Intelligent high level synthesis for customization on reconfigurable platforms

Sharad Sinha

2014

Sharad Sinha. (2014). Intelligent high level synthesis for customization on reconfigurable platforms. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/61691>

<https://doi.org/10.32657/10356/61691>

**INTELLIGENT HIGH LEVEL SYNTHESIS FOR
CUSTOMIZATION ON RECONFIGURABLE
PLATFORMS**

**SHARAD SINHA
SCHOOL OF COMPUTER ENGINEERING**

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

2014

Abstract

High level synthesis (HLS) using C/C++ has increasingly become a critical step in the realization of complex digital systems. One of the major research focus areas in this space has been to realize efficient synthesis of complex systems without violating stringent time-to-market constraints. Most of the related work cited in the literature has been mainly confined to supporting language constructs in C/C++, scheduling of operations and hardware binding for area reduction phase. The main motivation of the research work presented in this thesis is to develop novel algorithms for intelligent high level synthesis *without* designer intervention.

Using timing constraint as a knob, an algorithm called *Extended Compatibility Path Based Binding* (ECPB) has been proposed for resource sharing during hardware binding to minimize area utilization. It was demonstrated that the proposed method can be automated and has been shown to yield 12.49% and 29.21% on average lower area-delay product compared to compatibility path based (CPB) and weighted bipartite matching (WBM) based binding respectively. Building upon ECPB, a *latency-preserving* algorithm for *area-delay optimization* has been proposed to reduce area without violating timing constraint.

Data-initiation-interval aware graph partitioning algorithm has been proposed to partition an application's dataflow graph. This has also paved the way for further area reduction by invoking resource sharing without violating the initiation interval constraints. In addition, a systematic technique for area-delay trade-off analysis has been developed to establish multiple design points. The proposed method for initiation-interval-aware area optimization can be deployed for analyzing large dataflow graphs efficiently, making it highly scalable.

Technique for the efficient utilization of DSP resources available in FPGA platforms has been proposed to maximize application performance. It relies on the systematic investigation for the existence of multiplication and allied operations within the frequently executed code blocks of an application. Model based inferences for different types of

multiplication were developed to facilitate the rapid identification of profitable regions for maximizing performance. Investigations confirm that operating clock frequency can be increased by up to 3 times when compared to a commercial (Vivado-HLS) tool.

In order to combine the strengths of IP-core based design and high level synthesis, concepts of *program recognition* and *automatic algorithm replacement* are relied upon to develop lexical and pattern based analysis. This has led to the automatic identification of in-built arithmetic functions in a C/C++ application and compiler specific patterns. A novel IP-core selection algorithm has been proposed to facilitate the binding to available IP cores. Our investigations confirm that it lends well for notable area reduction when compared with that possible using a commercial HLS tool (Vivado-HLS). In addition, multiple design points can be generated to facilitate area-delay tradeoff analysis by associating a combination of IP-cores at a time. Our investigations show that the LUT reduction can be from 60% to 75% while the clock period reduction can range from 16% to 40% for the benchmarks investigated.

While the proposed methods for an intelligent high-level synthesis flow are applicable across all application domains, digital signal and information processing applications benefit greatly due to the existence of operations such as multiplication and transcendental functions. Additionally, since these methods look for application characteristics and exploit architecture specifics, they lead to customized synthesis solutions. Finally, the proposed methods have contributed to the realization of an intelligent high-level synthesis framework, which paves the way for less reliance on hand-crafted designs and skilled hardware designers.

Acknowledgment

I would like to take this opportunity to thank a number of people who have directly or indirectly helped me in my research. I would like to express my sincerest gratitude to my supervisor, Dr. Thambipillai Srikanthan for his constant support and guidance. It has indeed been a pleasure to work with him. He has always been there to guide regarding technical as well as administrative issues concerning my PhD. His approach to challenging the results and questioning the methodologies developed have helped me truly in improving my work. I am also thankful to some faculty members at the School of Computer Engineering in NTU and elsewhere with whom I have had discussions about research in reconfigurable computing area.

I would like to thank Mr. Lam Siew Kei, for his fruitful discussions. I would also like to thank Mr. Udit Dhawan and Mr. Ankit Aggarwal, two interns who worked with me for some time. Their support was crucial to finishing some of the work in time. I also take this opportunity to thank the wonderful laboratory staff of Ms. Nah, Ms. Marilyn and Mr. Jeremiah who have always given the best IT and administrative support in the lab.

It would not be fair if I did not thank my colleagues and friends. Of particular mention are Alok Prakash at NTU and Vipin K Menon, at my previous employer, Processor Systems (India) Pvt. Ltd., with whom I had several fruitful discussions regarding FPGA based designs and their verification. Looking back, I can say that I learned a lot in the two years that I spent there working on FPGA based designs. Things that I learned there have helped me significantly in my current research and framing its future direction.

Last but not the least, it gives me immense pleasure to thank my parents and my sister for supporting me in pursuing a research degree and constantly encouraging me to scale newer heights. Shreeja deserves special mention for putting up with the ups and downs of PhD life. I would like to thank God for His blessings, guidance and support in my life so far.

Table of Contents

Abstract.....	i
Acknowledgement.....	iii
List of Figures.....	vii
List of Tables.....	x
1. Introduction.....	1
1.1 Addressing Challenges of Design Complexity and Reduced time to Market.....	1
1.2 High Level Synthesis to Manage Design Complexity.....	3
1.3 Motivations for Research.....	4
1.4 Research Objectives.....	5
1.5 Main Contributions.....	5
1.6 Organization of Thesis.....	6
1.7 Publications Resulting from Research Presented in Thesis.....	7
2. Literature Review.....	9
2.1 FPGA Based Digital Hardware Design.....	9
2.2 Constraints in Digital Hardware Design.....	11
2.2.1 Economic Constraints.....	11
2.2.2 Technical Constraints.....	12
2.2.3 Hardware Design Complexity.....	13
2.3 Hardware Design Methodologies.....	15
2.3.1 Model Based Hardware Design.....	15
2.3.2 Intellectual Property (IP) Core Based Hardware Design.....	16
2.3.3 High Level Synthesis for Hardware Design.....	17
2.3.4 Comparison of Hardware Design Methodologies.....	17
2.4 Research in High Level Synthesis Methodology.....	19
2.4.1 Language of Design Specification.....	21
2.4.2 Compilation of Design Specification.....	23

2.4.3	Allocation, Scheduling and Hardware Binding.....	27
2.5	Summary.....	38
3.	Exploiting FPGA DSP Block for Architecture and Application Aware	
	Synthesis.....	40
3.1	Introduction.....	40
3.2	Contributions.....	40
3.3	Automatic Identification of Multiplication & Allied Operations.....	41
3.3.1	Types of Multiplication & Allied Operations in Applications.....	41
3.3.2	Algorithm for Identification of Multiplication & Allied Operations.....	43
3.4	Mapping Multiplication & Allied Operations to DSP Blocks.....	46
3.4.1	DSP Block Architecture and Functionality.....	47
3.4.2	LUT Utilization Model for Multiplication Operations.....	51
3.4.3	Selection Algorithm for Mapping of Multiplication Operations.....	62
3.5	Experiments and Results.....	63
3.6	Summary.....	66
4.	Hardware Binding for Area-Delay Efficient Synthesis	67
4.1	Introduction.....	67
4.2	Contributions.....	67
4.3	Extended Compatibility Path Based Hardware Binding Algorithm for Area Reduction.....	67
4.3.1	Experiments and Results.....	78
4.4	Joint Area-Delay Optimized Hardware Binding.....	81
4.4.1	Experiments and Results.....	86
4.5	Summary.....	87
5.	Application Dataflow Graph Partitioning Under System Level Constraint	88
5.1	Introduction.....	88

5.2	Contributions.....	88
5.3	Initiation Interval (II) Aware Dataflow Graph Partitioning.....	89
5.3.1	Graph Partitioning Flow.....	90
5.3.2	Extended Compatibility Path Based Hardware Binding.....	91
5.3.3	Initiation Interval (II) Aware Graph Partitioning Algorithm.....	93
5.4	Datapath Formation and Design Space Exploration.....	101
5.5	Experiments and Results.....	102
5.6	Summary.....	108
6.	Exploiting IP-Cores for Architecture and Application Aware Synthesis	109
6.1	Introduction.....	109
6.2	Contributions.....	111
6.3	Automatic Identification of Computation Blocks for Binding to IP Cores.....	112
6.3.1	Program Recognition.....	112
6.3.2	Compiler Specific Computational Patterns.....	114
6.3.3	Lexical and Semantic Analysis of in-Built C/C++Functions.....	115
6.3.4	Compiler Specific Patterns for Linear Algebra Operations.....	116
6.3.5	Data Structure for IP-Cores.....	123
6.4	IP Core Selection.....	127
6.5	Interface Synthesis for IP Core Mapped Computation Blocks.....	129
6.6	Experiments and Results.....	131
6.7	Summary.....	136
7.	Conclusion and Future Directions	138
7.1	Conclusion.....	138
7.2	Future Research Directions.....	141
	References	145

List of Figures

Figure	Page No.
2.1	Productivity gap and chip design complexity..... 14
2.2	Steps in high level synthesis..... 20
2.3	A control and dataflow graph (CDFG)..... 24
2.4	A dataflow graph for a basic block..... 24
2.5	Types of applications on the DSP precision spectrum..... 32
2.6	DSP block in 18-bit precision mode in Stratix V..... 33
2.7	DSP block in high precision mode in Stratix V..... 33
3.1(a)	Chain of multiply-add..... 44
3.1(b)	Chain of multiply-subtract..... 44
3.1(c)	Chain of multiply with a mix of add and subtract..... 44
3.2	Auto-Identify algorithm for identification of multiplication and allied operations..... 44
3.3	DSP48 architecture in Xilinx Virtex-4..... 49
3.4	DSP48E1 architecture in Xilinx Virtex-5..... 49
3.5	DSP48E1 architecture in Xilinx Virtex-6..... 50
3.6	Mapping of operations in a chain of allied operations to DSP blocks..... 50
3.7	A 3-bit by 3-bit binary multiplication..... 51
3.8	Actual LUT utilization corresponding to 8 different multiplications with same product bit-width (16)..... 53
3.9	Actual LUT utilization for A[7:0] x 8 bit constant..... 55
3.10	CLK period (ns) for A[7:0] x 8 bit constant..... 55
3.11	Joint Comparison and Selection (JCS) algorithm..... 62
4.1(a)	A scheduled DFG..... 69
4.1(b)	Hardware binding using WBM algorithm..... 69
4.1(c)	Hardware binding using CPB algorithm..... 70
4.1(d)	Hardware binding using proposed ECPB algorithm..... 71
4.2	Lifetimes of variables of DFG in Fig. 4.1(a)..... 71
4.3	Extended compatibility path based (ECPB) algorithm..... 74
4.4	Calculating the flow dependence factor F_{ij} 75

4.5	WOCG for the DFG in Fig. 4.1(a).....	76
4.6	Paths in WOCG of Fig. 4.5.....	76
4.7	Area-time product comparison.....	80
4.8	A time-constrained scheduled DFG with 6 addition operations.....	82
4.9	Algorithm for latency preserving slack based joint area-delay optimization.....	84
4.10	WOCG for the scheduled DFG in Fig. 4.8.....	85
5.1	A large DFG.....	89
5.2	Typical datapath for area minimization.....	89
5.3	Datapath for II=10 clock cycles.....	89
5.4	HLS flow with DFG partitioning.....	90
5.5	A scheduled DFG.....	91
5.6	WOCG of addition in Fig. 5.5.....	91
5.7	II-Aware dataflow graph partitioning algorithm.....	94
5.8	A higher level flow of II aware DFG partitioning.....	95
5.9	Graph traversal and formation of partitions.....	95
5.10	DFG type 1.....	96
5.11	DFG type 2.....	96
5.12	DFG type 3.....	97
5.13	DFG type 4.....	97
5.14(a)	DFG with operator chaining.....	99
5.14(b)	DFG with no operator chaining.....	99
5.15	A DFG with a multicycle operation <i>op-mc</i>	100
5.16	Partition ranking algorithm.....	101
5.17	Area-Clk Plot for DFG50.....	106
5.18	Area-Clk Plot for DFG100.....	106
5.19	Area-Clk Plot for DFG300.....	107
5.20	Area-Clk Plot for DFG500.....	107
6.1	Proposed IP-based HLS flow.....	112
6.2	For-loop code for 2 x 2 matrix-matrix multiplication in C.....	117
6.3	For-loop code for a matrix-vector multiplication in C.....	117
6.4	For-loop code for scaling a 2 x 2 matrix in C.....	117
6.5	For-loop code for 2 x 2 matrix-matrix addition in C.....	117

6.6	Directed acyclic graph (DAG) of IR for matrix-matrix multiplication.....	119
6.7	Directed acyclic graph (DAG) of IR for matrix vector multiplication.....	119
6.8	Directed acyclic graph (DAG) of IR for matrix scaling.....	119
6.9	Directed acyclic graph (DAG) of IR for matrix-matrix addition/subtraction.....	120
6.10	Algorithm for identification of matrix operations in LLVM IR.....	120
6.11	IP Core dependency resolution algorithm.....	128
6.12	A part of the state machine for datapath in the presence of an IP core.....	130

List of Tables

Table	Page No.
2.1 Some C like languages for hardware synthesis.....	21
3.1 Variation of <i>Actual</i> (LUT), <i>Count</i> (LUT) and DSP Implementation.....	54
3.2 Contributing rows for different columns for 4 bit variable multiplied by 4 bit constant.....	57
3.3 Contributing rows for different columns for $m * m$ multiplication.....	57
3.4 Example No. of FA for $m = 4$ (constant=4'b1010).....	57
3.5 Contributing rows for $m > n$	58
3.6 Contributing rows for $m = 5$ and $n = 3$	59
3.7 Contributing rows for $m < n$	59
3.8 Contributing rows for $m = 3$ and $n = 5$	59
3.9 Variation in LUT consumption for different 9 bit constants with same number of 1's in binary representation.....	61
4.1 Post place and route statistics for motivating example.....	72
4.2 MUX input count comparison.....	78
4.3 Global interconnect count comparison.....	79
4.4 LUT consumption comparison.....	79
4.5 Best achievable clock period (ns) comparison.....	80
4.6 LUT consumption results.....	86
4.7 MUX input count comparison.....	86
5.1 Adder area for different Xilinx Virtex 4 devices.....	102
5.2 Comparison of number of FUs for II=3 clock cycles(MediaBench).....	103
5.3 Comparison of number of FUs for II=3 clock cycles (ExPRESS).....	104
5.4 DFG vs. number of partitions for different II.....	105
5.5 Descending order of partitions for DFG50 for II = 3 clock cycles.....	105
5.6 Post place and route synthesis results for DFGs for II = 3 Clock Cycles.....	106
6.1 Representative list of standard arithmetic functions in C/C++.....	110

6.2	Matrix size determination for matrix-matrix multiplication.....	118
6.3	Matrix size determination for matrix-vector multiplication.....	118
6.4	Matrix size determination for vector-matrix multiplication.....	118
6.5	Matrix size determination for matrix scaling.....	118
6.6	A function f and corresponding IP cores.....	124
6.7	Configurable parameters in Xilinx CORDIC IP core.....	125
6.8	Available configurable parameters for Xilinx CORDIC IP core implementing $\text{sqrt}()$ function.....	125
6.9	Representative characterization data for CORDIC configured for $\text{sqrt}()$	126
6.10	Mapping of some representative in-built C functions.....	132
6.11	Mapping of matrix multiplication operation.....	133
6.12	Mapping of exponential decay formula.....	133
6.13	Mapping of greater circle distance formula.....	134

Chapter 1

Introduction

1.1 Addressing Challenges of Design Complexity and Reduced Time to Market

Modern electronic systems are getting increasingly more complex. There is a rapid increase in the number of features to be supported resulting in designs getting bigger and involving complex calculations and interface protocols. The complexity is increasing both in hardware and software design aspects of a system [18]. However, the designer productivity using conventional design methodologies has not been increasing at the same pace leading to a productivity gap [17]. This has led to investigations into design methodologies that could reduce this productivity gap.

The global electronics industry is beset with the issue of reducing time to market (TTM) in many industries as researched very thoroughly in a report from the MIT Sloan School of Management [195]. The productivity gap as mentioned in [17] aggravates the challenge of dealing with decreasing TTM windows. This has led to investigations not only in design methodologies but also architectures and devices that can be used for rapid design of systems. Such architecture and devices, like Field Programmable Gate Arrays (FPGAs), make it easier to design a product in a shorter time span compared to previously used Application Specific Integrated Circuits (ASICs) etc. while still meeting the design requirements. Rapidly evolving industry standards in domains like telecommunications coupled with reduced time to market also demand newer design methodologies and flexible to use architectures.

In this aspect, reconfigurable platforms are a suitable candidate as they allow quick digital design. Reconfigurable platforms refer to platforms where the hardware can be reconfigured as a design undergoes changes and evolution. These platforms can be reconfigurable chips like FPGAs, FPGAs together with a processor, multiple FPGAs

connected together, coarse grain reconfigurable arrays (CGRAs) [47], [48] or any combination of all of these. Essentially, FPGAs form the reconfigurable logic as they can be reconfigured multiple times. Examples of FPGA technology includes devices from vendors like Altera [246], Xilinx [245], Lattice [247], Actel [248] etc. The work presented in this report focuses on high level synthesis targeted towards FPGAs as reconfigurable platforms. Most of these techniques will also be applicable to the physical synthesis part of multi-core processors if some processing elements (PEs) are in the form of reconfigurable logic similar to the reconfigurable logic found in FPGAs. CGRAs have been excluded from the current work because they are primarily used for accelerating loops in applications. Hence, they must be coupled with a processor which executes the non-loop code present in the application. A good example of such coupling is the Samsung Reconfigurable Processor where a Very Long Instruction Word (VLIW) processor [49] is coupled to a CGRA [50]. Thus, unlike CGRAs, FPGAs are more general purpose as they can be used to implement both loop code as well as non-loop code. Also, CGRAs operate on word-level or sub-word level data whereas designs in FPGA can operate at bit level in addition to any word size. The difficulty of compiling applications for CGRAs so that efficient schedules can be generated has been a major roadblock to their commercial adoption [217]. Additionally, there is a complete lack of standardization with the presence of a number of different CGRA architectures which are all supported by proprietary compilation flows which has also contributed to its lack of commercial adoption.

The primary reason for thrust on designing with FPGAs lies in the fact that they help to design, validate and prototype a product quickly. This helps in meeting the Time to Market (TTM) window which is becoming increasingly smaller in different market segments. Other major reasons include the flexibility to optimize design performance by speeding up execution in hardware because FPGAs offer massive spatial parallelism which can be used to exploit data and task parallelism in applications. Their reconfigurability makes them a natural choice for telecom products where communication standards change and evolve in short time spans. This puts tremendous pressure on designing products which can be reconfigured to adapt to the changing requirements. Also, their reconfigurability gives the flexibility to try

different options in an algorithm or design to be mapped without actually changing the target hardware. This enables one to refine the design until performance constraints are met while targeting the same device. One more reason which makes FPGAs attractive is the low non-recurring engineering cost (NRE) associated with them compared to ASICs. Designing an ASIC at today's latest technology nodes is resource intensive, time consuming, hideously complex and extremely expensive. An ASIC design at the latest or even slightly older technology nodes would cost more than US\$ 50 million. This requires that the end product should have a very high sales volume in order to amortize the multi-million dollar development cost. This can be achieved only in certain specific consumer electronics segments or highly mission and safety critical applications like those related to space and nuclear electronics. FPGAs based designs on the other hand have very low NRE. Designers need not worry about the expensive tape-out process. The design process in itself is a lot simpler and most of the electronic design automation (EDA) tools for FPGAs are almost free or very cheap. Though the price per piece of high end FPGAs may seem high, the fact that they can be reprogrammed in field easily, management of engineering change orders (ECOs) is very simple and design, verification and validation is easier compared to ASICs makes them an extremely attractive design alternative that makes both technical as well as business sense.

1.2 High Level Synthesis to Manage Design Complexity

The complexity of embedded applications is increasing day by day. It is increasingly becoming difficult to manage the register transfer level (RTL) complexity of designs and at the same time manage to meet the Time to Market (TTM) window. RTL complexity coupled with constraints is proving to be a design and verification nightmare for system designers. High Level Synthesis (HLS) has emerged as a major design methodology in dealing with increasing design complexity.

Research in high level synthesis has been conducted for more than two decades and yet there is no efficient high level synthesis (ITRS roadmap uses the word behavioral synthesis) "despite recent advances driven by C- and SystemC-based synthesis and transaction level modeling technologies" as noted in the 2009 edition of International

Technology Roadmap for Semiconductors (ITRS) on Design [249]. ITRS in its 2004 update on roadmap for design had stated that by raising the level of abstraction above RTL, productivity could be increased by 200K gates/designer-year [238], [239] and this would enable managing the growing design complexity. The main challenges in high level synthesis identified based on [60], [19], [239], [240] and discussions in various research forums are:

- Synthesizing from complex C constructs like pointers and recursive functions and reducing the requirement of code restructuring in C-based HLS flows.
- Improving the quality of results produced by HLS flows to match them with handcrafted RTL designs for large applications.
- Increasing area-time efficiency of designs to meet tighter design constraints.
- Rapid design space exploration to choose between alternate micro-architecture choices without significant run time.
- Power consumption estimation at a higher level of abstraction.

1.3 Motivation for Research

As FPGA design starts are increasing every year and high end FPGAs are increasing in density and performance, it is imperative that a suitable design methodology be developed for them keeping in mind the increasing design complexity. It is imperative that a design methodology not only improves the process of design and presents a set of better design techniques, but also increases designer productivity and helps design teams in meeting TTM windows without compromising on performance. As FPGAs are being widely used as hardware accelerators [241], [243] and increasingly finding their way into high performance computing [244], [242], it is necessary to have a design methodology that helps in quick design and is an improvement over existing HLS approaches in some way. It is extremely essential to address the issues of quality of results with the scaling of design problems, the two of which are not sufficiently advanced in existing approaches and need further research. In this aspect, application and target hardware architecture aware synthesis methodology can play a big role. The distinct hardware blocks available on FPGAs and the application specific

characteristics can be exploited to make *intelligent synthesis* a reality. Application and target hardware architecture aware concept is explained further in Chapter 2. Mapping of operations to operators on FPGAs in order to reduce area, improve timing and manage application complexity are challenging problems.

1.4 Research Objectives

The main objectives of the research work are:

- (i) To understand the existing high level synthesis methodologies, their different steps, identify their weaknesses and strengths, applicability and suitability.
- (ii) To become familiar with different types of resources available on Field Programmable Gate Arrays (FPGAs) and the use cases for these resources.
- (iii) To propose a suitable intelligent synthesis framework that takes into account constraints (functional as well non-functional), application characteristics and target hardware architecture specifics.
- (iv) To devise scalable algorithms and techniques that help in rapid design space exploration and synthesis guided by constraints, application characteristics and target hardware architecture specifics.

1.5 Main Contributions

The main contributions of this thesis encompass techniques and algorithms developed with focus on application and target hardware architecture aware high level synthesis for FPGAs. These are presented below:

- (i) Automatic identification of different kinds of multiplication operations as well as allied operations like multiplication followed by addition or subtraction has been proposed in chapter 3. Once these are identified, an architecture aware algorithm maps them to the limited number of DSP blocks on FPGAs in order to efficiently utilize on-chip DSP blocks to maximize performance and save LUTs for other operations.

- (ii) Using timing as a knob, an algorithm called “*Extended Compatibility Path Based Binding (ECPB)*” has been proposed which focuses on area reduction by resource sharing without any timing constraint. Its extension called “*Latency preserving slack based joint area-delay optimization*” builds upon it to share resources while meeting a given timing constraint. These are discussed in chapter 4.
- (iii) Application dataflow graph partitioning under the system level constraint “Data Initiation Interval” is presented with the aim to meet the constraint as well as to reduce resource utilization by just the right amount. Hardware binding techniques presented in (ii) are applied to partitions to reduce the resource utilization. The proposed algorithm is adaptive and multi-dimensional compared to existing temporal partitioning algorithms. These are discussed in chapter 5.
- (iv) An IP-based high level synthesis methodology that uses concepts of *program recognition* and *automatic algorithm replacement* is presented in order to exploit application characteristics like transcendental functions and bind them to hardware specific IP-blocks based on a cost measure. This method brings together the benefits of IP-core based design and high level synthesis and is described in detail in chapter 6.

1.6 Organization of Thesis

Chapter 2 includes the detailed literature survey on high level synthesis for reconfigurable platforms and also explains the key concepts guiding this research. Chapter 3 presents the work on mapping different kinds of multiplication operations to limited on-chip DSP blocks on FPGAs. Chapter 4 presents the hardware binding algorithm methodology using timing as a knob to reduce resource utilization. Chapter 5 deals with application dataflow graph partitioning under the system level constraint called “Data Initiation Interval (II)”. In Chapter 6, an IP-core based high level synthesis technique is presented that is application and target hardware architecture aware and is directed at specific arithmetic functions supported in C/C++ language as

well as compiler specific patterns. Chapter 7 concludes with the work presented in this thesis with pointers to related future work.

1.7 Publications Resulting from Research Presented in Thesis

Journal Papers

- (1) Sharad Sinha and Thambipillai Srikanthan, “IP-Enabled C/C++ based high level synthesis: A step towards better designer productivity and design performance,” *International Journal of Reconfigurable Computing*, vol. 2014, Article ID 418750, 17 pages, 2014. doi:10.1155/2014/418750.
- (2) Sharad Sinha, Udit Dhwan and Thambipillai Srikanthan, “Extended compatibility path based hardware binding: an adaptive algorithm for high level synthesis of area-time efficient designs”, *Journal of Circuits, Systems and Computers*, vol. 23, no. 9, Article 1450131, October 2014.
- (3) Sharad Sinha and Thambipillai Srikanthan, “High level synthesis: boosting designer productivity and reducing time to market”, *IEEE Potentials* (Accepted).
- (4) Sharad Sinha and Thambipillai Srikanthan, “Dataflow graph partitioning for area-efficient high level synthesis with systems perspective”, *ACM Transactions on Design Automation of Electronic Systems* (Accepted).

Conference Papers

- (1) Sharad Sinha and Thambipillai Srikanthan, “Constraint-aware synthesis of embedded applications on reconfigurable platforms”, PhD Forum, International Conference on Field Programmable Technology (FPT 2012), Seoul, South Korea, Dec 10-12, 2012.
- (2) Sharad Sinha and Thambipillai Srikanthan, “Dataflow graph partitioning for high level synthesis”, 22nd International Conference on Field Programmable Logic and Applications, FPL 2012, Oslo, Norway, August 29-31, 2012, pp. 503-506.

- (3) Sharad Sinha, Udit Dhawan, Siew-Kei Lam and Thambipillai Srikanthan, “A novel algorithm to reduce critical path delay during high level synthesis”, VLSI(ISVLSI), 2011 IEEE Computer Society Annual Symposium on, Chennai, India, July 4-6, 2011, pp. 278-283.
- (4) Sharad Sinha and Thambipillai Srikanthan, “A framework for high level synthesis of transcendental functions”, VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on, Hsinchu, Taiwan, April 25-28, 2011, pp. 1-4.
- (5) Udit Dhawan, Sharad Sinha, Siew-Kei Lam and Thambipillai Srikanthan, “Extended compatibility path based hardware binding algorithm for area-time efficient designs”, Quality Electronic Design (ASQED) 2010, 2nd Asia Symposium on, Penang, Malaysia, August 3-5, 2010, pp. 151-156.

Workshop Presentations

- (1) **Best Speaker Award:** Sharad Sinha and Thambipillai Srikanthan, “High level synthesis: past, present and future”, 2013 Graduate Students’ Workshop on Circuits, Systems and Signal Processing, IEEE Circuits and Systems Society, Singapore Chapter, October 1, 2013.

Programming Challenges

- (1) **Special Mention Award:** II Quick FPGA Programming Contest with Vivado-HLS, Reconfigurable Computing and Applications Conference 2013, Madrid, September 18-20, 2013

Chapter 2

Literature Review

2.1 FPGA Based Digital Hardware Design

Field Programmable Gate Array (FPGA) has been a major architecture on which digital designs are implemented. It has come a long way from the days when only glue logic was implemented on it [38], [39]. The opportunities for spatial parallelism and reconfiguration that a FPGA offers has made it a choice for digital design in industries like telecommunications [1] where standards evolve rapidly. For instance, 2G[3], 3G[4], 4G[5] and 4G LTE [6] have all appeared in a short span of 20 years along with a host of other related mobile broadband standards like 3GPP [7]. Because of the massive spatial parallelism offered by FPGAs, they find numerous application as hardware accelerators in domains like web-search [8], linear algebra computations [9], image processing [10], biomolecular simulation [243] etc. and have been used to design complete high performance systems like the code breaking machine COPACOBANA [46].

The basic element in an FPGA is a look up table (LUT). Modern FPGAs have thousands of LUTs. They also provide a few tens of Mbits of on-chip memory as well as hard blocks for digital signal processing like the DSP48E1 in Virtex-5 [40] from Xilinx and DSP blocks in Stratix devices from Altera [41]. They also have on-chip modules like Media Access Control (MAC) block for implementing networking applications. For instance, Xilinx Virtex-4 and Virtex-6 devices have Tri-Mode Ethernet MAC (TEMAC) [42], [43]. Many devices also provide PCI-Express endpoint blocks on-chip [44]. Such resources are also available on Stratix IV, V etc. devices from Altera [45]. Thus FPGAs not only have fine grained elements like LUTs but also coarse grained resources like DSP, MAC and PCIe blocks. Availability of such resources on the chip itself has made FPGAs suitable for many applications and now

find usage in domains like automotive, medical, industrial, aerospace, networking, communications etc. [196], [197].

The availability of such resources has made it possible to implement complex and big digital designs on FPGAs [10], [46]. Thus, there is an increase in the complexity of designs being implemented on FPGAs with the complexity being the result of utilizing not only LUTs but other resources as well like the on-chip DSP blocks in Xilinx [40] and Altera FPGAs [42]. In addition to the complexity in terms of size, other factors like performance requirements and ever increasing constraints derived from market pressure, regulatory compliance, evolving standards and increased demand for superior user experience [231], have also made FPGA based digital design an increasingly more challenging task. Moreover verification and validation also add to this complexity [170], [173].

The traditional register transfer level (RTL) based design approach, where a design is described by a designer using a hardware description language like Verilog or VHDL, is increasingly becoming more time consuming because of increasing design complexity as well as associated verification challenges [172], [173]. Also, there is only so much by which designer productivity can increase whereas design complexity continues to increase unabated. This has been highlighted time and again in International Technology Roadmap for Semiconductors: Design [18]. There have been persistent calls for raising the levels of abstraction to describe functionality in order to explore different architecture options and to get a global view of the design. In fact, in [2] system level design is defined as “the utilization of appropriate abstractions in order to increase comprehension about a system, and to enhance the probability of a successful implementation of functionality in a cost-effective manner”.

Hardware-software co-design methodology has been studied in great detail [11], [12], [13] to reduce the design time gap between hardware and software. These methodologies look at developing software in parallel with hardware without having to wait for the latter to be ready. However, sometimes before one can go ahead with hardware-software co-design, it is necessary to determine which part of an application

will run in software and which in hardware. This adds to the complexity of software design as some parts will be implemented in hardware. This has been studied in great detail as hardware-software partitioning problem [14], [15], [16].

2.2 Constraints in Digital System Design

All FPGA based digital systems have to be designed and need to perform under a set of constraints. These constraints have an impact on design methodology and design goals. The constraints are derived from both the economic and the technical domains. These constraints are described in Sections 2.2.1 and 2.2.2.

2.2.1 Economic Constraints

The principal economic constraints are Time to Market (TTM) window, Non-Recurring Engineering (NRE) cost and volume of sales. In the consumer electronics space which spans product offerings in telecommunications, entertainment, information and home appliances industries, TTM windows are very small [195]. New and improved products need to be launched in a very short time span. This means that the design methodology should enable quick and rapid development without compromising on required performance. The NRE cost for FPGA based design is less compared to counterparts like Application Specific Integrated Circuits (ASICs) and structured ASICs. This is primarily because FPGAs by themselves can be considered to be devices at a higher level of abstraction than ASICs as one does not have to design at the transistor level in order to implement an algorithm on them. Shorter TTM window also means that there is less time to explore different architecture options. Therefore a design methodology that helps in exploring different architecture options while still meeting the TTM window is always welcome. Design methodologies at a higher abstraction level than RTL [19] have been under investigation because they have the potential to offer quick architecture exploration before a final architecture is selected for the design under consideration. It is now well accepted that computing platforms that afford quick design changes, design methodologies that afford quick architecture exploration under different performance constraints and enable quick and

easy comprehension of the design are needed to deal with the increasing design complexity.

2.2.2 Technical Constraints

Technical constraints arise from the fact that a greater number of features needs to be supported in modern systems. The support for increased features has to come from either the software or the hardware subsystem. Not only the features need to be supported but also the constraints related to them. Constraints can be categorized under two categories: a) functional and b) non-functional. Functional constraints guarantee correct input-output relationship. These are low level constraints related to the implementation of the system and the choice of the architecture for the design. They ensure the functionality of a system as outlined in its requirements and guide in what a system should do with the data presented as input to it.

Non-functional constraints on the other hand specify how a system should be with respect to performance metrics like area utilization, input-output delay, throughput, power consumption, reliability, fault tolerance etc. These constraints help in judging the operation of the system and do not define specific behavior or functions. They characterize properties of the system and not its specific functions [251]. Data oriented constraints like "*Initiation Interval (II)*", data type (real or integer, signed or unsigned) are also classified as non-functional constraints.

Not all of these constraints are necessary or required to be met in every design. For instance, area is a constraint when one wants to use as small a FPGA device as possible. This could be dictated by economic considerations. However, this may not be a constraint when a FPGA device has to be used for high performance computing as described in [242], [244] where performance is paramount and critical to success.

The increasing number of features to be supported and the associated constraints lead to complexities in design. These complexities can be understood at the levels of overall system, software subsystem and hardware subsystem. A proper understanding of all these constraints and their interplay is necessary for the success of a design.

2.2.3 Hardware Design Complexity

Hardware design complexity includes complexities related to design as well as verification. Increasing design complexity has led to increasing verification challenges. The increasing number of design constraints like power, area, timing (or delay), throughput, latency etc. add to both design and verification complexities. It should be noted here that all of these constraints are not exclusive to one another. This forces the designer to make a tradeoff analysis with respect to satisfying them. For instance area-delay tradeoff analysis [21], [22], [23] gives insight into various possible implementation architectures for a design with an area and a delay value associated with each architecture. With this tradeoff analysis, a designer is able to select an architecture that satisfies area and timing requirements. The architecture could correspond to an application to be implemented as in [23] or to the underlying hardware itself as in [22].

Once a design is ready, its verification to ensure that implementation matches the specification is necessary [167]. Functional verification which attempts to verify the functional correctness of an implementation is typically carried out using simulation [168] where test vectors are applied to the RTL design and its output compared against the desired output. However, simulation based verification can become extremely time consuming when the number of test vectors is very large [169]. For instance, simulation based verification of a 64-bit floating point division unit would require 10^{25} years at 1 test every 1 microsecond in order to simulate all the possible 2^{168} test vectors [169]. It is clear that in modern complex designs with many modules, simulation based exhaustive verification is nearly impossible. Formal verification [170], [171], which is based on mathematical reasoning, has been proposed as an alternative to simulation based verification. However, it is extremely difficult to formally verify very large designs which can have thousands of states [172]. Other methods of verification like model checking [173], constrained random simulation [174] etc. also have similar limitations in dealing with the increasing verification challenges associated with increasing design complexity. Hardware-software co-

verification [175] is another major challenge for hardware designs that involve significant software components in addition to complex hardware modules.

