection methodology will require prohibitively long runtime. The proposed methodologies would permit its use during iterative development cycles with an exhaustive search only being necessary if final performance is lacking. This is very similar to the trial and final routing approach taken in modern large scale ASIC designs.

Conclusion And Future Research

7.1 Conclusion

In this thesis several novel techniques have been proposed for the rapid realization of area-time efficient solutions on Configurable System-on-Chip (CSoC) platforms. The proposed techniques can be efficiently deployed to rapidly generate a high performance system with minimal designer intervention.

An efficient technique for instruction subsetting was first employed to realize a processor with minimum area without compromising performance. An estimation-based analysis was then proposed to accelerate the subsetting process without the need to rely on lengthy synthesis-based methodologies described in existing state-of-the-art techniques. The proposed approach relies on a one-time pre-characterization step of a given off-the-shelf soft-core processor to capture the influence of various functional units such as hardware multiplier, floating point units, etc. on the compute performance of an application. The pre-characterization stage was capable of successfully establishing the indirect dependencies between the application and functional units. This information was used to eliminate those redundant functional units. Experimental results obtained using widely used benchmarks showed that the proposed framework can be reliably deployed to minimize the area of a soft-core processor without

sacrificing performance. The proposed methodology can be easily extended to consider additional functional units and other micro-architectural features such as cache configurations for improved performance estimation.

In order to realize area-efficient custom instructions, a novel technique that considers the LUT structure of the target FPGA device during the pattern enumeration phase, was proposed. This allowed for the enumeration strategy to identify only those patterns that could fit into exactly one logic block of the target FPGA, thereby maximizing performance per unit FPGA area. Due to their relatively small size (3 to 5 basic operations), such patterns typically recur in multiple places of an application, leading to notable performance improvement. A pattern selection algorithm was also proposed to select fine-grained custom instructions (FGCI) that comprised of the largest set of instructions which could fit onto a single logic block of the target FPGA. This approach led to the selection of custom instructions with over 65% performance gain when compared to those obtained using the existing approaches for FPGA designs, especially with tight area constraints. It was also shown that the additional constraints used in the proposed strategies have resulted in the runtime reduction of the pattern enumeration and selection phases by 30% and 41% respectively on average when compared to existing state-of-the-art methods. The FGCI enumeration approach was subsequently extended to include larger custom instructions with high ILP but with relatively smaller critical path. A new template selection heuristic was also proposed to facilitate the selection of such coarse-grained custom instructions (CGCI) with high degree of parallelism for designs with less stringent area constraints. It was observed that such custom instructions do contribute to overall performance gain although they do not recur as frequently as those fine-grained custom instructions. Extensive evaluations based on widely used benchmark applications demonstrated that the proposed method achieved a maximum speedup of over 39%, with an average speedup of 19%, when compared to the processor with native instruction set. It is noteworthy that the

FGCI approach is very effective for implementations with very strict area constraints.

Unlike traditional custom instructions that do not incorporate memory operations, hardware accelerators need not adhere to such constraints as local memory units can be assigned to maintain the data manipulated by hardware accelerators. The proposed method for the selection of basic blocks for hardware acceleration ensures that only those basic blocks with common data-dependencies are preferred for hardware acceleration. A novel mathematical model that takes into account the communication time between processor and various memory subsystems, along with the traditional metrics such as area-time cost of accelerating a code segment in hardware, has been proposed to estimate the overall performance of accelerating application basic blocks in hardware.

Only statically allocated scalars and arrays were targeted in this work, where unlike existing approaches, the need for DMA transfers between the main memory and local memory of hardware accelerators was avoided. The local memory subsystems were maintained in the un-cached space, thereby avoiding the need for using additional cache-coherence mechanisms. A case study using the popular SHA benchmark application showed the feasibility and advantage of the proposed architecture, with SHA achieving almost 2X performance speedup with the right set of hardware accelerators. It is noteworthy that while the proposed strategy has been limited to statically allocated variables, it is greatly beneficial in embedded designs which are written with due considerations to the limited memory space and therefore contain many statically allocated variables. Additionally, such coding styles can be easily incorporated during application development to maximally exploit the proposed approaches to achieve high performance without the need for DMA or software/hardware cache-protocols.

In order to rapidly and reliably select the suitable basic blocks in an application for hardware acceleration while considering often convoluted memory dependencies, a

greedy heuristic search algorithm was proposed that utilizes the performance model proposed in chapter 5. It was shown that, while the complexity of an exhaustive approach rises exponentially with increasing number of candidate blocks, the heuristic method scales almost linearly, while still achieving almost 80% accuracy of the exhaustive approach. In order to avoid the local minimum problem, which results in the selection of sub-optimal accelerators, various refinements to the selection methodology were also proposed. In particular, multiple threads were intelligently chosen and considered during the search process to reduce the error bounds further without necessitating either exhaustive or manual selection process. A strategy was proposed in the end to determine the optimum number of threads that need to be considered in order to satisfy a user-specified error boundary. Based on the experiments, it was shown that the basic blocks recommended by the extended technique often achieved the same compute performance as those identified through exhaustive search and falls within less than 10% for almost all cases evaluated.

With increasing complexity of embedded applications, the proposed heuristic method can be effectively deployed in a rapid design exploration and customization framework, especially when involving exhaustive selection process will incur prohibitively long runtime. The heuristic search technique can also be easily adapted for use with other performance estimation models, making it suitable for examining a wider variety of architectures that employ DMA and/or cache-coherence protocols by considering the equivalent overhead of using DMA for data transfer and cache-coherence systems for synchronization.

7.2 Recommendations for Future research

While this thesis presents various techniques and strategies to automatically generate application-specific constraint-aware configurable system-on-chip, by leveraging the extensive customizability of the individual components, future research can be undertaken to strengthen the proposed ideas. The following identifies avenues for future research based on the work in this thesis:

(i) **Processor selection:**

In chapter 3 a framework was described for automatic customization of soft-core processors through the use Instruction Subsetting. The proposed framework can be extended for estimation of additional configurable parameters such as the cache configurations by analyzing the distribution of memory Load/Store operations in the applications. The framework can also be expanded to select a suitable processor from a pool of processors which can enable a designer to not only get the best configuration for a given application but also the most ideal processor for the purpose.

(ii) **FPGA technology-aware custom instruction generation:**

The FPGA-aware custom instruction generation methodology presented in this thesis has been extensively explored for a low end Xilinx FPGA. It was shown that even on a low end Xilinx FPGA with 4-Input LUT structure, the proposed approach achieved significant improvement when compared to existing approaches. However, the proposed methodologies will need to be extended to support modern high end FPGAs with different LUT architectures. This will require the development of a systematic approach to characterize LUT architecture variations.

