METHODOLOGY TO DERIVE RESOURCE AWARE CONTEXT ADAPTABLE ARCHITECTURES FOR FIELD PROGRAMMABLE GATE ARRAYS

by

Harikrishna Samala

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Engineering

Approved:

Dr. Aravind Dasu Major Professor	Dr. Brandon Eames Committee Member
Dr. Scott Budge Committee Member	Dr. Byron R. Burnham Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2009

Abstract

Methodology to Derive Resource Aware Context Adaptable Architectures for Field Programmable Gate Arrays

by

Harikrishna Samala, Master of Science

Utah State University, 2009

Major Professor: Dr. Aravind Dasu
Department: Electrical and Computer Engineering

The design of a common architecture that can support multiple data-flow patterns (or contexts) embedded in complex control flow structures, in applications like multimedia processing, is particularly challenging when the target platform is a Field Programmable Gate Array (FPGA) with a heterogeneous mixture of device primitives. This thesis presents scheduling and mapping algorithms that use a novel area cost metric to generate resource aware context adaptable architectures. Results of a rigorous analysis of the methodology on multiple test cases are presented. Results are compared against published techniques and show an area savings and execution time savings of 46% each.

(60 pages)

To my family.

Acknowledgments

I would like to thank my advisor, Dr. Aravind Dasu, for giving me the opportunity to work in the area of my interest and for his constant motivation and guidance throughout my research. I thank my committee members, Dr. Brandon Eames and Dr. Scott Budge, for extending their support.

I thank all my friends for their moral support throughout my graduate studies at Utah State University. Especially, I acknowledge Arvind Sudarsanam for his technical support and for reviewing my work. Last, but not the least, I would like to thank my parents for their love and support.

Harikrishna Samala

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this introductory chapter, the motivation for developing a methodology to create context adaptable architectures for Field Programmable Gate Arrays (FPGAs) is discussed, together with the issues of using FPGAs as the target hardware. The principal contributions of this thesis are presented next followed by an overview of the content presented in this thesis.

1.1 Motivation

The malleable logic and routing fabric of FPGAs have always held an attraction to Very Large Scale Integration (VLSI) architecture designers, since they allow for highly customized designs to be created in Register Transfer Level (RTL), synthesized, mapped, placed, routed, and tested. In a sense, the use of FPGAs offers instant feedback to the architect when compared with the traditional Application Specific Integrated Circuit (ASIC) design process. While designers have always taken advantage of creating custom architectures on FPGAs for Data-Flow Graphs (DFGs), more recently there have been a variety of investigations published to explore rapidly adaptable designs running on an FPGA. These efforts can be broadly classified into two categories: (i) polymorphic designs which are soft configurable (i.e., do not need a change in the underlying bit-stream, hence allowing for rapid adaptability), and (ii) Partial and Dynamic Reconfiguration (PDR) methods, which require reconfiguration of the bit-stream and are relatively much slower than configuring polymorphic designs.

In this thesis, the discussion is restricted to efforts carried out in the first category. The domain of generating polymorphic designs for a Control-Data Flow Graph (CDFG) composed of multiple DFGs is still not completely explored. The heterogeneous nature

of FPGA architectures adds a new dimension to the existing problem. As FPGAs offer multiple options to implement a circuit, it is no longer possible to find one single solution which is superior to all others. As such, this thesis develops a methodology for creating efficient context adaptable architectures targeting FPGAs using a combination of high-level synthesis techniques, including resource allocation, scheduling, and binding algorithms.

1.2 Principal Contributions

The principal contributions of this thesis are:

1. Context adaptable architecture template that can support context switching within a CDFG,
2. Methodology to derive the components of the data-path,
3. Heuristic-based resource selection algorithm targeting FPGAs, and
4. Heuristic-based resource binding algorithm (mapper).

1.3 Thesis Overview

Following this introductory section, in Chapter 2 basics of FPGAs are discussed along with existing techniques for generating data-paths for control and data-flow applications, and some techniques for area estimation on FPGAs. Chapter 3 presents some prelimaries. In Chapter 4 the proposed methodology, including scheduling and mapping algorithms, is presented. Complexity analysis of the proposed algorithm is derived in Chapter 5. Results of the proposed approach in comparison with few prior algorithms are presented in Chapter 6, and conclusions are presented in Chapter 7.

Chapter 2

Background and Related Work

2.1 Field Programmable Gate Arrays

An FPGA is a silicon device with a tightly interconnected matrix of user-programmable logic blocks, and a number of Input/Output Blocks (IOBs) to communicate with external devices as shown in fig. 2.1. Both the logic blocks and the interconnect structure are reprogrammable and are based on Static Random Access Memory (SRAM) cells. SRAM-based FPGAs hold their configurations in a static memory. Hence, FPGAs can be reconfigured multiple times to support any complex design. Xilinx [1] and Altera [2] are the two major players in today's reconfigurable market.

A Configurable Logic Block (CLB) has Look Up Tables (LUTs) and Flip-Flops (FFs). LUTs have multiple inputs and a single output, which can be programmed to represent any boolean function. As an example, consider a boolean function described by

$$f = a(b + c) + d, \tag{2.1}$$

and a Xilinx virtex-4 FPGA, which has four-to-one LUTs. Table 2.1 shows the truth table

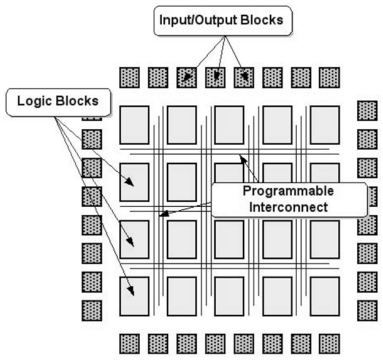


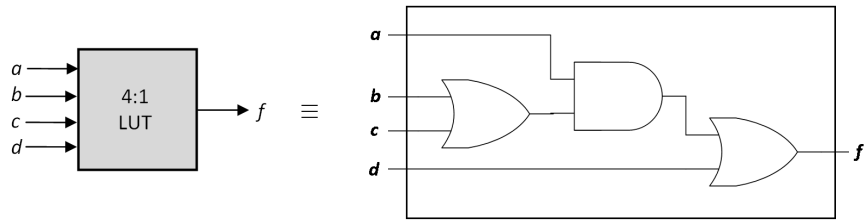
Fig. 2.1: General structure of an FPGA.

Table 2.1: Truth table.