Taking the number of transistors in a design as a measure of design complexity, Fig. 2.1 shows that there is a growing gap between increasing design complexity on one hand and designer productivity on the other [17] which means that there is more effort involved in design as well as verification.

Fig. 2.1. Productivity gap and chip design complexity [26]

In order to address the hardware design complexity, higher levels of abstraction for describing hardware subsystems have been studied. These include descriptions in C/C++ [19] and description using executable blocks [20]. All of these descriptions are followed by automatic generation of register transfer level (RTL) description for implementation on FPGA. These take away the pain of manually designing the RTL of complex digital systems. Instead, designers could focus on refining the algorithm or block level implementation in order to meet functional requirements and then let automation generate the RTL. Using the same testbench used to verify C/C++ description of a design to verify the auto-generated RTL, it has also been shown that higher levels of abstraction bring benefits to verification [176]. This obviates the need to write separate testbench to verify the auto-generated RTL for functional correctness thus speeding up the verification effort [176].

Structured ASICs [186], [187] have also been around as a device alternative to ASICs to achieve reduction in design and verification effort compared to ASICs. These devices are mask-programmable compared to FPGAs which are field programmable and thus tend to have more capacity than a similar size FPGA because programmable switches have been removed by vias between metal layers resulting in additional logic capacity [186], [187]. However, being only mask programmable reduces their flexibility when compared to FPGAs. Since the metal layers can be programmed only once, the flexibility is limited only to the design effort pre-mask programming.

2.3 Hardware Design Methodologies

As already discussed in previous sections, hardware design complexity is growing and there is an increasing productivity gap. Also, the evolving nature of standards in industries like telecommunication and the push for greater performance need to be taken into consideration when analyzing design methodologies.

2.3.1 Model Based Hardware Design

Model based hardware design methodology has been applied to embedded systems hardware design. Pre-defined configurable blocks can be assembled to model a hardware subsystem in a visual environment and once testing and verification at the model level is complete, RTL description can be automatically generated which can be targeted to FPGAs. For example, Xilinx System Generator for DSP [24] can be used to model a digital signal processing (DSP) system using Xilinx Blockset that contains functions for signal processing (e.g. FIR filters, FFT etc.), error correction (e.g. Reed-Solomon encoder/decoder etc.), memory etc. Different data types like fixed-point and floating point can be specified as well as different parameters of the blocks in the blockset can be configured to simulate the system in Simulink. Once the system has been simulated and results verified, RTL description in Verilog or VHDL can automatically be generated. Thus this design method does not require a designer to manually prepare the RTL description. The Altera DSP Builder tool [25] also follows a model based hardware design methodology similar to Xilinx System Generator. As can be easily understood, these design methodologies are restricted to designing

hardware subsystems for DSP algorithms and hence are limited in scope. It should be noted that in order to use these model based systems, it is still important to have at least a basic understanding of hardware and memory modules present in targeted architectures so as to be able to maximize the quality of result.

2.3.2 Intellectual Property (IP) Core Based Hardware Design

IP core based hardware design methodology is adopted to reduce the design effort and time. It is based on the principle of reusing already verified modules [26], [27]. The modules can exist as *hard-IP* which is laid-out standard cell level design that has been characterized and verified for a set of technology nodes. Modules can also exist as *soft-IP* which is verified and characterized netlist level description that can be imported into a design flow. The netlist could be a synthesizable RTL description or in EDIF (Electronic Design Interchange Format) [28] format either of which can be imported into a design flow. Thus, IP cores provide a level of abstraction [29] for hardware design where some or all functional modules are implemented using third-party or in-house IP cores. Since these have already been tested, verified and validated for a range of operational requirements, a designer can be more sure about the quality of hardware design. However, it should not be forgotten that there is effort involved in IP core selection, IP assembly and testing and verification of a hardware system assembled using IP cores [49]. Generally, an IP core comes with its own verification set up so that when a system is assembled using IP cores, the entire system can be tested and verified. Not all IP cores support all technology nodes or all FPGA families. Nevertheless, the hardware subsystem design task becomes easier for the supported technology nodes or FPGA families. The IP-XACT standard [30] was developed to promote a standard way of exchanging IP related data between different tools and IP vendors. IP related data contains information on configurable parameters, register file size, clocking requirement etc. This data is packaged in XML format [31]. An example of IP based hardware design would be a Xilinx FPGA based “Video over Internet Protocol” system that makes use of the different related IP cores offered by Xilinx [32].

2.3.3 High Level Synthesis for Hardware Design

High Level Synthesis (HLS), also referred to as behavioral synthesis in its very early days of research, is a hardware design methodology to manage design complexity and afford architecture explorations [19]. Algorithms to be implemented in an ASIC or programmable hardware like FPGA are described in C/C++/System C. This description is then verified against a C/C++/System C level testbench and then compiled under certain performance constraints like required operating clock frequency and finally RTL description is auto-generated. This RTL description is then processed further by physical synthesis tools. Thus this design methodology abstracts the hardware details for a designer who can now concentrate on refining the C/C++/System C description of algorithms. Verification of C/C++/System C description ensures that the description is functionally correct. Once the RTL has been generated, it can also be verified using the same testbench that was used to verify C/C++/System C description. This ensures that the generated RTL is functionally correct. The reasons for choosing C/C++ as languages for describing algorithms include the dominance of these languages in algorithmic implementations as well as the vast amount of legacy code available in these languages. Academic HLS tools include SPARK [33], GAUT [34], LegUp [164] while commercial HLS tools include Xilinx Vivado HLS [35], Calypto Catapult-C [36] and Cadence C-to-Silicon Compiler [37]. The design methodology makes use of FPGA device or ASIC technology node specific library of components (arithmetic, memory, bus, registers etc.) to generate the RTL code for a given target FPGA or ASIC technology node [71].

2.3.4 Comparison of Hardware Design Methodologies

Sections 2.3.1, 2.3.2 and 2.3.3 presented model based hardware design, IP core based hardware design and high level synthesis based hardware design methodologies respectively. Of all the three methodologies presented, IP core based hardware design is at the lowest level of abstraction with high level synthesis offering the highest levels of abstraction and requiring minimal designer intervention or expert hardware knowledge. The model based hardware design methodology is restricted to digital

signal processing applications. It also requires manually configuring the blocksets (models) [44], [45] in order to achieve the desired functionality. In order to configure the blocksets, one needs to have sufficient knowledge of the target hardware architecture in order to guide such tools towards generation of better RTL descriptions. Some efforts in alleviating this need have been presented in [51] where blocksets are still used for designing an algorithmic system but the auto RTL generation is now achieved using data trace driven analysis instead of auto-configuration of RTL modules (present in a library) corresponding to the blocksets. This has been implemented in Altera DSP Builder tool [45]. Model based and high level synthesis based hardware design methodologies abstract a limited set of digital circuit design principles and hardware architecture features. Thus a designer can instead focus on algorithm development and refinement. Existing IP core based methodology still exposes these principles to a designer when the IP cores are integrated into a design at RTL level. There are system integration tools like Xilinx EDK [52], Altera Qsys [53] and Microsemi Libero [185] where a processor can be connected to different IP cores in a visual representation and these tools automatically generate the entire digital system. These require IP cores which are compliant with the bus interfaces like Advanced eXtensible Interface (AXI) [54], Avalon [55] supported by these tools. It can therefore be understood that both model based design and high level synthesis methodologies can themselves be applied for developing IP cores as an IP core can be considered as a system in itself. The requirement of blocksets in case of model based design has a restrictive effect as it ties down a designer to developing hardware only for those applications which can be modeled using the available blocksets. Also, one would need to use lower level blocksets (like add, multiply, look up table etc.) if one has to implement an algorithm that cannot make use of any existing higher level blockset. For instance, [56] discusses the FPGA implementation of primitives for homomorphic encryption using Simulink. Design methods like in this particular case [56] tend to gravitate more towards an actual digital system design with knowledge of hardware resources and thus depart from the principle of abstracting away from a designer target architecture details and digital circuit design principles. Also, the quality of auto-generated RTL description by MATLAB is claimed to be

within 20% of hand crafted RTL [57] which is a major issue when it comes to implementing large algorithms or targeting an area-constrained hardware, for instance an FPGA with limited resources. Thus, it is evident that C-based high level synthesis based hardware design methodology (section 2.4.3) can abstract away architecture details and digital circuit design principles to a far greater extent than any of the other two methodologies. The simple reason for this strength lies in the fact that a true C-based description can only be algorithmic in nature with no other implementation detail. Since it is also not tied to any blockset like design method, it can cover almost all fields of application like signal processing, cryptography, networking, control intensive applications etc. Since the description is not tied to a target architecture in any way, there is far greater degree of freedom in choosing not only implementation options but also a target architecture. It should be noted that all the three design methodologies help in managing design complexity which results in quicker design closure and increases the ability to meet TTM window. However, as discussed in this section, they have their own strengths and weaknesses relative to the design goal. From a digital system design perspective, C-based design description offers the most flexibility not only for design but also for verification.

2.4 Research in High Level Synthesis Methodology

High level synthesis methodology involves describing an algorithm in C/C++/SystemC, refining this description to meet the input output relationship and then auto-generating the RTL description for a target architecture under a set of constraints. The constraint that is typically applied is the required clock frequency of the design and enabling resource sharing so as to reduce area [35], [36]. Figure 2.2 shows the HLS design steps based on [19]. The HLS flow includes the following steps starting from a high level description, a library of RTL components and design constraints:

- (i) Specification in a high level language: C, C++, SystemC
- (ii) Compilation of the specification (formal model generation)

- (iii) Allocation of hardware resources (functional units, storage units and buses from the RTL component library) to operations in the application
- (iv) Scheduling of operations in different clock cycles
- (v) Binding operations to allocated functional units, variables to storage elements, buses/interconnect for data transfers
- (vi) Generating the RTL micro-architecture

Each step in HLS flow has been an area of research in itself. Research has been conducted into studying the suitability of different languages for design description [59], compilation methods including automatic extraction of parallelism from sequential C-code [10], [11], formal model analysis for simplifying the type and the number of operations, scheduling of operations based on data dependency and other design constraints [58], binding operations to different functional units and variables to registers based on resource allocation [19].

Fig. 2.2. Steps in high level synthesis (HLS) [19]

While binding and scheduling have an inter-dependence relationship because scheduling is based on required clock period constraint which affects the kind of functional unit that an operation will be bound to, these steps are executed in a sequential manner [19] while taking into consideration the effects of binding

(knowledge of library modules used to implement operations) during the scheduling step [58].

2.4.1 Language of Design Specification

C or C like (C++, SystemC) languages have been proposed for design specification for HLS. Languages like Hardware C, Ocapi [59] etc. have also been proposed and developed. Table 2.1 shows a list of various C-based languages developed for hardware synthesis and their main characteristics [59]. Most of these languages have been restricted to research only purpose. BlueSpec SystemVerilog[165], [166] is another language that has found some acceptance with designers. Since it is an extension of System Verilog it is close to hardware design while providing a certain level of abstraction above Verilog and VHDL. However, the only languages that have gained a respectable degree of acceptance and generated debate on which language should be used for HLS are C, C++ and System C [60], [61].

Table 2.1. Some C like languages for hardware synthesis [59]

Language	Characteristics
Cones	Early, combinational only
Hardware C	Behavioral synthesis centered
Transmogriifier C	Limited scope
Ocapi	Algorithmic structural descriptions
C2Verilog	Comprehensive
BDL	Many extensions and restrictions (NEC)
Handel-C	C with CSP (Celoxica)
SpecC	Resolutely refinement based
Bach C	Untimed semantics (sharp)
CASH	Synthesizes asynchronous circuits

The debate [95], [96] has largely revolved around issues of easy algorithm development, verification, parallelism, appropriateness for hardware description, timing specification embedded in algorithm for timing sensitive designs, virtual

prototype modeling etc. C and C++ are generally the languages of choice for algorithm implementation and there is a vast amount of legacy code in C/C++ of different algorithms. Hardware designs can be parallel and targets like FPGAs support high degree of parallelism whereas C/C++ code is inherently sequential. C/C++ code has no notion of time [95] and is hence considered “*untimed*” model of the algorithm whereas hardware design is generally timed and works on a clock (synchronous designs). Though there has been some amount of research on high level synthesis for asynchronous designs [62], [63], [64] it has not gained enough attention compared to high level synthesis for synchronous designs primarily because asynchronous circuit design in itself is not widely practiced because of area overhead [65], lack of compatibility with industrial electronic design automation tools [66], problems with verification etc [65].

As far as languages are concerned, SystemC [67] is closer to hardware description though at an abstraction level above Verilog and VHDL. It is essentially a set of C++ macros and classes and is now a standard system modeling language. While SystemC allows transaction level modeling (TLM) where a digital system is modeled in a way that details of communication among models are separated from the details of the implementation of functional units as well as communication architecture, a subset of it is also used for synthesis of hardware [68], [69]. It allows an entire system to be modeled including peripherals. Thereafter those that need to be implemented in hardware can be synthesized by refactoring the model code using the synthesizable subset. This allows for use of only one description for both TLM as well as synthesis and this is a major argument put forward by proponents of SystemC for high level synthesis. Also, SystemC[67] allows explicit parallelism which can be exploited during hardware synthesis. As is evident from these facts, typically only a hardware designer can use SystemC whereas an algorithm designer would prefer C or C++. Nevertheless, because SystemC based description still embodies digital design principles, it has less flexibility compared to C/C++ based description in architecture exploration for implementation/synthesis. Since the design and development

community remains divided over the choice of language as well as a universally agreed level of abstraction, all commercial HLS tools support all the three languages.

OpenCL [72] , [74] has also been proposed as a language for design specification when a system is to be built using multiple processors. It facilitates parallel programming for heterogeneous computing systems comprising multiple processing elements where an FPGA can also be seen as a processing element. Altera has adopted OpenCL [73] to facilitate off-loading of compute intensive functions to FPGA while a host processor executes the non-compute intensive part of an application. It also allows describing algorithms using OpenCL and then auto-generates the RTL description for physical synthesis. Similarly, CUDA [13] has also been used for exploiting explicit parallelism and mapping compute intensive code to FPGA [75]. Here CUDA code is annotated with FPGA implementation guidelines and then translated to C code which is input to AutoESL/Vivado-HLS[35].

2.4.2 Compilation of Design Specification

A suitable compiler is used for the chosen language for design specification. Compilation involves transforming an input specification model into a formal description. Compilation techniques like dead-code elimination, code motion, false data dependency elimination, constant folding, loop transformations, common sub expression elimination etc. are used to generate an optimized formal model based on these techniques [70]. The effect of different compiler optimization on high level synthesis has been studied in [76], [77].

Compilation leads to an intermediate directed graph model of formal representation called *Control and Data Flow Graph (CDFG)* [71]. The CDFG model of the application consists of basic blocks (which basically consist of a straight line sequence of statements) and branching edges incident on basic blocks. The branching edges capture the control dependencies. An example CDFG is shown in Fig. 2.3. A sequence of straight line statements constitute a basic block (BB). Thus, operations in a basic block are executed in order. Hence, a basic block can also be represented as a Directed Acyclic Graph (DAG) where nodes in the DAG represent operations as shown in Fig.

2.4. The different nodes denote same or different operations in an application as scheduled by the compiler. The presence of two edges exiting a basic block means that the control can transfer to either of the two downstream BBs based on the condition in the source BB. For instance, control can transfer from BB2 to BB3 or BB4 in Fig. 2.3 based on some condition in BB2. Compilers like Trimaran [78] try to extract parallelism from sequential C-code. Trident [79] is another such compiler based on the LLVM [80] framework. The degree to which these compilers can extract parallelism depends on their strengths as well as on the quality of C-code.

Fig. 2.3. A control and dataflow graph (CDFG)

Fig. 2.4. A dataflow graph (DFG) for a basic block

Automatic extraction of functional parallelism from ordinary programs has been investigated in [10] while pipeline parallelism extraction has been investigated in [11]. Some authors have proposed modifying the CDFG/DFG before hardware binding [81], [82]. In [81], the authors describe a technique using Taylor Expansion Diagrams (TED) [83], to optimize DFGs of arithmetic expressions for hardware synthesis. In [82], authors concentrate on optimizing DFGs with min-max, adding and relational operations by adding/deleting nodes. Pattern finding in CDFG has also been studied to identify oft repeated patterns and synthesize them as individual hardware blocks. In [84] where the authors propose an algorithm to find recurring patterns in an application through subgraph enumeration, pruning and matching techniques in compute intensive applications. These patterns are then mapped to FPGAs. In [85], the

authors have proposed a scheduling methodology based on behavioral templates which are essentially sub graphs. Here also, the templates are created by enumerating the CDFG of an application. Since CDFG is the intermediate representation of the program, it has all the details regarding the type of operations, data dependence and control flow relationships. This information is very useful as it can be processed, analyzed and modified as shown in previous work to generate a RTL description that shows improved quality of results.

For C-based languages, the major research areas have been focused on supporting different kinds of language constructs for synthesis. They are highlighted below:

2.4.2.1 Mapping C/C++ Language Constructs to Hardware

This group consists of work done on supporting different constructs in C-based languages for their synthesis. Constructs include arrays, pointers, loops and recursive functions.

A. Loop Optimization

Loops have gained considerable attention for high level synthesis because a lot of algorithms involve computational steps within a loop body. From hardware perspective, loop pipelining leads to increased throughput and therefore has been a major area of research. From a compiler perspective, many different kinds of loop optimizations have been studied that could be applied by a compiler during the compilation phase. Different kinds of loop optimization methods like loop unrolling, loop fusion, loop fission etc. are described in [120]. These methods are typically invoked by using compiler flags or by choosing an appropriate compilation optimization switch like -O1, -O2 and -O3 in LLVM etc. In commercial HLS tools like Vivado HLS [35], such optimization techniques need to be explicitly specified as a compilation directive or inserted at the appropriate place in the code. Nested loop pipelining using polyhedral model [123] has been studied in [121] for high level synthesis. Optimization of memory hierarchy allocation for both on-chip and off-chip memories along with loop

transformations for high level synthesis has been studied in [122]. The polyhedral model has itself been used for high level synthesis of loops [124].

B. Memory Synthesis

Memory constructs like arrays have been investigated in [125], [126]. In [125], the authors present a memory partitioning algorithm for multidimensional arrays during high level synthesis. An optimal solution based on Ehrhart points counting on an Ehrhart polynomial [127] is presented and a heuristic solution based on memory padding is also presented. One dimensional array partitioning for throughput and power reduction has been studied in [126]. Either row or column based partitioning based on footprint of data access has been studied in [128]. Combining scheduling with memory allocation and array mapping to memories has been studied in [129]. A methodology to implement memory accesses during high level synthesis of an accelerator for a processor-accelerator system under design constraints like the memory addresses where data are stored is presented in [130]. Synthesis of pointers as well as dynamic memory allocation construct like *malloc/free* (in C) using a hardware memory allocator has been studied in [131] while [132] is one of the earliest works to study synthesis of pointers in C. All commercial HLS tools support pointers for synthesis that can be analyzed at compile time but they do not support dynamic memory allocation. Pointers which can only be dereferenced at run time are not supported.

C. Recursive Functions

Synthesis of recursive functions has remained a challenge. In [133], the authors present a methodology for high level synthesis of recursive functions by representing programs as term-rewriting systems. Its scalability to medium and large size applications was not studied in that work. The work in [134] investigates the use of dynamic reconfiguration of FPGAs to support the synthesis of recursive functions. On the other hand, the work in [135] uses recursive hierarchical state machines to design a control unit that implements a

mechanism that permits the contents of an execution unit to be stored/restored between hierarchical calls/returns. Hardware synthesis of recursive functions expressed in certain languages that are examples of Reynolds's affine Syntactic Control of Interference [137], [138] is explored in [136]. However, C/C++/SystemC do not fall under the category of this type of language.

2.4.3 Allocation, Scheduling and Hardware Binding

Allocation involves allocating functional units (FUs), storage elements (like registers) and connectivity components (like buses or point to point connections). These elements are stored in a library [71]. The library also stores their area, delay and power characteristics as these characteristics are needed in the later stages of the synthesis process. Allocation ensures that there is at least one functional unit for each type of operation in the application and required storage and connectivity components are also available.

Scheduling involves scheduling the operations in the CDFG model into different clock cycles [71]. Since hardware works on a cycle accurate formalism, each operation in the CDFG is scheduled in a specific clock cycle. Depending on the degree of parallelism extracted earlier during the compilation step, a number of operations may be scheduled in the same clock cycle thus speeding up the performance of the design in hardware by exploiting parallelism [71]. Scheduling is generally done under design constraints like latency and timing and is an NP-complete problem [115] and hence heuristics based algorithms have been developed though integer linear programming (ILP) based algorithm has also been proposed [116] which by nature is unsuitable for large applications. Different scheduling algorithms have been developed in the literature like *As Soon As Possible (ASAP)*, *As Late As Possible (ALAP)*, *force-directed scheduling* [71], [86]. *Iterative modulo scheduling* [87] is another scheduling technique for software pipelining of loops that is also widely used. It is the default scheduling algorithm in *Trimaran* compiler [78]. A scheduling methodology based on behavioral templates has been proposed in [85]. Constraint programming [89] technique has been applied in [90] to solve the scheduling and resource assignment

problem. Scheduling operations in control flow intensive applications has been studied in [91], [92]. The most widely used scheduling algorithm that also found acceptance in modern commercial HLS tools is presented in [88]. It is based on a system of difference constraints which model timing relationships as well as data dependencies.

Hardware binding deals with binding operations in the CDFG to allocated functional units chosen from the library of RTL components [19], [71]. The functional units can be single cycle in nature meaning that their output is available in one clock cycle or they can be multi-cycle in nature meaning that their output is available after a certain number of clock cycles [71]. This information is crucial for scheduling also and it is here that interdependence between hardware binding and scheduling can be observed [19]. Hardware binding, in general, proceeds with a design objective of either reducing area, or delay or both. Hence resource sharing based hardware binding [93], [95] has been studied extensively. Since functional units are shared during resource sharing, multiplexers are required before them. Minimizing these interconnects leads to less delay as well as less interconnect area. Variables are also bound to storage elements like registers in this step. Since not all variables are "alive (i.e. being used)" at the same time, a life-time analysis of variables makes it possible to share registers among them for storage thus leading to a minimization of registers [98]. The existing research literature on hardware binding and scheduling can be grouped under the following broad categories.

A. *Area, Delay Minimization*

Area and delay reduction during hardware binding have traditionally been the objectives. Functional units (FUs) and registers are bound to operations and variables respectively in a way so as to minimize LUT consumption in FPGAs. The basic idea is to share FUs and registers among multiple operations and variables. Since FUs are shared, multiplexers (MUXes) are needed in front of them to steer relevant data to the FUs at the right time. The presence of MUXes leads to additional interconnects. The different hardware binding algorithms try to control the extent of FU sharing and thus control the number of interconnects as well. The

input to a hardware binding algorithm is a scheduled DFG as mentioned previously in Section 2.4.3.

An automated datapath synthesis procedure called "*Facet*" has been described in [93]. The hardware binding algorithm described in [93] is based on *clique-partitioning* technique borrowed from Graph theory. Mapping of storage elements (registers), operators and interconnection units is formulated as a clique-partitioning problem in [93]. In [94], the authors describe a set of heuristics which guide a global branch and bound hardware allocator. An area estimation heuristic is used as the cost function to be minimized. A design space bounding heuristic is also proposed to conduct a pruned design space exploration. One of the most commonly used and widely understood datapath synthesis method is based on weighted bipartite matching (WBM) presented in [95]. Again, variables and operations are represented in a graph and are matched to registers and FUs to minimize interconnects. A cost function is used which represents the possibility of a MUX being used for resource sharing. Network flow based methods are presented in [96], [97]. The hardware binding problem is presented as a minimum cost, maximum flow problem in these works and attempts are made to simultaneously bind FUs and registers. An integer linear programming (ILP) based formulation of the problem is also presented in [97]. While ILP solutions are understood to give optimal results, they are not suitable for large applications as ILP formulation and solution is very time consuming [97].

Methods that deal with only register binding or FU binding have also been proposed [98], [99]. They do not exploit the interdependency that can exist between FUs and registers. The work in [98] introduces the *Left Edge Algorithm* for register binding only. Simultaneous resource binding and interconnect optimization based on a distributed register-file micro-architecture (DRFM) is proposed in [100]. The authors propose using a distributed register file for each FU so that data can be accessed locally as well as there is reduction in MUXes and interconnects compared to distributed register architecture. However, they do not address explicitly the complex issues regarding reading from and writing to a

register file while operations are being shared. The work in [100] is an improvement by the same authors over their prior work in [101]. Compatibility path based (CPB) hardware binding is proposed in [102] where compatible operations (operations scheduled in different control steps or clock cycles) of same type are grouped together and assigned to one FU for that particular operation. A weight function is used to assign weights to edges in the compatibility graph and then longest path algorithm is used. A path compatibility graph is also defined to reduce interconnects. All of the previous methods try to reduce heuristically the total number of inputs to multiplexers. However, in [103], a SAT (stands for Boolean Satisfiability) based resource binding algorithm has been proposed which makes use of SAT formulas and a SAT solver. The drawback with this approach is that formulating SAT formulas and running a SAT solver is complex and can take a lot of time for large applications.

Since area-delay analysis plays a crucial role in hardware binding, techniques have also been developed to estimate area and delay for the entire design without doing physical synthesis. These methods can be incorporated during hardware binding as well as design space exploration (DSE). An area and delay estimator is presented in [104] which takes in high level signal processing applications described in MATLAB and performs automatic design space exploration (DSE). Area and delay estimation specifically for LUT based FPGAs has been proposed in [105]. However, they use logic netlist to estimate area and delay for the design which means that the estimate is available only after logic synthesis. The technique is also not general in nature and is focused on Xilinx XC4000 device only.

The work in [106] focuses on rapid estimation of control delay from high level specifications. A controller is treated as a Finite State Machine (FSM) and the number of states as well as the number of nodes in datapath DFG are used to estimate control delay. This is based on the understanding that a datapath may get invalidated if the controller, which is synthesized after the datapath, has a very high delay. Hence, an early estimation of the control delay is likely to prevent such a condition. Similar work on area and delay estimation for control

path has also been proposed in [108]. Macro models for high level area and power estimation have been presented in [107]. The models of the FUs are in the form of arithmetic functions with bit width and switching activity as parameters and hence there is no need to store area and power data in a table. The models are generated based on actual back-end physical synthesis of the FUs.

All these hardware binding algorithms assume “a one method fits all” approach because they do not distinguish between different types of FUs that can do the same job. For instance, a multiplier can be implemented in LUT as well as using DSP blocks. Thus, there is a lack of architecture-awareness in their approach to hardware binding. Similarly, there is a lack of consideration for IP-cores during hardware binding including but not limited to their different configuration options which could give different area-delay results.

B. *Process Variation Aware*

Timing variation aware binding is considered in [109], [110], [111], [112]. In [109] and [110], the authors take into account the delay probability density function (PDF) of FUs for scheduling and hardware binding. They argue that instead of using the worst case delay, as is generally the case in HLS, datapath performance can be improved by exploiting the timing variation due to process variation. They also argue that for FUs with multiple outputs, it is necessary to take into account the delay PDF of the FU because each output of the FU may have a different delay. This is a departure from most of the work on hardware binding where FUs with only a single output are considered. In [112] and [113], simulated annealing based timing variation aware iterative hardware binding algorithms have been proposed while [110] uses a branch and bound technique. Process variation aware resource sharing and binding in high level synthesis has also been studied in [114]. Awareness of process variation is more applicable to devices in advanced technology nodes [114]. However, it has not been investigated whether such an approach at the binding stage is necessary and if it could lead to

pessimistic designs given the fact that timing closure in the later part of the design exercise is supposed to lead to timing sign-off.

C. *Low Power Driven*

Low power design is becoming increasingly important in domains like handheld computing because the devices operate on batteries. On the one hand battery lifetimes are not increasing significantly and on the other consumers want a higher degree of performance and rich user experience. In [117], a low power high level synthesis framework called *LOPASS* has been presented. A simulated annealing engine carries out the tasks of resource selection, FU and register binding and scheduling. An enhanced weighted bipartite matching algorithm is used to share resources while reducing the total number of MUX ports and overall achieving a low power design. A high level power estimation for low power design space exploration targeting FPGAs has been presented in [118]. Interconnect aware low power high level synthesis has been presented in [119] and is more applicable to HLS for ASIC than FPGAs. This is because interconnect length cannot be deterministically determined in the latter where it is highly dependent on how modules have been placed on the FPGA during the placement phase of physical synthesis.

D. *Bitwidth Aware*

Bit width analysis or word length optimization is important for HLS because it affects the size of FUs. This in turn has an effect on the resource utilization as well as timing performance of a design. Bit width analysis is an NP-hard problem [139]. In real world designs, the bit width and the bit growth across the nodes of an application's CDFG cannot be neglected as they impact the area and the timing [140]. The work in [140] considers the problem of multiple bit width scheduling and binding. It also presents a lower bound estimation for the total bit width of FUs with various bit widths. The fact that variable precision (different bit widths) FUs are essential to reduce area and improve performance has also been acknowledged with the introduction of variable precision DSP block in Altera's

new 28nm Stratix V devices [141]. These new DSP blocks (Fig. 2.6 and 2.7) can be mapped to three 9x9, two 18x18 or a single 27x27 or 18x36 multiplier mode to efficiently support the wide variation in precision in DSP domain as shown in Fig. 2.5.

Fig. 2.5. Types of applications on the DSP precision spectrum [141]

Fig. 2.6: DSP block in 18-bit precision mode in Stratix V [141]

Fig. 2.7: DSP block in high-precision mode in Stratix V[141]

The work in [142] presents a general technique for automatic determination and optimization of word length (bit width) during floating point to fixed point conversion. It presents a metric that handles several error models like those proposed by IEEE for inverse discrete cosine transform (IDCT) [143], user specifications and hardware cost models. Application specific floating point to fixed point bit width analysis techniques have also been presented in [144], [145]. General techniques for bit width analysis are based on simulation based approaches which involve using an input data set [146], [147], [148]. The input data set is used to find out the maximum and the minimum values at each node in an application's DFG and using these values the operations (and hence FUs) at those nodes are sized. Thus, it is clear that variable precision nodes will be present in an application's DFG because of this analysis. The techniques cited in these papers differ in the extent to which they support error analysis. A very simple, general and effective bit width aware synthesis process for integer operations is described in [149]. It also presents a technique that makes use of simulation using an input dataset. This technique is incorporated in the HLS tool GAUT [34] and is thus different from other works which are generally done offline. A system called *Minibit+* is proposed in [150] to optimize the bit widths of fixed point and floating point designs. It uses a combination of affine and interval arithmetic to determine optimal bit widths under error constraints. The results cited however do not indicate how hardware binding was performed.

The work in [151] uses output accuracy as a cost measure during wordlength optimization and proposes a greedy search algorithm for word length selection in case of quantized inputs. The authors introduce a symbolic noise analysis method for HLS in [152] which is based on symbolic modeling of error bounds. Optimization of polynomial datapaths using Finite Ring Algebra [153] is presented in [154]. The paper discusses a HLS design flow using MATLAB. The main issue with bit width aware synthesis lies in incorporating a bit width analysis and optimization technique in HLS flow that is not time consuming, can work on a DFG model and gives accurate results. While many techniques exist for word

length optimization, given the tighter TTM windows, one needs to investigate which techniques to incorporate into an HLS flow.

E. *Design Space Exploration Focused*

Given the challenges involved in design and verification, a designer would typically want to have multiple implementations to choose provided they all meet the design constraints. This is called “Design Space Exploration” [177]. This involves exploring the design space with respect to different functional units to which operations could be bound, different ways in which memory blocks can be configured for the same operations, different ways of scheduling the operations etc. while meeting the design constraints all the while. An adaptive simulated annealer based design space exploration approach has been proposed in [178] while a machine learning based predictive method has been proposed in [179]. Design space exploration has also been explored for low power high level synthesis in [180]. These techniques generate a set of pareto-optimal designs without actual physical synthesis because that would be too time consuming. Another design space exploration approach [181] considered the effect of different options (called ‘knobs’ in the paper) available in synthesis tools and their effect on implementation and used a *component-knob setting trade-off library* to prune the design space while selecting components.

F. *Application Aware/Domain Specific*

Application aware synthesis research has largely concentrated around two ideas. One idea focuses on domain specific HLS language and flow. The most prominent work in this area has been done on the Streams-C high level language and its associated HLS flow for stream oriented computing [155]. The Streams-C compiler synthesizes hardware circuits for single as well as multiple FPGAs for applications characterized by high data rate flow of one or more data sources, small stream payload, compute intensive low precision fixed point operations on the data stream and access to small local memories. Such applications can be found in telecommunications and multimedia domains. The area and timing results

as reported are inferior to those using hand crafted design but there is a significant improvement in productivity when it comes to meeting the TTM window. A model based high level synthesis for image processing applications has been presented in [156].

The other idea focuses on identifying often repeated patterns in an application's DFG and application specific loops. This is claimed to be application aware because such patterns are in general specific to an application. The work presented in [157] focuses on the generation of application specific macros by doing logic synthesis on the fly. All possible macro subgraphs of a given size (number of operations in the subgraph) are identified from an application graph as constructed by GUB tool kit [158] and then corresponding nodes in the application are replaced by a *macro node* such that there is maximal gain in overall design latency. The work in [159] clusters operations in a CDFG into macros while satisfying logical and physical constraints using macro formulation algorithm. As a final step, the macros are relationally placed to minimize delay. Multi-function loop accelerators are synthesized in [160] where single function loops are combined using positional and ILP methods. Since loops are present in many image and signal processing applications, these methods are considered as application aware. In [84], graph matching based pattern recognition [161] techniques have been employed to enable pattern based synthesis for FPGA resource reduction.

Borrowing the idea of custom instructions from the domain of processor design and acceleration, [162], [163] present works where patterns in an application's DFG are enumerated based on a set of rules and then profitable ones are selected for mapping onto FPGAs as standalone accelerators or for datapath merging for the entire DFG. The work in [162] also claims to be architecture aware because it makes use of the fact that a k -input LUT can map a k -input function. While loop acceleration and custom instruction generation require a careful analysis of an application and are subject to compilation efficiency, recognition of coarse

granularity computational structures in the higher level description of an application could possibly lead to another level of efficiency in hardware binding.

G. Graph Partitioning for Run Time Reconfiguration

Dataflow graphs can grow very big for large applications. Therefore, there is a need to manage DFG complexity and size. Temporal partitioning of DFGs has been studied with a view on run time reconfiguration (RTR) [167], [168], [169]. In [167], the authors propose a *level based partitioning* and a *cluster based partitioning* algorithm. The DFG is scheduled using ASAP algorithm [40]. The ASAP levels are used to partition DFG nodes in *level based partitioning* and nodes are clustered using a common parent formulation in *cluster based partitioning* under area constraints of reconfigurable space. The partitions are then scheduled on to the reconfigurable space in order of their existence in the ASAP schedule. A network flow based multi-way task partitioning algorithm to minimize communication cost is presented in [168]. Here also, the objective of partitioning is dynamic reconfiguration. The authors propose a temporal partitioning algorithm and a port reallocation algorithm in order to reduce reconfiguration time in [167] by analyzing critical path and trying to maximize area utilization of the reconfigurable fabric.

An algorithm to temporally partition DFGs in order to map different partitions to different processing elements in a reconfigurable space by minimizing number of partitions, communication cost between partitions and maximizing operation parallelism is presented in [170]. A temporal partitioning scheme that takes into account sharing of FUs during partitioning for reconfiguration was proposed in [171]. Since a DFG is a graph, techniques from graph theory like vertex cover, graph coloring, vertex coloring etc can also be applied to partition them [172].