(iii) **Performance estimation model for hardware accelerators:**

In chapter 5, a mathematical model was proposed to estimate the performance of basic blocks for hardware acceleration while considering the penalties associated with data placement in various memory subsystems. The values used to estimate the memory access latencies were obtained from the Altera NIOS II data sheets. However, a more accurate measurement of memory latencies would be useful in improving the accuracy of the performance model. Additionally, it was assumed that the data for software was always present in the cache to obtain the worst case penalty. In future, additional application analysis can be performed using methods such as offline loop profiling to obtain a more realistic estimate for the data presence in the cache hierarchy based on loop size, frequency, cache size, etc. This would further improve the accuracy of the performance estimation model by not only analyzing the worst case penalty but also a realistic application-specific behavior.

- (iv) **Constraint-aware CSoC generation:** In this thesis various algorithms are proposed to customize the individual components and generate an application-specific constraint-aware CSOC. In future, these algorithms can be integrated into a single tool chain to automatically generate a CSOC for an application or domain. The processor core can be customized using the processor sub-setting algorithms proposed in chapter 3. A conflict-graph based approach along with the analysis of memory-dependencies is essential to ensure non-overlapping selection of both traditional custom instructions and larger hardware blocks as multi-grained accelerators in order to achieve maximum performance-area efficiency.

- (v) **Domain-specific hardware accelerators:** In order to cater for a set of domain-specific applications, the proposed methodologies for hardware accelerators in chapter 5 and 6 can be directly scaled to implement domain-specific hardware accelerators. The set of applications can be profiled using open-source compiler such as LLVM and analyzed simultaneously to identify common hot spots in the application set. The hardware selection algorithms described in chapter 6 can then be utilized to rapidly select data-dependency aware hardware accelerators that are common to all applications in a given domain.
- (vi) **Soft-core based heterogeneous Multi-core platform:** The techniques proposed in this thesis can be directly extended to generate application-specific heterogeneous multi-core processors based on soft-core processors and hardware accelerators on modern FPGAs for cheap, yet high performance embedded computing platform. The major challenges in such systems would be to manage the application partitioning design space exploration as well the communication between the different modules running on different processing cores. It will be interesting to explore how the proposed techniques could be deployed by decomposing a large applications into several small applications, each of which can then be managed by a locally customized soft core processor and hardware acceleration. The customization strategies proposed in chapter 3 and 4 can be directly used for customizing the soft-core processor while the design space exploration methodology proposed in chapters 5 and 6 can be effectively scaled to select the right set of hardware accelerators for such designs.

Acronyms

AMBA Advanced Microcontroller Bus Architecture. 13

ASIC Application-specific integrated circuit. 10

BRAM Block Random Access Memory. 115, 119

CHB Custom Hardware Block. 49, 52

CSI Configurable System Interconnect. 14

CSoC Configurable System-on-Chip. 3, 4, 6, 15, 41, 42, 45, 46, 60, 122, 125, 152

DCT Discrete Cosine Transform. 13, 14

DDF Dynamic Data Flow graph. 24

DFG Data Flow Graph. 28

DMA Direct Memory Access. 14, 48, 110

DSP Digital Signal Processor. 11

EDA Electronic Design Automation. 35

FPGA Field Programmable Gate Array. 10, 11

GPP General Purpose Processor. 10, 152

GPU Graphics Processing Unit. 10

IR Intermediate Representation. 55

ISA Instruction Set Architecture. 23

LM Local Memory. 49

LUT Lookup Table. 61, 92

NRE Non-Recurring Engineering. i, 9, 18

PCIe Peripheral Component Interconnect Express. 15

PSoC Programmable System-on-Chip. 24

SEU Single Event Upset. 22

SHA Secure Hash Algorithm. 42, 50, 52

SoPC System-on-Programmable-Chip. 19

TTM Time-to-Market. i, 9, 18

References

- [1] "BCC Research, "Embedded Systems: Technologies and Markets, IFT016D" [Online]. Available: (<http://www.bccresearch.com/report/embedded-systems-technologies-markets-ift016d.html>), January 2012 (Last visited: 21st November, 2013)."
- [2] "SEMI ISS: Scaling innovation. [Online]. Available: (<http://electroiq.com/blog/2014/02/semi-iss-scaling-innovation/>) (Last visited: 5th May, 2014)."
- [3] "International Technology Roadmap for Semiconductors, 2011 Edition - DESIGN."
- [4] "International Technology Roadmap for Semiconductors, 2012 Update - OVERVIEW."
- [5] M. J. C. Mazo, *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap*. Phd.
- [6] A. Prakash, S.-K. Lam, A. Singh, and T. Srikanthan, "Rapid design exploration framework for application-aware customization of soft core processors," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pp. 539–542, 31 2009–Sept. 2 2009.
- [7] A. Prakash, S.-K. Lam, A. Singh, and T. Srikanthan, "Architecture-aware custom instruction generation for reconfigurable processors," in *Reconfigurable Computing: Architectures, Tools and Applications* (P. Sirisuk, F. Morgan, T. El-Ghazawi, and H. Amano, eds.), vol. 5992 of *Lecture Notes in Computer Science*, pp. 414–419, Springer Berlin Heidelberg, 2010.

- [8] A. Prakash, S.-K. Lam, C. Clarke, and T. Srikanthan, "Instruction set customization for area-constrained fpga designs," in *SOC Conference (SOCC), 2011 IEEE International*, pp. 329–334, Sept.
- [9] A. Prakash, S.-K. Lam, C. T. Clarke, and T. Srikanthan, "Fpga-aware techniques for rapid generation of profitable custom instructions," *Microprocessors and Microsystems*, vol. 37, no. 3, pp. 259 – 269, 2013.
- [10] A. Prakash, C. Clarke, and T. Srikanthan, "Custom instructions with local memory elements without expensive dma transfers," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pp. 647–650, Aug.
- [11] A. Prakash, C. Clarke, S. Lam, and T. Srikanthan, "Modelling communication overhead for accessing local memories in hardware accelerators," in *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, pp. 31–34, 2013.
- [12] R. Brodersen, "System-on-a-chip vlsi-is it finally really here?," in *Advanced Research in VLSI, 1999. Proceedings. 20th Anniversary Conference on*, pp. 154–, 1999.
- [13] "TILE-Gx8072 Processor, Specification Brief. [Online]. Available: (http://www.tilera.com/sites/default/files/productbriefs/TILE-Gx8072_PB041-02.pdf) (Last visited: 21st November, 2013)."
- [14] J. Cong, V. Sarkar, G. Reinman, and A. Bui, "Customizable domain-specific computing," *Design Test of Computers, IEEE*, vol. 28, no. 2, pp. 6–15, 2011.
- [15] "Mixing Custom and Standard Parts. [Online]. Available: (<http://http://semiengineering.com/mixing-custom-and-standard-parts/>) (Last visited: 21st November, 2013)."
- [16] S. Saponara and L. Fanucci, "Homogeneous and heterogeneous mpsoc architectures with network-on-chip connectivity for low-power and real-time multimedia signal processing," *VLSI Design*, vol. 2012, pp. 1–17, 2012.