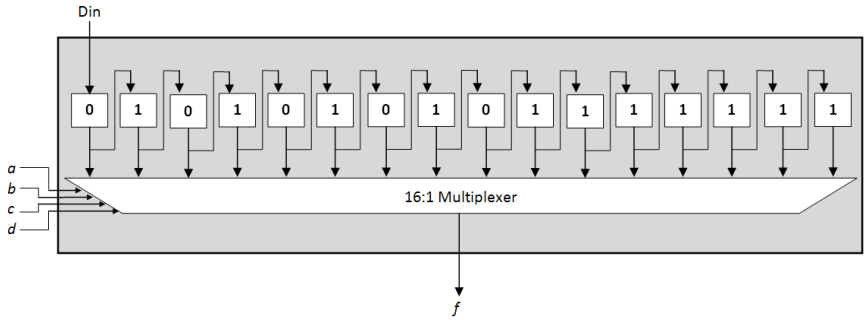
Inputs				Output
a	b	c	d	f
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

for the boolean function and fig. 2.2(a) shows the corresponding combinatorial logic. The internal structure of an LUT for the current example is shown in fig. 2.2(b). It can be seen that LUT has 16 1-bit wide memory elements each of which stores the output of the boolean function and a multiplexer is used to select a value depending on the input combination. The inputs to the LUT act as the select lines for the multiplexer. During FPGA configuration, the values of function f are written in the memory elements of the LUT through the Din port. It can be seen that all the memory elements are connected in a chained fashion, which decreases the number of input ports required but it takes 16 clock cycles to configure an LUT.

Though FPGAs are more power hungry and slower than custom designed ASICs, they save a significant amount of design cycle time as there is no wait time between completing the design and having a working chip. To increase the performance, modern FPGAs include embedded ASICs like Digital Signal Processors (DSPs) and Block Random Access Memories (BRAMs). For many years FPGAs were used only for prototyping, but today they are used inside digital systems.



(a)



(b)

Fig. 2.2: (a) LUT combinatorial circuit, (b) LUT internal structure.

FPGA Design Flow: Designing architectures for FPGAs starts with the designer describing the digital system using hardware description languages (HDLs) like Verilog and VHDL (Very high speed integrated circuit Hardware Description Language). Schematics are rarely used to describe logic, but are used at higher levels of system hierarchy to instantiate and connect lower-level modules. The FPGA design flow is shown in fig. 2.3. The design in HDL is synthesized to generate a netlist, which is then simulated for functional validation. A netlist is a collection of gates along with their attributes and their interconnections. The design is then implemented which includes the following steps: (i) *translate*: merge multiple design files into a single netlist; (ii) *map*: group logical symbols from the netlist (gates) into physical components (CLBs and IOBs); (iii) *place and route*: place components onto the chip, connect them, and extract timing data into reports.

Now that the timing data is available after place and route, the design is simulated for timing information and is analyzed for correctness. Once the design is implemented and meets the timing requirements, a bit stream is generated which can be understood by the target FPGA. As the last step, the FPGA is configured with the bit stream data.

2.2 Scheduling Techniques

A CDFG representation of an application is a commonly used intermediate format in high-level synthesis (HLS) tools. A CDFG consists of basic blocks (BBs) embedded in different forms of control structures. A DFG is a graphical representation of a set of operations and their data dependencies. It is denoted as $G(V, E)$, where V is the set of operations (nodes) and E is the set of edges that represents data-flow among the set of nodes. In this thesis, a DFG is referred to as a "context." The derivation of an architecture (at the RTL level) for a given CDFG involves three well known tasks in the area of HLS: (i) scheduling, (ii) allocation, and (iii) binding/mapping. Scheduling assigns operations to particular time steps of execution. Allocation determines the number and types of functional units to be used in the design. Binding maps the operations in the scheduled CDFG to functional units. Binding is also responsible for determining the resources for data routing.

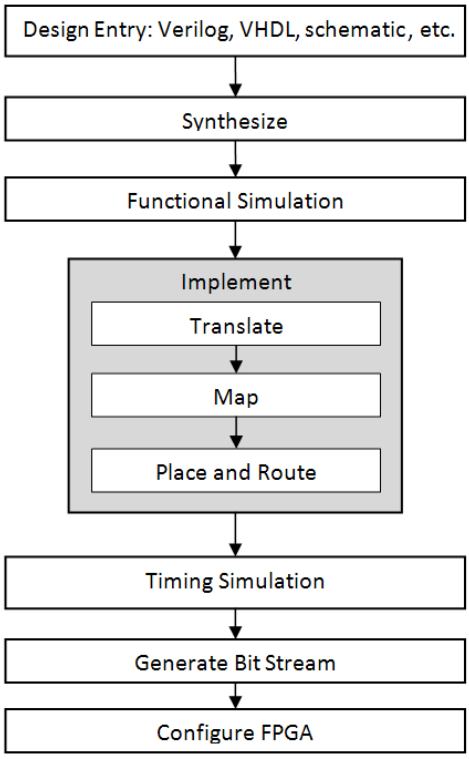


Fig. 2.3: FPGA design flow.

While some scheduling algorithms just deal with assigning operations to time steps, other scheduling algorithms include all the above three tasks (i - iii). Also, note that the above three tasks are not always done in the order they are listed. As scheduling provides multiple architectural options, there is a need to estimate FPGA resources for each option and select the best one. In this chapter, background and related work are presented in the following three categories: (a) DFG scheduling techniques, (b) CDFG scheduling techniques, and (c) FPGA resource estimation techniques.

2.2.1 DFG Scheduling Techniques

The two classes of DFG scheduling algorithms are: (i) time-constrained scheduling (TCS) and (ii) resource-constrained scheduling (RCS). TCS algorithms try to reduce the number of resources required to schedule a DFG within the specified execution time (time constraint), whereas RCS algorithms attempt to minimize the execution time by finding the

best possible schedule using the given set of resources (resource constraint). Integer linear programming (ILP) techniques have been proposed for both TCS and RCS algorithms [3] to generate optimal solutions, but the only downside of these techniques is their large execution times (exponential in the worst case). Alternately, heuristic techniques have been developed. Such techniques include, but not limited to, as soon as possible (ASAP) scheduling [4], as late as possible (ALAP) scheduling [5], list scheduling (LS) [6] and force-directed scheduling (FDS) [7]. The limitation of ASAP and ALAP algorithms is that they do not give priority to critical path nodes. Hence, they result in a set with number of resources which is larger than the number of resources in a set obtained from other scheduling techniques (like FDS).

The LS algorithm is a RCS algorithm, which prioritizes the nodes based on urgency and generates a schedule satisfying the input constraints specified as the number of resources of each type (i.e., *mul*, *add*, *div*, etc.). FPGAs are heterogeneous mixture of look-up tables (LUTs), flip-flops (FFs), digital signal processor units (DSPs), and embedded block RAMs (BRAMs), which are collectively termed as device primitives. A resource to be mapped on an FPGA can have multiple flavors of implementations and each flavor can consume a different mix of device primitives. A resource constraint for a design targeting an FPGA will be specified in terms of its device primitives. Therefore, there is a need to convert the input resource constraint that is specified in terms of device primitives to a set of possible resource constraints in terms of the number of resources of each type, before using the LS algorithm for FPGAs.