Temporal partitioning has also been studied for fitting very large designs into multiple-FPGAs. However, these work at the physical synthesis level where it is the gate level netlist compared to the high level abstracted dataflow graph in HLS. In [182], the authors have proposed an algorithm for circuit bi-partitioning focusing

on an implementation of time-multiplexed I/Os. Some facets of an improved version of Fiducia-Matheyses algorithm are exploited in a multi-way FPGA partitioning procedure called FART in [183]. An integrated partitioning and synthesis approach for multi-FPGA reconfigurable computers is proposed in [184].

It can be seen that while temporal partitioning has been studied with respect to run time reconfiguration and multi-FPGA designs, its utility (if any) has not been studied with respect to managing design complexity. Also, combining partitioning with area reduction has not been explored with respect to both managing design complexity as well as reduced utilization of resources.

2.5 Summary

In view of the presented literature survey, it is clear that there are many challenges in HLS. With the availability of heterogeneous resources like LUTs, BRAMs and DSP slices on FPGAs, it is important to consider their effective utilization as well as the tradeoffs that they offer. An application and architecture aware design methodology will help in exploiting application specific features and mapping them more effectively to these resources while ensuring the primary HLS objective of managing design complexity and improving productivity. This has also been highlighted in [240].

Exploiting application specific characteristics without too much code restructuring or writing C/C++ code in a very particular way is an important objective. This will ensure a smooth transition from higher level C/C++ description to hardware implementation as well as minimize designer intervention. Automatically extracted information about an application's characteristics can help in combining such characteristics with architecture specific hardware IP cores as well as on-chip resources like DSP blocks. This will bring HLS methodology on par with the methodology adopted by an expert hardware designer without the design time penalty incurred by the latter.

Graph partitioning techniques should be exploited to control DFG complexity for hardware binding. Existing hardware binding algorithms enforce a data initiation interval constraint on the datapath and are not suitable for very large DFGs. This needs to be investigated to ensure that unnecessarily large initiation interval constraints are not placed on the datapath by the hardware binding algorithms. Also, when user specified initiation interval constraints are present, the hardware binding algorithms should take that into account in addition to the area and delay constraints. Factoring in such system level constraints could help explore the space between an over-engineered design (deep pipelines, high resource utilization) and a minimal-area (less pipelines, heavy resource sharing,) but dysfunctional design. The design knowledge generated by an understanding of computation to resource mapping by exploiting application and architecture awareness could then be further refined for area and delay optimization by improved resource sharing focused hardware binding algorithms.

Chapter 3

Exploiting FPGA DSP Block for Architecture and Application Aware Synthesis

3.1 Introduction

FPGAs are now available with on-chip DSP blocks for high-performance multiplication and allied operations like multiplication followed by an addition/subtraction. For instance, multiplication followed by an addition is typically found in Finite Impulse Response (FIR) filters. The work presented in this chapter looks at mapping different kinds of multiplications and allied operations, which are automatically identified in an application, efficiently to on-chip DSP blocks based on a cost measure in order to improve performance as well as save FPGA logic for other usage.

Frequently executed code blocks in an application, that have already been identified through established methods like code profiling, can be further analyzed for multiplication and allied operations for mapping to available DSP blocks on the FPGA based on a priority ordering of these operations in order to maximize performance.

3.2 Contributions

This chapter presents an algorithm (A1) called “*Auto-identify*” which automatically identifies the different kinds of multiplication and allied operations in an application described in C/C++. Another algorithm (A2) then maps them to DSP blocks in a prioritized order based on a cost measure to maximize performance. This automatic mapping with the aim to maximize performance is crucial as this exercise pipelines the DSP blocks to maximize performance. This is what a hardware designer would

typically do instead of relying on inference by downstream physical synthesis tools which have limited ability to do pipelining. The algorithm A2 abstracts this hardware design view-point and automates it for HLS.

3.3 Automatic Identification of Multiplication & Allied Operations

In this section, the different kinds of multiplication and allied operations that could exist in an application are discussed. Also, the algorithm to identify these operations in an application without designer intervention is presented.

3.3.1 Types of Multiplication & Allied Operations in Applications

Multiplication on FPGAs is a costly operation in terms of area and requires careful timing considerations. Adding to this complexity is the fact that there are three major kinds of multiplication operations namely: 1) two variables multiplied together, 2) a variable multiplied by one constant (single constant multiplication or SCM) and 3) a variable multiplied by more than one constant (multiple constant multiplication or MCM). This chapter does not consider the case of two constants being multiplied together because in a HLS flow this multiplication will be replaced by the actual result of multiplication during the compilation phase. Each of these types has been studied in detail individually and various mapping strategies have been proposed for each of these by assuming *a-priori* information about the existence of such types in an application. Modern FPGA devices have dedicated on chip resources for multiplication for speeding up the multiplication process. Examples of such resource include DSP48 [193] and DSP48E [40] blocks in Xilinx Virtex-4 and Virtex-5/6/7 devices respectively. Similarly, the 25 x 18 DSP block on Altera FPGAs [42] serve the same purpose. With the presence of a large number of on chip multiplication resources like these, it is a challenge to map operations to them in a way that is most effective with respect to area and timing performance when the number of multiplication operations exceeds the number of resources and in the presence of different types of multiplication operations. Current high level synthesis flows identify multiplication operations by the presence of multiplication operator in the program. The

multiplication operator is “*” in C, C++, Verilog and VHDL. Every instance of “*” in the program is treated equally without any distinction as to whether a particular instance belongs to any of the three categories of multiplication operations mentioned earlier. In contrast, for hand-crafted digital designs, a designer is aware of the different kinds of multiplication operations present in the application and hence can manually choose a particular mapping strategy for each of them. However, this is a time consuming process and dependent on designer expertise.

Allied operations refer to: 1) multiplication followed by addition, 2) multiplication followed by subtraction and 3) multiply and accumulate. These operations are supported by the DSP blocks provided by all major FPGA vendors and each such operation can be executed on one single DSP block. These operations are typically found in implementations of digital filters like Finite Impulse Response (FIR) filter, polynomial expressions, matrix multiplication, molecular dynamics simulation [189] etc. Such operations can also form a chain like in FIR filter design, matrix multiplication etc. Equation 3.1 shows the calculation of an element of a 3 by 3 product matrix formed by multiplying two 3 by 3 matrices. The calculation is done using a chain of multiply and add operations. Mapping those chains to DSP blocks using internal cascaded connections can improve the timing performance of the chain. Therefore, chains of such operations are also automatically identified from the DFG of an application.

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \times \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} = \begin{bmatrix} aj + bm + cp & ak + bn + cq & al + bo + cr \\ dj + em + fp & dk + en + fq & dl + eo + fr \\ gj + hm + ip & gk + hn + iq & gl + ho + ir \end{bmatrix} \quad (3.1)$$

Multiplication, when implemented using LUTs in FPGA, consumes resources based on the size of operation: bigger multiplications consume more LUTs. Due to the associated logic depth with large multiplications, their timing performance is also poor. Therefore, larger multiplication operations are critical to the timing performance of a design whether they exist independently or as part of a chain.

3.3.2 Algorithm for Identification of Multiplication and Allied Operations

Since an application can be represented by its control and dataflow graph (CDFG) which is extracted from its C/C++ description, no assumption is made regarding prior information about the type of multiplication operations available in the application. It is also an error prone and time consuming exercise to manually analyze the application for the presence or absence of different types of multiplication operations. Hence, their automatic identification is important. In existing design methodologies, it is left to the designer to manually identify different categories of multiplication operations and select a hardware implementation strategy for them and implement that strategy in the design. While it is possible to do so at the RTL level, it is not always possible at C/C++ level because there can be instances where there is no appropriate C/C++ based description possible for a given hardware implementation strategy. Besides it goes against the spirit of HLS if a hardware structure is explicitly coded (where possible) in C/C++.

This entire manual process is also less amenable to optimization as it is extremely time consuming to generate the results of different implementation strategies and their combinations. Besides, it cannot be assumed that a designer will have all the *domain knowledge* to decide which implementation strategies and their combinations to choose as it is humanly impossible to do so in a large application. Automatic identification of chains of allied operations helps in ensuring that they can be mapped to DSP blocks using internal cascaded signals. This is achieved in the work presented in this chapter by instantiating DSP blocks with cascaded connections during RTL code generation phase of HLS. If these chains are not identified and RTL code is not generated appropriately, leaving such chains to be handled by inference (as is the case with existing HLS methodologies) by downstream physical synthesis cannot ensure such a mapping. Automatic identification of multiplication and allied operations existing in an application enables in applying a joint implementation strategy to three categories and allied operations with respect to mapping them to DSP blocks on FPGAs for better area and timing performance.

Fig. 3.1. (a) Chain of multiply-add, b) Chain of multiply-subtract, c) Chain of multiply with a mix of add and subtract

Auto-Identify Algorithm for Identification of Multiplication and Allied Operations

Input: A wordlength optimized dataflow graph (DFG)

Output: Consolidated list of multiplication operations in three categories

1. Perform *Breadth First Search* (BFS) on DFG
2. At each level of DFG
3. **If** a constant value node is found
4. Perform a *Depth First Search* (DFS) with constant value node as root node to find its successor node
5. **If** the second input to successor node is labeled a variable, store the variable identity and the constant in a hash table: H(MUL)
6. **EndIf**
7. **EndIf**
8. **End BFS**
9. Identity (ID) = 0
10. **For** each multiplication node
11. Check if successor node, *SN* is addition or subtraction else step 9
12. Check if successor node *SNN* of *SN* is addition or subtraction
13. **If** (11) is **TRUE**
14. Check if second operand source of *SNN* is a MUL node
15. **If** (13) is **TRUE**
16. Collect all nodes in 9, 10, 11 and 13 as a partial graph
17. $SN = SNN$
18. Repeat 11 to 16 until (11) or (13) returns **FALSE**
19. Chain Detected = Yes
20. Length of chain = Number of ADD/SUB nodes
21. Chain ID = ID+1
22. **EndFor**
23. Remove from H(MUL) all MUL nodes collected in step 15.

Fig. 3.2. Auto-Identify algorithm for identification of multiplication and allied operations

The joint implementation strategy is discussed in section 3.4. The automatic identification algorithm, *Auto-Identify*, for multiplication categories and allied operations is given in Fig. 3.2. The patterns related to chains of allied operations shown in Fig. 3.1 are automatically identified. It should be noted here that the different multiplication operations shown in Fig. 3.1 could be scheduled in different c-steps and not necessarily in the same c-step though they are shown like that in Fig. 3.1 for the sake of clarity. These patterns show that an addition or a subtraction operation should be immediately followed in the next c-step by another addition or subtraction operation which will take another input as the result of some other multiplication operation.

The breadth first search (BFS) returns constant value nodes at each level of the DFG (line 1). DFS search (lines 4-5) at each of these nodes then returns their target multiplication nodes (if any) with other input to such nodes being variables. These variables and the corresponding constants are then stored in a hash table: $H(MUL)$ with variable identities as hash keys (line 5). Clearly, there can be more than one constant associated with a variable in the hash table. This way the hash table stores all instances of single constant multiplication as well as multiple constant multiplication. Hash table is a useful data structure to query the different constants that a variable is multiplied with because the variable name can be used as the key. Removing all such multiplication nodes from the DFG would leave us with those multiplication nodes where both the operands are variables. In this way all the three different kinds of multiplication operations are identified automatically.

It should be noted here that the DFG is obtained from the internal representation that a compiler generates for a C-based description as a result of compilation. Hence, the effect of optimizations like replacing multiplication by a multiplier which is a power of two (an example of integer multiplication) by left shifting the multiplicand variable have already been considered. For instance, multiplication by 4 is equivalent to left shifting the multiplicand by 2 bits. Therefore, *auto-identify* algorithm is applied to a DFG which has already undergone such transformations.

For the MCM case, the DFG might have undergone a transformation. Such a transformation could have been brought about by the compiler or it could be another post processing step on the DFG. Such transformations could be related to merging of operations or reducing operations to a series of add and shift with adder for addition operation being shared etc. For the work presented in this chapter, a DFG that has already undergone such transformations is considered. However, it is not necessary that a DFG *should* have undergone such transformations. Thus, the work presented in this chapter is independent of such transformations and is to be considered as the next processing step if such transformations have taken place.

The DFG is also traversed in lines 10-22 to locate the patterns identified in Fig. 3.1 (a), (b) and (c). Once these patterns have been identified, multiplication nodes belonging to them are removed from the hash table H(MUL) to nullify duplicate identification of such nodes. The length of the identified chains is also calculated (line 20) and it equals the number of ADD or SUB nodes.

3.4 Mapping Multiplication and Allied Operations to DSP Blocks

Multiplication has been a widely researched topic. In this section, an overview of work relevant to mapping of the different kinds of multiplication operations identified in Section 3.3 is presented. In [198], the authors have investigated the single constant multiplication (SCM) problem with respect to implementing such operations in FPGA look up tables (LUTs). More work on LUT based SCM mapping has been carried out in [199], [200] etc. Their efforts have focused on replacing a multiplication operation by a series of shift and add or subtract because multiplication is a costly operation in hardware compared to addition and subtraction whereas shift operation requires only wires as resources and no other computational resource. Multiple Constant Multiplication (MCM), where a variable is multiplied by more than one constant, has been studied in [201], [202]. Again, the effort is directed at replacing multiplication by a series of shift and add or subtract. Besides, these two works have also focused on time multiplexing the operations so that resources can be shared within the multiplier block resulting in overall reduced resource utilization. In [203], the authors have

focused on using carry save arithmetic for time multiplexed MCM. Reconfigurable multiplier blocks called ReMB for time multiplexed MCM have been studied in [204] and [205] where the authors present the fundamentals in [204] and the algorithm itself in [205]. Mapping to DSP blocks on Xilinx FPGAs have been investigated in [206] and on both Xilinx and Altera DSP blocks in [207] with the aim to reduce the number of DSP blocks needed when the basic multiplication would involve more than one DSP resource. Their work has relied on the Karatsuba-Ofman algorithm [208] for large multiplication. However, they have restricted the size of operands for multiplication to a maximum of 68 x 65 bits. The FPGA vendor Altera also has a design guideline on implementing multipliers in FPGAs [209]. The work that comes closest to the work presented here is [210] where the authors have worked on automatic exploration of arithmetic-circuit architectures. However, their work uses the Boolean functions to represent the arithmetic function. Common subexpression elimination is applied on the Boolean network to select a set of expressions for implementation. However, they do not investigate the role of existing architectural advancements (*the body of knowledge*) in selecting the set of expressions and hence is very different from the work presented here in this sense. Their method does explore a restricted set of architectures but it is unclear from the paper how it is done. Partitioning of multiplication operations into smaller multiplications and additions has also been studied in [211] as a pre-synthesis optimization step in high level synthesis with a view to decreasing the circuit delay. However, it does not make use of Karatsuba-Ofman algorithm for partitioning the multiplication operations.

3.4.1 DSP Block Architecture and Functionality

All major FPGA vendors i.e. Xilinx, Altera, Actel etc. provide on-chip DSP blocks on their FPGA devices. This chapter presents work that is applicable to FPGA devices from any vendor and is therefore not restricted to any particular vendor though for the sake of discussion Xilinx FPGAs are considered. These DSP blocks are “hard” blocks and are implemented as ASIC-like blocks on the FPGA. They are distinct from the configurable fabric comprising LUTs, registers, multiplexers etc. All registers and

multiplexers within a DSP block are also part of this hard block and are distinct from their counterparts in the soft configurable space of the FPGA. On Xilinx devices, DSP blocks are referred to as ‘DSP48’, ‘DSP48E’ and ‘DSP48E1’ depending on the device family. On Altera devices, they are referred to simply as ‘DSP blocks’ while on Actel devices they are referred to as ‘mathblocks’. Functionally they are all the same with minor variations and enhancements in architecture in newer device families. They were all introduced to provide high performance on-chip resources for multiplication, multiply and accumulate, multiply and add, and multiply and subtract. Fig. 3.3 shows the DSP48 architecture in Xilinx Virtex-4 [193], Fig. 3.4 shows the DSP48E1 architecture in Xilinx Virtex-5 [41] and Fig. 3.5 shows the DSP48E1 architecture in Xilinx Virtex-6 [194]. Their details are easily available in the user guides [193], [41], [194].

These DSP blocks have configurable pipeline registers: at the input A (2 registers), B (2 registers), a register for multiplier output and another register for output of ALU or Add/Subtract unit as can be seen in Figs. 3.3, 3.4 and 3.5. Ensuring that all these registers are made use of during hardware binding leads to maximum operating frequencies of the DSP blocks [193],[41],[194] reaching operating frequencies of more than 450 MHz. Therefore, if fully pipelined DSP blocks are used to implement multiplication and allied operations, the operating frequency of a design increases compared to implementing those operations in FPGA fabric i.e. LUTs. Though this increase in operating frequency comes at the cost of increased latency, it is essential when a design is expected to operate at higher clock frequencies. In the work presented in this chapter, DSP blocks are fully pipelined during the hardware binding stage to achieve maximum increase in operating frequency compared to implementation in LUTs of multiplication and allied operations. As can be seen in Figs. 3.3, 3.4 and 3.5, each DSP block has a multiplier and an adder/subtractor block. The multiplier is 18 bits wide in Virtex-4 while it supports 25 bit by 18 bit multiplication in Virtex-5 and Virtex-6. The adder/subtractor block is enhanced with a logic unit in Virtex-5 and Virtex-6.

Fig. 3.3. DSP48 architecture in Xilinx Virtex-4 [193]

Fig. 3.4. DSP48E1 architecture in Xilinx Virtex-5 [41]

Fig. 3.5. DSP48E1 architecture in Xilinx Virtex-6 [194]

Figures 3.6 (a), (b) and (c) show how the chain of allied operations is mapped to DSP slices.

Fig. 3.6. Mapping of operations in a chain of allied operations to DSP blocks

Each curve represents a DSP block and the green arrows indicate the internal cascaded routing from one DSP block to another in a column of DSP blocks on a FPGA. This leads to greater performance as such signals do not come out to FPGA fabric which would result in performance degradation. The largest estimated LUT utilization of any

multiplication operation, $E_{max}(LUT)$, in a chain is taken as a measure of its criticality: chains with larger values of $E_{max}(LUT)$ are more critical from timing perspective.

3.4.2 LUT Utilization Model for Multiplication Operations

The binary multiplication of two binary numbers ‘a’ and ‘b’, each 3 bits wide can be represented as shown in Fig. 3.7.

		a_2	a_1	a_0	
	$\times$	b_2	b_1	b_0	
		p_{02}	p_{01}	p_{00}	
		p_{12}	p_{11}	p_{10}	
	p_{22}	p_{21}	p_{20}		
PB_5	PB_4	PB_3	PB_2	PB_1	PB_0

Fig. 3.7. A 3-bit by 3-bit binary multiplication

where b_i is the i^{th} bit of binary number ‘b’, a_j is the j^{th} bit of binary number ‘a’ and ‘ $\times$ ’ is Boolean multiplication. As can be seen, there is a circuit size C (number of gates) and a circuit depth D (maximum distance from an input to output) associated with the Boolean circuit implementing this multiplication. Clearly, the greater the number of bits in the operands, the bigger is the circuit size and the higher is the depth with the latter equal to the number of rows of partial products or the number of rows of partial products plus one when there is a carry in from some column to another column. When one of the two operands is a constant, all the partial products are reduced to constants and only the logical summation remains. This reduces the circuit size. Therefore, LUT utilization and timing performance depend on C and D .

For the multiplication in Fig. 3.7, the partial products ‘ p_{ij} ’ are given by equation (3.2)

$$p_{ij} = b_i \times a_j \quad \text{where } i, j \in [0,2] \quad (3.2)$$

The product bits ‘ PB_i ’ are given by the sum of the partial products in the respective columns and can be calculated using equation (3.3).

$$PB_i = C_{i-1} + \sum p_i \quad \text{where } i \in [0,5] \text{ and } C_{i-1} = 0 \text{ for } i = 0 \quad (3.3)$$

where p_i is the partial product belonging to i^{th} column, C_{i-1} being the carry out from the summation of partial products in $(i-1)^{\text{th}}$ column with its value being 0 for $i = 0$. Here summation refers to Boolean summation performed by an adder. Note that a m -bit by n -bit multiplication will result in a total of $(m + n)$ product bits and $(m + n - 1)$ columns of partial products with the carry out from $(m + n - 1)^{\text{th}}$ column forming the $(m + n)^{\text{th}}$ product bit.

The binary multiplication method in Fig. 3.7 is adopted to model the LUT resource utilization of different kinds of multiplication operations. Boolean multiplication is performed by an AND gate while Boolean summation is performed by an adder circuit. Therefore, the number of partial products affects the number of required AND gates and the number of summations affects the number of required adders. The subsections ahead model the different kinds of multiplication operations to give an estimate of number of required AND operations and adders. Ripple carry adder is assumed for modeling the number of full adders.

A. Variable multiplied with another variable

An example of this kind of multiplication is variable V1[16:0] multiplied with another variable V2[17:0] resulting in product P1[34:0]. Since the multiplication operator when mapped to the basic mapping element in a FPGA i.e. LUTs involves decomposing the multiplication operation into logical AND operations and use of adders, the utilization increases when the number of AND operations and the number of adders is high. This means that one can order two or more such instances of multiplication based on the estimated LUT utilization. As can be seen in Fig. 3.8, 8 different instances of multiplication where the product bit width remains constant (16 bits wide) result in different LUT utilization values and different clock periods. This is due to the effect of number of logical AND operations and number of full adders (FAs) for summation.

For a m -bit variable M multiplied with a n -bit variable N, the number of required AND operations i.e. $Count(AND)$ is given by equation (3.4) and the number of required FAs i.e. $Count(FA)$ is given by equation (3.5). Some of these FAs are

actually half adders (HA) because their carry ins are tied to logical ‘0’. Hence, no distinction is made between complete FAs and those used as HAs for the purpose of adder resource estimation.

$$Count(AND) = m \times n \quad (3.4)$$

$$Count(FA) = \{[(m + n - 1) - ((\min(m, n) - 1) \times 2)] \times \min(m, n)\} +$$

$$\sum_{q=1}^{q=\min(m,n)-1} q + \sum_{t=1, t \neq 2}^{t=\min(m,n)-1} t \quad (3.5)$$

Fig. 3.8. Actual LUT utilization corresponding to 8 different multiplications with same product bit-width (16)

In Figure 3.8, M1(1,15) means multiplier design 1 with operand bit widths 1 and 15 resulting in a product bit width of 16 (=1+15). A 1-bit logical AND operation requires ‘a’ LUTs while a FA requires ‘b’ LUTs. For Xilinx FPGAs, $a = 1$ and $b = 1$. Hence, the total number of LUTs estimated, $Count(LUT)$ to be required is given by equation (3.6).

$$Count(LUT) = a \times Count(AND) + b \times Count(FA) \quad (3.6)$$

This estimated value of *Count (LUT)* will always be higher than the actual number of LUTs used i.e. *Actual (LUT)* because this estimate does not take into account packing of logic functions at the Boolean logic level where both logical AND operation and FA operations will be considered. The difference between *Count (LUT)* and *Actual (LUT)* is of no significance in the present work because *Count (LUT)* will be used to order the multiplication operations and it is evident from Table 3.1 that the inequality in (3.7) holds.

$$\begin{aligned} & \text{If } Count(LUT, M_1) \geq Count(LUT, M_2), \\ & \text{then } Actual\ LUT(M_1) \geq Actual\ LUT(M_2) \quad \forall M_i \end{aligned} \quad (3.7)$$

where M_i are different multiplication instances and M_1 and M_2 are any two cases out of M_i .

Table 3.1: Variation of *Actual(LUT)*, *Count(LUT)* and DSP Implementation

V1	V2	<i>Actual(LUT)</i>	<i>Count (LUT)</i> (eq.2)	CLK(ns) (LUT Imp)	DSP Imp.
16	8	128	254	6.307	1
24	8	184	392	6.883	2
32	8	240	510	7.734	2
32	32	1113	2750	12.511	4

Table 3.1 shows the actual LUT utilization in ascending order with the maximum LUT utilization being for V1[31:0] multiplied with V2[31:0] (device: XC4VLX100-10FF1513). Column *DSP Imp.* shows the number of DSP blocks required if these multiplications were to be mapped to them and not to LUTs. Thus, when compared with the number of DSP blocks used, we can say that the “*Equivalent LUT Cost*” i.e. *Count (LUT)* of mapping a multiplication operation of this type is the number of LUTs used when the multiplication operator “*” is used and mapped to LUTs. Since, the DSP blocks are very fast with maximum clock frequency being 450 MHz for DSP 48 and 550 MHz for DSP48E, mapping these multiplication operations to the DSP blocks gives better timing performance compared to mapping them to LUTs which is evident from the high clock periods in Table 3.1 for mapping to LUT.

B. Multiple Constant Multiplication (MCM)

There can be multiple instances of MCM in an application. For instance, a variable $V1[7:0]$ can be multiplied with $\{33, 117, 221, 45, 98, 171\}$ while another variable $V2[7:0]$ can be multiplied with $\{3, 5, 6, 7, 9, 10, 11\}$. If there is limited availability of DSP blocks, then those sets of MCM operations should be mapped to DSP blocks which would benefit most in terms of performance. Fig. 3.9 shows the LUT utilization result of multiplying an 8-bit variable $A[7:0]$ with seven different 8 bit constants. Fig. 3.10 shows the clock period (ns) for the same set of designs. These figures clearly demonstrate that the actual value of a constant affects the LUT utilization as well as the minimum clock period (maximum clock frequency). This is in contrast to DSP blocks where the size or the actual value of operands does not affect the maximum clock frequency as long as the size of operands can be accommodated within the maximum input size available for one DSP block.

Fig. 3.9. Actual LUT utilization for $A[7:0]$ x 8-bit constant

Fig. 3.10. CLK period (ns) for $A[7:0]$ x 8-bit constant

It can be clearly seen from Fig. 3.9 that a variable (A, 8 bits wide) multiplied by another variable (B, 8 bits wide) consumes more LUT resources compared to A being multiplied by a constant with the same bit width as variable B. Similarly, Fig. 3.10 shows that a variable (A, 8 bits wide) multiplied by another variable (B, 8 bits wide) suffers from higher clock period compared to variable A multiplied by a constant with the same bit width as B in its binary representation. It can be seen easily seen from Figs. 3.9 and 3.10 that the value of the constant has no direct relationship with LUT utilization i.e. LUT utilization does not always increase with an increase in the value of the constant or vice-versa. The “*Equivalent LUT cost*” for each multiplication operation in a set of MCM is calculated in order to make comparison between sets of MCM as well as to compare with variable multiplied by variable. The formulas required to compute this cost for a variable multiplied by a constant are presented in section D.

C. *Single Constant Multiplication (SCM)*

SCM can be considered as a special case of MCM where a given variable is multiplied by just one constant. Therefore all the analytical arguments presented by way of Figs. 3.9 and 3.10 are applicable to SCM also.

D. *Analysis of Multiplication with a Constant*

Symmetric Multiplication: For symmetric multiplication, the number of partial product terms $PP(C_i)$ in each column C_i can be easily derived from the set of equations in (3.8). For a m -bit by m -bit multiplication, which will have $2m$ product bits and $2m-1$ columns:

$$\begin{aligned} PP(C_i) &= i + 1 \text{ for } i \in [0, m - 1], \text{ and} \\ PP(C_i) &= C_{i-1} - 1 \text{ for } i \in [m, 2m - 2] \end{aligned} \quad (3.8)$$

Each column C_i up to $i = m-1$ corresponds to the bit location b_i in the binary representation of the constant. Also, the number of rows of partial products equals the number of bits in the binary representation of the constant. Therefore if any bit $b_i = 0$, then all partial products in row R_i are zeros. The number of terms in each row R_i is

equal m . Table 3.2 shows which rows contribute partial products to which columns for a 4-bit variable multiplied by a 4-bit constant. The contributing rows for different columns for $m * m$ multiplication are given in Table 3.3. Thus, the number of FAs can be calculated based on which rows are all zeroes i.e. their partial products are all zeroes given the fact that row R_i corresponds to bit b_i in the binary representation of the constant. All AND operations are just wires when a variable is multiplied with a constant and hence they are ignored. For instance, for $A[3:0]$ multiplied by $4'b1010$, Table 3.4 shows calculation of the number of FA.

Table 3.2. Contributing rows for different columns for 4 bit variable multiplied by 4 bit constant

Column	Contributing Rows (each row contributes 1 partial product element to the column)
C0	R0
C1	R0, R1
C2	R0, R1, R2
C3	R0, R1, R2, R3
C4	R3, R2, R1
C5	R3, R2
C6	R3

Table 3.3. Contributing rows for different columns for $m * m$ multiplication

Column	Condition	Contributing Rows (each row contributes 1 partial product element to the column)
C_i	$i \in [0, m - 1]$	R_0 to R_i
C_i	$i \in [m, 2m - 2]$	R_{m-1} to $R_{m-PP(Ci)}$

Table 3.4. Example No. of FA for $m = 4$ (constant=4'b1010)

Column	Contributing Rows	No. of non-zero partial product	No. of FAs as a result of column addition
C0	R0	0	0
C1	R0, R1	0,1 = 1	0
C2	R0, R1, R2	0,1,0 = 1	0
C3	R0, R1, R2, R3	0,1,0,1 = 2	$1 + C_{out} = 2$
C4	R3, R2, R1	1,0,1 = 2	$1 + C_{out} = 2$
C5	R3, R2	1,0 = 1	C_{out}
C6	R3	1	0

A C_{out} in a column results in a FA in the next column. Hence, for the example in Table 3.4, the total number of FAs is 5.

Asymmetric Multiplication: For asymmetric m -bit by n -bit multiplication, there will be $(m + n)$ product bits and $(m + n - 1)$ columns with $(m+n)^{th}$ column being just the

$(m+n)^{\text{th}}$ product bit with a value either 0 or 1. The number of columns, NC_{equal} , with equal number of partial products is given by equation (3.9).

$$NC_{\text{equal}} = \max(m, n) - \min(m, n) + 1 \quad (3.9)$$

The beginning column number for such a column is given by equation (3.10).

$$C_{\text{begin}} = \min(m, n) - 1 \quad (3.10)$$

The end column number for such a column is given by equation (3.11).

$$C_{\text{end}} = C_{\text{begin}} + NC_{\text{equal}} - 1 \quad (3.11)$$

The number of partial product terms $PP(C_i)$ in each column C_i can be easily derived from the set of equations in (3.12).

$$\begin{aligned} PP(C_i) &= i + 1 \text{ for } i \in [0, C_{\text{begin}}), \text{ and} \\ PP(C_i) &= C_{\text{end}} - i + 1 \text{ for } i \in [C_{\text{end}}, m + n - 1] \end{aligned} \quad (3.12)$$

The rows which contribute to a particular column can be obtained from the expressions in Table 3.5 for $m > n$ where m is the number of bits in variable and n is the number of bits in the constant.

Table 3.5. Contributing rows for $m > n$

Column	Condition	Contributing Rows (each row contributes 1 partial product element to the column)
C_i	$i \in [0, C_{\text{begin}})$	R_0 to R_i
C_i	$i \in [C_{\text{begin}}, C_{\text{end}}]$	R_0 to R_{m-i}
C_i	$i \in [C_{\text{end}}+1, m + n - 1]$	R_{n-i} to $R_{(n-1)-PP(C_i)+1}$

Table 3.6 shows which rows contribute partial products to which columns for a 5-bit variable multiplied by a 3-bit constant. The rows which contribute to a particular column can be obtained from the expressions in Table 3.7 for $m < n$ where m is the number of bits in variable and n is the number of bits in the constant. Table 3.8 shows which rows contribute partial products to which columns for 3-bit variable multiplied by a 5-bit constant.

Table 3.6. Contributing rows for $m = 5$ and $n = 3$

Column	Contributing Rows (each row contributes 1 partial product element to the column)
C0	R0
C1	R0, R1
C2	R0, R1, R2
C3	R0, R1, R2
C4	R0, R1, R2
C5	R2, R1
C6	R2

Table 3.7. Contributing rows for $m < n$

Column	Condition	Contributing Rows (each row contributes 1 partial product element to the column)
C_i	$i \in [0, C_{begin})$	R_0 to R_i
C_i	$i \in [C_{begin}, C_{end}]$	$R_{i-(m-1)}$ to $R_{i-(m-1)+NC_{equal}-1}$
C_i	$i \in [C_{end+1}, m+n-2]$	R_{n-1} to $R_{(n-1)-PP(C_i)+1}$

Table 3.8. Contributing rows for $m = 3$ and $n = 5$

Column	Contributing Rows (each row contributes 1 partial product element to the column)
C0	R0
C1	R0, R1
C2	R0, R1, R2
C3	R1, R2, R3
C4	R2, R3, R4
C5	R4, R3
C6	R4

Each column C_i where $i \in [1, m+n-3]$ contributes a maximum of $PP(C_i)$ number of FAs which includes the FA required to add the carry out from C_i to C_{i+1} . No FA is contributed by C_0 and C_{m+n-2} under this analysis. Let $R_k = 0$ mean that all elements (partial products) in row k are zero and it will correspond to bit b_k in the binary representation of the constant. Therefore, the number of FAs contributed by a column will decrease by as many rows that are zeros. The contribution of the adder for carry out from column C_i will also vanish if all rows or all rows except the one that contribute to C_i are zeros.

E. Analysis of Chain of Allied Operations

For each chain of allied operations identified in the DFG, the multiplication operation

with the highest estimated LUT consumption is taken into consideration by the selection algorithm (section 3.4.3). If a multiplication operation belonging to a chain of allied operations is selected for mapping to DSP block, then the entire chain is mapped to a series of DSP blocks as shown in Fig. 3.6 as this makes use of fast internal routing within the DSP column and splitting a chain between DSP blocks and FPGA general fabric will result in degraded performance.

F. Why the proposed estimation models are better?

The expected LUT utilization for multiplication of two variables of bit widths m and n can be also approximated as $m \times n$ when calculated to full precision i.e. $(m + n)$ bits in product. However, this leads to inconsistent estimation for symmetric and asymmetric multiplication. For instance, from Table 3.1, 16-bit multiplied by 8-bit uses 128 LUTs (actual) which equals $m \times n = 128$. However, 24-bit multiplied by 8-bit uses 184 LUTs (actual) while $m \times n = 192$. In this case, there is over estimation. Similarly, 32-bit multiplied by 32-bit uses 1113 LUTs (actual) while $m \times n = 1124$ resulting in over estimation. Thus, it can be seen that there is no consistency in result when estimated using the approximate formula $m \times n$. Also, the biggest drawback with this estimation method is that it disregards the effect of one of the multiplicands being a constant. It also does not work when the result of a multiplication is not calculated to full precision. Therefore, using this formula, an m -bit variable multiplied by a n -bit variable would be estimated to use the same number of LUTs as a m -bit variable multiplied by a n -bit constant. On the other hand, the proposed model also works when the product is not calculated to full precision. In this case, $(m + n)$ take the value of the number of bits in the product instead of their full precision value and all relevant columns, partial products and full adders can be easily identified from the formulas in the previously mentioned tables.

Quadratic curve fitting based estimation proposed in [191], [192] also suffer from the drawback that they cannot take into account the effect of one of the multiplicands being a constant. Besides, curve fitting is based on using physical synthesis data for a set of multiplication operations thus exposing the estimation method to variance in

physical synthesis approaches. The resource utilization model proposed in this chapter is independent of physical synthesis approach and relies on the basic multiplication method to provide consistent approximation results. It should be noted and emphasized here that the approximation method proposed in this chapter is meant for the purpose of ordering different multiplication operations in an application. It is not its objective to give an estimate which is close to the actual synthesis result as such an objective is not necessary when ordering multiplication operations. The proposed resource utilization model also enables to estimate LUT utilization for any LUT based device either existing or the ones to appear in the future as it is independent of both device architecture and synthesis algorithms and is based entirely on mathematical reasoning.

Table 3.9 shows the variation in LUT consumption for an 8 bit variable multiplied by different 9 bit constants. It can be seen that the number of LUTs actually consumed is proportional to the number of 1's in the binary representation of the constant.