- [17] J. Becker, T. Pionteck, C. Habermann, and M. Glesner, "Design and implementation of a coarse-grained dynamically reconfigurable hardware architecture," in *WVLSI '01: Proceedings of the IEEE Computer Society Workshop on VLSI 2001*, (Washington, DC, USA), p. 41, IEEE Computer Society, 2001.
- [18] P. Stravers, "Homogeneous multiprocessing for the masses," in *Embedded Systems for Real-Time Multimedia, 2004. ESTImedia 2004. 2nd Workshop on*, pp. 3–, 2004.
- [19] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar, "An 80-tile 1.28tflops network-on-chip in 65nm cmos," in *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pp. 98–589, 2007.
- [20] "Raw Architecture Workstation (RAW). [Online]. Available: (<http://groups.csail.mit.edu/cag/raw/>) (Last visited: 21st November, 2013)."
- [21] "Asynchronous Array of Simple Processors (AsAP). [Online]. Available: (<http://www.ece.ucdavis.edu/vcl/asap/>) (Last visited: 21st November, 2013)."
- [22] "MPOC: A Chip Multiprocessor for Embedded Systems. [Online]. Available: (<http://www.hpl.hp.com/techreports/2002/hpl-2002-186.pdf>) (Last visited: 21st November, 2013)."
- [23] "WaveScalar processor. [Online]. Available: (<http://wavescalar.cs.washington.edu/wavecache.shtml>) (Last visited: 21st November, 2013)."
- [24] N. Saint-Jean, G. Sassatelli, P. Benoit, L. Torres, and M. Robert, "Hs-scale: a hardware-software scalable mp-soc architecture for embedded systems," in *VLSI, 2007. ISVLSI '07. IEEE Computer Society Annual Symposium on*, pp. 21–28, 2007.
- [25] "Intel Core. [Online]. Available: (http://en.wikipedia.org/wiki/intel_core) (Last visited: 21st November, 2013)."

- [26] "Tilera launches new generation of multicore embedded chips for communications [Online]. Available:
(<http://venturebeat.com/2008/09/21/tilera-launches-new-generation-of-multicore-embedded-chips-for-communications/>) (Last visited: 21st November, 2013)."
- [27] "Tilera Multicore Development Environment MDE [Online]. Available:
(http://www.tilera.com/sites/default/files/productbriefs/tileramde_pb014-04_web.pdf) (Last visited: 21st November, 2013)."
- [28] "Commentary: Revisiting heterogeneous vs. homogeneous - jeff bier [Online]. Available:
(http://www.eetimes.com/document.asp?doc_id=1271570) (Last visited: 21st November, 2013)."
- [29] "Xilinx corporation, [Online]. Available:
(<http://www.xilinx.com/>) (Last visited: 21st November, 2013)."
- [30] "Altera corporation, [Online]. Available:
(<http://www.altera.com/>) (Last visited: 21st November, 2013)."
- [31] G. Estrin, "Organization of computer systems: the fixed plus variable structure computer," in *Papers presented at the May 3-5, 1960, western joint IRE-AIEE-ACM computer conference*, pp. 33–40, ACM, 1960.
- [32] G. Estrin, "Reconfigurable computer origins: the ucla fixed-plus-variable (f+v) structure computer," *Annals of the History of Computing, IEEE*, vol. 24, no. 4, pp. 3–9, 2002.
- [33] H. Svensson, *Reconfigurable Architectures for Embedded Systems*. Phd.
- [34] M.-H. Lee, H. Singh, G. Lu, N. Bagherzadeh, F. J. Kurdahi, E. M. C. Filho, and V. C. Alves, "Design and implementation of the morphosys reconfigurable computing processor," *J. VLSI Signal Process. Syst.*, vol. 24, no. 2/3, pp. 147–164, 2000.

- [35] H. Zhang, V. Prabhu, V. George, M. Wan, M. Benes, A. Abnous, and J. Rabaey, "A 1-v heterogeneous reconfigurable dsp ic for wireless baseband digital signal processing," *Solid-State Circuits, IEEE Journal of*, vol. 35, pp. 1697–1704, Nov 2000.
- [36] "Pleiades group, [Online]. Available: (http://bwrc.eecs.berkeley.edu/research/configurable_architectures/) (Last visited: 21st November, 2013)."
- [37] J. Becker, A. Thomas, M. Vorbach, and V. Baumgarte, "An industrial/academic configurable system-on-chip project (csoc): coarse-grain xpp-/leon-based architecture integration," in *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pp. 1120–1121, 2003.
- [38] M. Vorbach and R. Becker, "Reconfigurable processor architectures for mobile phones," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pp. 6 pp.–, April 2003.
- [39] "Gaisler Research, [Online]. Available: (<http://www.gaisler.com/>) (Last visited: 21st November, 2013)."
- [40] "The XPP communication system, PACT Corporation, Technical Report 15, 2000."
- [41] "Triscend, [Online]. Available: (<http://www.triscend.com/products/dsa7csocsummary.pdf>) (Last visited: 21st November, 2013)."
- [42] "Xilinx press release, [Online]. Available: (http://www.xilinx.com/prsrsls/xilcorp/0435triscend_acquisition.htm) (Last visited: 21st November, 2013)."
- [43] S. Knapp and D. Tavana, "Field configurable system-on-chip device architecture," in *Custom Integrated Circuits Conference, 2000. CICC. Proceedings of the IEEE 2000*, pp. 155–158, 2000.

- [44] S. Wallner, "Design methodology of a configurable system-on-chip architecture," in *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pp. 283–284, April 2004.
- [45] H. Tan and Y. Sun, "Design of a configurable system-on-chip for audio application," in *ASIC, 2007. ASICON '07. 7th International Conference on*, pp. 740–743, Oct. 2007.
- [46] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pp. 365–376, June 2011.
- [47] M. Taylor, "A landscape of the new dark silicon design regime," *Micro, IEEE*, vol. 33, pp. 8–19, Sept 2013.
- [48] L. Wang and K. Skadron, "Implications of the power wall: Dim cores and reconfigurable logic," *Micro, IEEE*, vol. 33, pp. 40–48, Sept 2013.
- [49] N. HARDAVELLAS, "The Rise and Fall of Dark Silicon," 2012.
- [50] B. Da Silva, A. Braeken, E. H. DăŃŽHollander, A. Touhafi, J. G. Cornelis, and J. Lemeire, "Performance and programming environment of a combined gpu/fpga desktop," in *Presented at the 21st ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2013.
- [51] S. Che, J. Li, J. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with gpus and fpgas," in *Application Specific Processors, 2008. SASP 2008. Symposium on*, pp. 101–107, 2008.
- [52] C. Grozea, Z. Bankovic, and P. Laskov, "Facing the multicore-challenge," ch. FPGA vs. multi-core CPUs vs. GPUs: hands-on experience with a sorting application, pp. 105–117, Berlin, Heidelberg: Springer-Verlag, 2010.
- [53] H. P. Huynh, Y. Liang, and T. Mitra, "Efficient custom instructions generation for system-level design," in *Field-Programmable Technology (FPT), 2010 International Conference on*, pp. 445–448, 2010.