The FDS algorithm is a TCS algorithm which relies on ASAP and ALAP algorithms. This algorithm tries to reduce the number of resources by uniformly distributing operations of the same type. For designs targeting FPGAs, the output resource set from the FDS algorithm needs to be converted to the number of device primitives, to see if the design can be accommodated in the available FPGA area. The criterion used in FDS algorithm is the force associated with each node (calculated using the probabilities of scheduling predecessor and successor nodes) and this criterion is updated dynamically, whereas in the LS algorithm, the criterion used is the critical path length which is not updated. Hence, FDS results in a

better schedule than LS, and therefore, FDS is preferred over LS in the algorithm proposed in this thesis. Although work has been carried out to improve FDS algorithm and new algorithms were proposed for scheduling DFGs ([8–10]), the classical FDS algorithm [7] is used in this thesis.

2.2.2 CDFG Scheduling Techniques

As the major contribution of this paper is to generate architectures for applications involving different forms of control structures, this section evaluates some of the existing techniques that address this issue.

For applications involving control flow, the algorithms discussed in the previous section cannot be used directly to generate schedules. Attempts were made by Camposano [11], Al-Sukhni *et al.* [12], and Bergamaschi *et al.* [13] to schedule a control flow graph (CFG) by using path-based scheduling (PBS) method and its variations. PBS tries to minimize the number of control states under given timing and area constraints. CFG is first converted to a directed acyclic graph (DAG) by removing the feedback edges of loops. All paths in the DAG are scheduled using as-fast-as-possible schedule. All the schedules are combined (by overlapping) to obtain a finite state machine with least number of control states that can support the execution of the CFG. In PBS, the concurrency of operations and data dependency among operations is ignored, the execution time of the CFG is not addressed, and the order in which operations are executed in the DAG is fixed (same as the order in which operations are present in the input description). This results in slower and inefficient schedules.

A conditional resource-sharing algorithm using hierarchical reduction is proposed by Kim *et al.* [14]. In this approach, a CDFG (not containing loops) is transformed to a DFG by replacing conditional blocks by equivalent nonconditional blocks. This is achieved by first determining the time frame of operations using ASAP and ALAP algorithms. Operations from the conditional branches are paired, only if their time frames overlap. A ratio is associated with each node and pairing is done in the decreasing order of these ratios. The resulting DFG is scheduled using the force-directed list scheduling (FDLS) algorithm [8].

The schedule of the original CDFG with conditional branches is obtained by transforming the schedule information of the DFG. Kim *et al.* assume unit latency for all the operation nodes and fork nodes. Though this algorithm was proven to perform better than the PBS algorithm [11] in terms of number of control steps, there is no support for handling loops and complex control structures.

Lakshminarayana *et al.* [15] propose a more comprehensive scheduling algorithm, called *Wavesched-spec*, which can support branches and loops. Their algorithm is a RCS algorithm that simultaneously speculates branches and loops (by performing loop unrolling) in order to minimize the number of clock cycles. All the speculated values are stored in registers. The downside of this approach is the excessive use of registers to store the speculated values of all the unrolled loops.

Moreano *et al.* [16] propose a datapath merging approach to synthesize a single datapath that can be partially reconfigured (through multiplexers) to support multiple DFGs in a CDFG. Their algorithm merges only those DFGs that are present inside loops, as they are the major candidates for hardware acceleration. The remaining DFGs are executed on a SPARC-v8 microprocessor. Their approach first builds a compatibility graph and then uses maximum weighted clique partitioning to find the common data-path. They do not support sharing of a resource among multiple operations within the same DFG. Though sharing of resources across DFGs is achieved by adding MUX trees, the edge conectivity when multiple DFGs are merged makes the interconnect and the MUX tree very complex.

Guo *et al.* [17], Kastner *et al.* [18], and Cong *et al.* [19] propose to accelerate control and data-flow applications by extracting hardware templates for configurable processors. All three works try to minimize the number of distinct templates and the total number of templates. A template here refers to a set of connected nodes. Guo *et al.* [17], first generate a set of sub-graphs of varying sizes, from size one to a predefined size ($maxsize$), using the process of node growing. These sub-graphs are then used to cover the input CDFG, and a set of templates is generated such that the number of distinct templates and the total number of sub-graphs are minimized. As the template selection is computationally

intensive, the authors propose a heuristic method. They associate an objective function for each template which is based on the number of nodes present in a template and the number of nonoverlapped matches of the template in the original CDFG. The objective function is used as the heuristic and a set of templates is selected such that the objective function is maximized. However, they do not provide the exact representation for the objective function.

Kastner *et al.* [18] propose a template generation algorithm for hybrid reconfigurable architectures. Their algorithm starts by profiling the graph for frequency of edge types (ex., *mul-mul*, *add-mul*, *mul-add*, etc.). Based on the frequency of edge types, nodes are clustered to form super nodes. The resulting graph is again profiled and nodes are clustered. This algorithm is repeated until sufficient number of super nodes (templates) are generated to cover the graph. These templates are then used to cover the graph such that both the number of distinct templates and the number of instances of each template are minimized. Also, each edge is clustered in a locally optimal manner, which results in global sub-optimal solution.

Cong *et al.* [19] propose an algorithm for generating application-specific instructions to improve the performance of configurable processors. Patterns (templates) are first generated by a node clustering algorithm. Such patterns can have multiple inputs ($|IN(p_i)| \leq N_{in}$ $\forall$ i) but only a single output ($|OUT(p_i)| \leq 1$ $\forall$ i) and are subject to the area constraint ($\sum area(p_i) \leq A$). $IN(p_i)$ and $OUT(p_i)$ are the input set and output set of pattern p_i and A is the input area constraint. The value of N_{in} limits the number of parallel nodes present in the pattern. Each pattern is then characterized by gain and area. For a pattern, the hardware execution time (T_{hw}) is calculated to be its critical path length, and software execution time (T_{sw}) is calculated as the summation of the software execution times of all nodes. The speedup of a pattern is calculated as the ratio of T_{sw} and T_{hw}. Finally, the gain of the pattern is calculated as the product of the speedup and the occurrence of that pattern. From the set of templates generated so far, a set of templates is obtained such that the overall gain is maximized and the area constraint is satisfied. This set of templates is

then used to cover the input graph while minimizing the execution time. This step is carried out using a 0-1 Knapsack problem [20]. In all three template-based approaches ([17–19]), the objective to minimize the total number of templates results in architectures with longer execution times. Also, the number of templates generated can be exponentially large for CDFGs with large number of nodes, thereby increasing the computational complexity of the algorithm. The complexity analysis of the algorithm is provided by Phillips *et al.* [21].