Table 3.9. Variation in LUT consumption for different 9 bit constants with same number of 1's in binary representation

Constant	Binary	No. of 1's	LUT	CLK(ns)	#FAs
260	100000100	2	9	2.077	9
259	100000011	3	16	2.509	22
261	100000101	3	15	2.479	21
383	101111111	8	63,47	5.490	62
447	110111111	8	63,48	5.638	62
479	111011111	8	62,50	5.781	62
495	111101111	8	64,51	5.855	64
503	111110111	8	61,50	5.556	63
510	111111110	8	56,45	5.687	62
511	111111111	9	60,49	5.501	70

Therefore, LUT consumption could also have been estimated by assuming a proportionality relationship between LUT consumption and number of 1's in the binary representation of a number. However, when the number of 1's is the same, there still is variation in LUT consumption. This information cannot be captured by the proportionality logic but is captured by the number of FAs (the proposed model in this chapter) though with some deviations as evident in row for '511'. Thus, the count of FA and AND operations (no AND gates for multiplication with a constant) is a more

reliable measure to make comparisons between the three classes of multiplication operations.

3.4.3 Selection Algorithm for Mapping of Multiplication & Allied Operations

This section presents the Joint (J) Comparison (C) and Selection (S) algorithm (Fig. 3.11) for joint comparison between the three classes of multiplication operations leading to the selection of operations to be prioritized for mapping to DSP blocks on FPGA.

Fig. 3.11. Joint Comparison and Selection (JCS) algorithm

In the proposed algorithm, the number of FAs, AND gates and LUTs are calculated first (lines 1-4) for SCM, MCM and other multiplication operations. Thereafter for the identified MUL-(ADD/SUBTRACT) chains, the maximum LUT cost is estimated (lines 5-5.1). Once these are done, multiplication operations common to H(MULT) hash table and MULT-(ADD/SUBTRACT) chains are removed from H(MULT). Thus, H(MULT) only has multiplication operations which are not part of MULT-(ADD/SUBTRACT) chains (line 6). Thereafter multiplication operations are arranged

in descending order of estimated LUT consumption to form list D (line 7). Operations beginning from the top in list D are mapped to DSP blocks. Since list D may also contain a multiplication operation belonging to a MULT-(ADD/SUBTRACT) chain, all operations in such a chain are mapped to DSP blocks as explained in Section 3.4.1. Once all DSP blocks are exhausted this way, remaining multiplication operations and MULT-(ADD/SUBTRACT) chains are mapped to LUTs. During the automatic RTL datapath generation phase, DSP blocks are instantiated automatically in the RTL description and internal cascaded connections between the DSP blocks are utilized where necessary (as in Fig. 3.6) to ensure that such routing does not use low speed FPGA fabric. The internal cascaded connections being made from high speed routing lead to increased operating frequency.

In order to maintain the generality of the proposed design methodology, it has been assumed that bit-width optimization techniques (Sub-section D, Section 2.4.3) have already been applied to the DFG before the algorithms presented in this chapter process the DFG. However, it should be pointed out that there are specific techniques [252], [253], [254] related to adder width optimization without compromising on operand values which can help in reducing the number of full adders employed for multiplication.

3.5 Experiments and Results

In this section, case studies are presented to demonstrate the better quality of results obtained using the methodology presented in this chapter. The proposed methodology was implemented within a C-to-RTL framework and experiments were carried out using Xilinx XC4VLX100 device, Vivado HLS 2012.1 and Xilinx ISE 14.2. The experiments were carried out for two different scenarios: 1) when the number of multiplication operations (M) in an application exceeds the number of available DSP resources (D) and 2) when the number of multiplication operations (M) in an application is less than the number of available DSP resources (D). While it would appear that the methodology proposed in this chapter is more applicable for the case $M > D$, the experiments demonstrate that it is also applicable for the case $M < D$ when

maximizing performance is an objective. For the selected Xilinx device, $D = 32$. Vivado-HLS was chosen for comparison because it is a tool representative of all the steps in a HLS flow and hence a comparative analysis with the proposed methodology is a reflection on the limitations of existing HLS approaches.

A. FIR Filter

Different FIR filters were implemented, with the number of taps being more than 32. Direct form filter structure was used. Similar results were obtained for them. For instance, a 53 tap filter was implemented using Vivado HLS and the methodology proposed in this chapter. In this case $M = 53$. Vivado-HLS could not meet 7 ns, 8 ns, 10 ns clock period requirements and hence could not implement the design on the chosen device. It could implement the design only at 20 ns clock period requirement. On the other hand, using the proposed methodology, the fastest possible implementation could be clocked at a clock period of 6.97 ns and it made use of all the 32 DSP blocks before mapping the remaining multiplication operations to LUTs. Both Vivado and the proposed approach used 32 DSPs but the proposed approach was superior because it chose which operations to map to DSP based on the filter coefficients.

B. 40 by 40 Matrix Multiplication

In this case, a large matrix A with 40 elements in each row was multiplied with another large matrix B with 40 elements in each column. The total number of multiplications and additions required to get one element in the product matrix is 40 and 39 respectively. As a result $M = 40$. In this case, data for the elements in one row and data for the elements in one column for matrix A and B respectively were input to the function to be implemented. Vivado-HLS could not meet 7 ns, 8 ns, 10 ns clock period requirements and hence could not implement the design on the chosen device. It could implement the design only at 15 ns clock period requirement. On the other hand, using the proposed methodology, the fastest possible implementation could be clocked at a clock period of 6.512 ns and it made use of all the 32 DSP blocks before mapping the remaining multiplication operations to LUTs.

C. 7-Tap FIR Filter

In this case $M (=7) < D (=32)$. The results are shown in Table 3.10.

Table 3.10. Result comparison between Vivado-HLS and Proposed Approach for 7-Tap FIR filter

Design	Vivado-HLS			Proposed Approach	
	CLK Period Required(ns)	CLK Period Achieved(ns)	DSP48	CLK Period Achieved(ns)	DSP48
7-tap FIR filter	3	5.819	9	2.413	7
	4	5.837	9		
	5	6.083	9		
	6	5.902	9		

When implemented using Vivado- HLS, it was found that Vivado-HLS could not meet 3 ns, 4 ns and 5 ns clock period requirements. For 3 ns, it reported utilization of 9 DSP blocks with an achievable clock of 5.819 ns. For 4 ns, it reported utilization of 9 DSP blocks with an achievable clock of 5.837 ns while for 5 ns, the figures were 9 DSP blocks and 6.083 ns. It succeeded for a clock period requirement of 6 ns reporting 9 DSP blocks and achievable clock of 5.902 ns. Using the methodology proposed in this chapter, only 7 DSP blocks were utilized with a much smaller clock period of 2.413 ns.

D. Quadratic Polynomial

The quadratic polynomial: $49199 x^2 + 27133x + 41237$ was implemented using Vivado-HLS as well as the proposed methodology. The Vivado-HLS generated result failed the design rule check (DRC) during implementation and hence though it did generate RTL description of the design, it could not be implemented. On the other hand, using the proposed methodology, 4 DSP blocks were used and the smallest clock period achieved was 2.567 ns.

E. MCM-1

This consisted of multiplying a 16 bit variable A with 8 different randomly selected constants (33, 117, 221, 45, 98, 171). The proposed methodology used one DSP48 achieving a maximum clock frequency of 2.2 ns while Vivado-HLS could implement it only at a clock period constraint of 4 ns using 2 DSP blocks.

3.6 Summary

This chapter has presented a methodology to automatically identify and map multiplication and allied operations to DSP blocks on FPGAs thus bringing in both application and architecture awareness in HLS. It has been shown that the proposed LUT consumption model serves the purpose of ordering these operations with respect to their criticality (in terms of resource utilization) and thus enables to map them in order to maximize performance. The automated methodology is relevant for all types of multiplication and allied operations. The result for 7-tap FIR filter also demonstrates that applying compiler optimizations like replacing a multiplication with a series of shift and add operations does not always result in better timing performance. It can be seen that increase in operating frequencies of up to 3 times was obtained (53 tap FIR filter). While results have been reported for Xilinx FPGAs, the methodology is equally applicable to FPGAs from any vendor that support on-chip DSP blocks. Once the steps in this chapter have been carried out, resource sharing algorithms as described in chapter 4 may be applied subsequently to achieve reduction in resource utilization.

Chapter 4

Hardware Binding for Area-Delay Efficient Synthesis

4.1 Introduction

Hardware binding involves binding functional units (FUs) to different kinds of operations in the dataflow graph of an application. These operations could have been identified and relevant FUs chosen through algorithms like those described in chapter 3. The simplest hardware binding involves assigning one FU for each operation. However, non-trivial hardware binding algorithms focus on area-delay efficiency. Reduction in area utilization and increase in maximum operating frequency by reducing delays in the datapath are the objectives. Previous work in this area has been mentioned in chapter 2, Section 2.5.3.

4.2 Contributions

Using timing constraint as a knob, algorithms to reduce area are proposed. A heuristics based algorithm named Extended Compatibility Path Based (ECPB) is proposed. It also forms the basis of a *joint area-delay optimization algorithm*. The ECPB algorithm tries to reduce area without any timing constraint whereas the *joint area-delay optimization algorithm* uses ECPB to reduce area under a given timing constraint.

4.3 Extended Compatibility Path Based (ECPB) Hardware Binding

This work extends the compatibility path based (CPB) hardware binding algorithm proposed in [102]. The ECPB algorithm is based on a modification of the operation compatibility criterion and weight relation described in [102] and is aimed at resource sharing technique to reduce area. It reduces the number of inputs to MUXes in front of FUs and registers resulting in less interconnects as well as smaller area due to both

smaller MUXes and lesser interconnects. Bit widths and multimode FUs are also taken into account. The working of ECPB in the presence of user specified timing constraints is also analyzed.

A. Preliminaries

The inputs to the hardware binding stage are a scheduled dataflow graph (DFG) and resource allocation data. The number of resources available for each operation type for the purpose of binding is determined by the maximum number of identical operations scheduled in a single c-step during the compilation phase. For instance, if the maximum number of addition operations scheduled in any c-step is 10, then there are 10 adders available as resource. Similarly if the maximum number of multiplication operations scheduled in any c-step is 5, then the resource input to the algorithm has 5 multipliers.

B. Definitions

Definition 1: Weighted and Ordered Compatibility Graph (WOCG)- A graph $G(V,E)$ is a directed acyclic graph (DAG) which has a vertex set V and an edge set E . The set V consists of operations of the same type from DFG and E is a set of edges representing compatibility between operations. It means that there will be one distinct graph G for every type of operation (addition, multiplication etc) in a DFG. For instance, for a DFG with twenty operations of four different types, there will be four WOCGs each with its own sets V and E giving rise to a total of four different 'V' and 'E' sets. Operations u and v of the same type are compatible if both are scheduled in different c-steps and this is represented by a directed edge between u and v in the set E assuming u is scheduled before v i.e. $c\text{-step}(u) < c\text{-step}(v)$. Thus, operation compatibility is checked only between operations of the same type. A weight is assigned to the edges to represent the strength of compatibility. The weight relation is explained later.

Definition 2: A *path* in a WOCG is defined as a set of mutually compatible operations $\{op_1, op_2, \dots, op_n\}$ which are in order of their control steps.

Definition 3: Side Variables: FUs can have *side variables* which are their output variables with overlapping lifetimes. For instance, from Fig. 4.1(a), $v5'$ is a side variable for the adder FU “+1”. Since lifetimes of $v5$ and $v5'$ overlap, they need separate registers for storage. The reduction in number of side variables results in less registers and interconnects in the design.

C. Motivating Example

Figure 4.1 presents a motivating example. Figure 4.1(a) shows a scheduled data flow graph (DFG) whereas Figs. 4.1(b) and 4.1(c) show its binding results due to WBM [95] and CPB [102] algorithms respectively.

Fig. 4.1(a). A scheduled DFG

Fig. 4.1(b). Hardware binding using WBM algorithm

Fig. 4.1(c). Hardware binding using CPB algorithm

Fig. 4.1(a) shows a portion of a scheduled DFG. It has 6 addition operations. Operations are scheduled in different control steps (c-steps) by taking into account dataflow dependencies between them. Edges represent flow dependencies of data from one operation node to another. WBM [95] algorithm binds operations in the order of c-steps resulting in Fig. 4.1(b). The binding problem for operations in any c-step is formulated as a weighted bipartite graph between the FUs and the operations. A minimal cost matching is then found where MUX input count is taken as cost. Since WBM progresses in order of c-steps, it does local optimization and thus lacks the strength to globally optimize binding for a scheduled DFG. CPB [102] algorithm takes a global view of the problem by traversing through the complete DFG to form WOCGs and then binds operations to FUs resulting in the binding in Fig. 4.1(c). It can be seen from Figs. 4.1(b) and (c) that both WBM and CPB bind same operations to corresponding adders. However, CPB algorithm results in one level of MUXes lesser than WBM algorithm. Consequently WBM generated datapath has higher delay because of the additional level of MUXes compared to CPB as well as ECPB algorithms. Figure 4.1(d) shows the binding result due to ECPB algorithm. It has the same number of MUXes as in CPB but the total number of MUX inputs is smaller than in CPB. It can be observed that these previous methods do not exploit inter-operation flow dependencies dependency and non-overlapping lifetimes of output variables. It is possible that taking these into consideration could lead to further

reduction in interconnect and resources. This forms the premise of the proposed ECPB algorithm which when applied to Fig. 4.1(a) leads to binding shown in Fig. 4.1(d).

Fig. 4.1(d). Hardware binding using proposed ECPB algorithm

Figure 4.2 shows the lifetime chart of variables in the DFG shown in Fig. 4.1(a).

Fig. 4.2. Lifetimes of variables of DFG in Fig. 4.1(a)

Operations +3 and +2 are bound to the same adder because lifetimes of variables $v5'$ and $v7$ overlap as seen in Fig. 4.2. 'P' and 'C' in Fig. 4.2 refer to the production and consumption times respectively of a variable. 'P' present in c-step k and 'C' present in c-step m means that the variable is produced in c-step k and consumed in c-step m . When k and m coincide, it means that the variable is an input variable. Also, when $k = 1$ and m is any positive integer, it means that the variable is an input variable. For instance, $v1$, $v2$, $v3$, $v4$ (all with $k = m = 1$) and $v8$ ($k = 1$ and $m = 3$) are all input

variables. In CPB, operations +3 and +1 would be bound to the same adder. Considering inter-operation flow dependencies between operations +4 & +3 and +5 & +3, +4 is bound to the same adder as +3 as it gets both of its inputs from same type of FUs i.e. adder FUs and sends its output to the same FU type as +3 i.e. adder. On the other hand, +5 gets one of its inputs from an input register and one from an adder and hence is less favorable compared to +4. As a result ECPB reduces the number of registers required for this DFG by 1. Also, the MUX input count is reduced from 9 in Fig. 4.1(c) (CPB) to 8 in Fig. 4.1(d) (ECPB). RTL code was also generated for these bindings and the post place and route results obtained by using Xilinx ISE 10.1 on the device XC4VFX140-11 are shown in Table 4.1. It can be seen that ECPB improved both area and delay compared to WBM and CPB algorithms. WBM uses 5 registers which are shared between input and output variables whereas CPB and ECPB use 8 and 7 registers respectively. In order to keep the design testable and easily verifiable, it is not a good idea to share registers between input and output variables as it leads to verification problems.

Table 4.1: Post place and route statistics for motivating example

Algorithm	Registers	MUX inputs	LUT	Clock Period(ns)
WBM	5	12	128	3.068
CPB	8	9	112	2.964
ECPB	7	8	82	2.830

D. ECPB Algorithm Overview

ECPB tries to minimize area and delay by reducing the number of interconnects in the design. This is achieved by reducing the number of inputs to MUXes. Reduction in interconnects leads to lesser area and power consumption. The task of physical synthesis also becomes easier as there is less routing congestion to deal with during physical placement and routing.

The architecture adopted for hardware binding consists of heterogeneous FUs capable of executing different operations present in a DFG and discrete registers. Distinct

registers are used to store the output variables of FUs and the input variables. There is no register sharing between input and output variables. FUs have dedicated output registers to store their output variables. Physical synthesis tools can be relied upon to place these registers close to the FUs to minimize delay during physical synthesis. This is because physical synthesis tools pack related logic together whenever possible. In ECPB, operations are bound to FUs and the output variable is simultaneously bound to the dedicated output registers of FUs.

As in the CPB [102] method, we bind operations to one FU type at a time. For each FU type in the DFG, one WOCG is made. The CPB method relies on the fact that if we bind operations of same type that have immediate flow dependency then a feedback interconnect can serve as the datapath for several such operations. Also, if operations with common inputs are bound to the same FU, no additional interconnects or MUX inputs would be required. The sum of flow dependency and the number of common inputs between any two operations is defined as the weight between them. Thereafter longest paths are found in a WOCG iteratively using the longest path algorithm [216]. The operations in each path are bound to a single FU. ECPB modifies and extends the CPB approach based on the following observations:

- (i) Inter-operation flow dependency is considered instead of only that between operations of same type. The flow dependency is split into input and output flow dependencies. Then operations with common input and/or output flow dependencies are assigned higher weight.
- (ii) If two compatible operations have common inputs then binding them to the same FU would not require additional interconnects. Here common inputs refer to variables and not registers as those variables may not have been bound to registers yet.
- (iii) Since there are dedicated registers for output variables, two compatible operations whose output variables have non-overlapping lifetimes will not require an additional register because a possible side variable is avoided. As additional registers are not required, this would also lead to reduction in interconnects.

In addition to the observations above, ECPB also uses partial binding information that is available after operations of a particular type have been bound. WOCGs for the different operation types are not built at the same time; instead a step by step approach is adopted where each step uses partial binding information from the previous step. The order in which operations are bound is determined by the delays of the FUs allocated for the operations. The operation binding procedure begins with the FU type with the least delay. In this way ECPB algorithm is able to use as much previous binding information as possible when binding operations to FUs with higher delays. It should be noted that if a DFG has 2-input as well as n -input operations of same type (say adder), then separate WOCGs need to be created for 2-input adder FU and n -input adder FU. This is because their delays will be different as well as their input counts are incompatible.

E. The Extended Compatibility Path Based (ECPB) Algorithm

The steps involved in ECPB algorithm are shown in Fig. 4.3.

Fig. 4.3. Extended compatibility path based (ECPB) algorithm

ECPB algorithm begins with the construction of WOCG for the operation type whose FU has least delay. The operations form the vertices of the WOCG and the compatibility between operations is denoted by directed edges which are assigned weight based on the following relation which is a modified version of the one in [102]:

$$W_{ij} = 1 + \alpha \times F_{ij} + \beta \times N_{ij} + \gamma \times R_{ij} \quad (4.1)$$

where i and j are indices of two operations of same type. F_{ij} denotes the compatibility due to flow dependence, N_{ij} is the number of common inputs between operations i and j and R_{ij} represents the possibility of operations i and j storing their output results in the same register. We have the following for R_{ij} :

$$R_{ij} = 1, \text{ if the lifetimes of output variables of operations } i \text{ and } j \text{ do not overlap,} \\ 0, \text{ otherwise}$$

As explained earlier F_{ij} is the sum of input flow dependency factor F_{ij}^{in} and output flow dependency factor F_{ij}^{out} . Let us consider the current operation to be a 2-input operation. If the operations that feed inputs operations i and j have not been bound and are of same type, then we can speculate that these predecessor operations will be bound to the same FU and hence operations i and j should also be bound to that FU to reduce MUX inputs. In this case, F_{ij}^{in} is incremented by 1 for each predecessor operation of i and j that are of the same type and have not been bound yet. Similarly F_{ij}^{out} is computed. Thus if the predecessor operations in the current case happen to be bound to a single FU, F_{ij}^{in} (or F_{ij}^{out}) is incremented by 2. The coefficient α can be set high to put more emphasis on flow dependence factor. Calculation of F_{ij} is explained using Fig. 4.4.

Fig. 4.4. Calculating the flow dependence factor F_{ij}

It can be seen in Fig. 4.4 that operations +1 and +2 have flow dependencies with operations of same type (in this case additions) and hence binding these two to the same FU can save MUX inputs. If we assume that we are in the first pass of binding operations then we have $F_{ij}^{in} = 1$ and $F_{ij}^{out} = 1$. Thus $F_{12} = 2$. Since +1 and +2 have no common inputs, $N_{ij} = 0$. The output variables of +1 and +2 do not have overlapping lifetimes and this $R_{12} = 1$. Thus $w_{12} = 1 + \alpha * 2 + \beta * 0 + \gamma * 1$. For the experiments and

results reported in this chapter $\alpha = 1$, $\beta = 1$ and $\gamma = 2$. Using these values, the WOCG for the DFG in Fig. 4.1(a) is shown in Fig. 4.5.

Fig. 4.5. WOCG for the DFG in Fig. 4.1(a)

After making the WOCG, longest path algorithm was used iteratively to find longest paths. The longest path algorithm uses the weight values to find the longest path. The paths found in the WOCG are shown in Fig. 4.6.

Fig. 4.6. Paths in WOCG of Fig. 4.5

All operations in a single path are bound to the same FU. These operations are then removed from the current graph and the next longest path is found operations in which are bound to another single FU. This process is repeated until no more operations are left in WOCG. Then the algorithm starts processing the WOCG of next FU_{type} . The CPB method [102] continues to form path compatibility graphs to consider possible

path merging in order to reduce the number of registers used due to side variables. However, in ECPB algorithm, there is no merging of paths as it leads to extra multiplexers in front of registers and consequently more interconnects due to MUX input and select signals, and more area. It is to be noted that although partial binding information is used from one FU_{type} to another, the weights are still assigned in a global manner as evident from the calculation of F_{ij} . Thus, ECPB adopts a global approach in weight assignment and hence there is no risk of getting stuck in a local optimum as in WBM algorithm.

F. Managing FU bit width

When a WOCG is constructed, the nodes are also annotated with the bit width of the incident. For instance, a WOCG for 2-input addition operations can have varying bit widths of the operands on its different nodes. Since all these operations will be bound to the same 2-input adder, the bit width of the adder is adjusted to accommodate the largest operands in the WOCG. It should be noted that separate WOCGs are constructed for 2-input, 3-input etc operations even if the operation type is same.

G. Managing multi-mode FUs

Multi-mode FUs refer to FUs which can perform more than one function. For instance, a single adder-subtractor FU can perform both addition and subtraction based on a select line. The ECPB algorithm can easily manage such FUs as well. However, each instance of such FUs would be able to execute more than one mode only if the different modal operations are scheduled in different control-steps. This puts a restriction on scheduling and therefore exploiting parallelism which is otherwise possible with separate FUs for each mode, for instance a different FU each for addition and subtraction. The algorithm presented in the current work will report higher area if only multi-mode FUs were considered. This is because it will associate a multi-mode FU for each operation for which a WOCG is constructed. Since multi-mode FUs use more area than normal FUs for same operation type, the area reported will be higher. For the results in this thesis, multi-mode FUs were not considered.

4.3.1 Experimental Results

ECPB algorithm was implemented as part of a C to RTL framework. The benchmarks used to test the performance of the algorithm are the commonly used discrete cosine transforms (DCT) kernels extracted from the *MediaBench Suite* [214]. These are signal processing related benchmarks with a lot of arithmetic operations like multiplication, addition, subtraction etc. with considerable data dependencies between the operations. The benchmark *motion* is the main computational kernel in motion estimation procedure used in MPEG compression and *large* is a randomly generated DFG. For comparison, these benchmarks were also implemented using WBM and CPB algorithms. The FU allocation for all the algorithms was same in order to justify comparison. After the hardware binding stage, the framework also generated RTL codes in Verilog HDL. The RTL code was then synthesized using Xilinx ISE 10.1 for Virtex 4 (XC4VFX140-11) device, no timing constraints were set and maximum bit growth at any node was assumed to be 16 bits. The MUX input count, number of global interconnects, LUT consumption and the best achievable clock period using the three algorithms (WBM, CPB and ECPB) have been compared for all the benchmarks. Tables 4.2, 4.3, 4.4 and 4.5 show the various results.

Table 4.2: MUX input count comparison

Benchmark	Absolute values			Normalized values		
	WBM	CPB	ECPB	WBM	CPB	ECPB
dtint	149	124	114	1.307	1.087	1
dct1	78	69	65	1.200	1.062	1
dct2	81	71	67	1.209	1.059	1
cjpegdct	55	48	47	1.170	1.021	1
motion	153	109	119	1.429	1.019	1
large	205	179	169	1.213	1.059	1
dft9	335	315	272	1.394	1.201	1
Average	-	-	-	1.271	1.071	1

Table 4.2 presents the number of MUX inputs in the designs using the three algorithms. We have assumed 16-bit wide data for our experiments and hence each MUX input is 16 bits wide. Thus, the actual number of interconnects due to MUX inputs only would be 16 multiplied by the number of MUX inputs. In the right half of Table 4.2, the values have been normalized with respect to the values due to ECPB

algorithm. It can be seen that ECPB algorithm generates designs with 25.4% and 5.1% lesser MUX inputs on average than WBM and CPB respectively.

Table 4.3 shows the total number of global interconnects in the datapath. This includes MUX inputs, connections from MUXes to FUs, and from MUXes and FUs to registers. It can be seen that WBM and CPB create 34.2% and 6.1% more global interconnects on average compared to ECPB algorithm.

Table 4.3: Global interconnect count comparison

Benchmark	Absolute values			Normalized values		
	WBM	CPB	ECPB	WBM	CPB	ECPB
dctint	227	187	176	1.289	1.063	1
dct1	167	124	114	1.465	1.088	1
dct2	173	126	116	1.490	1.086	1
cjpegdct	117	95	94	1.244	1.011	1
motion	203	164	163	1.245	1.006	1
large	285	240	216	1.319	1.111	1
dft9	497	413	373	1.332	1.107	1
Average	-	-	-	1.340	1.067	1

Table 4.4 shows the LUT consumption metric. The LUT consumption metric includes total LUTs consumed by FUs, MUXes and registers. These results are the post place and route statistics. The right half of Table 4.3 shows the values after normalizing with respect to those obtained by ECPB algorithm. It can be seen that ECPB algorithm results in lesser LUT consumption than WBM and CPB algorithms by a margin of 23.8% and 6.9% respectively on average.

Table 4.4: LUT consumption comparison

Benchmark	Absolute values			Normalized values		
	WBM	CPB	ECPB	WBM	CPB	ECPB
dctint	3014	2568	2367	1.273	1.085	1
dct1	2227	1890	1727	1.289	1.094	1
dct2	2262	1926	1793	1.261	1.074	1
cjpegdct	2150	1876	1828	1.176	1.026	1
motion	2542	2275	2134	1.191	1.067	1
dft9	3789	3112	2745	1.380	1.133	1
Average	-	-	-	1.258	1.078	1

Table 4.5 shows the best achievable clock period after post place and route. The right half of Table 4.5, where values have been normalized with respect to those obtained

by ECPB algorithm, shows that the best clock period achievable with WBM and CPB algorithms is higher than that achievable by ECPB algorithm by a margin of 11.2% and 4% respectively. Thus, ECPB algorithm results in designs with lesser delays.

Table 4.5: Best achievable clock period (ns) comparison

Benchmark	Absolute values			Normalized values		
	WBM	CPB	ECPB	WBM	CPB	ECPB
dctint	9.15	8.67	8.093	1.131	1.071	1
dct1	9.23	7.9	7.86	1.174	1.005	1
dct2	9.06	8.19	8.19	1.106	1.000	1
cjpegdct	8.32	8.36	7.6	1.094	1.100	1
motion	9.064	8.775	8.563	1.058	1.025	1
dft9	9.78	9.13	8.73	1.120	1.045	1
Average	-	-	-	1.113	1.040	1

The area-time product for all the benchmarks using the three algorithms is shown in Fig. 4.7. It shows the product of average area (in terms of LUT) and average delay (in ns) for all the designs. One can observe average improvement in the area-time product of the designs generated with ECPB algorithm as 29.21% and 12.49% lower than that with WBM and CPB respectively.

Fig. 4.7. Area-time product comparison

H. Discussion of Results

The ECPB algorithm clearly outperforms WBM and CPB algorithms. The algorithm run time is proportional to the number of operations in a DFG. If all the operations are

of the same type, then the run time is directly proportional to the number of operations. However, if all operations are not of the same type, then there is no direct relation between the run time and the number of operations. In the presence of operations of different types, the run time will be higher compared to the case of homogeneous operations because different WOCGs will need to be constructed and traversed. Although the results here have been cited for FPGAs, this algorithm can as well be applied for C to RTL synthesis of full and semi-custom designs.

4.4 Joint Area-Delay Optimized Hardware Binding Algorithm

The previous algorithm focused on area reduction. It works well when resource constrained schedule of an application is used as input. However, when a timing constrained schedule is generated, it needs to be further refined so that resource sharing during hardware binding does not negate the schedule. Therefore, this section presents an algorithm that seeks to reduce area in the presence of a user defined timing constraint. It uses ECPB algorithm (section 4.3) as the basis for resource sharing but performs additional processing to ensure that the timing constraint is not violated after resource sharing. The distinctive characteristics of the proposed hardware binding algorithm is that it can be executed sequentially after scheduling. Thus it avoids the iterative scheduling and binding strategy proposed in works like [190] and is hence puts a smaller demand on run time compared to other iterative approaches [190].

I. Scheduling Under Timing Constraint

This section presents in brief the idea of timing constrained scheduling. A timing constrained scheduling algorithm schedules each operation in a DFG so as to not violate the timing constraint which is specified as the maximum required clock frequency (minimum clock period, T_{req}) at which the design should operate. Thus, each control step of the schedule has only as many operations as the delay when mapped to a FU from the resource library would not exceed T_{req} . Figure 4.8 which is a reproduction of Fig. 4.1(a) shows six addition operations scheduled in four control steps (c-steps). In a time constrained schedule, this would mean that the delay of an adder FU, Add_{delay} in the resource library obeys the constraint in equation (4.2).

$$Add_{delay} < T_{req} \quad (4.2)$$

If operation chaining were to take place during scheduling resulting in ‘ n ’ addition operations being performed in a single c-step, then the constraint in equation (4.3) must be obeyed. Operation chaining takes place when timing requirement is relaxed i.e. T_{req} has a large value.

$$n \times Add_{delay} < T_{req} \quad (4.3)$$

Fig. 4.8. A time-constrained scheduled DFG with 6 addition operations

Both equations (4.2) and (4.3) can be generalized for any type of operation and any combination of operation chaining. A heterogeneous operation chaining would involve chaining different kinds of operations in the same c-step. Thus, for every operation ‘ op ’ to be executed by a functional unit FU_{op} with delay $FU_{op-delay}$ that belongs to the set of operations ‘ S_{op} ’, equation (4.2) can be generalized to equation (4.4).

$$FU_{op-delay} < T_{req} \quad \forall op, op \in S_{op} \quad (4.4)$$

Similarly, for any operation chain ‘ op_{chain} ’ with delay ‘ $op_{chain-delay}$ ’ present in the schedule, equation (4.3) can be generalized to equation (4.5).

$$op_{chain-delay} < T_{req} \quad (4.5)$$

Since operation chaining reduces the chances of resource sharing, chained operations are not considered during resource sharing in both ECPB as well as the algorithm proposed in this section. The inequality in (4.6) shows the actual condition that must be satisfied by a compute logic ‘ L ’ with delay L_{delay} present in a c-step. T_{c-q} is the clock to output delay of the register (source register) feeding the data to ‘ L ’ and T_{su} is the set up time requirement of the destination register which is the output register holding the result of computation by ‘ L ’.

$$T_{c-q} + L_{delay} + T_{su} \leq T_{req} \quad (4.6)$$

Since a c-step is assumed to execute in one clock cycle and operation chaining is not considered, equation (4.7) holds.

$$L_{delay} = FU_{op-delay} \quad (4.7)$$

In the presence of resource sharing i.e. FU_{op} being used to execute ‘ n ’ instances of operation ‘ op ’, L_{delay} is given by equation (4.8) where $MUX(n/1)_{delay}$ is the delay of a n -to-1 multiplexer.

$$L_{delay} = FU_{op-delay} + MUX(n/1)_{delay} \quad (4.8)$$

The timing slack available (prior to resource sharing) for a functional unit FU_{op} , $slack^+(FU_{op})$ is given by inequality (4.9) where timing slack refers to the positive time duration available within a clock period. It is the duration in which no computation is performed.

$$slack^+(FU_{op}) \leq T_{req} - FU_{op-delay} - T_{c-q} - T_{su} \quad (4.9)$$

This positive timing slack can be utilized to share the resource. Hence, The multiplexer delay i.e. $MUX(n/1)_{delay}$ should be less than or equal to the positive timing slack as shown in inequality (4.10).

$$MUX(n/1)_{delay} \leq T_{req} - FU_{op-delay} - T_{c-q} - T_{su} \quad (4.10)$$

Therefore, the largest multiplexer that can be used to share a resource for maximum number of operations (n_{max}) must satisfy equation (4.11).

$$MUX(n_{max}/1)_{delay} = T_{req} - FU_{op-delay} - T_{c-q} - T_{su} \quad (4.11)$$

Since T_{c-q} and T_{su} are of the order of a few picoseconds in FPGAs, they can be ignored in equation (4.11) and therefore, equation (4.11) reduces to equation (4.12).

$$MUX(n_{max}/1)_{delay} = T_{req} - FU_{op-delay} \quad (4.12)$$

Clearly, since $FU_{op-delay}$ will be different for different FU_{op} , the maximum sharing will also be different for different types of operations. Thus, the maximum sharing (n_{max}) of each FU_{op} is dictated by the amount of positive timing slack available for it under a given timing constraint. Therefore, it is possible that some FU_{op} may not get shared at all while others get shared.

J. Latency Preserving Slack Based Joint Area-Delay Optimization

In the timing analysis presented in section I, the latency of the initial schedule is preserved. The sharing of resources does not lead to any increase in latency. The latency preserving slacked based joint area-delay optimization algorithm is presented in Fig. 4.9. It is based on ECPB algorithm (section 4.3) to which additional processing steps (2), (7) and (8) have been added.

Latency Preserving Slack Based Joint Area-Delay Optimization

1. sort FU_{types} in order of *decreasing* slack
2. find n_{max} for each FU_{type}
3. **for** all FU_{type} **do**
4. form $WOCG_{type}$
5. **repeat**
6. find *longest path*
7. divide longest path into sub-paths of length n_{max} nodes
8. bind operations in each sub-path to a single FU
9. bind output variables to registers dedicated to FUs
10. remove bound operations from $WOCG_{type}$
11. **until** all operations in $WOCG_{type}$ have been bound
12. **end for**
13. bind input variables to independent registers

Fig. 4.9. Algorithm for latency preserving slack based joint area-delay optimization

Figure 4.10 shows how the WOCG for the addition operation in Fig. 4.8 is processed for a n_{max} value of 2. The dashed line AB divides the longest path (4 nodes) into two sub-paths (2 nodes each) and each path is mapped to one adder FU for a total of 2 adders. Thereafter all these nodes (+2, +3, +4, +6) are removed and a new longest path is found which happens to be 2 nodes long (+1, +5) and hence requires no further sub-division. These operations are shared onto one adder FU resulting in a total of three adder FUs for the entire design compared to two adder FUs when ECPB is used in its original form (section 4.3) alone.

Fig. 4.10. WOCG for the scheduled DFG in Fig. 4.8

The determination of longest path using the weight relationship described in section 4.3 ensures that all the factors chosen for describing weight relationship remain considered even during the application of latency preserving slack based joint area-delay optimization.

Depending on the value of n_{max} for a FU_{op} , it is possible that no sub-division of the longest paths may be required. For instance if $n_{max} = 15$ for some FU_{op} and the longest paths are less than 15 nodes long, then there will not be any further sub-division of the longest paths. Since n_{max} could be different for different FU_{op} , it is possible that the longest paths for some FU_{op} may be partitioned while for others they may not be partitioned. The run time of this algorithm depends on the number of different types of FU_{op} used as well as to the n_{max} value for each FU_{op} .