- [54] "Maxeler technologies [Online]. Available:
(<http://www.maxeler.com/>) (Last visited: 21st November, 2013)."
- [55] "Convey computer, [Online]. Available:
(<http://www.conveycomputer.com/>) (Last visited: 21st November, 2013)."
- [56] "Convey MX Series - Architectural Overview, [Online]. Available:
(<http://www.conveycomputer.com/products/mx-series/>) (Last visited: 21st November, 2013)."
- [57] A. K. Singh, A. Kumar, and T. Srikanthan, "Accelerating throughput-aware run-time mapping for heterogeneous mpsoCs," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 18, pp. 9:1–9:29, Jan. 2013.
- [58] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: survey of current and emerging trends," in *Proceedings of the 50th Annual Design Automation Conference, DAC '13*, (New York, NY, USA), pp. 1:1–1:10, ACM, 2013.
- [59] I. Lebedev, C. Fletcher, S. Cheng, J. Martin, A. Dougnik, D. Burke, M. Lin, and J. Wawrzynek, "Exploring many-core design templates for fpgas and asics," *International Journal of Reconfigurable Computing*, vol. 2012, pp. 1–15, 2012.
- [60] M. Asri, LiaoHsuan-Chun, H. C. Liao, T. ISSHIKI, and hiroaki kunieda et al., "Bridging the gap between asic and gpp : A high-performance and c-programmable asip for image processing," in *Bridging the Gap between ASIC and GPP : A High-Performance and C-Programmable ASIP for Image Processing*, 2012-03.
- [61] E. Roza, "Systems-on-chip: what are the limits?," *Electronics Communication Engineering Journal*, vol. 13, no. 6, pp. 249–255, 2001.
- [62] J. M. Rabaey, "Silicon platforms for the next generation wireless systems - what role does reconfigurable hardware play?," in *Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications, FPL '00*, (London, UK, UK), pp. 277–285, Springer-Verlag, 2000.

- [63] C. Edwards, "Speeding up is hard to do [multiprocessing systems for faster software execution]," *IEE Review*, vol. 50, no. 9, pp. 44–46, 2004.
- [64] K. Benkrid *et al.*, "High performance reconfigurable computing: From applications to hardware," *IAENG International Journal of Computer Science*, vol. 35, no. 1, 2008.
- [65] "Consider time to market early in test process. [Online]. Available: (<http://www.edn.com/electronics-blogs/test-voices/4414388/consider-time-to-market-early-in-test-process>) (Last visited: 21st November, 2013)."
- [66] "Accelerating Time-to-Market using an FPGA and Customizable SoC Methodology. [Online]. Available: (<http://www.rtcmagazine.com/articles/view/102721>) (Last visited: 21st November, 2013)."
- [67] "We Must Hold The Line On Soaring ASIC Design Costs. [Online]. Available: (<http://electronicdesign.com/archive/we-must-hold-line-soaring-asic-design-costs>) (Last visited: 21st November, 2013)."
- [68] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 2, pp. 203–215, 2007.
- [69] X. Wu and T. Vladimirova, "A self-reconfigurable system-on-chip architecture for satellite on-board computer maintenance," in *Advances in Computer Systems Architecture* (C. Jesshope and C. Egan, eds.), vol. 4186 of *Lecture Notes in Computer Science*, pp. 552–558, Springer Berlin Heidelberg, 2006.
- [70] "Is NASA's design opportunity for FPGAs in space vanishing in favor of privatized platforms. [Online]. Available: (<http://www.edn.com/electronics-blogs/fpga-gurus/4418246/is-nasa-s-design-opportunity-for-fpgas-in-space-vanishing-in-favor-of-privatized-platforms->) (Last visited: 21st November, 2013)."

- [71] "If it's tiny and embedded, is it still a software-defined radio? [Online]. Available:
(<http://www.edn.com/electronics-blogs/fpga-gurus/4418702/if-it-s-tiny-and-embedded-is-it-still-a-software-defined-radio->) (Last visited: 21st November, 2013)."
- [72] F. Fons, M. Fons, E. CantÃş, and M. LÃşpez, "Deployment of run-time reconfigurable hardware coprocessors into compute-intensive embedded applications," *Journal of Signal Processing Systems*, vol. 66, no. 2, pp. 191–221, 2012.
- [73] B. Osterloh, H. Michalik, B. Fiethe, and K. Kotarowski, "Socwire: A network-on-chip approach for reconfigurable system-on-chip designs in space applications," in *Adaptive Hardware and Systems, 2008. AHS '08. NASA/ESA Conference on*, pp. 51–56, June 2008.
- [74] "Zynq-7000 All Programmable SoC, [Online]. Available:
(<http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/index.htm>) (Last visited: 21st November, 2013)."
- [75] "Altera SoC FPGAs, [Online]. Available:
(<http://www.altera.com/devices/processor/soc-fpga/overview/proc-soc-fpga.html>) (Last visited: 21st November, 2013)."
- [76] A. Ferrari and A. Sangiovanni-Vincentelli, "System design: traditional concepts and new paradigms," in *Computer Design, 1999. (ICCD '99) International Conference on*, pp. 2–12, 1999.
- [77] K. Keutzer, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: orthogonalization of concerns and platform-based design," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 19, no. 12, pp. 1523–1543, 2000.
- [78] "Platform Based Design [Online]. Available:
(http://embedded.eecs.berkeley.edu/respep/research/asves/trade_magazine/asv_eedesign02.pdf) (Last visited: 21st November, 2013)."

- [79] A. Sangiovanni-Vincentelli and G. Martin, "Platform-based design and software design methodology for embedded systems," *Design Test of Computers, IEEE*, vol. 18, no. 6, pp. 23–33, 2001.
- [80] "<http://www.st.com/stonline/stappl/cms/press/news/year2007/p2142.htm>."
- [81] "Atmel corp.FPSLIC, [Online]. Available: (<http://www.digikey.com/catalog/en/partgroup/fpslic/14188>) (Last visited: 21st November, 2013)."
- [82] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, (New York, NY, USA), pp. 33–36, ACM, 2011.
- [83] "Altera excalibur devices. [Online]. Available: (<http://www.altera.com/products/devices/arm/arm-index.html>) (Last visited: 21st November, 2013)."
- [84] "Xilinx microblaze, [Online]. Available: (<http://www.xilinx.com/tools/microblaze.htm>) (Last visited: 21st November, 2013)."
- [85] "Altera corp. NIOS II processors., [Online]. Available: (<http://www.altera.com/products/ip/processors/nios2/ni2-index.html>) (Last visited: 21st November, 2013)."
- [86] P. Yiannacouras, J. G. Steffan, and J. Rose, "Exploration and customization of fpga-based soft processors," *Computer Aided Design of Integrated Circuits and Systems IEEE Transactions on*, vol. 26, pp. 266–277, Feb. 2007.
- [87] D. Sheldon, R. Kumar, R. Lysecky, F. Vahid, and D. Tullsen, "Application-specific customization of parameterized fpga soft-core processors," in *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, (New York, NY, USA), pp. 261–268, ACM, 2006.