Bilavarn *et al.* [22] propose an algorithm to generate FPGA based architectures for CDFGs using area and delay estimations. They perform design space exploration for a specific RTL architecture template (bus-based architecture), by defining a parameterized architecture, rather than building one. They use a time-constrained list-scheduling algorithm [8] for scheduling DFGs with multiple time constraints to generate multiple schedules. The schedules of the DFGs are combined hierarchically based on the control structure resulting in a set of solutions. Nonoptimal solutions are then eliminated through Pareto optimality [23]. They categorize FPGA resources into (i) logic cells and (ii) dedicated cells. Logic cells include LUTs and FFs, and dedicated cells include DSP48s and BRAMs. The output of the algorithm is a set of architectures with different area-delay trade-offs for each type of FPGA resource. It is up to the designer to choose a common architecture from the solution sets of each FPGA resource that best fits the FPGA. After all the DFGs are processed, area estimation for each type of FPGA resource is computed separately for each solution. Unlike Bilavarn *et al.* [22], in the proposed methodology area estimation was performed during architecture exploration offering instant feedback to the scheduling algorithm, which might yield better results.

None of the algorithms discussed in this section (excluding Bilavarn *et al.* [22]) consider FPGAs as the target architectures, and none of them consider the possibility of different implementations of a resource mapped on an FPGA. The algorithm proposed in this thesis takes advantage of these facts to generate architectures for a CDFG. A quantitative comparison of Moreano *et al.* [16], Guo *et al.* [17], Kastner *et al.* [18], and Cong *et al.* [19], in terms of resource utilization and execution time, relative to each other and the proposed algorithm

is provided in the results chapter. Results for Bilavarn *et al.* [22] were not presented in this paper because sufficient information was not available in the publication.

2.3 FPGA Resource Estimation Techniques

One of the most important constraints required when designing architectures for FPGAs is that the design fits inside the specified FPGA area. When exploring multiple architectures for a design, there is a need for early area estimations. This section discusses three approaches ([22, 24, 25]) towards generating FPGA architectures using some form of area estimation.

Nayak *et al.* [24] perform design space exploration on the input behavioral specification in MATLAB and pessimistically estimate the number of configurable logic blocks (CLBs) utilized by the generated hardware architecture. The MATLAB code is first converted to VHDL, which is scheduled using FDS [8] to obtain concurrency of operations, from which the number of function generators are determined. Register allocation is performed to calculate the total number of registers. The number of CLBs is then estimated using the number of function generators and number of registers. Their estimation is targeted specifically at the Xilinx XC4010 device and they do not include memory units.

Kulkarni *et al.* [25] propose an iterative compilation-based algorithm that translates high-level single-assignment code (SA-C) into hardware using quick estimations. The objective of their estimation tool is to reduce the development time by quick estimations rather than to increase the accuracy. The SA-C code is first translated to a DFG representation whose nodes are associated with approximation formulae. Their estimation does not incorporate scheduling, resource allocation, and binding algorithms, but it takes into account some synthesis optimizations. In Nayak *et al.* [24] only one type of FPGA resource (LUT in this case) is considered for implementing arithmetic/logic units. Also, memory units are not handled.

Bilavarn *et al.* [22] propose area estimation of RTL architectures in terms of logic cells (lc) and dedicated cells (dc) computed separately. The total number of logic cells is estimated using

$$N_{lc} = \sum_{k} (n_k * N_{lc}^k), \tag{2.2}$$

where n_k is the number of operations of type k and N_{lc}^k is the number of logic cells to implement a resource on FPGA that supports execution of operation of type k. The total number of dedicated cells is estimated in a similar manner. Unlike Nayak *et al.* [24] and Kulkarni *et al.* [25], Bilavarn *et al.* estimate the resources for memory units. A comparison of the above approaches is provided in the results section in Table 6.2.

Chapter 3

Preliminaries

This chapter presents some preliminaries on area estimation of architectures implemented on an FPGA and architecture representation, necessary to understand the architecture exploration algorithm which will be explained in the next chapter.

As FPGAs have fixed logic and routing resources, certain designs might not fit on a given FPGA. For CDFGs involving multiple DFGs and control structures, the designer has to explore multiple designs which satisfy latency constraints and that fit on the chip with satisfactory area requirements. A heuristic algorithm is presented in this thesis for resource selection, which is based on resource estimations, without having to go through the time-consuming synthesis, map, and Place and Route (P&R) tools. As mentioned in Chapter 2, FPGAs are heterogeneous mixture of device primitives (LUTs, FFs, DSP48s, and BRAMs). A resource to be mapped on an FPGA can have multiple flavors of implementations, and each flavor can consume a different mix of device primitives. Therefore, a Weighted Sum of Device Primitives (WSDP), a metric to evaluate the relative area cost of resources mapped onto FPGAs, is presented here. Work related to this metric was presented by Phillips [26]. WSDP for a resource r is calculated using

$$W_r = \sum_i \left(\frac{k_i}{a_i}\right) \forall\ i \in \{LUT,\ FF,\ DSP48,\ BRAM\}, \tag{3.1}$$

where k_i is the number of device primitives of type i required to implement the resource r and a_i is the number of available device primitives on the FPGA. If any two resources have the same value of WSDP, then the standard deviation for that resource r is calculated using

$$SD_r = \sqrt{\frac{\sum_i \left(\frac{k_i}{a_i} - \frac{W_r}{4}\right)^2}{4}} \quad \forall\ i \in \{LUT,\ FF,\ DSP48,\ BRAM\}. \tag{3.2}$$

The final area cost of a circuit can then be calculated using

$$W_{cumulative} = \Big(\sum_{r \in R} n_r W_r\Big) + W_{mux} + W_{reg}, \tag{3.3}$$

where R represents the set of distinct resources, $R = \{add,\ mul,\ div,\ \text{etc.}\}$, and n_r is the number of resources of type $r \in R$, obtained from the scheduling algorithm (explained later in sec. 4.4). SD_r can be used in the place of W_r whenever the WSDP of a resource is calculated using (3.2). $W_{cumulative}$ represents the relative area cost and hence can be used to compare area cost of two different circuits. The WSDPs for multiplexers (W_{mux}) and delay registers (W_{reg}), necessary for routing data among functional units (FUs) based on the scheduled DFG, are calculated by using (3.1) in the mapping algorithm (explained later in sec. 4.5).

A solution (architecture for a CDFG) is represented as a 4-tuple (S, W, A, T) calculated using (3.3), (3.4), (3.5), and (3.6). A resource set S (shown in (3.4)), represents the set of number of resources obtained after scheduling a DFG/CDFG using a time constraint. W represents the cumulative WSDP of the circuit (from (3.3)). A is the set of available device primitives. T represents the execution time of the CDFG (shown in (3.6)). An estimate of the execution time of a CDFG (T_{CDFG}) can be calculated as a summation of product of execution time (T_i), weighting factor (f_i), and total number of effective iterations (N_i) of DFG i, over all the DFGs. The weighting factor (f_i) of a DFG can be either probabilistic (i.e., when the branch conditions and loop terminating conditions surrounding a DFG depend on the input data set), or deterministic if they can be determined at compile time. The probability values can be obtained by profiling the application on various input data sets.

$$S = \{n_r \mid r \in R\} \tag{3.4}$$

$$A = \{a_i \mid i \in \{LUT,\ FF,\ DSP48,\ BRAM\}\} \tag{3.5}$$

$$T_{CDFG} = \sum_{i=1}^{n_{DFG}} T_i f_i N_i \tag{3.6}$$

Chapter 4

Proposed Methodology

This chapter presents the architecture template, algorithm overview, heuristic resource selection, the resource aware scheduling algorithm, and the mapping algorithm.