4.4.1 Experiments & Results

The proposed algorithm was tested on the same benchmark circuits as those used for ECPB in section 4.3.1. The same DFGs were assumed to be scheduled under time-constraints. However, different timing constraints were applied to them to accommodate for the most delay critical operations in them. The same DFGs were chosen to see the effect of area minimization under a given timing constraint compared to area minimization without a timing constraint. The target device was Virtex 4 (XC4VFX140-11) and the post place and route results were obtained using Xilinx ISE 10.1.

Table 4.6.LUT consumption results

Benchmark	Joint Area-Delay Optimization with CLK period constraint 'CC'			ECPB	
	CC = 5ns	CC = 6ns	CC = 7ns	LUT	CLK(ns)
dctint	3124	2867	2546	2367	8.093
dct1	2211	2101	1879	1727	7.860
dct2	2312	2013	1910	1793	8.190
cjpegdct	2431	2214	2033	1828	7.600
motion	2745	2543	2349	2134	8.563
dft9	3214	3102	2815	2745	8.730

Table 4.7. MUX input count comparison

Benchmark	Joint Area-Delay Optimization with CLK period constraint 'CC'			ECPB	
	CC=5ns	CC = 6ns	CC = 7ns	MUX	CLK(ns)
dctint	134	124	121	114	8.093
dct1	87	79	71	65	7.860
dct2	93	81	73	67	8.190
cjpegdct	53	50	49	47	7.600
motion	137	133	122	119	8.563
dft	334	314	292	272	8.730

Table 4.6 shows the LUT consumption results for joint area-delay optimization under three different timing constraints. As can be seen, with tighter timing constraints, the LUT utilization increases because less number of operations are shared onto a FU leading to an increase in the FU count. Table 4.7 shows the MUX input count comparison. It can be seen in general that the MUX input count increases with a

tighter timing constraint because of the proliferation of a large number of smaller MUXes compared to a small number of large MUXes when the timing constraint is relaxed. These observations are in line with the principle that a faster design uses more resources and there is less degree of resource sharing though there could be a large number of smaller MUXes in such designs.

4.5 Summary

This chapter has presented algorithms for area-delay efficient hardware binding. All the algorithms have focused on resource sharing to reduce area as well as to manage delay through the datapath. The result of hardware binding is the actual datapath which describes the microarchitecture produced as a result of HLS steps. This microarchitecture, presented as auto-generated RTL is then synthesized by downstream physical synthesis tools like Xilinx ISE etc. for final implementation on a FPGA device. Comparisons were not made with any commercial tool because they apply optimization passes on the used benchmarks. Use and investigation of these passes is beyond the scope of the work described in this chapter. Hence, comparisons were restricted to resource sharing algorithms reported in research literature.

Chapter 5

Application Dataflow Graph Partitioning Under System Level Constraint

5.1 Introduction

System level constraint like “Data Initiation Interval (II)” gives an opportunity to reduce resource utilization during HLS. An II constraint limits the throughput requirement and therefore the computational load on the system is also reduced. This can be exploited to share resources during the hardware binding phase. This also helps in managing design complexity as the dataflow graph can be partitioned based on II constraint. From a run-time perspective, each of these partitions can be processed in parallel during hardware binding which would result in reduced run time.

The availability of partitions also facilitates *Design Space Exploration* during datapath formation by stitching the partitions together in time selectively. Looking at the design space generated by varying the II constraint can also facilitate selection of a suitable II constraint in a system design exercise.

5.2 Contributions

An *II-aware dataflow graph partitioning algorithm* is proposed to take into account II-constraint. At the same time, a resource sharing algorithm is applied to such partitions in order to reduce area. This results in designs with just the right amount of resource sharing for a given II-constraint. Partitions are also ranked based on estimated area savings and temporally joined together to generate multiple design points with different area and time measures to enable a designer to make decisions.

5.3 Initiation Interval (II) Aware Graph Partitioning

Figure 5.1 shows a simple large dataflow graph (DFG) with 100 addition operations. In this figure, inputs 1 and 2 are the primary set of inputs while the remaining inputs 3 to 101 are read from a register file or are constants. Traditional hardware binding algorithms focused on resource sharing to reduce area would share one adder among all the 100 addition operations. This would lead to a 100 to 1 multiplexer in front of the adder and the datapath would like in Fig. 5.2. There are two adverse effects on design because of this. First, the large multiplexer will have a very negative impact on timing because of its size. Second, since 100 addition operations have been mapped to one adder, a new set of inputs 1 and 2 cannot be processed until all the 100 addition operations are over. Depending on the architecture of the adder, this may require 100 clock cycles (assuming the adder completes its operation in one clock cycle) or more (in case of a multicycle adder). Therefore, there is a significant constraint put on the design because of the way the micro-architecture got architected and this may not be a desirable outcome.

Fig. 5.2. Typical datapath for area minimization

Fig. 5.3. Datapath for II=10 clock cycles

Ideally, it is the design requirements which should guide the datapath generation. However, in the motivation example it is the datapath which generates a design constraint or invalidates some pre-existing design objective on the data line rate. If we set an II constraint of 10 clock cycles, where a new set of primary inputs 1 and 2 arrive every 10 clock cycles, the datapath in Fig. 5.2 would not be functional. In order to meet this constraint and reduce the area at the same time, the datapath shown in Fig. 4.3 should be generated. The datapath in Fig. 5.3 has 10 sub-circuits named C1 to C10. Each sub-circuit shares one adder among 10 addition operations. When C1 has finished processing the 10 addition operations shared on its adder, C2 begins the next processing stage thus leaving C1 to process the next set of 10 primary inputs. In this way, all the ten sub-circuits can remain functional at the same time but processing data corresponding to different sets of primary inputs. The design meets the II constraint and only 10 adders are used.

5.3.1 Graph Partitioning Flow

The initiation interval aware dataflow graph partitioning is presented in this section. It should be noted that this partitioning algorithm aims at meeting the II constraint while a reduction in area is also taken into account. The high level synthesis flow with graph partitioning is shown in Fig. 5.4. We can see from Fig. 5.4 that there is a feedback (dashed arrow AB) from the hardware binding algorithm to the graph partitioning algorithm. The feedback path contains information regarding the resource sharing opportunity in a given partition and the graph partitioning algorithm uses this information to decide the partition boundary. It is the underlying hardware binding algorithm that determines the resource sharing.

Fig. 5.4. HLS flow with DFG partitioning

Before the description of the II aware graph partitioning flow, a brief description of the hardware binding algorithm used to produce the results and as part of the flow shown in Fig. 5.4 is presented for the sake of completeness. The extended compatibility path based hardware binding (ECPB) algorithm presented in chapter 3 has been used to produce the results. It should be noted again that the partitioning algorithm is not tied to any particular hardware binding algorithm. Any hardware binding algorithm that focuses on resource sharing can be used.

5.3.2 Extended Compatibility Path Based (ECPB) Hardware Binding

The ECPB algorithm was proposed in [218] and is described in detail in chapter 4. Its salient points are described here for the sake of completeness. Fig. 5.5 shows a scheduled DFG with six addition nodes (+1 to +6). Figure 5.6 shows the weighted and ordered compatibility graph (WOCG) for the addition operations in Fig. 5.5.

Fig. 5.5. A scheduled DFG

Fig. 5.6. WOCG of addition in Fig. 5.5

A WOCG has operations arranged in order of their control steps (c-steps). In ECPB algorithm, a WOCG is formed for each type of operation in a DFG. Hence, if a DFG has three different kinds of operations, there will be a WOCG for each type. The

weight relation between the different operations of the same type is given by equation (5.1).

$$W_{ij} = 1 + \alpha F_{ij} + \beta N_{ij} + \gamma R_{ij} \quad (5.1)$$

Here i and j are indices of two operations of the same type. F_{ij} denotes the compatibility between the two operations due to flow dependence. N_{ij} denotes the number of common inputs between them while R_{ij} is the possibility of their outputs being stored in the same register based on variable lifetime analysis. R_{ij} is 1 if the lifetimes do not overlap and 0 otherwise. The weight values thus calculated are the edge weights in WOCG as shown in Fig. 5.6. The longest path algorithm [216], which uses edge weights for weighted graphs, is run on the WOCG to extract the longest path. All the operations in the longest path are bound to one functional unit (adder in case of Fig. 5.6). These operations are then removed from the WOCG and longest path algorithm is run again on the remaining nodes in the WOCG. This process is repeated until no more nodes are left in the WOCG and all nodes (operations) are thus bound to functional units. The path “+2, +3, +4, +6” is the first longest path in Fig. 5.6. After these addition operations have been bound to an adder, they are removed and it results in a second longest path “+1, +5”. Operations +1 and +5 are then bound to another adder. In this way, all addition operations are bound. All functional units (adders in Fig. 5.6) have an associated output register which holds their results. There are more than one output register associated in case of registers with overlapping lifetimes. We set $\alpha = 1$, $\beta = 1$ and $\gamma = 2$ to give equal weight to flow dependence and common inputs and more to register sharing.

The resource sharing opportunity is calculated based on the longest path available in the WOCG. Referring to Fig. 5.6 as an example, the number of addition operations in the longest path “+2, +3, +4, +6”, which is four, gives the maximum existing sharing opportunity i.e. $SO(WOCG_i)$ for the adder to which these operations are bound. Similarly we check for other adders resulting in the WOCG corresponding to the addition operation. This is repeated for every WOCG corresponding to different

operations in a partition. It should be noted here that during formation of WOCG and subsequent analysis, the distance metrics D_{c-step} proposed in section 5.4 and D_{op-mc} proposed in section 5.6 are also taken into account.

5.3.3 Initiation Interval (II) Aware Graph Partitioning Algorithm

The initiation-interval aware graph partitioning algorithm is shown in Fig. 5.7. Let **NoOpportunityList** maintains a list of FUs which cannot be shared further in a partition because they are already maximally shared with number of operations shared $= k$, where k is the II constraint or the D_{c-step} constraint (equation 5.3, section 5.4) is not satisfied. Similarly, **OpportunityList** maintains a list of FUs in a partition which can be shared further as they have still not been shared k times. Lines 9-15, expand the size of the partition one c-step at a time if the **OpportunityList** is not empty. This way the size of a partition is not always restricted to k number of c-steps as that would be a trivial partition without exploiting sharing opportunities that may be available as the DFG is traversed from top to bottom.

Lines 16-25 take into account the fact that new FUs (different from those in a given partition) can be encountered as one moves beyond the current partition and these can be incorporated into the current partition. Hence, even if **OpportunityList** is empty, these new FUs can be added to the **OpportunityList**.

At line 24, one reaches the end of a partition. Thus, the creation of a partition is an incremental exercise where the initial size of partition in terms of number of c-steps is k . Thereafter, based on resource sharing opportunity calculated, the size of the partition is increased until a partition boundary is reached when no more sharing is possible. Fig. 5.8 shows a simplified flowchart for the II aware graph partitioning algorithm.

In line 25, a new partition is created beginning from the c-step immediately after the last c-step included in the previous partition as shown in Fig. 5.9. Each {begin, stop} pair forms a partition. The DFG is traversed until the last c-step is reached. This creation of partitions by traversing the DFG top to down as explained in the algorithm

ensures that the algorithm can deal with DFGs of any size. Section 5.6 explains the case of those operations where $SO(WOCG_i)$ cannot be defined.

II Aware Graph Partitioning Algorithm

Input: Scheduled DFG, II constraint = k

Output: Temporal partitions

1. Take partition P1 beginning for the first c-step with no. of c-steps = k
 2. Make the WOCG for each operation in partition
 3. **For** each WOCG in partition **do**
 4. Calculate the maximum existing sharing opportunity: $SO(WOCG_i)$
 5. **If** $SO(WOCG_i) = k$, go to step 3 and update **NoOpportunityList**
 6. **Else** assign operation i to **OpportunityList**
 7. **EndIf**
 8. **EndFor**
 9. **If** **OpportunityList** is not Empty
 10. **For** every operator in **OpportunityList** **do**
 11. **If** operator in the next immediate c-step beyond current partition $\notin$
 NoOpportunityList and $\in$ **OpportunityList**
 12. Include the next immediate c-step beyond current partition into the
 partition
 13. **EndIf**
 14. **EndFor**
 15. **EndIf**
 16. **If** **OpportunityList** is Empty
 17. **If** operator m in the next immediate c-step beyond current partition $\notin$
 NoOpportunityList
 18. Include the next immediate c-step beyond current partition into current
 partition
 19. Calculate $SO(WOCG_i)$ for the newer operator
 20. **If** $SO(WOCG_i) = k$, go to step 25 and update **NoOpportunityList**
 21. **Else** assign operation m to **OpportunityList**
 22. **EndIf**
 23. **Else** go to step 25
 24. **EndIf**
 25. **EndIf**
 26. Create a partition (with no. of c-steps = k) beginning after the last included c-step in the
 previous partition
 27. Go to step 2 and repeat until the DFG is fully traversed
-

Fig. 5.7. II-Aware dataflow graph partitioning algorithm

Fig. 5.8. A higher level flow of II aware DFG partitioning

Fig. 5.9. Graph traversal and formation of partitions

A. Effect of Operation Schedule and II Constraint

The investigation into the effect of operation schedule and II constraint on resource sharing is presented here. Fig. 5.10 shows part of a DFG with five addition (A1, A2, A3, A4, A5) and four multiplication (M1, M2, M3, M4) operations scheduled in four control steps (c-steps).

Fig. 5.10: DFG type 1

Fig. 5.11: DFG type 2

If an II constraint of four clock cycles is set and assuming that each operation executes in one clock cycle, we can see that four addition operations, A1 or A2, A3, A4 and A5 can be shared onto a single adder A_0 while the fifth addition operation is scheduled on another adder A_1 . Similarly, the four multiplication operations can also be shared onto a single multiplier M_0 . It can be seen that sharing any more addition operations on A_0 is not possible as then it would violate the II constraint. Similarly, one cannot share any more multiplication operation on M_0 as then it would violate the II constraint. Thus one can share only as many operations as the II constraint as long as those operations are scheduled in as many c-steps.

Figure 5.11 shows another part of a DFG. It has three additions operations (a1, a2, a3) and four multiplication operations (m1, m2, m3, m4) scheduled over five c-steps. In this particular case, only a1 and a2 can be shared on a single adder A_2 and a3 will need to be bound to another adder A_3 . This is because a3 can be executed only in c-step 5. Similarly, only m1, m2 and m3 can be shared on a single multiplier M_1 . Operation m4 will have to be bound to another multiplier M_2 because it is scheduled in c-step 5.

Thus we see that in this particular case, the number of operations shared with a functional unit is less than Π and therefore if the difference of the two c-steps for two operations of the same type exceeds the Π constraint, those two operations cannot be shared onto a single functional unit. We therefore propose a distance metric $D_{c\text{-step}}$ which is defined as per equation (5.2).

$$D_{c\text{-step}} = cstep(op2) - cstep(op1) + 1 \quad (5.2)$$

Here $op1$ and $op2$ are two operations of the same type but scheduled in different c-steps with $op2$ scheduled later than $op1$. This distance metric is incorporated during the formation of WOCG by hardware binding algorithm. It is important because it helps in expanding the size of a partition beyond Π c-steps thus facilitating resource sharing. For operation $op2$ to be included in the partition and to be bound to the same functional unit to which $op1$ will be bound, the constraint in equation (5.3) must be satisfied:

$$D_{c\text{-step}} = cstep(op2) - cstep(op1) + 1 \leq \Pi \quad (5.3)$$

It can be seen from Fig. 5.12 that it is still possible to share the three multiplication operations (M1, M2, M3) because $D_{c\text{-step}}$ between c-steps of M1 and M3 is 4. Thus the partition boundary is at c-step 5. C-steps 1 to 4 was the initial partition size (for $\Pi = 4$) for which different WOCGs were formed (as mentioned in section 5.2) and then it was incrementally increased to up to c-step 5 where lay the final partition boundary. Note that though one of the two inputs for operations M3 and A5 come from some registers or are constants, this does not affect the partition.

Fig. 5.12. DFG type 3

Fig. 5.13. DFG type 4

Figure 5.13 shows a part of a DFG where a new type of operation *subtraction* is scheduled in c-steps 5 and 6. Beginning with the base partition, which is up to c-step 4, and based on the explanation earlier in section 5.3 (lines 16-25 of the graph partitioning algorithm), subtraction operations S1, S2 and S3 will also be included in the partition as it is expanded, with S1 and S3 mapped to one subtractor FU and S2 to another and thus the partition grows to have its boundary at c-step 6. When c-step 7 is analyzed for inclusion in or exclusion from this current partition, it can be seen that though subtraction operation S4 satisfies D_{c-step} inequality constraint (equation (5.3)), addition operation A5 fails this constraint with respect to A1, A2 and A3, though satisfies it with respect to A4. Since in the base partition, A4 shared the adder either with A1 or A2, sharing A5 on the same adder will violate the II constraint. Therefore A5 cannot be included in this partition. In such cases, the partition algorithm retains the previous partition boundary which was at c-step 6 and does not include c-step 7 into the current partition.

B. Effect of Operation Chaining

In this section, the effect of operation chaining on dataflow graph partitioning is considered. Operation chaining involves scheduling more than one operation in a c-step. This is generally employed to make use of the extra timing slack that may be available in a c-step to execute more than one dataflow dependent operation as shown in Fig. 5.14. Operation chaining is a fairly well established technique[212], [213] to reduce area (by getting rid of additional intermediate output registers/latches where possible) and reducing the latency of a design. For schedules generated under a timing constraint, one can see examples like in Fig. 5.14(a) where operation chaining is employed. Figure 5.14(b) shows the same scheduled DFG but without considering operation chaining.

It is evidently clear that if the delay of a multiplier is more than or equal to the delay of three adders, the schedule in Fig. 5.14(a) will have a lower latency and input to output delay compared to the schedule in Fig 5.14(b) for the same clock period. During graph partitioning, the partitioning algorithm works as before, however, the

hardware binding algorithm treats a “chain of operations” as one consolidated operation. Hence, the chain of addition operations “A2, A3, A4” is treated as one consolidated operation “CO1”.

Fig. 5.14(a). DFG with operator chaining

Fig. 5.14(b). DFG with no operator chaining

Similarly, the chain of addition operations “A5, A6, A7” is treated as another consolidated operation “CO2”. Hence, a WOCG for each such consolidated operation is formed. In this particular example, since the two consolidated operations CO1 and CO2 are of the same type, they can be shared onto one consolidated operator which is a series of three adders in this case.

C. Effect of Multicycle Operations

Multicycle operations in a DFG need to be handled in a special way during DFG partitioning if the internal latency of such operations exceeds the II constraint. The hardware binding algorithm creates independent datapath for multicycle operations with internal latencies more than the II constraint. However, multicycle operations with internal latencies less than or equal to the II constraint are absorbed in the partitions. Figure 5.15 shows a multicycle operation *op-mc* with a self-latency of 4 clock cycles in a DFG where $II = 2$. It can be easily seen that operations M1, M2, A1

and A2 can form first partition, M3,M4,A3 and A4 second partition and M5,M6, A5 and A6 the third partition with each partition having one adder and one multiplier.

Fig. 5.15. A DFG with a multicycle operation *op-mc*

When the base partition of two -steps (c-steps1 and 2) is formed for analyzing the DFG, the multicycle operation *op-mc* cannot be included in the partition because it starts functioning in c-step 2 (which is in the base partition) but finishes in c-step 5. Thus, the maximum sharing opportunity $SO(WOCG_i)$ is undefined for such an operation as WOCG cannot be formed.

If *op-mc* has been allocated an operator that in itself uses resource sharing in its own internal implementation (which has led to its latency), then its inputs P & Q will need to be delayed by 2 clock cycles for $II=2$ to be satisfied. On the other hand, if *op-mc* has been allocated a pipelined operator (which has led to its latency) and this pipeline can be activated every clock cycle once it is full, then there is no need to delay the inputs P&Q inputs.

The graph partitioning algorithm keeps track of the multicycle operations and operators allocated to them and uses the metric D_{op-mc} to ensure temporal coherency:

$$D_{op-mc} = L_{op-mc} - II \quad (5.4)$$

where L_{op-mc} is the internal latency of *op-mc*.

The graph partitioning algorithm inserts D_{op-mc} number of register stages to all those output edges which share the same c-step as the output edges driving the inputs of $op-mc$. For instance, inputs to M2 and A2 will also be delayed by two clock cycles to ensure latency matching along the entire datapath corresponding to the DFG.

5.4 Datapath Formation and Design Space Exploration

Once the partitions and any register stages (based on section 5.6) are generated, the datapath is formed by stitching the partitions and register stages in time to maintain the temporal coherency. A partition ranking algorithm is proposed in Fig. 5.16 to choose profitable partitions to be included in the datapath. The ranking algorithm gives information on which partitions contribute most to area reduction.

Partition Ranking Algorithm

Input: Partitions of a scheduled DFG under a user defined II constraint, operator characterization library

Output: Rank of partitions based on area reduction

1. **For** all partitions **do**
 2. Calculate area utilization (A_b) without resource sharing under the given II constraint
 3. Calculate area utilization (A_r) with resource sharing under the given II constraint
 4. Calculate area savings (A_s) = $A_b - A_r$
 5. **EndFor**
 6. Perform descending order sort on all A_s
-

Fig. 5.16. Partition ranking algorithm

The different area utilization values are calculated as shown in equations (5.5) and (5.6).

$$A_b = \sum_{i=1}^t n_i A_i \quad (5.5)$$

$$= \sum_{i=1}^p n_i A_i + \sum_{i=1}^q s_i A_{mux,i} \quad (5.6)$$

where n_i is the number of instances of operator i with area A_i and s_i is the number of instances of MUX i with area $A_{mux,i}$ on a target FPGA architecture. All area values are in terms of number of LUTs used in the target FPGA architecture. At this point, it should be made clear that the area of MUXes or FUs remains the same across members of a FPGA device family. For example, an adder will use same number of LUTs across all devices in Virtex-4 family from Xilinx as shown in Table 5.1. Hence a one step characterization process for a particular device family only is required and not for each device in a device family.

Table 5.1. Adder area for different Xilinx Virtex 4 devices

FU	Device	LUT
Adder (8 bit)	XC4VLX15-10SF363	8
	XC4VLX25-10SF363	8
	XC4VLX40-10FF668	8
	XC4VLX60-10FF668	8
	XC4VLX80-10FF1148	8
	XC4VLX100-10FF1148	8
	XC4VLX160-10FF1148	8

As mentioned earlier, the datapath is formed by stitching the partitions in time to maintain temporal coherency. The profitable partitions are chosen based on the partition ranking algorithm. At this point, a *design space exploration (DSE)* can be done by choosing different groups of profitable partitions which would result in multiple datapath designs corresponding to the different groups. This is shown in Table 5.6 where D2_2 and D2_4 for DFG(50) corresponds to two designs with top two and four partitions respectively. The remaining section of DFG(50) is implemented in baseline mode without resource sharing. Such a DSE can assist a designer in choosing a design point based on other considerations like clock period, design feature enhancement, availability of suitable FPGA devices etc.

5.5 Experiments and Results

The graph partitioning algorithm was tested on MediaBench [214] and ExPRESS [215] benchmarks. For the MediaBench applications listed in Table 5.2, a basic block

(generated by Trimaran [79]) with a large number of c-steps was partitioned. The operations as scheduled by Trimaran in a basic block were used and no explicit additional scheduling was done. The selected basic blocks had a sequence of operations to be executed one after another with one operation in each c-step. The ExPRESS benchmarks cited here are used without any modification from the way their DFGs are available at the website. These DFGs have multiple parallel operations in the same c-step. For the ExPRESS benchmarks: smoothen (MESA) and interpolation (MESA), the right most independent and parallel DFG blocks in the overall DFG were used. Table 5.3 shows the partitioning results for ExPRESS benchmark.

Table 5.2. Comparison of number of FUs for II=3 clock cycles (MediaBench)

MediaBench		#c-steps	#ADD	#SHL	#SHR	#XOR	#AND	#OR	Functional?
Blowfish_cbc	Original	14	0	0	6	0	8	0	Yes
	II-Aware Partitioned	14	0	0	2	0	3	0	Yes
	Simple Way	14	0	0	3	0	4	0	Yes
	Without Partitioning	14	0	0	1	0	1	0	No
Blowfish_bf_enc	Original	354	144	64	48	50	48	0	Yes
	II-Aware Partitioned	354	48	32	16	17	16	0	Yes
	Simple Way	354	76	43	27	43	32	0	Yes
	Without Partitioning	354	1	1	1	1	1	0	No
Sha_bb9	Original	985	262	157	157	98	96	215	Yes
	II-Aware Partitioned	985	94	79	79	58	39	85	Yes
	Simple Way	985	160	157	157	98	83	156	Yes
	Without Partitioning	985	1	1	1	1	1	1	No
Pgencode_md5	Original	628	260	64	64	128	32	80	Yes
	II-Aware Partitioned	628	87	62	62	65	32	63	Yes
	Simple Way	628	189	64	64	87	32	66	Yes
	Without Partitioning	628	1	1	1	1	1	1	No

The results under "without partitioning" refer to the case where the hardware binding algorithm (as explained in section 5.2) was run on the entire DFG in one attempt

without any partitioning. It is clear that such a hardware binding without partitioning gave minimum resource utilization. However, none of those designs would work for an II of 3 clock cycles as a new set of primary inputs cannot be fed to these designs until the last operation shared on the FU with the largest number of operations shared has been executed. The column "Functional?" answers the question whether the selected design will function under the given II constraint. The results under "Simple Way" refer to the case where a DFG is trivially partitioned where the partition size is II control steps. While the "Simple Way" of partitioning does lead to a functional design, it still uses more resources than the II-aware partitioned approach. The graph partitioning algorithm and datapath generation were both tested on synthetic DFGs where each DFG had multiple parallel operations in a c-step. Table 5.4 shows the different synthetic DFGs considered and the number of partitions generated for II=3,4 and 5.

Table 5.3. Comparison of number of FUs for II=3 clock cycles (ExPRESS)

ExPRESS		#c-steps	#MUL	#ADD	#SUB	Functional?
Smoother Triangle (MESA)	Original	11	16	16	4	Yes
	II-Aware Partitioned	11	8	4	2	Yes
	Simple Way	11	10	8	4	Yes
	Without Partitioning	11	4	4	2	No
Interpolation (MESA)	Original	8	9	13	1	Yes
	II-Aware Partitioned	8	4	4	1	Yes
	Simple Way	8	4	7	1	Yes
	Without Partitioning	8	2	2	1	No
Elliptic Wave Filter	Original	14	8	26	0	Yes
	II-Aware Partitioned	14	6	11	0	Yes
	Simple Way	14	6	11	0	Yes
	Without Partitioning	14	2	4	0	No

Table 5.4. DFG vs. number of partitions for different II

DFG	Number of Partitions		
	II=3	II=4	II=5
DFG(50)	6	5	4
DFG(100)	9	7	6
DFG(300)	29	22	18
DFG(500)	53	42	34

In the notation DFG(N), N means the number of operational nodes in the DFG. Table 5.5 shows the effect of partition ranking algorithm on the partitions of DFG 50 for an II constraint of 3 clock cycles.

Table 5.5. Descending order of partitions for DFG50 for II = 3 clock cycles

DFG	Partition Number	A_b (LUT)	A_r (LUT)	$A_s = A_b - A_r$
DFG(50)	P1	464	328	136
	P4	492	403	89
	P5	525	451	74
	P6	493	419	74
	P3	508	435	73
	P2	311	265	46

Table 5.6 shows the partitioning and datapath synthesis results for all the synthetic DFGs for an II constraint of 3 clock cycles. In the notation “DP_Q”, D refers to a design point, P refers to the design point number and Q refers to the number of partitions selected from the top in the list sorted in descending order as shown in Table 5.5 for the case of DFG50. For instance, D2_5 means design point 2 and it has 5 partitions. P = B refers to the baseline design which has no partitions. As noted earlier, only designs with profitable ($A_s > 0$) partitions were synthesized. When $Q < \text{maximum number of partitions}$ as shown in Table 5.4, the design is a mix of Q partitions and the remaining sections of DFG are implemented in baseline mode. For example, D2_2 for DFG50 means that this design has top 2 partitions (P1,P4) and the remaining parts of DFG corresponding to partitions P5, P6, P3 and P2 are implemented without resource sharing. The percentage area reduction is calculated with respect to the base design (D1_B) where the DFG has not been partitioned and no resource sharing is done.

Fig. 5.17, 5.18, 5.19 and 5.20 show the scatter plot of area-clk for all the synthetic DFGs mentioned in Table 5.4 for three different II constraints of 3, 4 and 5 clock

cycles. It can be seen that based on graph partitioning and selection of profitable partitions, multiple design points can be generated which satisfy a given II constraint and there is a reduction in area as well. A design space exploration can also be done based on varying the II constraint to look at the area variations which can be used for decision making as is evident from Fig. 5.17 which shows the area-clk plot for DFG50. There are multiple design points with different area-clk parameters for three different II constraints of 3, 4 and 5 clock cycles.

Table 5.6. Post place and route synthesis results for DFGs for II = 3 Clock Cycles

DFG	Design Point	Synthesis (no DSP 48)			Synthesis (with DSP 48)			
		LUT	CLK(ns)	% Area Reduction (LUT)	LUT	DSP48	CLK(ns)	% Area Reduction DSP
DFG(50)	D1 B	2390	7.429	-	400	24	6.496	-
	D2 2	2258	7.273	5.5	640	16	5.485	33.33
	D3 4	2097	7.633	12.2	755	12	7.597	50.00
	D4 6	2025	8.159	15.2	805	10	6.131	58.33
DFG(100)	D1 B	3688	7.106	-	976	31	6.929	-
	D2 3	3277	7.648	11.1	1272	23	6.026	25.80
	D3 6	3195	7.682	13.3	1561	13	5.829	58.06
DFG(300)	D1 B	12086	7.824	-	2368	123	6.987	-
	D2 5	11303	8.344	6.4	3061	101	7.156	17.88
	D3 10	10869	8.129	10.07	3687	84	7.111	31.70
	D4 15	10609	8.477	12.22	4187	71	7.544	42.27
	D5 20	10429	8.431	13.71	4687	53	7.367	56.91
DFG(500)	D1 B	22028	7.509	-	12203	146	8.978	-
	D2 5	20777	8.181	5.6	9470	152	8.345	-4.11
	D3 10	20023	8.53	9.1	7580	153	7.961	-4.79
	D4 15	19345	8.098	12.18	6031	152	7.578	-4.11
	D5 20	19041	8.274	13.56	6497	137	7.53	6.16
	D6 25	18720	8.23	15.01	6994	120	7.274	17.80
	D7 30	18403	8.363	16.45	7367	112	7.185	23.28
	D8 35	18283	7.827	17.00	7677	106	8.655	27.39

Fig. 5.17. Area-Clk Plot for DFG50

Fig. 5.18. Area-Clk Plot for DFG100

The baseline design point (with the suffix _B) remains the same for all II constraints as this design is just a fully pipelined implementation without any resource sharing and with no partitions. It is evident that because of the use of partition ranking algorithm to select profitable partitions to be implemented, design points which would result in an increase in area from baseline design point would not be selected for implementation.

Fig. 5.19. Area-Clk Plot for DFG300

Fig. 5.20. Area-Clk Plot for DFG500

The run time for the proposed synthesis approach was obtained on a computer running 64-bit Windows 7 with 8 GB RAM, Intel(R) Core(TM)2 Duo CPU E8500 @ 3.16 GHz and the C-code implementation of the synthesis approach was executed in NetBeans IDE 7.1.2 [251]. For the biggest DFG i.e. DFG(500) run time for the partitioning algorithm was 5.72 seconds for II = 3 clock cycles, 7.85 seconds for II = 4 clock cycles and 8.32 seconds for II = 5 clock cycles. The run time for the partitioning algorithm is dependent on the size of the DFG, number of different types of operations in the DFG as well as the II constraint. It depends on the number of different types of operations in the DFG because for each unique operation type, a unique WOCG needs to be formed as mentioned in section 5.2 thus increasing the run time. The reason it depends on the II constraint is that the partition size (see Fig. 5.7) varies and as a result the time to traverse the entire DFG also varies. Our experiments demonstrate that there is no linear relationship between II constraint and run time because of the influence of other factors like the different types and number of operations present in different

partitions. The worst case computational complexity of the proposed algorithm is $O(C(V+E))$ where ‘C’ is the number of control-steps, V the number of vertices(nodes) and E the number of edges in the application DFG.

5.6 Summary

In this chapter, we have presented a DFG partitioning and a partition ranking algorithm to synthesize the datapath of DFGs under a given data initiation interval constraint. It can be clearly seen that this method is scalable to address DFGs of arbitrary size. The synthesized designs will meet the real world II constraint and will not instead pose a constraint on the system as is generally the case with traditional hardware binding approaches. This methodology also gives a design space to explore for a given II constraint as well as across different possible II constraint. This can greatly help in early system architecture when such constraints need to be defined. The partitioning algorithm will generate a number of partitions for a given DFG and a given II constraint. In the results shown above, a select number of partitions were selected for implementation. However, one can implement different combinations of partitions by making use of the result produced by partition ranking algorithm (e.g. Table 5.6) to get a denser area-clk tradeoff plot. The presented design methodology is particularly useful for those systems where a FPGA interfaces with a sensor that sends a new set of data every C clock cycles. This would act as II constraint. Thus, a datapath constructed with this sensing parameter in consideration and area reduction as a goal and using the proposed methodology will be better compared to the one which is unaware of such a sensing parameter. Unawareness of such parameters would result in over-engineered (fully pipelined, no resource sharing) or under-engineered (excessive resource sharing) solutions.

Chapter 6

Exploiting IP-Cores for Architecture and Application Aware Synthesis

6.1 Introduction

An application can have computational structures at different levels of granularity. For instance, it could make use of in-built function calls in the language of description (coarse granularity). There could be implementations of algorithms like FFT (coarser granularity) and finally, there will be atomic operations like addition, subtraction etc. (smallest granularity). This chapter focuses on the first two computational structures which are coarse grained and presents a methodology that seeks their automatic identification and subsequent binding to architecture specific resources at the same level of granularity. Resources like Intellectual property (IP) cores are at the same level of granularity. IP-core based design (section 2.3.2, chapter 2) and high level synthesis design methodology (section 2.3.3, chapter 2) both aim to make the task of designing a system easier. Therefore, a design methodology is presented that combines these two individual methodologies and is therefore more powerful. It is demonstrated that a careful analysis of *in-built functions in C-based languages* and *compiler specific pattern generation* for different types of computation structures can be used to bind these with architecture specific and optimized hardware circuits (IP-cores). The proposed methodology requires no alteration of original C/C++ source code. Relying on concepts from the domains of program recognition and optimized implementations of such arithmetic functions, the described design methodology is a step towards intelligent synthesis where application characteristics are matched with specific architectural resources and relevant IP cores in a transparent manner for improved area-delay results. The combined methodology is not only powerful in enabling

automatic IP reuse in HLS but also gives better results because it is *more aware* of the target hardware architecture than the conventional HLS flow.

A representative list of standard arithmetic in-built functions available in C/C++ is shown in Table 6.1. The complete list can be found in [219] and [220]. Traditional HLS flow involves the processes of resource allocation, scheduling and hardware binding. Every operation in the C/C++ description is treated as an atomic level operation and the bigger operations, like those involving the in-built functions, are replaced by their atomic level descriptions thus resulting in a flattened description of complete design. Atomic level operations refer to addition, subtraction, multiplication, comparison etc. This methodology completely ignores the fact that hardware design is different from software design.