- [88] S. Padmanabhan, R. K. Cytron, R. D. Chamberlain, and J. W. Lockwood, "Automatic application-specific microarchitecture reconfiguration," in *In Proc. of 13th Reconfigurable Architectures Workshop*, 2006.
- [89] R. Lysecky and F. Vahid, "Design and implementation of a microblaze-based warp processor," *ACM Trans. Embed. Comput. Syst.*, vol. 8, pp. 22:1–22:22, Apr. 2009.
- [90] "Nios II embedded processors. [Online]. Available: (http://www.altera.com/devices/fpga/cyclone/features/cyc-cyclone_nios.html)(last visited: 27th october, 2013)."
- [91] "3G Infrastructures. [Online]. Available: (<http://www.altera.com/end-markets/wireless/cellular/wir-cellular-infrastructure.html>) (last visited: 27th october, 2013)."
- [92] "Radiation-Hardened, Space-Grade Virtex-5QV Family Overview [Online]. Available: (http://www.xilinx.com/support/documentation/data_sheets/ds192_v5qv_device_overview.pdf) (last visited: 27th october, 2013)."
- [93] P. Bonzini and L. Pozzi, "Recurrence-aware instruction set selection for extensible embedded processors," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 16, no. 10, pp. 1259–1267, 2008.
- [94] K. Atasu, L. Pozzi, and P. lenne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *DAC '03: Proceedings of the 40th annual Design Automation Conference*, (New York, NY, USA), pp. 256–261, ACM, 2003.
- [95] N. Clark, H. Zhong, and S. Mahlke, "Automated custom instruction generation for domain-specific processor acceleration," *Computers, IEEE Transactions on*, vol. 54, pp. 1258–1270, Oct. 2005.
- [96] N. Pothineni, A. Kumar, and K. Paul, "Application specific datapath extension with distributed i/o functional units," in *VLSI Design, 2007. Held jointly with 6th International Conference on Embedded Systems., 20th International Conference on*, pp. 551–558, Jan. 2007.

- [97] X. Chen, D. L. Maskell, and Y. Sun, "Fast identification of custom instructions for extensible processors," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 2, pp. 359–368, 2007.
- [98] C. Pilato, F. Ferrandi, and D. Sciuto, "A design methodology to implement memory accesses in high-level synthesis," in *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, CODES+ISSS '11*, (New York, NY, USA), pp. 49–58, ACM, 2011.
- [99] "Opencores.org, [Online]. Available: (<http://www.opencores.org/>) (Last visited: 21st November, 2013)."
- [100] "Choosing an Intellectual Property Core. [Online]. Available: (http://www.mips.com/media/files/whitepapers/choosing_an_intellectual_property_core.pdf)."
- [101] Y. Jiao, C. Guo, B. Wu, and H. Luo, "Extended-ddf modeling embedded system design: Adapting to ip technology," in *CIT '05: Proceedings of the The Fifth International Conference on Computer and Information Technology*, (Washington, DC, USA), pp. 829–835, IEEE Computer Society, 2005.
- [102] Y. Iosifidis, A. Mallik, S. Mamagkakis, E. De Greef, A. Bartzas, D. Soudris, and F. Catthoor, "A framework for automatic parallelization, static and dynamic memory optimization in mpsoc platforms," in *Proceedings of the 47th Design Automation Conference, DAC '10*, (New York, NY, USA), pp. 549–554, ACM, 2010.
- [103] "Xilinx's Analog Mixed Signal (AMS) technology. [Online]. Available: (<http://www.xilinx.com/products/technology/analog-mixed-signal.html>)(Last visited: 21st November, 2013)."
- [104] "PSoC - Programmable System-on-Chip. [Online]. Available: (<http://www.cypress.com/psoc/>)(Last visited: 21st November, 2013)."
- [105] D. Jones, A. Powell, C. Bouganis, and P. Y. K. Cheung, "Gpu versus fpga for high productivity computing," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pp. 119–124, 2010.

- [106] P. Yiannacouras, J. Rose, and J. G. Steffan, "The microarchitecture of fpga-based soft processors," in *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, (New York, NY, USA), pp. 202–212, ACM, 2005.
- [107] "LatticeMico32 open, free 32-bit Soft Processor, [Online]. Available: (<http://www.latticesemi.com/products/intellectualproperty/ipcores/mico32/index.cfm>) (Last visited: 21st November, 2013)."
- [108] "Lattice Semiconductors, [Online]. Available: (<http://www.latticesemi.com>) (Last visited: 21st November, 2013)."
- [109] D. Mattsson and M. Christensson, "'evaluation of synthesizable cpu cores" master's thesis,chalmers univ. of technology, 2004."
- [110] "Soft core war, [Online]. Available: (http://www.eejournal.com/archives/articles/20060926_softcore/) (Last visited: 21st November, 2013)."
- [111] J. Turley, "'survey: Who uses custom chips", embedded systems programming, aug 2005."
- [112] W. E. Dougherty, D. J. Pursley, and D. E. Thomas, "Instruction subsetting: Trading power for programmability," in *VLSI '98. System Level Design. Proceedings. IEEE Computer Society Workshop on*, pp. 42–47, 1998.
- [113] D. Sheldon, F. Vahid, and S. Lonardi, "Interactive presentation: Soft-core processor customization using the design of experiments paradigm," in *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, (San Jose, CA, USA), pp. 821–826, EDA Consortium, 2007.
- [114] P. Yiannacouras, J. G. Steffan, and J. Rose, "Application-specific customization of soft processor microarchitecture," in *FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, (New York, NY, USA), pp. 201–210, ACM, 2006.

- [115] J.-e. Lee, K. Choi, and N. D. Dutt, "Energy-efficient instruction set synthesis for application-specific processors," in *ISLPED '03: Proceedings of the 2003 international symposium on Low power electronics and design*, (New York, NY, USA), pp. 330–333, ACM, 2003.
- [116] K. Keutzer, S. Malik, and R. Newton, "From ASIC to ASIP: The next design discontinuity," in *ICCD '02: Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02)*, (Washington, DC, USA), p. 84, IEEE Computer Society, 2002.
- [117] A. Ray, T. Srikanthan, and W. Jigang, "Practical techniques for performance estimation of processors," in *System-on-Chip for Real-Time Applications, 2005. Proceedings. Fifth International Workshop on*, pp. 308–311, July 2005.
- [118] A. Ray, W. Jigang, and S. Thambipillai, "Estimating processor performance of library function," in *Embedded Software and Systems, 2005. Second International Conference on*, pp. 6 pp.—, Dec. 2005.
- [119] "LLVM, Compiler Infrastructure, [Online]. Available: (<http://llvm.org/>) (Last visited: 21st November, 2013)."
- [120] F. Sun, S. Ravi, A. Raghunathan, and N. Jha, "A scalable application-specific processor synthesis methodology," in *Computer Aided Design, 2003. ICCAD-2003. International Conference on*, pp. 283–290, 2003.
- [121] F. Sun, S. Ravi, A. Raghunathan, and N. Jha, "A scalable synthesis methodology for application-specific processors," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 14, no. 11, pp. 1175–1188, 2006.
- [122] S.-K. Lam and T. Srikanthan, "Rapid design of area-efficient custom instructions for reconfigurable embedded processing," *J. Syst. Archit.*, vol. 55, no. 1, pp. 1–14, 2009.
- [123] N. Pothineni, A. Kumar, and K. Paul, "Exhaustive enumeration of legal custom instructions for extensible processors," in *VLSI Design, 2008. VLSID 2008. 21st International Conference on*, pp. 261–266, Jan. 2008.