4.1 Context Adaptable Architecture Template

The architecture template used in the proposed approach is composed of a data path and a controller. The RTL model of this template is shown in fig. 4.1. The data path consists of a set of functional units (FUs) and data routing network (i.e., delay registers and multiplexers). The outputs of FUs can be connected to the inputs of other FUs directly or through multiplexers and/or through delay registers, governed by the data dependencies present in the scheduled DFGs. FUs are determined by the number, type, and implementations of the resources obtained through the scheduling algorithm. FUs are generated using the Xilinx CORE generator, whose characteristics are classified in an architecture library file and used by the scheduling and mapping algorithms. Characteristics here refer to the latency, implementation type, and area usage in terms of FPGA device primitives.

The multiplexers that are used to select the input data for the FUs are controlled by the schedule information (control words) stored inside the schedule table of the respective DFGs, which are controlled by the Global Controller depending on which DFG (context) of the CDFG is currently being executed. The Global Controller is a simple finite state machine, which uses the FU outputs (for conditional branches in the input CDFG). The delay registers used are SRL16 shift register look-up-tables present on the Xilinx Virtex 4 FPGAs and are also generated using the Xilinx CORE generator. This template is used to represent a custom architecture, defining the number of FUs, number of delay registers, number and size of multiplexers, and bus connections among all the blocks.

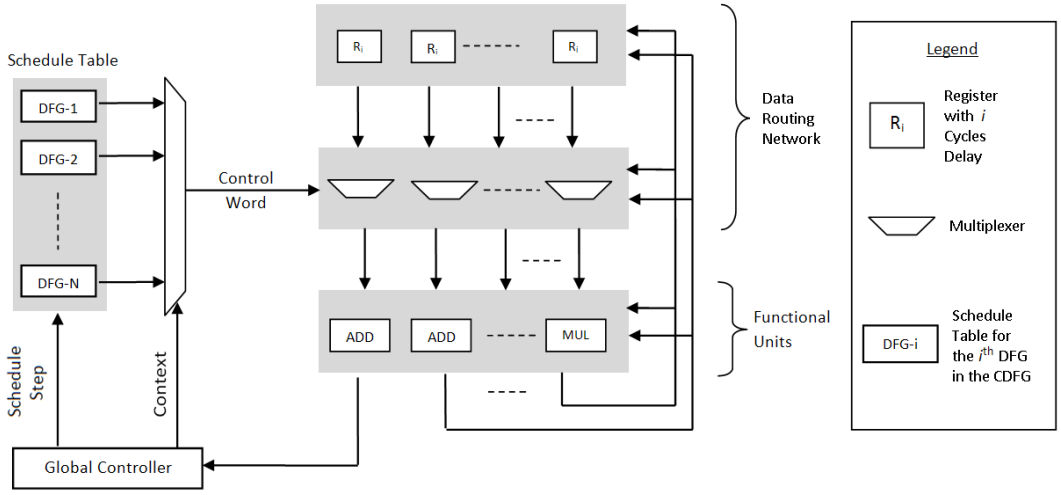


Fig. 4.1: Context adaptable architecture (CAA) template.

4.2 Algorithm Overview

The purpose of the architecture exploration algorithm (explained in sec. 4.4) is to generate an architecture that can support the execution of a CDFG and that can fit in the given FPGA area (area constraint). Given the heterogeneous nature of FPGAs and their flexibility to support multiple implementations of resources, the proposed algorithm explores multiple designs and generates a set of architectures with different area-time trade-offs which can support execution of the given CDFG.

Key features of the algorithm:

1. Updates available FPGA device primitives after resource allocation,
2. Updates relative area cost (WSDP) of multiple implementations of all resources,
3. Calculates the area cost of architectures implemented for FPGAs,
4. Uses Pareto optimality criterion [23] to retain optimal solutions.

Now, we present a high-level overview of the algorithm. Individual DFGs of a CDFG

are scheduled using FDS with different latencies (time constraints) and different resource sets are obtained. These resource sets are merged exhaustively across all DFGs (one-by-one) to obtain common resource sets (explained in sec. 4.4), each of which can support all the DFGs. During the merge process, resource selection is done by evaluating WSDPs for all resource implementations. The number of available device primitives is updated after resources are allocated for the implementation with the least WSDP. As the resource sets associated with individual DFGs have different latencies, the merged resource sets have different latencies (CDFG execution time) and different relative area costs. A common resource set represented by its area cost (WSDP) and CDFG execution time (calculated using (3.6)) is termed as a partial solution. A partial solution set is obtained after resource sets of two DFGs are merged. Pareto optimality criterion [23] is used to retain the set of local optimal resource sets from partial solution set, before merging the resource sets of another DFG. Note that the resource sets obtained here are not global optimal resource sets, as the algorithm used to schedule individual DFGs is FDS, which is a heuristic technique.

The common resource sets obtained after all the DFGs are processed are passed to a mapping algorithm (explained in sec. 4.5). The mapping algorithm, which is also a heuristic algorithm, calculates the resources required to route the data among FUs in order to support all the DFGs. Data routing resources refer to multiplexers and delay registers. A resource set along with the data routing information and the corresponding CDFG execution time is termed as a solution. The relative area cost of the resource sets is updated with the area cost of data routing resources and again sub-optimal solutions are removed using pareto optimality criterion. The final set of pareto optimal solutions represent the final set of context adaptable architectures.