Table 6.1. Representative list of standard arithmetic functions in C/C++

C/C++ in-built functions type	Available in <math.h> and <cmath.h>
General	div(), ldiv(), lldiv(), abs(), labs(), llabs() etc.
Exponential functions	exp(), exp2(), log(), log10(), log2() etc.
Power functions	sqrt(), cbrt(), hypot(), pow()
Trigonometric functions	sin(), cos(), tan(), acos(), asin() etc.
Hyperbolic functions	sinh(), cosh(), tanh(), asinh(), acosh() etc.
Error and Gamma functions	erf(), erfc(), lgamma(), tgamma()

Specifically, it ignores the fact that there can be optimized hardware implementations (IP cores) of those bigger operations. Therefore, it is unable to generate RTL microarchitecture that is closer in spirit to the target hardware; instead the generated RTL microarchitecture is closer in spirit to the software description. As a specific example, HLS flows assume that transcendental functions like sine(), cosine() etc. will be replaced by their equivalent polynomial expansions. Equations (6.1) and (6.2) show the polynomial expansion of sine() and cosine() operators respectively for an angle x radians.

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \quad \text{for all } x \quad (6.1)$$

$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots \quad \text{for all } x \quad (6.2)$$

Synthesizing such polynomials can not only be expensive in terms of area but also result in poor timing. The number of terms in these expansions affects the accuracy of the calculations and with bit growth allowed, such implementations can consume a lot of resources. It would be ideal if such operations are mapped to corresponding hardware IP cores which are typically optimized implementations. Thus, such an IP core can facilitate calculation over a wide range of input arguments without the designer having to worry about the number of terms. For other cases, like linear algebra operations involving matrix computations, FPGA device family specific IP cores are available which have been optimized for these device families [221].

6.2 Contributions

Methods and techniques to support hardware binding of in-built arithmetic functions in C/C++ and *compiler specific patterns for different computational constructs* are presented. For the case of *compiler specific patterns*, mapping of different linear algebra operations (like matrix addition, multiplication, subtraction and scaling) to architecture specific hardware IP cores is presented as an example. An organizational view of IP cores for the purpose of design space exploration is combined with an understanding of the nature of application obtained via program recognition and compiler specific pattern generation to generate improved area-delay results. A compiler optimization called “*Pointer Condense*” is also presented to deal with program variation.

A. Proposed IP-Core Aware HLS Design Flow

The proposed design flow is shown in Fig. 6.1 with matrix operations as an example and makes use of LLVM [81] compiler framework. LLVM generated control and dataflow graph (CDFG) forms the IR of an application. However, this design flow is not restricted to LLVM framework. Instead, this design flow is generic and serves to prove the concepts and the ideas presented in this chapter. The choice of a compilation

framework will depend on factors like its ability to extract parallelism, supported compiler optimizations etc. which we do not discuss in this thesis.

Fig. 6.1. Proposed IP-based HLS flow

6.3 Automatic Identification of Computation Blocks for Binding to IP-Cores

Previous work on IP-based hardware design and high level synthesis has been discussed in sections 2.3.2 and 2.3.3 in chapter 2. In this section, concepts as well as previous work related to *program recognition* and *compiler based pattern synthesis* are presented which are crucial to the idea of automatic identification of computation blocks for binding to IP-cores.

6.3.1 Program Recognition

Program recognition is a branch of study in computer science that seeks to understand what a program does by analyzing and studying the source code. It relies on *lexical and semantic analysis* of the source code as well as *algorithm recognition*. An

excellent reference in this area is the work on automatic algorithm recognition and replacement [222] which focuses on such efforts directed at software optimization. In the work proposed here, the same principle has been investigated but applied to hardware optimization in a high level synthesis flow. Lexical and semantic analysis of the source code can tell us which in-built functions available in a language have been used. Since the in-built functions are standard, their interpretation is also constant and does not lead to any *program variation* with respect to their functionality. For instance, the function `cos()` in C would always return the cosine value of a parameter of data type *integer* and the function `cosf()` in C would always return the cosine value of a parameter of data type *float*. Since the in-built functions implement some algorithm to complete some computational process, for instance calculating the cosine value of some parameter x , the identification of in-built functions can be equated with *algorithm recognition*. Once these functions are identified and the algorithms thus recognized, they can be matched to optimized hardware implementations which achieve the same purpose. It should be noted here that the *actual algorithm* used (in a source language like C) to complete a computational process is not important because the optimized hardware implementation may or may not have used the same algorithm.

In [223], the authors have investigated a new approach to DSP intrinsics for DSP processors. Here also they have tried to identify specific instructions from a compiler's IR and promote them to C-keywords. Intrinsics were developed because often an engineer would like to use a target specific instruction in code but there is no way to describe that instruction in C/C++. In its most basic form, an intrinsic is an *asm* statement as found in gcc [224]. This concept is similar to the fact that a HLS user may not be aware of the different optimized IP cores available for different sets of operations in C and that there is no way to describe them in a language like C/C+. In [225], a very recent work, the authors have looked at program recognition in order to parallelize sequential programs. They use a pre-compiler phase which processes specific keywords in the main source code and replaces chosen sequential algorithms with their parallel implementations selected from a database. They have assumed that

most programmers are either unable or unwilling to write parallel implementations of their sequential code. This assumption is valid even for HLS. The current work tries to address this assumption. Most HLS users might be comfortable in describing an algorithm in C/C++ but they would not be aware of corresponding hardware implementations because they are outside their domain expertise. Even for hardware designers who adopt HLS, the barrier that existing design flows present with respect to exploiting optimized IP cores gives rise to issues with performance optimization. It is only sufficient to mention that the workload of these two classes of designers can be significantly reduced if knowledge about IP cores is somehow embedded in the compilation phase instead of them having to adopt an ad-hoc way of identifying candidates for IP core implementation in the source code.

It has also been mentioned in [226] that program and algorithm recognition can play an important role in hardware-software co-design.

6.3.2 Compiler Specific Computational Patterns

Since in any HLS flow, there will be a compilation step involved, in the work presented in this chapter, a *compiler-specific* pattern finding and matching approach is proposed. Thus in the real world, any HLS vendor will only need to define a set of patterns which is specific to the underlying compilation process. These set of patterns can correspond to different computational structures like Fast Fourier Transform (FFT), color-space conversion etc. In this chapter, linear algebra operations involving matrix operations is chosen as an example to demonstrate the proposed idea under the LLVM compiler [81] framework. This approach clearly distinguishes the current work from that of others as mentioned in the next few paragraphs.

Pattern based high level synthesis has been investigated in [227] where the authors propose an algorithm to find recurring patterns in an application through subgraph enumeration, pruning and matching techniques in compute intensive applications. These patterns are then mapped to FPGAs. In [228], the authors have proposed a scheduling methodology based on behavioral templates which are essentially sub

graphs. Here also, the templates are created by enumerating the CDFG of an application. Research in [229],[230],[231],[232],[233] etc. are all focused on different methods to identify potential custom instructions based on graph processing techniques applied to the CDFG of an application. Thus this class of work is focused on finding *often repeated* patterns in application where the patterns are either generated by enumeration or taken from a library of available patterns and finally sub-graph matching is performed. However, the patterns used are small with maximum size being 40 nodes in [227]. Sub-graph matching has its own drawback as it requires the patterns in the library to be very precise. Also, the compilation step may generate a pattern which would be *functionally identical* to a pattern in a library but *structurally dissimilar* thus making it impossible to match. Also, since different compilers may generate the intermediate representation (IR) in different ways, patterns in a pre-defined and *compiler-unaware* library may have a reduced significance as such patterns may not appear in the IR. The problem with enumeration based pattern finding and matching is that even in this case the design flow has to rely on a pre-defined set of rules to enumerate patterns during the enumeration phase. This can lead to a large number of patterns which may not necessarily yield any advantage. Also, it is extremely difficult to construct a set of rules to describe a pattern that matches any IP core as the number of rules is likely to be large given the different options to configure and describe an IP core. The work closest to using the concept of *compiler-aware* library is [234] where the authors have looked at writing DSP intrinsics to the compiler's IR and promoting them as C-keywords for code targeted to a DSP processor.

6.3.3 Lexical & Semantic Analysis for In-Built C/C++ Functions

In this step, the symbol table generated as a result of lexical and semantic analyses phases of the compilation process is analyzed to locate calls to those in-built functions in C/C++ which are listed in Table 6.1. These function calls also show up in the IR. The lexical analysis phase parses the source code to generate identifiers like variable names, operation symbols, reserved words (like in-built functions) etc. The semantic

analysis phase analyses the context in which an identifier is used, for instance as a variable or a procedure etc. These help in generating the symbol table where each identifier is stored as a symbol along with its usage as a variable, procedure etc., its data type, number of and list of parameters if used as a procedure, return data type etc. Thus, at this step there is complete information about the different in-built functions that are used in a given C/C++ application.

6.3.4 Compiler Specific Patterns for Identification of Linear Algebra Operations

The linear algebra operations involving matrix operations are identified by processing the different basic blocks (BB) in the IR for nested loop identification and thereafter sub-graph matching as shown in the design flow. Each basic block is a sequence of operations and has only one entry and exit points. Matrix operations were chosen because they find widespread usage in different applications like computer graphics, molecular dynamics simulation [224] etc. In computer graphics, a 3D object can be modeled by a set of polygons and the vertices of these polygons can be stored in a matrix which can then undergo different spatial transformations like rotation, scaling etc. Similarly, in molecular dynamics simulations, the updated location of each molecule can be calculated by performing addition operation on a matrix containing their present location.

A. *Structure of Linear Algebra Operations in LLVM IR*

The matrix operation identification problem is solved for matrix multiplication of different kinds as well as matrix addition and subtraction. Matrix multiplication can be classified into a) matrix-matrix multiplication, b) matrix-vector multiplication and c) matrix scaling where a matrix is multiplied by a constant. Vector-vector multiplication is a special case of matrix-matrix multiplication. Figure 6.2 shows the *for-loop* code for matrix- matrix multiplication in C for 2 x 2 matrices. Figure 6.3 shows the code for

matrix-vector multiplication. Figure 6.4 shows the code for matrix scaling. For a 2 x 2 matrix-matrix addition, the *for-loop* is shown in Fig. 6.5.

```
for(i=0;i<2;i++){
  for(j=0;j<2;j++){
    for(k=0;k<2;k++){
      prod[i][j] = prod[i][j] + mat1[i][k] * mat2[k][j];
    }
  }
}
```

Fig. 6.2. For-loop code for 2 x 2 matrix-matrix multiplication in C

It is the same for matrix-matrix subtraction with addition operator replaced by subtraction operator. Figures 6.6, 6.7, 6.8 and 6.9 show the loop body IR as a directed acyclic graph (DAG) for the code segments in Figures 6.2, 6.3, 6.4 and 6.5 respectively. It can be seen that the actual multiplication or addition or subtraction code block is always in the innermost *for loop* of a nested block of *for* loops.

```
for(i=0;i<2;i++){
  for(k=0;k<2;j++){
    prod[i][0]=prod[i][0]+A[i][k]*B[k][0];
  } }
}
```

Fig. 6.3. For-loop code for a matrix-vector multiplication in C

```
for(i=0;i<2;i++){
  for(k=0;k<2;j++){
    A[i][k]=A[i][k]*scalefactor;
  } }
}
```

Fig. 6.4. For-loop code for scaling a 2 x 2 matrix in C

```
for(i=0;i<2;i++){
  for(k=0;k<2;j++){
    sum[i][j]=A[i][j] +B[i][j];
  } }
}
```

Fig. 6.5. For-loop code for 2 x 2 matrix-matrix addition in C

- *Loop Identification.* The first step in identifying matrix multiplication operations in the IR is to identify the loops. Loop identification is a well-established technique in compiler design [71]. The dominator tree analysis is

used for this purpose. For the different kinds of matrix multiplication (a, b or c mentioned in the beginning of this section), there are different levels of dominating loops with the inner most loop being most dominated. We use ' Lx ' where x is a number to indicate the level of domination of a loop with a higher value of x meaning a higher domination of the said loop by other loops. Thus, $L3$ (loop k) is the most dominated loop while $L1$ (loop i) is the least and $L1$ dominates both $L2$ (loop j) and $L3$ as in Fig. 6.2.

- *Matrix Size Calculation.* The size of the matrices being multiplied and of the vector in case of matrix-vector multiplication is determined from the loop iteration limits. Therefore, if the iteration limit for loops $L1$, $L2$ and $L3$ are $I1$, $I2$ and $I3$ respectively, then the matrix sizes i.e. total number of elements and the number of rows and the number of columns is given in Table 6.2.

Table 6.2. Matrix size determination for matrix-matrix multiplication

	Left matrix	Left matrix, rows	Left matrix, cols	Right matrix	Right matrix, rows	Right matrix, cols	Product matrix	Product matrix, rows	Product matrix, cols
Size	$I1 * I3$	$I1$	$I3$	$I3 * I2$	$I3$	$I2$	$I1 * I2$	$I1$	$I2$

In the case of matrix multiplied by a vector or a vector multiplied by a matrix, there are only two loops $L1$ and $L2$ with iteration limits $I1$ and $I2$ respectively. $L2$ is considered the innermost loop. In case of a matrix multiplied by a vector, the sizes are given in Table 6.3. In case of a vector multiplied by a matrix, the sizes are given in Table 6.4 and Table 6.5 shows the sizes for matrix scaling.

Table 6.3. Matrix size determination for matrix-vector multiplication

	Matrix	Matrix, rows	Matrix, cols	Vector	Vector, rows	Product matrix	Product matrix, rows
Size	$I1 * I2$	$I1$	$I2$	$I2 * 1$	$I2$	$I1 * 1$	$I1$

Table 6.4. Matrix size determination for vector-matrix multiplication

	Vector	Vector, cols	Matrix	Matrix, rows	Matrix, cols	Product matrix	Product matrix, cols
Size	$1 * I2$	$I2$	$I2 * I1$	$I2$	$I1$	$1 * I1$	$I1$

Table 6.5. Matrix size determination for matrix scaling

	Matrix	Matrix, rows	Matrix, cols	Product matrix	Product matrix, rows	Product matrix, cols
Size	$I2 * I1$	$I2$	$I1$	$I2 * I1$	$I2$	$I1$

In case of matrix-matrix addition or subtraction, there are only two loops $L1$ and $L2$ with iteration limits $I1$ and $I2$ respectively. $L2$ is considered the innermost loop. The sizes are determined as per equation (6.3).

$$S(matrix1) = S(matrix2) = S(sum\ matrix) = I1 \times I2 \quad (6.3)$$

Fig. 6.6. Directed acyclic graph (DAG) of IR for matrix-matrix multiplication

Fig. 6.7. Directed acyclic graph (DAG) of IR for matrix vector multiplication

Fig. 6.8. Directed acyclic graph (DAG) of IR for matrix scaling

Fig. 6.9. Directed acyclic graph (DAG) of IR for matrix-matrix addition/subtraction

GEP() stands for the “getelementpointer inbounds” instruction in LLVM. It is a memory instruction that fetches the address of the array locations which store the matrix elements.

B. Algorithm for Identification of Matrix Operations

Algorithm 1. Identification of Matrix Operations in LLVM IR

Input: CDFG of Application in C/C++

Output: CDFG with cluster of operations(sub-graph) identified as matrix operations

1. **For** each Basic Block
 2. Perform *Dominator Analysis* to identify nested loops
 3. **For** each set of nested loops
 4. Label loops using ‘ L_x ’ as defined in section 6.3.4. *A*.
 5. Select nodes in CDFG bound by loop with highest x value
 6. Perform *subgraph-matching* for subgraphs shown in Fig. 6.6, 6.7, 6.8, 6.9
 7. **If true** for Fig. 6.6
 8. Mark nested loop set as matrix-matrix multiplication
 9. **If true** for Fig. 6.7
 10. Mark nested loop set as matrix-vector multiplication
 11. **If true** for Fig. 6.8
 12. Mark nested loop set as matrix scaling
 13. **If true** for Fig. 6.9
 14. Mark nested loop set as matrix add/subtract
 15. **EndFor**
 16. **EndFor**
-

Fig. 6.10. Algorithm for identification of matrix operations in LLVM IR

Since the size of matrices can be determined as shown in Tables 6.2, 6.3, 6.4 and 6.5 the relevant IP cores can be configured automatically in the design flow. We discuss this in more detail in section 6.3.5.

C. Dealing with Program Variation

The directed acyclic graph representation (DAG) of IR shown in Figs. 6.6, 6.7, 6.8 and 6.9 represent the C/C++ code as shown in Figs. 6.2, 6.3, 6.4 and 6.5 respectively. A key issue to address here is the structure of the DAGs when code is not written as demonstrated in these figures or when additional statements are inserted in the loop body. These additional statements could be unrelated to matrix operations or the matrix operation could be executed on more than one matrices or they could be some form of manual optimization. The code segments shown in Figs. 6.2, 6.3, 6.4 and 6.5 are the easiest way of coding the relevant matrix operation from a software perspective. Thus, when one is building an executable specification of a design in C/C++ and using HLS to map it to hardware (FPGA or ASIC), it is not necessary to write an optimized C/C++ implementation because the target of such a design description is not a processor but mapping to hardware. The executable nature of the specification is needed only to verify the hardware mapping against execution results from the C/C++ description to ensure functional correctness. The matrix operation DAGs as shown earlier will always be present in the loop body DAG irrespective of any other code in the loop body. Thus the matrix operation DAGs can always be located in the innermost loop body IR.

- **Loop Fission.** Traditionally loop fission optimization can be applied to cases of more than one matrix multiplication (or any other matrix operation) being done in the same loop body, assuming that the relevant matrices are all of similar sizes. This would replicate the loop constructs while putting each matrix operation in a different loop construct. However, we do not adopt this approach because in our graph matching based approach more than one instance of matrix multiplication (or any other matrix operation) DAG will be easily found. Similarly, if the loop body code has a mix of different matrix operations, instances of each DAG type can be found.
- **Different Loop Constructs.** Loops can also be constructed using *do-while* and *while* constructs also in C/C++. For each of these constructs the loop body IR

is different from the loop body IR for *for*-loops. However, the matrix operation DAGs are still present in the innermost loop body IRs in the same form as shown in Figs. 6.6, 6.7, 6.8 and 6.9. Thus, it does not matter which loop construct is used to form the loops.

- **Loop Rerolling.** If the loop body is partially unrolled in the code, we apply the *loop-rerolling* [235] optimization. This is because loop unrolling obscures the underlying computation algorithm (for example matrix multiplication) and as such DAG matching will fail. Loop rerolling cannot be applied to completely unrolled loops because there is no loop in the code in such cases. The unrolling is visible to and understood only by the designer.
- **Data Structure Variation.** The C codes given for different matrix operations make use of arrays whose elements are accessed in the usual way of accessing array elements. However, there are two main variations in data structure: 1) the array elements can be accessed using pointers even when declared as arrays and 2) the array elements are accessed using pointers when the two dimensional matrices' arrays are declared using pointers. Typically, pointer based operations and matrices will be required only for code targeting a processor and involving large matrices. For HLS, one can safely stick to the usual access methods irrespective of the size of matrices. Nevertheless, we have addressed this issue for the sake of accommodating different programming styles by introducing a new compiler optimization pass specifically for HLS. Equation (6.4) shows three equivalent ways of accessing a memory element $A[i][j]$.

$$A[i][j] \equiv * (A[i] + j) \equiv * (* (A + i) + j) \quad (6.4)$$

These three different ways of accessing an array element leads to different numbers of GEP() instruction in LLVM IR. $A[i][j]$ involves two GEP() while $* (A[i] + j)$ requires three GEP() and $* (* (A + i) + j)$ needs four GEP() instructions. The easiest way to accommodate these program variations would be to define three different DAGs corresponding to each type of memory

access. However, that would add to the DAG database as well as lead to additional time in the execution of DAG matching. The additional increase in the size of the DAGs would also add to the execution time. Therefore, to deal with this program variation, we have developed a compiler optimization pass called *Pointer Condense*. This pass converts both $*(A[i] + j)$ and $*(A + i + j)$ to $A[i][j]$ during semantic analysis phase applied to the syntax tree using the symbol table when A is an array.

6.3.5 Data Structure for IP-Cores

This section deals with the data structure that is necessary to store and provide information on an IP-core's configurable as well as static attributes. The attributes are separated into *computation* and *non-computation* attributes. The computation attributes are those which are essential and necessary for computation operations within the IP core to take place. They include available input-output port bit-widths, supported rounding modes and any internal precision attribute. The non-computation attributes area (in terms of LUT when FPGA is the target architecture), maximum clock frequency, serial and parallel interfaces, latency, throughput etc. The list of computation and non-computation attributes is determined by the commonly used parameters for IP cores targeted in the work presented here. Attributes like input-output port bit-widths, supported rounding modes, serial and parallel interfaces, use of LUT or DSP blocks for multiplication etc. can be configurable depending upon the nature of an IP core. It is assumed that these attributes have already been extracted from the IP-XACT [30] description of the IP cores. We use these attributes to arrange the IP cores in an IP core library based on their maximum clock frequencies. IP cores without any configurable attribute have only one maximum frequency. We use the maximum clock frequency as the head of a linked-list that stores the IP attributes. For any function f which is a candidate for IP core implementation, the different IP cores can be described as shown in Table 6.7. Note that the number of available attributes for different IP cores need not be same. Some IP cores may have architecture attributes like serial or parallel. These are different from interface attributes. Not all

attributes are shown in Table 6.7 for brevity. Also, the IP core selection algorithm, as discussed in section 6.4, does not use any attribute information besides maximum clock frequency, rounding mode, input and output bit-widths.

Table 6.6. A function f and corresponding IP cores

Function	IP Cores
f	$T1 \rightarrow RND(M1) \rightarrow InputBW1 \rightarrow OutputBW1 \rightarrow None \rightarrow Serial Arch \rightarrow Area1 \rightarrow IPCore1$
f	$T2 \rightarrow RND(M1) \rightarrow InputBW2 \rightarrow OutputBW2 \rightarrow Serial I/F \rightarrow Area2 \rightarrow IPCore2$
f	$T2 \rightarrow RND(M2) \rightarrow InputBW2 \rightarrow OutputBW2 \rightarrow Serial I/F \rightarrow Area2 \rightarrow IPCore2$
f	$T3 \rightarrow RND(M2) \rightarrow InputBW3 \rightarrow OutputBW3 \rightarrow Parallel I/F \rightarrow Area3 \rightarrow IPCore2$
f	$T3 \rightarrow RND(M1) \rightarrow InputBW3 \rightarrow OutputBW3 \rightarrow Parallel I/F \rightarrow Area3 \rightarrow IPCore2$
f	$T4 \rightarrow RND(M1) \rightarrow InputBW4 \rightarrow OutputBW4 \rightarrow None \rightarrow Area4 \rightarrow IPCore3$
f	$T5 \rightarrow RND(M1) \rightarrow InputBW5 \rightarrow OutputBW5 \rightarrow None \rightarrow Parallel Arch \rightarrow Area5 \rightarrow IPCore4$

As can be seen in Table 6.6, there are a total of 4 different IP cores that can perform function f . IP Core2 has 2 configurations and each can support two different rounding modes RND(M1) and RND(M2) for a total of four configurations. The input and the output port bit-widths are different for these two configurations and so is the interface (serial vs. parallel). Attributes not related to the current design requirements are walked through but not considered while traversing the list.

Description like in Table 6.6 is maintained for each FPGA device architecture in the IP library and therefore the flow takes into account the fact that whether a particular function f has support for IP core in a target FPGA device or not.

A. IP Core Characterization

It should be pointed out here that rigorous characterization of IP cores is very important especially with respect to their timing characterization for various configurations. It is this information that is crucial in the design flow and it affects all stages, from IP core selection to the synthesis of interfaces to calculation of latency and throughput. As an example, Table 6.7 shows the different configurable parameters of the Xilinx CORDIC IP core [236].

Not all configurable parameters will be available for all functions supported by the Xilinx CORDIC IP core. At the same time, some parameters cannot be configured

when a particular parameter is selected. For instance, when configured for square root operation with 32 bit unsigned integer input, the output width is automatically set to 17 in the core and the architectural configuration is set to parallel.

Table 6.7. Configurable parameters in Xilinx CORDIC IP core

Configurable Parameter (<i>attribute</i>)	Possible Values (<i>attribute values</i>)(to select one)
Functional Selection	Rotate, Translate, Sin and Cos, Sinh and Cosh, Arc Tan, Arc Tanh, Square Root
Architectural Configuration	Word Serial, Parallel
Pipelining Mode	No pipelining, Optimal, Maximum
Data Format	Signed Fraction, Unsigned Fraction, Unsigned Integer
Phase Format	Radians, Scaled Radians
Input Width	Range 8 to 48
Output Width	Range 5 to 48
Round Mode	Truncate, Round Positive Infinity, Round Positive Negative Infinity, Nearest Even
Iterations	Range 0 to 48
Precision	Range 0 to 48
Coarse Rotation	Yes, No
Compensation Scaling	No Scale Compensation, LUT Based, BRAM based, Embedded Multiplier
Optional Pins	Handshake Signals(Chip Enable, CE; Synchronous Clear, SCLR; New Data, ND; Ready, RDY), X OUT, Y OUT, Phase Output

Table 6.8. Available configurable parameters for Xilinx CORDIC IP core implementing $\sqrt{}$ function

Configurable Parameter (<i>attribute</i>)	Possible Values (<i>attribute values</i>)(to select one)
Pipelining Mode	No pipelining, Optimal, Maximum
Data Format	Unsigned Fraction, Unsigned Integer
Input Width	Range 8 to 48
Round Mode	Truncate, Round Positive Infinity (RPI), Round Positive Negative Infinity (RPNI), Nearest Even (RNE)
Optional Pins	Handshake Signals(Chip Enable, CE; Synchronous Clear, SCLR; New Data, ND; Ready, RDY)

On the other hand, both input and output widths are configurable for $\sin()$ calculation. This nature of the IP core is typical of the way IP cores are developed and can be observed in many available IP cores. Table 6.8 shows the configurable parameters and possible values when Xilinx CORDIC IP core is used to map the $\sqrt{}$ function.

For experimental results reported in this chapter, we performed the characterization of this core for the different functions that were looked at viz. $\sqrt{}$, $\sin()$ and $\cos()$. It should be noted that the resource utilization for a given design does not vary across members of a device family because the basic LUT, FF etc. architecture remains the

same and hence logic synthesis and technology mapping yield similar results during characterization. For example, all Xilinx Virtex 4 device family members will use the same amount of resources for a particular design. It is only the timing performance which can vary from one member to another. Table 6.9 shows a representative characterization data for $\sqrt{}$ mapping.

Table 6.9. Representative characterization data for CORDIC configured for $\sqrt{}$

Configurable Parameter (<i>attribute</i>)	Attribute Value (for 32 bit unsigned integer) (XC4VFX12FF668-10)			
Pipelining Mode	No pipelining			
Round Mode	Truncate	RPI	RPNI	RNE
Latency Attained	2	2	2	2
Fmax Obtained (MHz) (ns)	45.535 (21.961)	37.038 (26.999)	37.038 (26.999)	33.357 (29.979)
LUT/FF	327/27	375/27	375/27	416/27
Pipelining Mode	Optimal			
Round Mode	Truncate	RPI	RPNI	RNE
Latency Attained	10	10	10	11
Fmax Obtained (MHz) (ns)	147.558 (6.777)	149.298 (6.698)	149.298 (6.698)	149.365 (6.695)
LUT/FF	351/201	399/227	399/227	440/239
Pipelining Mode	Maximum			
Round Mode	Truncate	RPI	RPNI	RNE
Latency Attained	17	18	18	19
Fmax Obtained (MHz) (ns)	280.426 (3.566)	264.131 (3.786)	264.131 (3.786)	263.227 (3.799)
LUT/FF	358/363	403/380	403/380	444/419

B. Scheduling in the Presence of IP Cores

Once the matrix operation patterns and the keywords have been identified and extracted from the CDFG IR, the CDFG needs to be scheduled. Every IP core will have latency and throughput attributes associated with it. These attributes can take up different values depending on whether the architecture of the core is configurable or not. Since the library of IP cores has information on IP cores in a very structured manner, the scheduling step can easily query the library. It is evident from an examination of commonly available IP cores provided by FPGA vendors not only for the class of operations discussed in this paper but also for other functions that the IP

cores can be sequential multicycle in nature or combinational multicycle. Therefore, scheduling algorithms like SDC-based scheduling [89] can easily take care of IP cores.

6.4 IP Core Selection

The IP core selection space is pruned in a hierarchical way by walking over their linked-list descriptions. First stage of pruning involves checking for the truth of the inequality given in equation (6.5) below where T_i is the maximum clock frequency of an IP core configuration.

$$T_i \geq T_{req} \quad (6.5)$$

The second stage of pruning involves checking for the truth of the equality given in equation (6.6).

$$RND(M_i) = RND(req) \quad (6.6)$$

These are applied successively on each linked-list. Those configurations that satisfy equations (6.5) and (6.6) are further examined for the cost metric called *Difference of Attributes (DoA)* as given in equation (6.7).

$$DoA(n_i, k) = \sum_{j=1}^t \frac{1}{j} [AV_k(j, 0) - AV_{n_i}(j, 0)] \quad (6.7)$$

Here, n_i refers to the node (or set of nodes) which are being mapped to an IP core and k is the k^{th} IP core that can implement the function in that node (or set of nodes). AV_k and AV_{n_i} are the attribute vectors that store the input port bit-widths and the output port bit-widths. The scaling factor of $1/j$ gives more weight to input port bit-widths because if input port sizes do not match then the IP configuration is unusable. The upper summation limit t is the sum of the numbers of input and output ports. We restrict the results in this chapter to two input and one output ports. The IP core configuration that yields the minimum *DoA* value is selected for mapping. This would lead to one design point corresponding to the selected IP core configuration. However, it should be pointed out here that this very approach can be used for a localized, IP

core level design space exploration. As long as $DoA > 0$, the corresponding IP core configuration can be used in the design.

A. Area Reduction by IP Core Resolution. Let there be operations $f1$ and $f2$ to be computed in the same control step in the schedule. Let these operations belong to Table 6.1 so that they are candidates for mapping to IP cores. It is possible that some IP cores can provide more than one output at the same time on the same set of inputs. For instance, there could be an IP core that can compute both sine and cosine simultaneously for a given input. An example of such an IP core is the CORDIC IP core provided by Xilinx [236]. In an application, if $f1$ and $f2$ are two such operations which are scheduled in the same control step and they share the same input, then they could be mapped to an IP core which computes both simultaneously. Therefore once the code candidates for mapping to IP cores have been identified (sections 6.3.3, 6.3.4), we implement a IP Core Dependency Resolution Algorithm (Algorithm 2) as shown in Fig. 6.11.

Algorithm 2. IP Core Dependency Resolution Algorithm

Input: List of Candidate functions for mapping to IP cores, their C-step number, their operands

Output: Candidate functions to be mapped to same IP core in the same C-step

```

1. For each c-step number in the list
2.   Check for candidate functions with same input operands
3.   If same input operands
4.     Check for IP core selected
5.     If same IP core selected
6.       Check if IP core is concurrent multi-output
7.       If true, use only one instance of the IP core
8.       Else use more than one instance of the IP core
9.       EndIf
10.    EndIf
11.  EndIf
12.EndFor

```

Fig. 6.11. IP Core dependency resolution algorithm

The reason for doing this is that it can lead to reduced area. Instead of using two different IP cores or two instances of the same IP core (because it can implement both functions $f1$ and $f2$), only one IP core can be used since the operations $f1$ and $f2$ are scheduled in the same control step and have the same inputs.

6.5 Interface Synthesis for IP-Core Mapped Computation Blocks

Since a part of the IR is sliced away for implementation using IP cores from a library while the remaining IR is implemented using the traditional atomic level hardware binding method, it is important to account for interface synthesis for the point where the IR was sliced away and the point where the IP core output joins the IR. The interface is characterized by *data interface* and *control interface*. Data interface describes the input-output port sizes, data types (signed, unsigned, single precision, double precision etc.) and whether the input-output ports are serial, parallel or some other configuration. Control interface describes the control signals necessary for input and output of data to and from the IP core.

Once an IP core is selected as described in section 6.4, interface synthesis takes place. If the operation identified for mapping to IP core is the *only* operation in the design, then different microarchitectures corresponding to the different interfaces supported by the selected IP core can be generated leading to an area-delay tradeoff curve. For instance, if a *matrix operation* is the only operation in the C/C++ description of the application and if the target device is a Xilinx Virtex-6, Virtex-7 or Kintex-7 FPGA, then the Xilinx linear algebra IP core supports both serial and parallel input-output for data. Thus, the design flow returns *different microarchitectures* corresponding to each interface provided equations (6.5) and (6.6) are satisfied. Additionally, if there exists additional IP core configuration attributes like serial or parallel architecture, the design flow returns even more microarchitectures provided again equations (6.5) and (6.6) are satisfied. As can be seen in Table 6.7, each possible configuration of an IP core that can implement a function f is stored as a linked-list in the IP library. Thus, the design flow returns as many microarchitectures as successful (satisfying (6.5) and (6.6)) configurations of one IP core or more. On the other hand, if the application has other operations besides those listed in Table 6.1, then the interface synthesis involves understanding the control and data flow at the points where the CDFG was sliced for the sliced portion to be implemented in an IP core. Therefore in this case, a further pruning of IP cores selected in section 6.4 is done. The input data to the IP core can

come from the previous computation nodes in the CDFG or they can come from some memory block. In either case, the latency and the throughput attributes of the IP core are taken into account. Note that latency and throughput are functions of the IP core's internal architecture which may be a configurable attribute (serial or parallel architecture). Typically, all IP cores will have an elementary set of control signals to signal to the sender and the receiver modules the readiness to receive new set of input data from the sender and the readiness to send a new set of valid output data to the receiver. This information related to the flow control mechanism of the selected IP core is used in the design flow to design the state machine for the datapath. A part of the state machine relevant to the IP core section of the design is shown in Fig. 6.12.

In Fig. 6.12, S(p) refers to the state machine state prior to IP core computation phase and S(l) refers to the state later than IP core computation phase. The format *a/b* stands for *input signal a to state* and *state output signal b* when *input signal a* is present. RFD (stands for ready for data) is a control signal that the IP core asserts when it is ready to receive a new set of input data. ND stands for new data which the previous state asserts while sending *data in* to the IP core.

Fig. 6.12. A part of the state machine for datapath in the presence of an IP core

As can be seen that there is only output control signal RDY (stands for ready with valid output) from the state in which IP core computation is performed. This signal is automatically asserted by the IP core after it has calculated the valid output data. Based on these control flow signals the state machine transitions from state S(p) to

S(c) to S(l). Any additional flow control signals that the IP core has can be easily integrated in the three states.

If the data to the IP core comes from memory blocks, then choosing between (or if possible both) serial and parallel data interface is a matter of how the memory blocks are arranged. The sequence of instructions $GEP > GEP > LOAD$ as present in the DAGs shown in Figs. 6.6, 6.7, 6.8 and 6.9 are actually memory read operations. In a typical HLS flow, the matrices will be stored in arrays and mapped to a memory, for example block RAM(BRAM) on Xilinx FPGAs. Thus, this sequence of instructions would translate into read from these BRAMs. For the case of different kinds of matrix operations, the depth of the BRAM will be equal to the size of the matrices as given in Tables 6.3, 6.4, 6.5 and 6.6 and equation (6.3). Depending on the type of memory (single port, simple dual port, true dual port, RAM/ROM), the interface for the IP core is selected. We do not discuss mapping of arrays to memory blocks and selection of appropriate memory blocks as it is outside the scope of the current work.

Since the actual mapping to an IP core is done during the hardware binding phase in HLS flow, the information on the type of memory used for matrices is already available. Hence, the interface in the selected IP core is appropriately chosen. If the memory is a single port RAM or a simple dual port RAM or a true dual port RAM, *serial interface* on the IP core is automatically selected. Absolute addressing is used to generate addresses that are needed to read from the input matrices' BRAM and to write to the output matrix BRAM. Address generation for reading from input matrices and writing to output matrices is part of state S(c) in the state machine in Fig. 6.12. The number of addresses to be generated for input and output matrices is given by the size of left, right and product matrices as given in Tables 6.3, 6.4, 6.5 and 6.6 and equation (6.3).