- [124] K. Atasu, O. Mencer, W. Luk, C. Ozturan, and G. Dundar, "Fast custom instruction identification by convex subgraph enumeration," in *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on*, pp. 1–6, July 2008.
- [125] T. Li, W. Jigang, Y. Deng, T. Srikanthan, and X. Lu, "Fast identification algorithm for application-specific instruction-set extensions," in *Electronic Design, 2008. ICED 2008. International Conference on*, pp. 1–5, Dec. 2008.
- [126] S.-K. Lam, W. Li, and T. Srikanthan, "High level area estimation of custom instructions for fpga-based reconfigurable processors," in *Information, Communications & Signal Processing, 2007 6th International Conference on*, pp. 1–5, Dec. 2007.
- [127] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," in *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, (New York, NY, USA), pp. 183–189, ACM, 2004.
- [128] J. Henkel and R. Ernst, "An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation techniques," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 9, no. 2, pp. 273–289, 2001.
- [129] W. Wolf, "A decade of hardware/software codesign," *Computer*, vol. 36, no. 4, pp. 38–43, 2003.
- [130] R. Koenig, L. Bauer, T. Stripf, M. Shafique, W. Ahmed, J. Becker, and J. Henkel, "Kahrisma: A novel hypermorphic reconfigurable-instruction-set multi-grained-array architecture," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pp. 819–824, 2010.
- [131] W. Ahmed, M. Shafique, L. Bauer, and J. Henkel, "mrts: Run-time system for reconfigurable processors with multi-grained instruction-set extensions," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pp. 1–6, 2011.

- [132] P. Biswas, V. Choudhary, K. Atasu, L. Pozzi, P. lenne, and N. Dutt, "Introduction of local memory elements in instruction set extensions," in *Proceedings of the 41st annual Design Automation Conference, DAC '04*, (New York, NY, USA), pp. 729–734, ACM, 2004.
- [133] T. Kluter, P. Brisk, P. lenne, and E. Charbon, "Speculative dma for architecturally visible storage in instruction set extensions," in *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis, CODES+ISSS '08*, (New York, NY, USA), pp. 243–248, ACM, 2008.
- [134] T. Kluter, S. Burri, P. Brisk, E. Charbon, and P. lenne, "Virtual ways: efficient coherence for architecturally visible storage in automatic instruction set extensions," in *Proceedings of the 5th international conference on High Performance Embedded Architectures and Compilers, HiPEAC'10*, (Berlin, Heidelberg), pp. 126–140, Springer-Verlag, 2010.
- [135] P. Biswas, N. Dutt, L. Pozzi, and P. lenne, "Introduction of architecturally visible storage in instruction set extensions," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 3, pp. 435–446, 2007.
- [136] P. Biswas, N. Dutt, P. lenne, and L. Pozzi, "Automatic identification of application-specific functional units with architecturally visible storage," in *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, vol. 1, pp. 1–6, 2006.
- [137] F. Sun, S. Ravi, A. Raghunathan, and N. Jha, "A synthesis methodology for hybrid custom instruction and coprocessor generation for extensible processors," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 11, pp. 2035–2045, 2007.
- [138] P. R. Panda, N. D. Dutt, and A. Nicolau, "On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, pp. 682–704, July 2000.

- [139] T. R. Kumar, R. Govindarajan, and C. Ravikumar, "On-chip memory architecture exploration framework for dsp processor-based embedded system on chip," *ACM Trans. Embed. Comput. Syst.*, vol. 11, pp. 5:1–5:25, Apr. 2012.
- [140] H. Devos, J. Van Campenhout, I. Verbauwhede, and D. Stroobandt, "Transactions on high-performance embedded architectures and compilers iii," ch. Constructing application-specific memory hierarchies on FPGAs, pp. 201–216, Berlin, Heidelberg: Springer-Verlag, 2011.
- [141] M. Zuluaga, T. Kluter, P. Brisk, N. Topham, and P. lenne, "Introducing control-flow inclusion to support pipelining in custom instruction set extensions," in *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, pp. 114–121, 2009.
- [142] R. Baert, E. de Greef, and E. Brockmeyer, "An automatic scratch pad memory management tool and mpeg-4 encoder case study," in *Proceedings of the 45th annual Design Automation Conference, DAC '08*, (New York, NY, USA), pp. 201–204, ACM, 2008.
- [143] J. Cong, W. Jiang, B. Liu, and Y. Zou, "Automatic memory partitioning and scheduling for throughput and power optimization," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 16, pp. 15:1–15:25, Apr. 2011.
- [144] P. Li, Y. Wang, P. Zhang, G. Luo, T. Wang, and J. Cong, "Memory partitioning and scheduling co-optimization in behavioral synthesis," in *Proceedings of the International Conference on Computer-Aided Design, ICCAD '12*, (New York, NY, USA), pp. 488–495, ACM, 2012.
- [145] J. Cong, W. Jiang, B. Liu, and Y. Zou, "Automatic memory partitioning and scheduling for throughput and power optimization," in *Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, pp. 697–704, 2009.
- [146] J. Henkel, "Closing the soc design gap," *Computer*, vol. 36, no. 9, pp. 119–121, 2003.

- [147] “Bambu, A Free Framework for the High-Level Synthesis of Complex Applications, [Online]. Available: (http://panda.dei.polimi.it/wp-content/uploads/posterub_date.pdf).”
- [148] A. Putnam, D. Bennett, E. Dellinger, J. Mason, P. Sundararajan, and S. Eggers, “Chimps: A c-level compilation flow for hybrid cpu-fpga architectures,” in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pp. 173–178, 2008.
- [149] W. Luk, J. Coutinho, T. Todman, Y. M. Lam, W. Osborne, K. Susanto, Q. Liu, and W. S. Wong, “A high-level compilation toolchain for heterogeneous systems,” in *SOC Conference, 2009. SOCC 2009. IEEE International*, pp. 9–18, 2009.
- [150] M. Rashid, F. Ferrandi, and K. Bertels, “hartes design flow for heterogeneous platforms,” in *Quality of Electronic Design, 2009. ISQED 2009. Quality Electronic Design*, pp. 330–338, 2009.
- [151] I. Colacicco, G. Marchiori, and R. Tripiccion, “The hardware application platform of the hartes project,” in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pp. 439–442, 2008.
- [152] F. Capman, S. Thabuteau, and A.-M. Fouilliant, “In-car speech and audio processing - some experiments within hartes project,” in *Networking, Sensing and Control, 2009. ICNSC '09. International Conference on*, pp. 463–467, 2009.
- [153] K. Bertels, V.-M. Sima, Y. Yankova, G. Kuzmanov, W. Luk, G. Coutinho, F. Ferrandi, C. Pilato, M. Lattuada, D. Sciuto, and A. Michelotti, “Hartes: Hardware-software codesign for heterogeneous multicore platforms,” *Micro, IEEE*, vol. 30, no. 5, pp. 88–97, 2010.
- [154] “Bluebee Multi-core Technologies, [Online]. Available: (<http://www.bluebee-tech.com/index.html>) (Last visited: 21st November, 2013).”
- [155] “Bluebee - Your Gateway to Multicore Computing, [Online]. Available: (<http://www.bluebee-tech.com/downloads/bluebee-overview.pdf>) (Last visited: 21st November, 2013).”