Pareto optimality criterion: For a two-objective minimization problem (in my case area and execution time), it is desirable that the solutions have small values for each objective. Multiple solutions can be obtained for a problem (by changing the values of the input variables) which are evaluated for the two objectives and plotted in a 2D space as shown in fig. 4.2. The circles represent solutions evaluated for execution time and area. As we

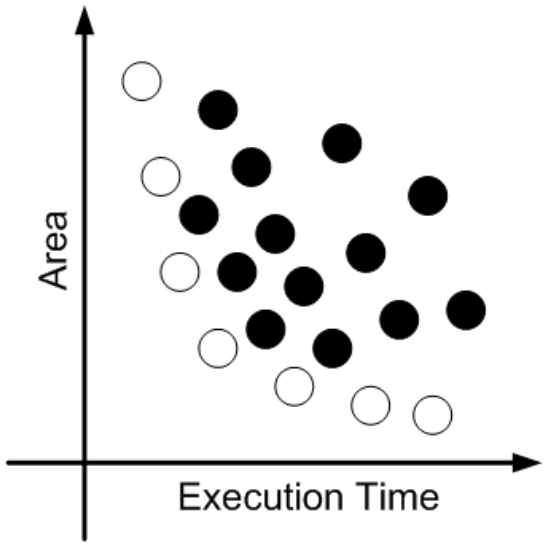


Fig. 4.2: Illustration of Pareto optimality criterion.

can see, there exists some solutions (shown in white circles) that are not entirely better or worse when compared to each other. These solutions are said to dominate the rest of the solutions (shown in black circles), which have higher values for both the objectives. The set of nondominant solutions (white circles) which are better in at least one objective, when compared to the others in the same set represet the Pareto optimal set. The criterion used to find the set of solutions which are the members of the Pareto set is called the Pareto optimality criterion.

4.3 Evaluate Algorithm

The *Evaluate* algorithm (Algorithm 4.1) takes in as inputs, a DFG, its corresponding resource sets, and an input solution. It returns a set of partial solutions with their associated area cost and execution time. In step 1, an empty solution set (X) is created. In step 2b, for each new additional resource present in S_j over S_{in}, the following three steps are repeated: (i) WSDPs of all resource implementations are recomputed based on the current available device primitives, (ii) the implementation with the smallest WSDP is selected for the current resource (i.e., resource selection) and resources are allocated, and (iii) WSDP of the current resource set and the number of available device primitives are updated. During

resource selection, if two implementations of a resource have the same WSDP, then the implementation that has the least standard deviation (3.2) is considered. The execution time (T) of the CDFG is estimated next using (3.6) for each set S_j. Execution times of individual DFGs corresponding to the input solution are used except for the current DFG, for which j is taken as the execution time. If the available device primitives are sufficient for the current DFG (i.e., if its value is nonnegative at the end of all resource allocations), the current set S_j, its corresponding WSDP, the updated available device primitives, and the execution time represent a solution and is appended to the solution set X (step 2d). The resulting solution set X (which represents a set of partial solutions) is returned to the $CAAE$ algorithm. If the available device primitives are not sufficient to accommodate at least one resource set of the input DFG, the *Evaluate* algorithm returns an empty set to the $CAAE$ algorithm indicating insufficient FPGA resources.

4.4 Context Adaptable Architecture Exploration (CAAE) Algorithm

The proposed $CAAE$ algorithm (Algorithm 4.2), which includes resource selection, scheduling and mapping algorithms, generates a set of context adaptable architectures for a given CDFG. It takes as inputs, a CDFG to be synthesized, weighting factors of DFGs, number of iterations of all loops, and a set of available FPGA device primitives for use. The only constraint to the scheduling algorithm is that the design should fit in the given area (A) and there is no hard constraint on the execution time. Hence, the algorithm generates a set of best possible solutions with different execution times and area tradeoffs. It is then left to the designer to choose a solution with desirable execution time.

The algorithm starts by calculating the critical path latencies, in clock cycles, of all the DFGs using ASAP algorithm (step 1). In step 2, the lower bound on the execution time of the CDFG (T_{min}) is estimated using critical path latencies and (3.6). Steps 3 through 7 of the algorithm generate an initial partial solution set using one DFG, and then iteratively update the solution set using the remaining DFGs one-by-one.

In step 3, an initial solution (S_{in}, W_{in}, A_{in}, T_{in}) is created, where S_{in} is an empty set of resources, initial WSDP W_{in} is zero, A_{in} is the input available device primitives, and

Algorithm 4.1 Evaluate

Input: Data-flow graph (G), resource sets of G, and input solution (S_{in}, W_{in}, A_{in}, T_{in}).
Output: Set of partial solutions (X).
Notations:

1) Let S_j represent the set of number of resources of a DFG scheduled with latency j. $S_j = \{n_r^{'} \mid r \in R\}$.
2) Let S_{in} represent the set of number of resources of the input solution. $S_{in} = \{n_r \mid r \in R\}$.

1. $X \leftarrow$ Empty solution set.
2. **For** each resource set (S_j) of G, **do**
   a. $W \leftarrow W_{in}$, $A \leftarrow A_{in}$
   b. **For** each $r \in R$, **do**
      If ($n_r^{'} > n_r$)
      Loop ($n_r^{'} - n_r$) times
      i. For the available device primitives (A), recompute WSDPs of all implementations of the resource r and select the least one ($w_r^{'}$). If any two implementations have the same WSDP, then select the implementation with the least standard deviation.
      ii. $W \leftarrow W + w_r^{'}$
      iii. Update available device primitives (A) by subtracting from A, the number of device primitives required by the implementation of resource r.
      End Loop
      End If
   End Loop
   c. Calculate the execution time (T) of the CDFG, using latency j for the current DFG.
   d. If the number of available device primitives (A) is non-negative, add the current solution (S_j, W, A, T) to the solution set X.
   End Loop
3. Return solution set X.

Algorithm 4.2 CAAE

Input: CDFG, Weighting factors (f), Number of loop iterations (N), Available device primitives (A).

Output: Set of context adaptable architectures.

Notation: Let n_{DFG} be the number of DFGs present in the CDFG.
Let G_i represent the i^{th} DFG in the CDFG.

1. Determine critical path latencies of all the DFGs. $CP(G_i)\ \forall\ i \in \{1, 2, ...n_{DFG}\}$.
2. Estimate the lower bound of execution time (T_{min}).
3. Create empty solution (S_{in}, W_{in}, A_{in}, T_{in}).
4. Using FDS find all schedules and sets of number of resources G_1.
5. For all resource sets, evaluate G_1 for WSDPs, CDFG execution time (using *Evaluate* algorithm) and generate a set of partial solutions ($P_{current}$).
6. **If** $P_{current}$ is an empty set, the algorithm is terminated.
7. **For** $i \leftarrow 2$ to n_{DFG}, **do**
   a. Using FDS, find all schedules of G_i.
   b. $X_{new} \leftarrow$ Empty solution set.
   c. **For** a solution m in the set $P_{current}$, **do**
      i. For all resource sets obtained in step-7a, evaluate G_i for WSDPs, CDFG execution time, and generate a set of partial solutions (X_m).
      ii. Append solutions in set X_m to set X_{new}.
      End For loop
   d. If X_{new} is an empty set, the algorithm is terminated.
   e. Retain set of partial Pareto optimal solutions ($P_{current}$) from set X_{new}.
   End For loop
8. Calculate W_{mux} and W_{reg} using a mapping algorithm for each solution in the Pareto optimal solution set ($P_{current}$), and add them to the WSDP of each solution, resulting in a new set $P'_{current}$.
9. Retain Pareto optimal solutions from $P'_{current}$, resulting in a new set P_{final}.
10. The final set of Pareto optimal solutions (P_{final}) along with their corresponding schedules represent the set of context adaptable architectures.