6.6 Experiments and Results

A set of compute kernels in different applications involving different in-built C/C++ functions and matrix operations were selected. Area and clock frequency results are

shown for the compute kernels when mapped using a commercial HLS tool-Vivado(v) HLS, and when mapped under proposed flow using IP cores to Xilinx FPGA devices. It should be noted here that the only reason to use a commercial HLS flow is because it embodies all the steps of high level synthesis. The results from the commercial HLS flow are not a reflection on the strength or weaknesses of the tool per se. Instead those results are reflection on the quality of results for some functions selected from Table 6.1 under the traditional synthesis flow which is IP core-unaware.

Table 6.10. Mapping of some representative in-built C functions

Design	XC4VFX12FF668-10								
	I/P Data	LUT	FF	DSP	SRL	BRAM	CLK (ns)	Latency /Through hput	IP Core
sqrt (v-HLS)	32 bit integer	3105	4308	0	135	0	5.419 ¹	75 ² /75	No
sqrt(proposed)	32 bit integer	358	363	0	0	0	5.323 ¹	17 ³ /1	CORDIC
div(v-HLS)	32 bit integer	4342	4105	0	136	0	7.572 ⁴	34 ² /34	No
div(proposed)-D1	32 bit integer	1230	1095	13	262/1230	0	5.674 ⁴	36 / 5	Divider ⁵
div(proposed)-D2	32 bit integer	1224	1011	13	262/1230	0	6.769 ⁴	34 / 5	Divider ⁵
div(proposed)-D3	32 bit integer	1150	937	13	234/1150	0	6.336 ⁴	30 / 5	Divider ⁵
div(proposed)-D4	32 bit integer	2279	6720	0	72/2279	0	6.820 ⁴	68 /1	Divider ⁵
div(proposed)-D5	32 bit integer	1284	3244	0	68/1284	0	5.607 ⁴	36/1	Divider ⁶
div(proposed)-D6	32 bit integer	1253	1971	0	4/1253	0	7.885	37/2	Divider ⁶
abs(v-HLS)	16 bit integer	51	19	0	0	0	2.462	1/1	No
abs(proposed)	16 bit integer	32	32	0	0	0	3.121	1/1	Yes

1.Clock period constraint of 4.5 ns with input jitter of 0.56ns (clock uncertainty).

2.Worst case and best case latency 3.Maximum latency

4.Clock period constraint of 8 ns with input jitter of 1ns (clock uncertainty).

5.CORDIC configuration of integer quotient and fractional remainder with maximum pipelining.

6.CORDIC configuration of integer quotient and integer remainder with maximum pipelining.

Table 6.10 shows the synthesis and implementation results for *sqrt()*, *abs()* and integer divide i.e. *div()* in-built functions in C which were implemented as top modules so that design space exploration could also be performed. Table 6.11 shows the results for matrix multiplication using the traditional flow (v-HLS) and the proposed flow. Table 6.12 shows the results for *exponential decay*(exp-decay)benchmark using single precision floating point (SPFP) data. Exponential decay

is given by equation (6.8) and is widely used in studying radioactive decay, RC-circuit analysis etc.

$$N_t = N_0 e^{-\alpha t} \quad (6.8)$$

In equation (6.8) N_t is some quantity measured at time t , N_0 is the same quantity measured at time $t = 0$, α is the decay constant and t is time.

Table 6.11. Mapping of matrix multiplication operation

Design (2 x 2 matrix multiplication; 16 bit integer elements)	XC4VFX12FF668-10								
	LU T	FF	DSP	BRA M	CLK (ns) ⁷	Latency /Throug hput ⁸	Wall Clock Time (ns)	IP Core	Comments
D1 (v-HLS)	31	62	1	0	2.856	70/70	199.9	No	External memory (EM)
D2 (v-HLS)	39	138	8	0	2.856	9/9	25.7	No	EM
D3 (v-HLS)	135	86	1	3	3.247	N.A	N.A	No	--
D4 (v-HLS)	129	198	8	3	3.112	N.A	N.A	No	--
D5 (v-HLS)	124	103	1	3	3.215	117/117	376.1	No	--
D6 (v-HLS)	155	214	8	3	2.988	59/59	176.2	No	--
D7 (proposed)	269	622	2	4	2.956	22/4	11.8	Yes	--
D8 (proposed)	514	1688	8	0	3.126	17/1	3.1	Yes	--
D9 (proposed)	183	576	4	7	3.005	17/2	3.0	Yes	--

7.Clock period constraint of 4 ns with 0.5 ns clock jitter.

8.The best case and the worst case latencies were the same for v-HLS flows.

Table 6.12. Mapping of exponential decay formula

Design	XC7VX330T-1FFG1157								
	I/P Data	LUT	FF	DS P	SRL	BRAM	CLK (ns) ⁹	Latency /Throug hput	IP Core
exp-decay (v-HLS)	SP FP	2656	2147	40	237	0	4.486	56/56	No
exp-decay (proposed)-D1	SP FP	4539	4679	8	39/ 4539	0	2.649	61/30	Yes
exp-decay (proposed)-D2	SP FP	4633	4807	6	65/ 4633	0	2.618	63/32	Yes
exp-decay (proposed)-D3	SP FP	4561	4566	8	17/ 4561	0	4.568	57/26	Yes
exp-decay (proposed)-D4	SP FP	4571	4675	6	45/ 4675	0	2.725	61/30	Yes
exp-decay (proposed)-D5	SP FP	4539	4679	8	39/45 39	0	2.649	26/20	Yes

9.Clock period constraint of 5 ns with zero input jitter.

Table 6.13 shows the synthesis and implementation results for a formula (equation (6.9)) used to calculate *great circle distance* (GC) [237]. This formula returns the central angle between two geographical locations and is widely used in global positioning systems (GPS) and is given in equation (6.9). Only the parameter of $\cos^{-1}$

has been synthesized to maintain fairness with the v-HLS tool which does not support synthesis of $\cos^{-1}$.

$$GC = \cos^{-1}(\sin(lat1) \times \sin(lat2) + \cos(lat1) \times \cos(lat2) \times \cos(lon1 - lon2)) \quad (6.9)$$

In equation (6.9), $lat1$ and $lat2$ stand for latitude1 and latitude2 respectively while $lon1$ and $lon2$ stand for longitude1 and longitude2 respectively and they correspond to two different geographical locations on the surface of the earth.

Table 6.13. Mapping of greater circle distance formula

Design	XC4VLX15-12SF363								
	I/P Data	LUT	FF	DS P	SRL	BRAM	CLK (ns) ¹¹	Latency /Through hput	IP Core
GC(v-HLS)	16 bit	22522	11937	68	0	20	5.31 (Est)	109/109	No
GC (proposed)-D1	16 bit	3113	3136	4	15/3113	0	5.47	24/1	CORDIC ¹⁰
GC (proposed)-D2	16 bit	1925	2062	4	15/1925	0	5.38	18/1	CORDIC ¹²
GC (proposed)-D3	16 bit	8567	8719	4	33/8567	0	5.45	50/1	CORDIC ¹³
GC(3-terms)	16 bit	894	3433	16	0	0	8.61	41/1	No
		3456	3664	0	0	0	9.95	41/1	No
GC(4-terms)	16 bit	2145	6729	21	84/2145	0	8.67	59/1	No
		5375	6959	0	84/5375	0	9.99	59/1	No

10. Parallel configuration, iteration and precision: automatic, coarse rotation.

11. Clock period constraint of 5.5 ns with 0 ns clock jitter.

12. Parallel architecture, iterations-8, precision: automatic, coarse rotation.

13. Parallel architecture, iterations-40, precision: automatic, coarse rotation.

A. Discussion of Results

In Table 6.10, it can be seen that when bound to the IP core, the synthesis and implementation results are better not only for area (LUT, FF etc.) and maximum clock frequency but also for latency. The square root function was mapped to a hardware IP core for Xilinx FPGAs. One main reason why the Vivado(v)-HLS flow resulted in high area values is because it converted 32-bit integer to 64 bit double while using the in-built C implementation of $\sqrt{}$ function. This clearly demonstrates that such an approach which ignores hardware level IP and instead relies on software level algorithmic description for in-built functions may not result in efficient synthesis and implementation. For the integer divide function $\text{div}()$, we again see that the proposed

IP based flow results in better area, clock , latency and throughput measures. It can also be seen that 6 different designs (D1 to D6) could be generated because at least 6 different configuration of the Xilinx divider IP core satisfied the conditions during IP selection phase as described in section 6.4.

Results in Table 6.11, which shows the synthesis and implementation results for matrix multiplication operations, show that it is possible to generate multiple designs even using v-HLS flow but these were obtained by *manually altering* the source code. Nevertheless, their latency and throughput results are very poor and some of them require memory blocks external to the compute block for the entire matrix multiplication exercise to yield results. On the other hand, graph based pattern matching resulted in three different designs corresponding to three different configurations of the Xilinx linear algebra toolkit IP core [221] *without altering* the source code. Also, their latency and throughput metrics were far better than the ones from traditional flow for the same clock period constraint though at the expense of some increase in area.

Results in Table 6.12 show that at least 5 different designs could be achieved using the different configuration of the IP core for $\exp()$ calculation and all satisfied the same clock period specification of 5 ns with no clock jitter (clock uncertainty). The v-HLS flow resulted in around 40% less LUTs but at least 400% more DSP blocks. On the other hand, the designs using proposed flow used more LUTs but a lot less DSP blocks. This demonstrates the fact the tradeoff between DSP block and LUT could be the result of variation in the internal design of the IP core and the type of C-implementation of $\exp()$ function. The IP core used for $\exp()$ calculation made use of CORDIC IP core for Xilinx FPGAs. Nevertheless, it is evident from the results that the designs with hardware IP core achieved better clock frequency with around 40% faster clock and around 50% higher throughput.

In Table 6.13, the v-HLS result shows the estimated (Est) clock period for synthesis and implementation of the *greater circle distance* benchmark. This is so because the LUT utilization of 22522 is so high that this design could not fit into the target device.

Again, these numbers are high because the c-HLS flow used single precision floating point for implementation as the available C-based implementations of $\sin()$ and $\cosine()$ use single precision floating point data type. Three designs were generated as per our proposed flow and they used only 4 DSP blocks while the LUT utilization varied from 3113 to 8567 depending on the IP core configuration. The IP core used was the Xilinx CORDIC IP core. All the designs could meet the clock specification of 5.5 ns period with zero clock jitter. When the polynomial expansion of $\sin()$ and $\cos()$ were used, the DSP utilization was 16 and 21 for 3 terms and 4 terms in the expansion respectively. When DSP blocks were not used, their LUT utilization was higher than two of the IP core based designs. However, the two polynomial expansion based designs did not meet the timing specification. The latency figures for the IP core based design are also much better ranging from 18 to 50 for the three design versus 109 for v-HLS and 41 and 51 for polynomial based synthesis. It should be noted here that in this particular case, algorithm for *area reduction by IP core resolution* (section 6.4, Algorithm 2) in the flow resulted in lesser area because $\sin()$ and $\cos()$ are calculated for both $lat1$ and $lat2$ in the benchmark. Hence, instead of using separate CORDIC IP core for each, which would have resulted in a total of 4 instances of the core, only 2 instances were used thus leading to area savings.

6.7 Summary

This chapter presents a framework for IP enabled high level synthesis and relevant methods and results to show that IP reuse and high level synthesis can be combined for better area-time results and more designs points *without any code restructuring* by a designer. It can be seen that the proposed approaches to deal with IP cores i.e. lexical and semantic analysis, and compiler aware pattern matching, can support all kinds of IP cores. The IP core view is an efficient abstracted view of the IP cores with only details relevant to high level synthesis. While the results in this chapter have been quoted for FPGAs, the framework as well as the techniques, including IP core characterization and view, are applicable to high level synthesis of ASICs also. The ASIC IP cores would require characterization for supported technology nodes in this

case. Also, since no source code restructuring or use of directives embedded in the source code is required (which is the case with all existing HLS flows), the proposed flow and methods are sure to increase designer productivity as well as acceptance of HLS design methodology. In addition to improving design performance, the presence of lower level optimized implementations of in-built functions in the IP library relieves a system designer from the worry of algorithms needed to implement such functions. This is particularly useful to those engineers who are more accustomed to software design and less to hardware design.

Chapter 7

Conclusion & Future Directions

This chapter presents the conclusion based on the work presented in this thesis and provides directions on future research in hardware binding during high level synthesis (HLS).

7.1 Conclusion

This thesis proposed novel methods for intelligent high level synthesis. These methods addressed some of the challenges in high level synthesis. These challenges were mentioned in the introductory chapters and were arrived at through a critical review of existing research literature in high level synthesis. The proposed methods and techniques have focused on applying the design principles of application and architecture awareness during the hardware binding phase of HLS. As a result, they lead to customized synthesis solutions.

Timing constraint was used as a knob by the proposed hardware binding algorithms to reduce resource utilization. The *Extended Compatibility Path based Binding* (ECPB) algorithm reduced resource utilization without a given timing constraint. Its extension, *latency preserving area-delay optimization algorithm*, reduced resource utilization by taking a given timing constraint into consideration. It exploits the available timing slack to reduce area without violating a given timing constraint. It was shown that consideration of both inter-operation and intra-operation dependencies lead to improved performance over state of the art as evidenced by the post place and route results: 12.49% and 29.21% on average lower area-delay product compared to compatibility path based (CPB) and weighted bipartite matching (WBM) based binding respectively. The ECPB algorithm helps to determine the best timing performance attainable with maximum area reduction while the latency preserving area-delay optimization algorithm determines the resource reduction achievable for a

given timing constraint. These findings are crucial for determining the implementation strategy for a design.

Dataflow-initiation-interval (II) aware graph partitioning algorithm was developed to partition an application's dataflow graph. Area reduction was further achieved by applying resource sharing algorithm like ECPB to each partition. These steps ensured that II-constraint was not violated and area was also reduced simultaneously. It was established that the proposed application of ECPB algorithm to individual partitions is significant because a uniform application of such resource reduction algorithms to an entire application is meaningful only when II-constraint does not exist. Design space exploration could also be done to generate multiple design points satisfying the constraint by deciding which partitions to select based on area savings within them. The experimental results also demonstrated the scalability of the algorithm to large sized dataflow graphs. It supports parallel processing of post partitioning for area reduction, thus reducing overall run time for the proposed methods. These methods are particularly suitable for designing hardware with just the right amount of area for digital signal and information processing applications where input data arrives at predefined intervals.

In order to create application awareness and exploit architectural information for maximizing timing performance, frequently executed code blocks in an application were analyzed for different combinations of multiplication and allied operations (like multiply-accumulate etc.). The proposed algorithm automatically identified these operations. These were then mapped to on-chip DSP blocks to maximize timing performance. This mapping was enabled by the proposed area modeling method which helped in judiciously choosing the operations. Experimental results show an increase in maximum operating frequencies of up to 3 times compared to a commercial HLS tool (Vivado-HLS). The non-simulation but mathematical reasoning based area modeling method for different kinds of multiplication operations keeps the estimation independent of different versions of vendor tools and free from the deficiencies of curve fitting based methods. Identification of these operations presents them as *anchor*

or hierarchical points in a large application so that the decision regarding their binding could be made judiciously.

The idea that the individual strengths of IP-core based design and high level synthesis can be brought together to generate a superior methodology was investigated in the work on automatic recognition of in-built arithmetic functions (in C/C++) and computational patterns. Relying on the concepts of *program recognition* and *automatic algorithm replacement* and the usage of a compiler, it was established that a library of *compiler specific computational patterns* can be relied upon to identify such patterns in an application. They are then bound to relevant IP-cores that already exist for efficient computation. Similarly, in-built arithmetic functions in C/C++ are bound to relevant hardware IP-cores instead of synthesizing from their equivalent polynomial based software implementations. This improved area (ranging from 60% to 75% reduced LUT) and clock period (reduction ranging from 16% to 40%) on benchmarks investigated compared to that possible using traditional HLS. The IP-core selection algorithm selected multiple IP-cores which also led to a greater number of functionally correct design points. This facilitates in the area-delay tradeoff analysis for making informed decisions. The proposed approach in this work has established yet another higher level of hierarchy in establishing *anchor or hierarchical points* in a large application to deal with efficient hardware binding of complex operations.

The significance of all these investigations lies not only in improved synthesis results but also the fact that no designer intervention, manual code restructuring or use of directives is required for the proposed methods. The implication of these findings is that an application can be analyzed automatically to identify different anchor points: coarse granularity code blocks related to in-built arithmetic functions and computational patterns, followed by relatively smaller granularity multiplication and allied operations and finally the remaining atomic compute operations. These can then be bound to architecture specific resources and IP-cores. The identification of anchor points can be put in a top-down hierarchical perspective. This analysis can be followed by dataflow graph partitioning and resource shared hardware binding bringing a constraints aware closure to the high level synthesis flow.

From an overall design methodology perspective, the proposed methods can be incorporated in a top-down hierarchical synthesis methodology. Here, the scheduled dataflow graph of an application can be first analyzed for identifying segments that could be bound to IP cores based on the methods described in Chapter 6. This can serve as the first level of hierarchy or anchor points. Thereafter, the dataflow graph can be further analyzed for multiplication and allied operations for binding to on-chip DSP blocks as described in Chapter 3. This can serve as the second level of hierarchy or anchor points. Once the candidates for binding to IP cores and on-chip DSP blocks have been identified this way, only atomic level compute operations remain to be bound to resources. Subsequently, the dataflow graph can be partitioned if there exists a given data initiation interval-constraint as described in Chapter 5 and resources shared to minimize area. Finally, resources are shared to further reduce area utilization under a timing constraint as described in Chapter 4. In this way, an application can be taken from its C/C++ description to hardware implementation in a top-down application and architecture aware synthesis methodology. While the proposed algorithms and methods are applicable to applications in any domain, digital signal and image processing applications stand to benefit more because of their richness in computational operations like multiplication and transcendental functions.. In this way, they make the high level synthesis methodology intelligent by bringing it closer in spirit to hardware design.

7.2 Future Research Directions

The work presented in this thesis has scope for future expansion. While this thesis has addressed certain gaps, future work could strengthen the overall design methodology by taking additional considerations which are highlighted below. Some of these considerations are related to hardware specific issues while others are related to compilation specific issues.

(1) Exploiting both Hard and Soft-IP Cores

While soft IP-cores have been introduced in Chapter 6 and DSP block as hard-IP core was discussed in chapter 3, judicious utilization of different types of

such cores is worthy of investigation. Since some functions could be implemented either as soft IP-core or hard IP-core, the effect of choosing either on the overall implementation could also be investigated. These investigations are likely to provide insights not only on design but also aspects of verification.

(2) Investigation of Compiler Specific Computational Structures and Complex Interface Protocols for Binding to IP-Cores

Compute operations and structures are wide and varied in scope as well as software level description. It is still a challenge to come up with graph or other suitable representations of different computational structures at the level of a compiler's intermediate representation and to establish their invariance or minimal deviation due to code variations. Essentially, this work would take further the notion of program recognition and automatic algorithm recognition. An ontological library of such structures will help in making HLS flow more transparent and easier for designers with minimal hardware knowledge.

Also, IP-Cores can have complex interface protocols. While the work presented in Chapter 6 of this thesis has explored automatic interface synthesis, complex interface protocols have not been considered. Such protocols are also going to affect the overall latency of the design as well as scheduling of operations in an application. Therefore, a precise and automated methodology is needed to enable the use of such IP cores which have complex interface protocols during high level synthesis.

(3) Investigating the Joint Criticality of Multiplication Operations and Multiply-Add/Subtract Operation Chains for Hardware Binding

The criticality of multiply-add/subtract chains has been investigated in this thesis (Chapter 3) with respect to multiplication operations in order to efficiently utilize the DSP blocks. However, it would be worthwhile to additionally investigate the effect of addition or subtraction operations as well

in respective chains to choose a suitable hardware binding strategy in FPGA. This can further strengthen the efficient utilization of DSP blocks. Also, while the proposed methodology in Chapter 3 adopts highest LUT consumption justified by the worst case design, a dynamic approach based on the actual bit-widths of operands could be investigated for improving efficiency and power.

(4) Tradeoff Analysis of on-chip resources between IP Core Candidates and Non-IP Core Candidates during Hardware Binding

IP cores also make use of on-chip resources like DSP blocks. In the work presented in this thesis, DSP block utilization by IP cores is treated as a black box and the left over DSP blocks are used for mapping multiplication and allied operations. However, it would be worthwhile to investigate trading off such resources between IP cores and non-IP core based operations where possible to see the impact of allocating or de-allocating such resources to IP cores on design performance.

(5) Power-Aware Area and Delay Optimization during Hardware Binding

The methods described in this thesis do not consider power directly. They are based on the premise that reduced resource utilization and appropriate selection of resources would lead to reduced power consumption. However, it would be worthwhile to investigate extensions of proposed methods when a power budget has been allotted to a design. Since the amount of switching activity within resources will depend on both their architecture and intermediate data, such considerations are important during hardware binding to manage power while reducing area and maximizing performance.

(6) Integrated Framework to Bring Together Different Design Constraints and Methods

Development of an integrated framework that takes into account different design constraints and the methods proposed in this thesis together with the future work identified in (1) – (5) would be beneficial. Such a framework

would make it possible to use these methods and see their cumulative impact in a HLS flow. The relative ordering of these methods in the framework is also an important investigation as this would affect the quality of results. Since this framework is driven by an application and architecture aware synthesis perspective, it could also be used to investigate appropriate hardware-software partitioning. This is likely to make hardware-software partitioning more deterministic and specific with respect to resources and performance in hardware like FPGA.

The future research directions as identified above could not only expand the scope of high level synthesis but also increase its value proposition for hardware design while helping to manage design complexity.

REFERENCES

- [1] ASICs and FPGAs for Telecom Applications: A Buyer Survey, Light Reading's Components Insider, vol. 2, no. 3, July 2006.
- [2] Brian Bailey, Grant Martin and Andrew Piziali, ESL Design and Verification: A Prescription for Electronic System Level Methodology. Morgan Kaufmann/Elsevier, 2007.
- [3] (2013) <http://en.wikipedia.org/wiki/2G>. (Last Accessed: October 28, 2013)
- [4] (2013) <http://en.wikipedia.org/wiki/3G>. (Last Accessed: October 28, 2013)
- [5] (2013) <http://en.wikipedia.org/wiki/4G>. (Last Accessed: October 28, 2013)
- [6] (2013) http://en.wikipedia.org/wiki/LTE_%28telecommunication%29. (Last Accessed: October 28, 2013)
- [7] (2013) <http://www.3gpp.org/>. (Last Accessed: October 28, 2013)
- [8] Yan J., Xu N.-Y, Cai X.-F., Gao, R., Wang, Y., Luo R. and Hsu, F.-H., "An FPGA-based accelerator for LambdaRank in web search engines", *ACM Transactions on Reconfigurable Technology and Systems*, vol. 4, issue 3, Aug 2011, article no. 25.
- [9] Dave, N., Fleming, K., King M., Pellauer, M. and Vijayaraghavan, M., "Hardware acceleration of matrix multiplication on a Xilinx FPGA", *Proc. 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE 2007)*, pp., 97-100, Nice, May 30 - June 2, 2007.
- [10] Draper, B.A., Beveridge, J.R., Bohm, A.P.W, Ross, C., Chawathe, M., "Accelerated image processing on FPGAs", *IEEE Transactions on Image Processing*, vol. 12, issue 12, pp. 1543-1551, Dec. 2003.
- [11] G. De Micheli, R. Ernst, W. Wolf, *Readings in Hardware/Software Co-Design*, 1st Ed., Morgan Kaufmann, 2001.
- [12] Teich, J., "Hardware/Software Codesign: the past, the present and predicting the future", *Proc. of IEEE*, vol. 100, pp. 1411-1430, May 2012.
- [13] P. Schaumont, *A Practical Introduction to Hardware/Software Codesign*, 2nd Ed., Springer, December 2012.

- [14] Vallejo, L. M., and Lopez, C. J., “On the hardware-software partitioning problem: system modeling and partitioning techniques”, *ACM Transactions on Design Automation of Embedded Systems*, vol. 8. no. 3, pp. 269-297, July 2003.
- [15] Eles, P., Peng, Z., Kuchcinski, K. and Doboli, K., “System level hardware/software partitioning based on simulated annealing and tabu search”, *Design Automation for Embedded Systems*, vol. 2, pp. 5-32, 1996.
- [16] Vanmeerbeek, G., Schaumont, P., Vernalde, S., Engels, M. and Bolsens, I., “Hardware/software partitioning of embedded system in OCAPI-xl”, *Proc. of 9th International Symposium on Hardware/Software Codesign (CODES 2011)*, Copenhagen, pp. 30-35.
- [17] (2013) <http://www.semtech.org/> (Last Accessed: October 28, 2013)
- [18] International Technology Roadmap for Semiconductors 2007 Edition: Design
- [19] Coussy, P., Gajski, D.D., Meredith, M. and Takach, A., "An introduction to high level synthesis", *IEEE Design & Test of Computers*, vol. 26, issue 4, July-Aug. 2009, pp 8-17.
- [20] (2013) <http://www.mathworks.com/products/hdl-coder/> (Last Accessed: October 28, 2013)
- [21] Bilavarn, S., Gognia, G., Philippe, L. J. and Bossuet, L., “Design space pruning through early estimations of area / delay trade-offs for FPGA implementations”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, issue 10, pp. 1950-1968, Oct. 2006.
- [22] Kuon, I. and Rose, J., “Area and delay trade-offs in the circuit and architecture design of FPGAs”, *Proc. of the 16th ACM/SIGDA Symposium on Field Programmable Gate Arrays (FPGA’08)*, pp. 149-158.
- [23] Zambreno, J., Nguyen, D. and Choudhary, A., “Exploring are/delay tradeoffs in an AES FPGA implementation”, *Proc. IEEE International Conference on Field Programmable Logic (FPL 2004)*, LNCS 3203, pp. 575–585, 2004
- [24] Xilinx System Generator for DSP User Guide, UG640 (v 14.1), April 24, 2012.
- [25] Altera DSP Builder Handbook Vol. 1: Introduction to DSP Builder, May 2013.
- [26] Tom Dewey, IP Reuse for FPGA Design: Rapidly Unravel Internal and Third-Party IP, Mentor Graphics, October 2002.

- [27] Sarkar, S., Chanciar, G.S. and Shinde, S., "Effective IP reuse for high quality SOC design", *Proc. of IEEE International SOC Conference*, pp. 217-224, Sept. 2005.
- [28] (2013) <http://en.wikipedia.org/wiki/EDIF> (Last Accessed: October 28, 2013)
- [29] (2013) <http://www.edn.com/electronics-news/4368295/SOCs-IP-is-the-new-abstraction> (Last Accessed: October 28, 2013)
- [30] IP-XACT STANDARD. IEEE 1685-2009. IEEE standard for IP-XACT, standard structure for packaging, integrating and reusing IP within tool flows
- [31] (2013)<http://www.w3.org/XML/> (Last Accessed: October 28, 2013)
- [32] (2013) http://www.xilinx.com/esp/video/refdes_listing.htm (Last Accessed: October 28, 2013)
- [33] Coussy P., Gajski D.D., Meredith M., Takach A., "An introduction to high level synthesis", *IEEE Design & Test of Computers*, vol. 26, issue 4, July-Aug. 2009, pp 8-17.
- [34] R. Gupta et al., *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*, Kluwer Academic Publishers, 2004.
- [35] Martin, E., Sentieys, O., Dubois, H. and Philippe, J.L. , "GAUT: An Architectural Synthesis Tool for Dedicated Signal Processors", *IEEE European Design Automation Conference*, EURO-DAC 1993, Hamburg, pp14-19.
- [36] Vivado-HLS User Guide, UG 910 (v 2013.2), June 19, 2013.
- [37] (2013) Catapult C website. [Online]. Available: <http://calypto.com/en/products/catapult/overview> (Last Accessed: October 28, 2013)
- [38] (2013) Cadence C to Silicon Compiler website. [Online]. Available: http://www.cadence.com/products/sd/silicon_compiler/pages/default.aspx (Last Accessed: October 28, 2013)
- [39] Betz, V., Marquardt, A. and Rose, J., *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, New York, Jan 1999.
- [40] Kuon, I., Tessier, R. and Rose J., "FPGA architecture: survey and challenges", *Foundations and Trends in Electronic Design Automation*, vol. 2, no. 2, pp. 135-253, 2007.
- [41] Virtex-5 FPGA XtremeDSP Design Considerations User Guide, UG193 (v3.5) January 26, 2012.

- [42] (2013) <http://www.altera.com/devices/fpga/stratix-fpgas/about/dsp/stx-dsp-block.html> (Last Accessed: October 28, 2013)
- [43] Virtex-4 FPGA Embedded Tri-Mode Ethernet MAC User Guide, UG704 (v2.2) February 22, 2010.
- [44] Virtex-6 FPGA Embedded Tri-Mode Ethernet MAC User Guide, UG368 (v1.3) March 1, 2011
- [45] (2013) <http://www.xilinx.com/technology/protocols/pciexpress.htm> (Last Accessed: October 28, 2013)
- [46] (2013) <http://www.altera.com/end-markets/wireline/applications/ethernet/wil-ethernet.jsp> (Last Accessed: October 28, 2013)
- [47] Guneyusu T., Kasper T., Novotny M., Paar C. and Rupp A., “Crytanalysis with COPACOBABA”, *IEEE Transactions on Computers*, vol. 57, no. 11, Nov’ 2008.
- [48] Park Y., Park K. J. J. and Mahlke S., “Efficient performance scaling of future CGRAs for mobile applications”, *Proc. of Field Programmable Technology Conference*, FPT 2012, pp. 335-342.
- [49] Sutter B.D., Raghavan P. and Lambrechts A., *Coarse-Grained Reconfigurable Array Architectures* in Handbook of Signal Processing, Bhattacharya S.S., Deprettere Ed F., Leupers R., Takala J. (Eds) pp. 449-484, 2010.
- [50] Fisher J., Faraboschi P. and Young C., “VLIW processors: once blue sky, now commonplace”, *IEEE Solid-State Circuits Magazine*, vol. 1, issue 2, pp. 10-17, Spring 2009.
- [51] Suh D., Kwon K., Kim S., Ryu S. and Kim J., “Design space exploration and implementation of a high performance and low area coarse grained reconfigurable processor”, *Proc. of 2012 International Conference on Field Programmable Technology* (FPT 2012), pp. 67-70, Seoul, Dec. 10-12, 2012.
- [52] Perry, S., “Model based design needs high level synthesis - a collection of high level synthesis techniques to improve productivity and quality of results for model based electronic design”, *Proc. of Design, Automation & Test in Europe Conference & Exhibition*, (DATE’09), pp. 1202-1207, Nice, April 2009.
- [53] Xilinx EDK Concepts, Tools and Techniques: A Hands-on Guide to Effective Embedded System Design, EDK 13.1, UG683, April 13, 2011.
- [54] Altera Qsys System Design Tutorial, TU-01006, April 2011.

- [55] Xilinx AXI Reference Guide, UG761 (v13.4), January 18, 2012.
- [56] Altera Avalon Interface Specifications, MNL-AVABUSREF-2.1, May 2013.
- [57] Cosuins, D.B., Rohloff, K., Peikert, C. and Schantz, R., “An update on SIPHER (Scalable Implementation of Primitives for Homomorphic EncRyption) - FPGA implementation using SIMULINK”, *Proc. of 2012 High Performance Extreme Computing (HPEC)*, pp. 1-5, Waltham, MA, Sept. 10-12, 2012.
- [58] (2013) <http://www.bdti.com/InsideDSP/2012/09/05/MathWorks> (Last Accessed: October 28, 2013)
- [59] Vivado Design Suite User Guide, High-Level Synthesis, UG902(v2013.1) March 20, 2013.
- [60] Edwards, S.A., "The challenges of synthesizing hardware from C like languages", *IEEE Design & Test of Computers*, September-October 2006, pp 375-386.
- [61] (2013) <http://www.cadence.com/Community/blogs/ii/archive/2010/06/30/what-language-is-best-for-high-level-synthesis.aspx> (Last Accessed: October 28, 2013)
- [62] Gajski, D., Austin, T. and Svoboda, S., “What input-language is the best choice for high level synthesis (HLS)?”, *Proc. of 47th ACM/IEEE Design Automation Conference (DAC 2010)*, pp. 857-858, Anaheim, CA.
- [63] Leuken, R.V., Leeuwen, T.V. and Arriens, H.L., “High level synthesis of asynchronous circuits from dataflow graphs”, *Integrated Circuit and System Design. Power and Timing Modeling , Optimization and Simulation*, Lecture Notes in Computer Science, vol. 6951, pp. 317-330, 2011.
- [64] Sparso, J., “Current trends in high level synthesis of asynchronous circuits”, *Proc. of 16th IEEE International Conference on Electronics, Circuits and Systems (ICECS 2009)*, pp. 347-350, Dec. 2009.
- [65] Badia, R.M. and Cortadella, J., “High level synthesis of asynchronous systems: scheduling and process synchronization”, *Proc. 4th European Conference on Design Automation*, 1993, with the European Event in ASIC Design, Paris, pp. 70-74, Feb. 1993.
- [66] Jens Sparso, Steve Furber. *Principles of Asynchronous Circuit Design- A Systems Perspective*. Springer, 2002.

- [67] Leeuwen, T.M. van, “Implementation and automatic generation of asynchronous scheduled dataflow graph”. M.Sc.Thesis, TU Delft. [Available]: <http://ens.ewi.tudelft.nl/Education/msctheses/vanleeuwen-thesis.pdf>
- [68] IEEE Standard for Standard SystemC[®] Language Reference Manual. IEEE Std 1666[™]-2011. 9th January 2012, IEEE.
- [69] Describing Synthesizable RTL in SystemC, version 1.0, May 2001, Synopsys. [Available]: http://www.es.ele.tue.nl/mininoc/doc/rtl_systemc.pdf
- [70] Schlebusch, H.-J., “SystemC based hardware synthesis becomes reality”, *Proc. of the 26th Euromicro Conference*, pp. 434, Maastricht, 2000.
- [71] A. V. Aho, M.S. Lam, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques & Tools*, 2nd Edition, Addison Wesley, 2006.
- [72] D. Gajski et al., *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.
- [73] (2013) <http://www.khronos.org/opencl/> (Last Accessed: October 28, 2013)
- [74] Implementing FPGA Design with the OpenCL Standard, Altera Whitepaper WP-01173-2.0, November 2012.
- [75] Jaaskelainen, P.O., de la Lama, C.S., Huerta, P. and Takala, J.H., “Open-CL based design methodology for application-specific processors”, *Proc. of 2010 International Conference on Embedded Computer Systems*, pp. 223-230, Samos, July 2010.
- [76] Papakonstantinou, A., Gururaj, K., Stratton, J.A., Chen, D., Cong, J. and Hwu, W.-M.V., “FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs”, *Proc. IEEE 7th Symposium on Application Specific Processors*, pp. 35-42, San Francisco, July 2009.
- [77] Cong, J., Liu, B., Prabhakar, R. and Zhang, P., “A study on the impact of compiler optimizations on high-level synthesis”, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, vol. 7760, pp. 143-157, 2013.
- [78] Huang, Q., Lian, R., Canis, A., Choi, J., Xi, R., Brown, S. and Anderson, J., “The effect of compiler optimizations on high-level synthesis for FPGAs”, *Proc. of IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM 2013)*, Seattle, pp. 89-96, April 2013.