- [156] “Bluebee Toolchain Presentation, [Online]. Available:
(http://www.bluebee-tech.com/downloads/bluebee_toolchain_presentation.pdf)
(Last visited: 21st November, 2013).”
- [157] “The BlueBee Toochain Whitepaper, [Online]. Available:
(http://www.bluebee-tech.com/downloads/bluebee_whitepaper.pdf) (Last visited: 21st November, 2013).”
- [158] “Synopsys. [Online]. Available:
(<http://www.synopsys.com/home.aspx>) (Last visited: 21st November, 2013).”
- [159] “Mentor Graphics. [Online]. Available:
(<http://www.mentor.com>) (Last visited: 21st November, 2013).”
- [160] “Synopsys System-Level Design Solutions. [Online]. Available:
(<http://www.synopsys.com/systems/pages/default.aspx>) (Last visited: 21st November, 2013).”
- [161] “Mentor Graphics, Electronic System Level (ESL). [Online]. Available:
(<http://www.mentor.com>) (Last visited: 21st November, 2013).”
- [162] “The XPRES Compiler. [Online]. Available:
(http://www.tensilica.com/uploads/pdf/xpres-triple-threat_solution.pdf).”
- [163] R. Gonzalez, “Xtensa: a configurable and extensible processor,” *Micro, IEEE*, vol. 20, pp. 60–70, Mar/Apr 2000.
- [164] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, “Autopilot: A platform-based esl synthesis system,” in *High-Level Synthesis* (P. Coussy and A. Morawiec, eds.), pp. 99–112, Springer Netherlands, 2008.
- [165] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, “High-level synthesis for fpgas: From prototyping to deployment,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, pp. 473–491, 2011.
- [166] “Xilinx Vivado Design Suite [Online]. Available:
(<http://www.origin.xilinx.com/products/design-tools/vivado/index.htm>) (Last visited: 21st November, 2013).”

- [167] "Space Codesign. [Online]. Available:
(<http://www.spacecodesign.com/>) (Last visited: 21st November, 2013)."
- [168] "Spacestudio, Space Codesign. [Online]. Available:
(<http://www.spacecodesign.com/#space-studio-design-embedded-systems-hw-sw-co-design>) (Last visited: 21st November, 2013)."
- [169] "Y Explorations eXCite tool, [Online]. Available:
(<http://www.yxi.com/products.php>) (Last visited: 21st November, 2013)."
- [170] "Altera C2H Compiler - User Guide [Online]. Available:
(http://www.altera.com/literature/ug/ug_nios2_c2h_compiler.pdf) (Last visited: 21st November, 2013)."
- [171] D. Calderín and S. Vassiliadis, "Soft core processors and embedded processing: a survey and analysis," in *Proceedings of -ProRISC*, pp. 483–488, November 2005.
- [172] J. Tong, I. Anderson, and M. Khalid, "Soft-core processors for embedded systems," in *Microelectronics, 2006. ICM '06. International Conference on*, pp. 170–173, Dec 2006.
- [173] I. D. L. Anderson and M. A. S. Khalid, "Design space exploration using parameterized cores: A case study," in *Electrical and Computer Engineering, 2006. CCECE '06. Canadian Conference on*, pp. 1893–1896, 2006.
- [174] N. Clark, H. Zhong, and S. Mahlke, "Processor acceleration through automated instruction set customization," in *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), p. 129, IEEE Computer Society, 2003.
- [175] T. Li, W. Jigang, S.-K. Lam, T. Srikanthan, and X. Lu, "Efficient heuristic algorithm for rapid custom-instruction selection," in *ICIS '09: Proceedings of the 2009 Eighth IEEE/ACIS International Conference on Computer and Information Science*, (Washington, DC, USA), pp. 266–270, IEEE Computer Society, 2009.

- [176] Y. Guo, G. J. Smit, H. Broersma, and P. M. Heysters, "A graph covering algorithm for a coarse grain reconfigurable system," in *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, (New York, NY, USA), pp. 199–208, ACM, 2003.
- [177] "Trimaran compiler [Online]. Available: (<http://trimaran.org/>) (Last visited: 21st November, 2013)."
- [178] "xilinx Application note, Design tips for HDL Implementation of Arithmetic Functions, XAPP215", [Online]. Available: (http://www.xilinx.com/support/documentation/application_notes/xapp215.pdf) (Last visited: 21st November, 2013)."
- [179] "Xilinx 7 series Overview, [Online]. Available: (http://www.xilinx.com/support/documentation/data_sheets/ds180_7series_overview.pdf) (Last visited: 21st November, 2013)."
- [180] "The VFLib graph matching library, version 2.0, [Online]. Available: (<http://mivia.unisa.it/wp-content/uploads/2013/05/vflib.pdf>) (Last visited: 21st November, 2013)."
- [181] "Mibench Version 1.0, [Online]. Available: (<http://www.eecs.umich.edu/mibench/>) (Last visited: 21st November, 2013)."
- [182] "Mediabench Consortium, [Online]. Available: (<http://euler.slu.edu/fritts/mediabench/>) (Last visited: 21st November, 2013)."
- [183] S.-K. Lam, M. Shoaib, and T. Srikanthan, "Modeling arbitrator delay-area dependencies in customizable instruction set processors," in *Electronic Design, Test and Applications, 2006. DELTA 2006. Third IEEE International Workshop on*, pp. 6 pp.–, Jan.
- [184] "ARM Ltd. [Online]. Available: (<http://www.arm.com>) (Last visited: 21st November, 2013)."
- [185] "Introduction to AMBA 4 ACE. - Whitepaper, [Online]. Available: (http://www.arm.com/files/pdf/cacheoherencywhitepaper_6june2011.pdf) (Last visited: 21st November, 2013)."