T_{in} is the lower bound of the execution time (T_{min}). In step 4, DFG G_1 is scheduled using FDS with different latency constraints and multiple resource sets are obtained. The latency constraint is relaxed starting from critical path latency until the resource set obtained from FDS contains one resource for each operation type. This terminating condition is chosen because relaxing the latency constraint beyond this point does not result in a smaller resource set. The resource set corresponding to the critical path latency supports the most parallel schedule (fastest execution - in terms of clock cycles) that can be achieved for G_1. Similarly, the resource set corresponding to the terminating latency constraint supports the most sequential schedule (slowest execution) that can be achieved for G_1. In step 5, each resource set of G_1 is evaluated for area cost (WSDP) and execution time of the CDFG using the *Evaluate* algorithm.

The solution set obtained from the *Evaluate* algorithm is represented as $P_{current}$ in the $CAAE$ algorithm. In step 7 of the $CAAE$ algorithm, another DFG is selected and multiple resource sets are obtained as explained before. For each partial solution in the set $P_{current}$, resource sets of the new DFG are evaluated for WSDPs and CDFG execution times, resulting in a new set of partial solutions represented by X_{new}. As X_{new} can have nonoptimal solutions with a new DFG added, Pareto optimality criterion [23] is used to retain local optimal solutions in the current partial solution set. The optimal solution set is again represented as $P_{current}$. The above process (steps 7a through 7e) is repeated for all the remaining DFGs.

After all the DFGs are processed in step 7, each solution in the set $P_{current}$ represents a resource set that can support all the DFGs. In step 8, the area cost of data routing resources (i.e., multiplexers and delay registers) is estimated using the mapping algorithm (explained later in sec. 4.5) for each solution in $P_{current}$ and added to the area cost of the solution itself resulting in a new solution set $P'_{current}$. All invalid solutions, which require more device primitives than those available, are pruned in the mapping algorithm. The final step is to retain the Pareto optimal solutions from the set of updated solutions ($P'_{current}$) resulting in the final set of context adaptable architectures represented as P_{final}.

4.5 The Mapping Algorithm

The mapping algorithm is used to calculate the resources required to route the data among FUs, in order to support all the DFGs of a CDFG using the schedule information obtained from the $CAAE$ algorithm. Data routing resources refer to the multiplexers, which steer data from one FU to another, and delay registers, which delay the output data of an FU for an appropriate number of clock cycles before it is consumed by another FU. In step 8 of the $CAAE$ algorithm, the mapping algorithm is invoked for each solution in the Pareto optimal solution set ($P_{current}$). The mapping algorithm takes as inputs, schedules of all the DFGs, a set of number of resources of a solution and initial available device primitives (before the $CAAE$ algorithm is invoked). It outputs the estimated WSDP values for multiplexers and delay registers.

Before presenting my mapping algorithm, I will briefly discuss a couple of works already published for register and functional unit binding/mapping. Chen *et al.* [27] build a global compatibility graph for each type of operation present in the input CDFG. Each node in this graph represents an operation and the edge between two nodes represents if the two operations are compatible. All the compatible nodes can be mapped on to the same resource. As a first step, register mapping is performed such that each dataflow occupies its own register. Then, FU mapping is carried out. Their mapping algorithm iterates for as many times as the number of nodes present in the CDFG. In each iteration, the current node is mapped to all the possible FUs to create multiple solution points. The number of solutions at the end of each iteration keeps on growing, so only a fixed number of solution points (10) are retained for the next iteration. Solution points are pruned based on power consumption and not on area. Also, they have no way to guide the mappings such that the number and size of multiplexers is reduced.

Cromar *et al.* [28] iteratively constructs weighted bipartite graphs for the given scheduled CDFG. Each node in this graph represents an operation or operations. Edges are created between compatible nodes. All the compatible nodes can be mapped on to the same resource. Edge weights are assigned based on the difference of multiplexer widths

obtained as a result of mapping the two compatible nodes to the same FU. Nodes are combined based on maximum matching and the bipartite graph is reconstructed. Their method considers multiplexer balancing such that the size of multiplexers is reduced. However, their approach has no consideration for reusing the existing bus connections among the FUs which will potentially reduce the size and number of multiplexers.

A heuristic algorithm for mapping is therefore proposed in this section to overcome the deficiencies of the above approaches. The algorithm is based on creating and iteratively updating (i) a weighted bipartite graph for the scheduled CDFG like Cromar *et al.* [28] and (ii) FU mapping list ($FUList$). The weight of the edges in the bipartite graph depend on whether there exists a bus connection in the current FU mapping and the size of the multiplexer already present on an FU. $FUList$ stores the information regarding the node to FU mapping, FU-to-FU bus connections and the size of multiplexer at each input of the FU.

In order to perform efficient register mappings (i.e., reuse existing registers), register nodes (with latency equal to the number of delay cycles required) are first inserted between the data nodes of the scheduled CDFG where ever delay is required. Unlike Chen *et al.* [27], register nodes are mapped to delay registers during (and not before) FU mapping such that the number of delay registers is minimized. As a CDFG has multiple DFGs, FU mapping is performed for each DFG indivudually, but the weighted bipartite graph is used across the DFGs as the main goal is to find the mapping for the entire CDFG.

Mapping of a scheduled DFG is carried out using a matrix called the *reservation table* with the number of rows equal to the latency of the DFG and the number of columns equal to the total number of available resources present in the final resource set corresponding to the input solution. This table holds the associations of data nodes present in the DFG to the hardware resources. In order to reduce the number and size of multiplexers, when a node is mapped to a specific resource, the mappings of its parents' nodes are compared with the mappings of the parent nodes of the nodes mapped to the current resource. If a match is found, then the existing bus connections can be reused, and hence the current node is

mapped to the current resource. If a match is not found, the current node is mapped to a resource with the lowest number of node mappings to distribute resource mappings. Since the parent node mappings are used during child node mappings, data nodes are processed in increasing order of their schedules. Once the *reservation table* is populated with the mapping information of a DFG, a functional unit list ($FUList$) is generated. As a resource has two input ports, for each entry, $FUList$ maintains two lists - one for the left port and one for the right port. Each port list is populated with only distinct node and/or register mappings. The *reservation table* along with the $FUList$ is the manifestation of the weighted bipartite graph. As there are multiple DFGs, the *reservation table* is cleared and reused for each DFG and the port lists inside $FUList$ are updated. After all the DFGs are processed, the number of entries in each port list indicates the size of multiplexers required. The optimal order in which the DFGs should be processed in order to obtain an optimal mapping, is not explored in this algorithm.