- [79] (2013) Trimaran, A Research Compiler: www.trimaran.org. ((Last Accessed: October 28, 2013))
- [80] Tripp, J. L., Peterson, K. D., Ahrens, C., Poznanovic, J. D. and Gokhale, M. B., "Trident: an FPGA compiler framework for floating-point algorithms", *Proc. of the International Conference on Field Programmable Logic and Applications*, FPL 2007, pp. 317–322. [Available]:<http://trident.sourceforge.net/>
- [81] (2013) LLVM Compiler, <http://llvm.org/> (Last Accessed: October 28, 2013)
- [82] Prado, D.G., Ren, Q., Ciesielski, M., Guillot, J. and Boutillon, E., "Optimizing data flow graphs to minimize hardware implementation", *Proc. of Design Automation and Test in Europe*, 2009, DATE 2009, pp. 117-122.
- [83] Perez, J., Sanchez, P. and Fernandez, V., " Optimizing Data-Flow Graphs with min/max, adding and relational operations", *Proc. of Design Automation and Test in Europe*, 2010, DATE 2010, Dresden, March 2010, pp 1361-1364.
- [84] Ciesielski, M., Askar, S., Gomez-Prado, D. et al., "Data-Flow transformations using taylor expansion diagrams", *Proc. of Design Automation and Test in Europe*, 2007, DATE 2007, pp 455-460.
- [85] Cong, J. and Jiang, W., "Pattern-based behavior synthesis for FPGA resource reduction", *Proc. of 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays (FPGA'08)*. ACM, New York, NY, 107-116.
- [86] Ly, T., Knapp, D., Miller, R. and Macmillen, D., "Scheduling using behavioral templates", *Proc. of 32nd Conference on Design Automation (DAC'95)*, San Francisco, pp.101-106, 1995.
- [87] Christophe Bobda, *Introduction to Reconfigurable Computing: Architectures, Algorithms and Applications*, 1st Edition, Springer, 2007.
- [88] Rau, B.R., "Iterative modulo scheduling: an algorithm for software pipelining loop", in *Proc. of the 27th Annual International Symposium on Microarchitecture*, 1994, pp.63-74.
- [89] Cong, J. and Zhang, Z., "An efficient and versatile scheduling algorithm based on SDC formulation", *Proc. of 43rd ACM/IEEE Design Automation Conference*, San Francisco, pp. 433-438, 2006.
- [90] Francesca Rossi, Peter van Beek, Toby Walsh, *Handbook of Constraint Programming*, 1st Ed., Elsevier Science, 2006.

- [91] Kuchcinski, K., "Constraints-driven scheduling and resource assignment", *ACM Transactions on Design Automation of Embedded Systems*, vol. 8, issue 3, pp. 355-383, July 2003.
- [92] Lakshminarayana, G., Khouri, K.S. and Jha, N.K., "Wavesched: a novel scheduling technique for control-flow intensive designs", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, issue 5, pp. 505-523, May 1999.
- [93] Ku, D.C. and De Mitcheli, G., "Relative scheduling under timing constraints: algorithms for high-level synthesis of digital circuits", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, issue 6, pp. 696-718, June 1992.
- [94] Tseng, C.-J. and Siewiorek, D. "Automated synthesis of datapaths in digital systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 5, no. 3, July 1986, pp. 379-395.
- [95] Septien, J., Mozos, D., Tirado, F., Hermida, R. and Fernandez, M., "Heuristics for branch-and-bound global allocation," *Proc. of the conference on European Design Automation*, EURO-DAC '92, Los Alamitos, CA, USA. IEEE Computer Society Press, 1992, pp. 334-340.
- [96] Huang, C.-Y., Chen, Y.-S., Lin, Y.-L. and Hsu, Y.-C., "Data path allocation based on bipartite weighted matching," *Proc. of Design Automation Conference 1990*, Jun 1990, pp. 499-504.
- [97] Kim, T. and Liu, C., "An Integrated Data Path Synthesis Algorithm Based on Network Flow Method," *Proc. of the IEEE Custom Integrated Circuits Conference 1995*, May 1995, pp. 615-618.
- [98] Cong, J. and Xu, J. "Simultaneous FU and register binding based on network flow method," *Proc. of Design, Automation and Test in Europe 2008*, DATE '08, March 2008, pp. 1057-1062.
- [99] Kurdahi, F. J. and Parker, A. C., "Real: A program for register allocation," *Proc. of the 24th ACM/IEEE Design Automation Conference*, DAC '87, New York, 1987, pp. 210-215.
- [100] Chen, D. and Cong, J., "Register binding and port assignment for multiplexer optimization," *Proc. of the Asia and South Pacific Design Automation Conference*, ASPDAC 2004, Jan 2004, pp. 68-73.

- [101] Cong, J., Fan,Y. and Xu, J., " Simultaneous resource binding and interconnection optimization based on a distributed register-file microarchitecture", *ACM Transaction on Design Automation of Electronic Systems*, vol. 14, no. 3, May 2009, pp. 1-31.
- [102] Cong, J., Fan,Y. and Jiang, W., "Platform based resource binding using a distributed register-file microarchitecture", *Proc. of IEEE/ACM International Conference on Computer-Aided Design , 2006,ICCAD'06*, Nov 2006, pp. 709-715.
- [103] Kim,T. and Liu ,X., "Compatibility Path Based Binding Algorithm for Interconnect Reduction in High-Level Synthesis," *Proc. of IEEE/ACM International Conference on, Computer-Aided Design, 2007, ICCAD 2007*, Nov 2007, pp. 435-441
- [104] Seto, K., Nonka, Y., Maruizumi, T. and Y. Shiraki, "SAT-based resource binding for reducing critical path delays", *Proc. of International Conference on Field Programmable Logic and Applications, 2008, FPL 2008*, Heidelberg, Sept 2008, pp. 507-510.
- [105] Nayak,A., Haldar, M., Choudhary, A.and Banerjee, P., "Accurate area and delay estimators for FPGAs", *Proc. of Design Automation and Test in Europe Conference and Exhibition, 2002*, Paris, March 2002, pp. 862-869.
- [106] Xu, M. and Kurdahi,F.J., " Area and timing estimation for lookup table based FPGA", *Proc. of European Design and Test Conference, 1996*, Paris, March 1996, pp. 151-157.
- [107] Gupta, G. R., Gupta, M. and Panda, P.R., "Rapid estimation of control delay from high level specifications", *Proc. of 43rd IEEE/ACM Design Automation Conference, 2006*, San Francisco, July 2006, pp. 455-458.
- [108] Jiang,T., Tang, X. and Banerjee, P., "Macro models for high level area and power estimation on FPGAs", *Proc. of the 14th ACM Great Lakes Symposium on VLSI, GLSVLSI'04*, Boston, April 2004.
- [109] Menn, C., Bringmann,O. and Rosenstiel ,W., "Controller estimation for FPGA target architectures during high-level synthesis", *Proc. of the 15th International Symposium on System Synthesis, ISSS'02*, Kyoto, October 2002.
- [110] Jung, J. and Kim, T., "Timing variation-aware high level synthesis: current results and research challenges", *Proc. of IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, December 2008, pp1004-1007.

- [111] Jung, J. and Kim, T., "Timing variation-aware high level synthesis", *Proc. of IEEE/ACM International Conference on Computer-Aided Design, 2007, ICCAD 2007*, San Jose, Nov 2007, pp. 424-428.
- [112] Liou, J.-J., Cheng, K.-T., Kundu, S. and Kristic, A., "Fast statistical timing analysis by probabilistic event propagation," *Proc. of ACM/IEEE Design Automation Conference*, 2001, pp. 661-666.
- [113] Hung, W.-L., Wu, X. and Xie, Y., "Guaranteeing performance yield in high-level synthesis," *Proc. of ACM/IEEE International Conference on Computer-Aided Design, ICCAD 2006*, San Jose, Nov 2006, pp. 303-309.
- [114] Wang, F., Sun, G. and Xie, Y., "A variation aware high level synthesis framework," *Proc. of Design, Automation and Test in Europe, 2008, DATE 2008*, Munich, March 2008, pp. 1063-1068.
- [115] Wang, F., Xie, Y. and Takach, A., "Variation-aware resource sharing and binding in behavioral synthesis", *Proc. of Asia and South Pacific Design Automation Conference (ASP-DAC 2009)*, Yokohama, pp. 79-84, Jan. 2009.
- [116] A. Raghunathan, N. K. Jha, S. Dey. *High-level Power Analysis and Optimization*. Kluwer Academic Publishers, 1998.
- [117] Hwang, C.-T., Lee, J.-H. and Hsu, Y.-C., "A formal approach to the scheduling problem in high level synthesis", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, issue 4, pp. 464-475, April 1991.
- [118] Chen, D., Cong, J. and Fan Y., "Low-power high level synthesis for FPGA architectures", *Proc. of the 2003 International Symposium on Low Power Electronics and Design, ISLPED 2003*, pp. 134-139.
- [119] Chen, D., Cong, J. Fan, Y. and Zhang, Z., "High level power estimation and low-power design space exploration for FPGAs", *Proc. of Asia Pacific Design Automation Conference, ASP-DAC 2007*, Jan. 2007, Yokohama, pp. 529-534.
- [120] Zhong, L. and Jha, N.K., "Interconnect-aware low-power high-level synthesis", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, issue 3, pp. 336-351, March 2005.
- [121] (2013) http://en.wikipedia.org/wiki/Loop_optimization (Last accessed: October 28, 2013)
- [122] Morvan, A., Derrien, S. and Quinton, P., "Efficient nested loop pipelining in high level synthesis using polyhedral bubble insertion", *Proc. of 2011 International*

Conference on Field-Programmable Technology (FPT 2011), New Delhi, pp. 1-10, Dec. 2011.

- [123] Cong, J., Zhang, P. and Zou, Y., “Optimizing memory hierarchy allocation with loop transformations for high-level synthesis”, *Proc. of the 49th Annual Design Automation Conference (DAC’12)*, pp. 1233-1238, 2012.
- [124] Walid, M., Benabderrahmane, Pouchet, L.-N., Cohen, A. and Bastoul, C., “The polyhedral model is more widely applicable than you think”, *Proc. of the 19th Joint European Conference on Theory and Practice of Software, International conference on Compiler Construction (CC’10/ETAPS’10)*, in Lecture Notes in Computer Science, vol. 6011, pp. 283-303, 2010.
- [125] Derrien, S., Rajopadhye, S., Quinton, P. and Risset, T., *High-level synthesis of loops using the Polyhedral Model*, chapter 12 in High-Level Synthesis: from Algorithm to Digital Circuit. Philippe Coussy, Adam Moraweic (Eds.), Springer, 2008.
- [126] Wang, Y., Peng, L., Zhang, P., Zhang, C. and Cong, J., “Memory partitioning for multidimensional arrays in high level synthesis”, *Proc. of 50th ACM/EDAC/IEEE Design Automation Conference*, Austin, pp. 1-8, May 29-June 7, 2013.
- [127] Cong, J., Jiang, W., Liu, B., and Zou, Y., “Automatic memory partitioning and scheduling for throughput and power optimization,” in *Proc. ICCAD*, Nov. 2009, pp. 697–704.
- [128] Clauss, P. and Loechner, V., “Parametric Parametric Analysis of Polyhedral Iteration Spaces”, *Journal of VLSI signal processing systems for signal, image and video technology*, Volume 19, Issue 2, pp 179-194, 1998
- [129] Gong, W., Wang, G. and Kastner, R., “Storage assignment during high-level synthesis for configurable architectures”, *Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD ’05)*, pp. 3-6, Nov. 2005.
- [130] Seo, J., Kim, T. and Panda, P.R., “Memory allocation and mapping in high-level synthesis-an integrated approach”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, issue 5, pp. 928-938, Oct. 2003.
- [131] Pilato, C., Ferrandi, F. and Sciuto, D., “A design methodology to implement memory accesses in high-level synthesis”, *Proc. of 7th IEEE/ACM/IFIP international Conference on Hardware/Software Codesign and System Synthesis*, pp. 49-58, 2011.

- [132] Semeria, L., Sato, K. and De Micheli, G., "Synthesis of Hardware models in C with pointers and complex data structures", *IEEE Transactions on Very Large Scale Integrations (VLSI) Systems*, vol. 9, issue 6., pp. 743-756, Dec. 2001.
- [133] Semeria, L. and De Micheli, G., "SpC: Synthesis of Pointers in C application of pointer analysis to the behavioral synthesis from C ", *Digest of Technical Papers, 1998 IEEE/ACM International Conference on Computer-Aided Design*, San Jose, pp. 340-346, Nov. 1998.
- [134] Middendorf, L., Bobda, C. and Haubelt, C., "Hardware synthesis of recursive functions through partial stream rewriting", *Proc. of 49th ACM/EDAC/IEEE Design Automation Conference*, San Francisco, pp. 1203-1211, June 2012.
- [135] Ferizis, G. and Gindy, H.E., "Mapping recursive functions to reconfigurable hardware", *Proc. of International Conference on Field Programmable Logic and Applications (FPL'06)*, Madrid, pp. 1-6, Aug. 2006.
- [136] Sklyarov, V., "FPGA-based implementation of recursive algorithms", *Microprocessors and Microsystems*, vol. 28, issues 5-6, pp. 197-211, August 2004.
- [137] Ghica, D.R., Smith, A. and Singh, S., "Geometry of synthesis iv: compiling affine recursion into static hardware", *Proc. of the 16th ACM SIGPLAN International Conference on Functional Programming*, pp. 221-233, ACM SIGPLAN Notices, vol. 46, issue 9, pp. 221-233, Sept. 2011.
- [138] Reynolds, J.C., "Syntactic control of interference", *ACM Symposium on Principles of Programming Languages (POPL)*, pages 39-46, 1978.
- [139] McCusker, G., "A graph model for imperative computation", *Logical Methods in Computer Science*, vol. 6, issue 1, 2010.
- [140] Constantinides, G. and Woeginger, G., "The complexity of multiple wordlength assignment", *Applied Mathematical Letters*, vol. 15, issue 2, Feb 2002, pp.137-140.
- [141] Cong, J., Fan, Y. Han, G. et al., "Bit-width aware scheduling and binding in high level synthesis", *Proc. of Asia and South Pacific Design Automation Conference, 2005*, vol.2, Jan 2005, pp. 856-861.
- [142] Enabling High-Precision DSP Applications with the FPGA Industry's First Variable-Precision Architecture, Altera WP-01131.

- [143] Cantin, M. -A., Savaria, Y., Prodanos, D. and Lavoie, P., "A metric for automatic word-length determination of hardware datapaths", *IEEE Transactions on Computer Aided Design of Integrated Circuits*, vol. 25, no. 10, October 2006, pp. 2228-2231.
- [144] CAS Standards Committee of the IEEE Circuits and Systems Society, IEEE Standard Specifications for the Implementations of 8×8 Inverse Discrete Cosine Transform, 1991, IEEE Std 1180-1990.
- [145] Yli-Kaakinen, J. and Saramaki, T., "An efficient algorithm for the design of lattice wave digital filters with short coefficient wordlength," *Proc. Int. Symposium Circuits and Systems*, vol. 3, Orlando, May 1999, pp. 443–448.
- [146] Lim, I.-T. and Bahn, J., "Optimal wordlength determination of AC-3 decoding hardware based on fixed-point analysis and simulations of AC-3 algorithm," *Proc. IEEE Workshop Signal Processing*, Leicester, U.K., Nov. 1997, pp. 301–310.
- [147] Kum, K., and Sung, W., "Combined word-length optimization and high-level synthesis of digital signal processing systems," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 20, no. 8, August 2001, pp. 921-930.
- [148] Abdul Gaffar, A., Mencer, O., Luk, W., Cheung, P. Y. and Shirazi, N., "Floating-point bitwidth analysis via automatic differentiation," *Proc. of IEEE International Conference on Field Programmable Technology (FPT)*, December 2002, pp. 158-165.
- [149] Roy, S. and Banerjee, P., "An algorithm for trading off quantization error with hardware resources for MATLAB-based FPGA design," *IEEE Transactions on Computers*, vol. 54, no. 7, July 2005, pp. 886-896.
- [150] Gal, B. L., Andriamisaina, C. and Casseau, E., "Bit-width aware high level synthesis for digital signal processing systems", *Proc. of IEEE International SOC Conference*, 2006, Taipei, September 2006, pp. 175-178.
- [151] Osborne, W. G., Cheung, R.C.C., Coutinho, J.G.F. et al., "Automatic accuracy-guaranteed bit-width optimization for fixed and floating point systems", *Proc. of International Conference on Field Programmable Logic and Applications*, 2007, Amsterdam, Aug 2007, pp. 617-620.
- [152] L. Zhang, Y. Zhang, W. Zhou, "Fast trade-off evaluation for digital signal processing systems during wordlength optimization", *Proc. of IEEE/ACM*

International Conference on Computer Aided Design, ICCAD 2009, San Jose, November 2009, pp. 731-738.

- [153] Ahmadi, A. and Zwolinski, M., "Symbolic noise analysis approach to computational hardware optimization", *Proc. of 45th ACM/IEEE Design Automation Conference, 2008*, Anaheim, June 2008, pp. 391-396.
- [154] Reg Allenby, *Rings, Fields, and Groups: An Introduction to Abstract Algebra*, 2nd Edition, Butterworth-Heinemann, 1991.
- [155] Gopalakrishnan, S. and Kalla, P., "Optimization of polynomial datapaths using finite ring algebra", *ACM Transactions on Design Automation of Electronic Systems*, vol. 12, issue 4, September 2007.
- [156] Gokhale, M., Stone, J., Arnold, J. and Kalinowski, M., "Stream oriented FPGA computing in the Streams-C high level language", *Proc. of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2000*, Napa, April 2000, pp. 49.
- [157] Keinert, J., Dutta, H., Hannig, F. et al., "Model-based synthesis and optimization of static multi-rate image processing algorithms", *Proc. of Design Automation and Test in Europe, 2009, DATE 2009, Nice, April 2009*, pp. 135-140.
- [158] Sundaraman, S., Govindarajan, S. and Vemuri, R., "Application specific macro based synthesis", *Proc. of 14th International Conference on VLSI Design*, Bangalore, 2001, pp. 317-324.
- [159] Bruno T. Messmer, *Efficient Graph Matching Algorithms for Preprocessed Model Graphs*, PhD thesis, University of Berne, November 1995.
- [160] Huang, R. and Vemuri, R., "Forward-looking macro generation and relational placement during high level synthesis to FPGAs", *Proc. of 18th International Parallel and Distributed Processing Symposium, 2004, IPDPS 2004, April 2004*, pp. 139.
- [161] Fan, K., Kudlur, M., Park, H. and Mahlke, S., "Increasing hardware efficiency with multifunction loop accelerators", *Proc. of the 4th International Conference on Hardware/Software Codesign and System Synthesis, CODES +ISSS 2006*, pp. 276-281.
- [162] S. Theodoridis, K. Koutroumbas. *Pattern Recognition*. Academic Press, 1999.

- [163] Lam, S. K., Srikanthan, T. and Clarke, C. T., "Architecture-aware technique for mapping area-time efficient custom instructions onto FPGAs", *IEEE Transactions on Computers*, vol. 60, issue 5, May 2011, pp. 680-692.
- [164] Lam, S. K., Srikanthan, T. and Clarke, C. T., " Selecting profitable custom instructions for area–time-efficient realization on reconfigurable architectures", *IEEE Transactions on Industrial Electronics*, vol. 56, issue 10, October 2009, pp. 3998-4005.
- [165] Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Czajkowski, T., Brown, S.D. and Anderson, J.H., "LegUp: an open source high-level synthesis tool for FPGA-based processor/accelerator Systems", *Proc. of the 19th ACM/SIGDA International Symposium on Field programmable Gate Arrays (FPGA'11)*, pp. 33-36.
- [166] Nikhil, R.S., *Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions*, pp. 129-146, Chapter in *High-Level Synthesis: from Algorithm to Digital Circuit*. Philippe Coussy, Adam Moraweic (Eds.), Springer, 2008.
- [167] (2013) <http://www.bluespec.com/high-level-synthesis-tools.html> (Last Accessed; October 28, 2013)
- [168] (2013) http://www.mpsoc-forum.org/previous/2003/slides/soc_verification.pdf (Last Accessed; October 28, 2013)
- [169] (2013) http://en.wikipedia.org/wiki/Functional_verification (Last Accessed; October 28, 2013)
- [170] (2013) <http://www.mcsoc-forum.org/2012/files/MCSOC12-Tahar.pdf> (Last Accessed; October 28, 2013)
- [171] (2013) http://www.cerc.utexas.edu/~jay/fv_surveys/McFarland.pdf (Last Accessed; October 28, 2013)
- [172] (2013) http://en.wikipedia.org/wiki/Formal_verification (Last Accessed; October 28, 2013)
- [173] Ayewah, N., Kikkeri, N., Seidel, P.-M. and Beyer, S., "Challenges in the formal verification of complete state-of-the-art processors", *Proc. of 2005 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 603-606, Oct. 2005.

- [174] McMillan, K.L., “A methodology for hardware verification using compositional checking”, *Science of Computer Programming*, vo. 37, issues 1-3, May 2000, pp. 279-309.
- [175] Kitchen, N. and Kuehlmann, A., “Stimulus generation for constrained random simulation”, *Proc. of the 2007 IEEE/ACM International Conference on Computer-Aided Design (ICCAD’07)*, pp. 258-265.
- [176] Xie, F., Yang, G. and Song, X., “Component-based hardware/software co-verification”, *Proc. of 4th ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, 2006 (MEMOCODE’06), pp. 27-36, July 2006, Napa, CA.
- [177] Varma, D., Mackay, D. and Thiruchelvam, P., “Easing the verification bottleneck using high level synthesis”, *Proc. of 28th VLSI Test Symposium*, pp. 253-254, Santa Cruz, CA.
- [178] Jha, P.K. and Dutt, N.D., “Rapid estimation for parameterized components in high-level synthesis”, *IEEE Transactions on Very Large Scale Integration Systems*, vol. 1, issue 3, pp. 296-303, August 2002.
- [179] Schafer, B.C., Takenaka, T. and Wakabayashi, K., “Adaptive simulated annealer for high level synthesis design space exploration”, *Proc. of International Symposium on VLSI Design, Automation and Test (VLSI-DAT’09)*, pp. 106-109, Hsinchu, April 2009.
- [180] Schafer, B.C. and Wakabayashi, K., “Machine learning predictive modelling high-level synthesis design space exploration”, *IET Computers & Digital Techniques*, vol. 6, issue 3, pp. 153-159, May 2012.
- [181] Chen, D., Cong, J., Fan, Y. and Zhang, Z., “High level power estimation and low power design space exploration for FPGAs”, *Proc. of Asia and South Pacific Design Automation Conference*, 2007 (ASP-DAC’07), pp. 529-534, Yokohama, Jan. 2007.
- [182] Liu, H.-Y., Petracca, M. and Carloni, M.P., “Compositional system-level design exploration with planning of high-level synthesis”, *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2012, pp. 641-646, Dresden, March 2012.
- [183] Inagi, M., Takashima, Y., Nakamura, Y., and Kajitani, Y., “A performance driven circuit bipartitioning algorithm for multi-FPGA implementation with time-

- multiplexed I/Os”, *Proceedings of the IEEE Intl. Conf. on Field Programmable Technology*. IEEE, Piscataway, NJ. 361-364.
- [184] Roncheng, Z., Jiarong, T., and Pushan, T., “FPART: A multi-way FPGA partitioning procedure based on the improved FM algorithm”, *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC’98)*. IEEE, Piscataway, NJ. 513-518.
 - [185] Srinivasan, V., Govindarajan, V., and Vemuri, R., “Fine grained and coarse grained behavioral partitioning with effective utilization of memory and design space exploration for multifpga architectures”, *IEEE Transactions on VLSI* 9,1. 140-158.
 - [186] (2013) <http://www.microsemi.com/products/fpga-soc/design-resources/design-software/libero-soc> (Last Accessed; October 28, 2013)
 - [187] (2013) http://www.altera.com/products/software/flows/asic/qts-structured_asic.html (Last Accessed; October 28, 2013)
 - [188] (2013) <http://www.toshiba-components.com/ASIC/structuredasic.html> (Last Accessed; October 28, 2013)
 - [189] (2013) http://en.wikipedia.org/wiki/Molecular_dynamics (Last Accessed; October 28, 2013)
 - [190] Kondratyev, A., Lavagno, L., Meyer, M., and Watanabe, Y., “Exploiting area/delay tradeoffs in high-level synthesis”, in *Proceedings of DATE, 2012*, pp. 1024-1029.
 - [191] Kulkarni, D., Najjar, W.A, Rinker, R. and Kurdahi, F.J., “Compile-time area estimation for LUT-based FPGAs”, *ACM Transactions on Design Automation of Electronic Systems*, vol. 11, no. 1, January 2006, pp. 104-122.
 - [192] Jiang, T., Tang, X. and Banerjee, P., “Macro-models for high level area and power estimation on FPGAs”, *Proc. of the 14th ACM Great Lakes Symposium on VLSI (GLSVLSI’04)*, pp. 162-165.
 - [193] Xtreme DSP for Virtex-4 FPGAs User Guide, UG073(v2.7), May 15, 2008
 - [194] Virtex-6 FPGA DSP48E1 Slice User Guide, UG369(v1.3), February 14, 2011.
 - [195] Elena Jean Folgo, *Accelerating Time to Market in the Global Electronics Industry*, MIT Sloan School of Management, May 9, 2008.

- [196] (2013) <http://www.xilinx.com/applications/index.htm> (Last Accessed; October 28, 2013)
- [197] (2013) <http://www.altera.com/end-markets/end-index.html> (Last Accessed; October 28, 2013)
- [198] Wirthlin, M. J., "Constant coefficient multiplication using look-up tables", *Journal of VLSI Signal Processing Systems*, Vol. 36, Issue 1, pp. 7-15, 2004.
- [199] Boullis, N. and Tisserand, A., "Some optimization of hardware multiplication by constant matrices", *IEEE Transactions on Computers*, Vol. 54, Issue 10, pp. 1271-1282, 2005.
- [200] Meher, P. K., "LUT optimization for memory based computation", *IEEE Transactions on Circuits and Systems II: Express Briefs*, Vol. 57, Issue 4, pp. 285-289, 2010.
- [201] Voronenko, Y. and Puschel, M., "Multiplierless multiple constant multiplication", *ACM Transactions on Algorithms*, Vol. 3, Issue 2, Article No. 11, 2007.
- [202] Tummeltshammer, P., Hoe, J. C. and Puschel, M., "Time-multiplexed multiple-constant multiplication", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 26, Issue 9, pp. 1551-1563, 2007.
- [203] Gutierrez, R., Valls, J. and Perez-Pascual, A., "FPGA-implementation of time-multiplexed multiple constant multiplication based on carry-save arithmetic", *Proc. of FPL 2009*, pp. 609-612, 2009.
- [204] Demirsoy, S. S., Kale, I. and Dempster, A. G., "Synthesis of reconfigurable multiplier blocks:part I-fundamentals", *Proc. of ISCAS 2005*, pp. 536-539, 2005.
- [205] Demirsoy, S. S., Kale, I. and Dempster, A. G., "Synthesis of reconfigurable multiplier blocks:part I- algorithm", *Proc. of ISCAS 2005*, pp. 540-543, 2005.
- [206] de Dinechin, F. and Pasca, B., "Large multipliers with fewer DSP blocks", *Proc. of FPL 2009*, pp. 250-255, 2009.
- [207] de Dinechin, F. and Pasca, B., "Large multipliers with less DSP blocks", LIP Research Report RR2009-03.
- [208] Karatsuba, A. and Ofman, Yu., "Multiplication of multidigit numbers on automata", *Soviet Physics Doklady*, Vo. 7, pp. 595

- [209] Implementing Multipliers in FPGA Devices, Altera Application Note, AN 306(ver 3.0), July 2004.
- [210] Verma, A. K. and Ienne, P. , “Towards the automatic exploration of arithmetic-circuit architectures”, *Proc. of DAC’06*, pp. 445-450, 2006.
- [211] Ruiz-Sautua, R., Molina, M.C., Mendias, J. M. and Hermida, R., “pre-synthesis optimization of multiplications to improve circuit performance”, *Proc. of DATE’06*, pp. 1-6, 2006.
- [212] Sivaraman, M. and Aditya, S., “Cycle-time aware architecture synthesis of custom hardware accelerators”, *Proceedings of the 2002 International Conf. on Compilers, architecture and synthesis of embedded systems (CASES’02)*. ACM, New York, NY. 35-42.
- [213] Zaretsky, D.C., Mittal, G., Dick, R.P., and Banerjee, P., “Balanced scheduling and operation chaining in high-level synthesis for FPGA designs”, *Proceedings of the 8th International Symposium on Quality Electronic Design, 2007 (ISQED ’07)*. IEEE, Piscataway, NJ. 595-601.
- [214] C. Lee, M. Potkonjak, and Mangione-Smith, W., “MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems,” *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, Dec 1997, pp. 330-335.
- [215] (2013) ExPRESS DFG Benchmarks. ExPRESS Specifications. <http://express.ece.ucsb.edu/benchmark> (Last Accessed: October 28, 2013)
- [216] T. Cormen, C Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms, 2nd Edition*. The MIT Press, 2001.
- [217] Huang, Y., Ienne,P. Temam, O., Chen, Y. and Wu, C., “Elastic CGRAs”, *Proceedings of the ACM’SIGDA International Symposium on Field Programmable Gate Arrays (FPGA’13)*, pp. 171-180.
- [218] Dhawan,U., Sinha, S., Lam, S.-K. and Srikanthan, T., “Extended compatibility path based hardware binding algorithm for area-time efficient designs”, *Quality Electronic Design (ASQED) 2010, 2nd Asia Symposium on*, Penang, Malaysia, August 3-5,2010, pp 151-156.
- [219] ISO/IEC C 11 STANDARD. 2011. ISO/IEC 9899 - Programming languages - C.
- [220] ISO/IEC C++ 11 STANDARD. ISO/IEC 14882:2011 Programming Language C++.

- [221] Xilinx LogicCORE IP Linear Algebra Toolkit (LAT) v1.0. March 1, 2011. DS829.
- [222] Metzger, R. And Wen, Z. 2000. *Automatic Algorithm Recognition and Replacement*. MIT Press.
- [223] Batten, D., Jinturkar, S., Glossner, J. Schulte, M. and D'arcy, P., "A new approach to DSP intrinsic functions", *Proc. of the 33rd Annual Hawaii International Conference on System Sciences*. IEEE, Los Alamitos, CA.
- [224] Stallman, R. *Using and Porting GNU CC*. Free Software Foundation, June 1996, version 2.7.2.1.
- [225] Li, H., He, W., Chen, Y., Eeckhout, L. Temam, O. and Wu, C., "SWAP: Parallelization through algorithm substitution", *IEEE Micro*, vol. 32, issue 4, 2012, pp.54-67.
- [226] Alias, C. and Barthou, D., "Algorithm recognition based on demand-driven data-flow analysis", *Proc. of the 10th Working Conference on Reverse Engineering*. IEEE, Los Alamitos, CA. 296.
- [227] Cong, J. and Jiang, W., "Pattern-based behavior synthesis for FPGA resource reduction", *Proc. of 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays (FPGA'08)*. ACM, New York, NY, 107-116.
- [228] Ly, T., Knapp, D., Miller, R. and Macmillen, D., "Scheduling using behavioral templates", *Proc. of 32nd Conference on Design Automation (DAC'95)*. IEEE, Los Alamitos, CA, 101-106.
- [229] Prakash, A., Lam, S.K., Clarke, C.T and Srikanthan, T., "FPGA-aware techniques for rapid generation of profitable custom instructions", *Microprocessors and Microsystems*, vol. 37, issue 3, 2013, pp. 259-269.
- [230] Atasu, K., Pozzi, L., and Ienne, P., "Automatic application-specific instruction-set extensions under microarchitectural constraints", *Proc. of 40th Conference on Design Automation (DAC'03)*. IEEE, Los Alamitos, CA, 256-261.
- [231] Bonzini, P. and Pozzi, L., "Polynomial-time subgraph enumeration for automated instruction set extension", *Proc. of Design Automation and Test in Europe (DATE'07)*. IEEE, Los Alamitos, CA, 1-6.
- [232] Brisk, P., Kaplan, A., Kastner, R. and Sarrafzadeh, M., "Instruction generation and regularity extraction for reconfigurable processors", *Proc. of the 2002 International Conference on Compilers, Architecture and Synthesis of Embedded Systems (CASES'02)*. ACM, New York, NY, 262-269.

- [233] Yu, P. and Mitra, T., “Scalable custom instructions identification for instruction-set extensible processors”, in *Proc. of the 2004 International Conference on Compilers, Architecture and Synthesis of Embedded Systems (CASES’04)*. ACM, New York, NY, 69-78.
- [234] Batten, D., Jinturkar, S., Glossner, J. Schulte, M. and D’arcy, P., “A new approach to DSP intrinsic functions”, *Proc. of the 33rd Annual Hawaii International Conference on System Sciences*. IEEE, Los Alamitos, CA.
- [235] Metzger, R. Automated recognition of parallel algorithms in scientific applications, 1995
- [236] Xilinx LogiCORE IP CORDIC v4.0. March 1, 2011. DS249.
- [237] Sinnott, R.W., “Virtues of the haversine”, *Sky and Telescope*, vol. 68, issue 2, pp. 159.
- [238] Semiconductor Industry Assoc. (SIA), San Jose, CA, International Technology Roadmap for Semiconductors Update (Design), 2004.
- [239] Sangiovanni-Vincentelli A. , "Quo Vadis, SLD? Reasoning about the trends and challenges of system level design", *Proc. of the IEEE*, vol. 95, issue 3, March 2007, pp 467-506.
- [240] Cardoso J.M.P., Diniz P.C., Weinhardt M., "Compiling for reconfigurable computing: a survey", *ACM Computing Surveys*, vol. 42, no. 4, June 2010, pp 1-65.
- [241] Herbordt M.C., Gu Y., Vancourt T. et al., "Computing models for FPGA based accelerators", *Computing in Science & Engineering*, vol. 10, issue 6, November 2008, pp 35-45.
- [242] Gokhale M.B., Rickett C.D. and Hsu C.H., “Promises and pitfalls of reconfigurable supercomputing,” *Proc. 2006 Conf. Eng. of Reconfigurable Systems and Algorithms*, CSREA Press, 2006, pp. 11-20.
- [243] Alam S.R., Agarwal P.K., Smith M.C. et al., “Using FPGA devices to accelerate biomolecular Simulations,” *IEEE Computer*, vol. 40, no. 3, March 2007, pp. 66–73.
- [244] Herbordt M.C., Vancourt T., Gu Y. et al., "Achieving high performance with FPGA-based computing", *IEEE Computer*, vol. 40, issue 3, March 2007, pp 50-57.
- [245] (2013) Xilinx Inc. <http://www.xilinx.com> (Last Accessed: October 28, 2013)

- [246] (2013) Altera Corp.. <http://www.altera.com> (Last Accessed: October 28, 2013)
- [247] (2013) Lattice Semiconductor. <http://www.latticesemi.com> (Last Accessed: October 28, 2013)
- [248] (2013) Actel now part of Microsemi. <http://www.microsemi.com/products/fpga-soc/fpga-and-soc> (Last Accessed: October 28, 2013)
- [249] International Technology Roadmap for Semiconductors 2009 Edition: Design
- [250] (2013) NetBean IDE. <https://netbeans.org/> (Last Accessed: October 28, 2013)
- [251] (2014) Non-functional Requirements. http://en.wikipedia.org/wiki/Non-functional_requirement (Last Accessed: April 15, 2014)
- [252] Vinod, A.P. and Lai, E.M-K., “An efficient coefficient partitioning algorithm for realizing low complexity digital filters”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 12, December 2005, pp. 1936-1940.
- [253] Choo, H., Muhammad, K. and Roy, K., “Complexity reduction of digital filters using shift inclusive differential coefficients,” *IEEE Transactions on Signal Processing*, vol. 52, no. 6, Jun. 2004, pp. 1760–1772.
- [254] Vinod, A.P. and Lai, E.M-K., “On the implementation of efficient channel filters for wideband receivers by optimizing common subexpression elimination methods,” *IEEE Transactions on Computer- Aided Design of Integrated Circuits and Systems*, vol. 24, no. 2, Feb. 2005, pp. 295–304.