- [186] “AMBA 4 ACE (AXI Coherency Extensions) Verification IP, Synopsys. [Online]. Available:
(<http://www.techdesignforums.com/practice/technique/amba4-ace-cache-coherency-vip/>) (Last visited: 21st November, 2013).”
- [187] T. Kluter, P. Brisk, E. Charbon, and P. lenne, “Way stealing: A unified data cache and architecturally visible storage for instruction set extensions,” 2013.
- [188] J. Cong and K. Gururaj, “Architecture support for custom instructions with memory operations,” in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA ’13, (New York, NY, USA), pp. 231–234, ACM, 2013.
- [189] P. Crosthwaite, J. Williams, and P. Sutton, “Profile driven data-dependency analysis for improved high level language hardware synthesis,” in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pp. 207–214, 2009.
- [190] T. M. Chilimbi, “Efficient representations and abstractions for quantifying and exploiting data reference locality,” in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI ’01, (New York, NY, USA), pp. 191–202, ACM, 2001.
- [191] “Altera NIOS II Core Implementation Details., [Online]. Available:
(http://www.altera.com/literature/hb/nios2/n2cpu_nii51015.pdf) (Last visited: 21st November, 2013).”
- [192] R. Chaves, G. Kuzmanov, L. Sousa, and S. Vassiliadis, “Cost-efficient sha hardware accelerators,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, no. 8, pp. 999–1008, 2008.
- [193] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, “Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis,” *Journal of Information Processing*, vol. 17, pp. 242–254, 2009.

- [194] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy, "The suif compiler system: a parallelizing and optimizing research compiler," tech. rep., Stanford, CA, USA, 1994.
- [195] "LLVM, Language Reference Manual, [Online]. Available: (<http://www.llvm.org/docs/langref.html>) (Last visited: 21st November, 2013)."
- [196] O. Almer, R. Bennett, I. Bohm, A. Murray, X. Qu, M. Zuluaga, B. Franke, and N. Topham, "An end-to-end design flow for automated instruction set extension and complex instruction selection based on gcc," in *Proc. 1st International Workshop on GCC Research Opportunities (GROW'09)*, 2009.
- [197] A. Murray and B. Franke, "Compiling for automatically generated instruction set extensions," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, (New York, NY, USA), pp. 13–22, ACM, 2012.

Appendix

Open source compilers used for application analysis

Although the work discussed in this research is not about compilers, we still need to use and even tweak them in order to get more information about the application. We also need compilers to help us map the application on the customized heterogeneous platform. In the past various researchers have developed open source compilers such as Trimaran [177], LLVM [119], SUIF [194] etc. to compile and analyze applications at an intermediate level. This is done by compiling the application source code through the front end of the compiler generating an intermediate representation. The intermediate representation is largely target-independent which allows a hardware designer to analyze and understand the requirements of the application from a possible custom hardware component.

The Trimaran compiler [177] extensively to compile the application to an intermediate level and perform various transformations and compiler passes. These compiler passes were helpful in generating information regarding the schedule steps in the application block as well as highlighting the instruction level parallelism in the program.

The trimaran compiler generates the intermediate representation by targeting it for a Very Large Instruction Word (VLIW) based architecture. This was especially helpful in the custom instruction generation step described in Chapter 4 due to various reasons some of which are mentioned below.

- (i) Since the dataflow graph was readily available using Trimaran, the identification and selection of custom instructions was easier to perform.

- (ii) The operations in the intermediate representation is scheduled, which makes it easier to identify instruction level parallelism (ILP). The ILP information helps us to identify larger custom instructions with high parallelism and low critical path.

The LLVM compiler [119] was introduced by researchers in University of Illinois at Urbana-Champaign. The compiler has gained wide popularity in academia as well as industry for its various features ranging from easily understandable intermediate representation format to modular compiler passes that can be added by an user for a custom application transformation. LLVM is widely supported by an active community of developers who constantly update the compiler with new features and bug fixes. The front end compiler in LLVM targets a LLVM virtual machine with RISC-like instruction set architecture [195]. The LLVM compiler also allows profiling the application on the LLVM virtual machine to identify the critical code segments in the application. In this thesis we have used LLVM extensively to analyze the application at the Intermediate level for selecting hardware accelerators in Chapters 5, 6 as well for processor customization in Chapter 3. The specific usage of LLVM in this research has been discussed in more details as and when it is used.

Lastly, with the ability to generate application-specific architectures, we need a compiler that can take advantage of the additional capabilities of the architecture by mapping the appropriate sections of the code on the extended hardware accelerators [196]. To this end, the authors in [197] have proposed the use of subgraph isomorphism checking to map and exploit the appropriate code sections on the new hardware functional units. As stated earlier, while the generation of a compiler is out of the scope of this thesis, such compilers can prove to be immensely beneficial in harnessing the full potential of configurable platforms.

IR2Verilog: Converting LLVM Intermediate Representation To Verilog

The IR2Verilog tool was written specifically to enable us to convert specific portions of the applications (in our case at the basic block level of granularity) into Verilog which could be synthesized along with the processor. We used the LLVM compiler to pass

the application “C” Code through the front-end compilation process and generate an Intermediate Representation (IR) for the application. The generated IR was profiled on the LLVM’s virtual target to identify the frequently executed basic blocks. A set of basic blocks is then selected for acceleration based on certain criteria discussed earlier in Sections 5.3 and 5.4 (more details on the selection process in Chapters 5 and 6). After selecting the right set of basic blocks to accelerate, we proceeded with converting the LLVM IR of these basic blocks into Verilog using the IR2Verilog tool. In the next subsection we will briefly explain the main components of this tool.

Stages Of IR2Verilog

The IR2Verilog tool is a custom hardware block generator that takes LLVM intermediate representation as its input and outputs a Verilog module capable of implementing multiple hardware blocks. The higher level tool calling IR2Verilog can specify the basic blocks from the intermediate representation for which an implementation is desired. The input LLVM IR of the entire application is parsed on a line by line basis. Initially the lines are examined for a block delimiter. If there is a block delimiter and the block is indicated for hardware implementation, the parser switches to interpreting and storing each line in a linked list for that block. Whilst parsing a line in a candidate block, the parser checks that the line is implementable, adds all variables used on the line to the local variables list if they are not already on the list. Other information, such as the operation, word length etc. is stored in the parsed line structure that is placed at the end of the block linked list.

The output Verilog implementation of the block hardware consists of a small number of constant code output sections, multiple passes through the block and variable lists to achieve the following:

- (i) Provide a constant module header and I/O allowing different hardware blocks to be implemented without there being a need to change any hardware outside the custom hardware block.
- (ii) Declaration of all variables use in the block based upon the list of variables used in the intermediate representation combined with state and control variables.

Typically, individually named variables are implemented as registers, and arrays as sections of a synchronous on chip memory.

- (iii) State machine generation. In a simple implementation, this identifies all instructions that require clock cycles to access synchronous memories and adds a new state for all subsequent instructions. State transitions can include direct transitions from the last state of one block to the first state of that or another block. Conditional branches occur based upon the state of single bit variables that can be accessed by the state machine logic.
- (iv) Slave bus implementation is split into two sections: the registers and the on-chip memory that are provided to the bus in un-cached memory space. This section also produces an access list to allow translation of accesses made to the variable in other sections of the code (unimplemented blocks).
- (v) Local access to the memory based arrays. Since the processor is paused waiting for the hardware blocks to complete there is no possibility of contention or out of order access between the two memory interfaces.
- (vi) Implementation of non-memory access lines. Because of the state machine generated earlier, this is implemented as a case statement with a group of lines per state with no regard for the timing implications of that group. Additional states could be added by considering the need for pipelining at the state machine generation stage. Resource sharing is possible between states and will be implemented automatically by the synthesis engine.