As multiplexers are pure combinational circuits, they will be mapped only to LUTs on the FPGA. Xilinx Virtex-4 FPGAs have 4-input LUTs, so the problem is to find the number of 4-input LUTs to realize an n input multiplexer. A simple intuitive pseudo-code for obtaining a rough estimate on the number of LUTs required for a multiplexer is shown in fig. 4.3. The LUT usage for all the multiplexers is then accumulated as Mux_{LUT}.

Register entries in each port list of the $FUList$ indicate the number and depth (number of delay cycles) of the delay registers used in the architecture. To obtain an estimate of the resource utilization of delay registers, several 32-bit delay registers were created with various depths using the Xilinx CORE generator and the post P&R area requirements are observed. These values are tabulated in Table 4.1. By analyzing columns 2 and 3 in this table, formulae for LUT and FF usage can be derived as a function of number of delay cycles as

$$Reg_{LUT} = \left\lceil \frac{number\ of\ delay\ cycles}{16} \right\rceil * dataWidth, \tag{4.1}$$

Table 4.1: Post P&R vs. estimated values of resources used for delay registers.

Number of delay cycles	Post P&R		Estimated	
	LUT	FF	LUT	FF
8	32	32	32	32
16	32	32	32	32
24	64	32	64	32
32	64	32	64	32
40	96	32	96	32
48	96	32	96	32
56	128	32	128	32
64	128	32	128	32
96	192	64	192	64
128	256	64	256	64
160	320	96	320	96
192	384	96	384	96

$$Reg_{FF} = \left\lceil \frac{number\ of\ delay\ cycles}{64} \right\rceil * dataWidth. \tag{4.2}$$

For the purpose of comparison, the LUT and FF usage is estimated using (4.1) and (4.2) and is shown in Table 4.1 (columns 4 and 5). WSDPs of multiplexers and delay registers are returned to the $CAAE$ algorithm which are calculated using

$$W_{mux} = \frac{Mux_{LUT}}{a_{LUT}}, \tag{4.3}$$

$$W_{reg} = \left(\frac{Reg_{LUT}}{a_{LUT}}\right) + \left(\frac{Reg_{FF}}{a_{FF}}\right). \tag{4.4}$$

Estimate_LUT_MUX(Number of inputs (n), $dataWidth$)
Let p, L be two integer variables
If(n $\leq$ 1)
    return 0
Else
{
    $p = n$ + ceil($log_2 n$)
    $L = 0$
    while($\frac{p}{4} \neq 0$)
    {
        $L = L + \frac{p}{4}$
        $p = \frac{p}{4} + p\%4$
    }

    If($p > 1$)
        $L = L + 1$
}
return($L * dataWidth$)

Fig. 4.3: Pseudo code for estimating number of LUTs for a multiplexer.

Chapter 5

Complexity Analysis

This chapter presents the complexity analysis of the proposed $CAAE$ algorithm shown in Algorithm 4.2. The complexity of an algorithm is a theoretical measure of the running time of the algorithm for a given input problem size. For an input problem size of n, the algorithm is associated with a function $f(n)$ that characterizes the running time in terms of n using assymptotic notation or Big-Oh notation as

$$f(n) = O\big(g(n)\big). \tag{5.1}$$

Formal definition: Let $f(n)$ and $g(n)$ be functions mapping nonnegative integers to real numbers. If there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq cg(n)$ for every integer $n > n_0$, then we say that $f(n)$ is $O\big(g(n)\big)$ [20]. The input problem size cannot always be specified using only one variable. For multiple variables (represented as $\vec{x}$), the complexity of an algorithm can be associated with the function shown in (5.2).

$$f(\vec{x}) = O\big(g(\vec{x})\big) \tag{5.2}$$

For example, $f(n, m) = O\big(g(n, m)\big)$.

To simplify the complexity computation of large examples, few rules have been proposed by Goodrich and Tamassia [20], which are repeated here for the sake of clarity. Let $d(n)$, $e(n)$, $f(n)$, and $g(n)$ be functions mapping nonnegative integers to nonnegative reals.

Rule 1: If $d(n)$ is $O\big(f(n)\big)$, then $ad(n)$ is $O\big(f(n)\big)$, for any constant $a > 0$.
Rule 2: If $d(n)$ is $O\big(f(n)\big)$ and $e(n)$ is $O\big(g(n)\big)$, then $d(n) + e(n)$ is $O\big(f(n) + g(n)\big)$.

Rule 3: If $d(n)$ is $O\big(f(n)\big)$ and $e(n)$ is $O\big(g(n)\big)$, then $d(n)e(n)$ is $O\big(f(n)g(n)\big)$.

Rule 4: If $d(n)$ is $O\big(f(n)\big)$ and $f(n)$ is $O\big(g(n)\big)$, then $d(n)$ is $O\big(g(n)\big)$.

Rule 5: If $f(n)$ is a polynomial of degree d (that is $f(n) = a_0 + a_1 n + \cdots + a_d n^d$), then $f(n)$ is $O\big(n^d\big)$.

Rule 6: n^x is $O\big(a^n\big)$ for any fixed $x > 0$ and $a > 1$.

Rule 7: $logn^x$ is $O\big(logn\big)$ for any fixed $x > 0$.

Rule 8: $log^x n$ is $O\big(n^y\big)$ for any fixed constants $x > 0$ and $y > 0$.

For the CAAE algorithm, the input problem has two variables N and M, where N represents the number of nodes in a DFG and M represents the number of DFGs in a CDFG. The total complexity of the algorithm can be calculated by determining the complexity of individual steps and then adding them up according to Rule 2.

Step 1. Determine critical path latencies of all the DFGs. $CP(G_i)\ \forall\ i \in \{1, 2, ...n_{DFG}\}$.

Complexity to find critical path of one DFG (using Rules 2 and 1)

= (Complexity of ASAP algorithm) + (# of nodes in the DFG)

= (# of edges in the DFG) + (# of nodes in the DFG)

= $(2N)$ + (N)

$\simeq O(N)$

Complexity of Step 1 (using Rule 3)

= (# of DFGs) * (Complexity to find CP of one DFG)

= (M)*(N)

= $O(MN)$

Step 2. Calculate T_{min}

Complexity of Step 2 = $O(k)$; k represents constant.

Step 3. Create empty solution (S_{in}, W_{in}, A_{in}, T_{in})

Complexity of Step 3 = $O(k)$

Step 4. Using FDS find all schedules and resource sets of G_1.

In the $CAAE$ algorithm, the FDS algorithm is invoked N times with different timing constraints (relaxed from critical path latency untill the resource set obtained from FDS has only one resource of each type). So, N is the worst case number of schedules and worst case number of resource sets generated for G_1.

Complexity of Step 4 (using Rules 3 and 5)

= (# of times FDS is called) * (Complexity of FDS)

= (N) * (N^3)

$\simeq O(N^4)$

Step 5. For all resource sets, evaluate G_1 for WSDPs, CDFG execution time (using *Evaluate* algorithm) and generate a set of partial solutions ($P_{current}$).

Complexity of Step 5 (using Rules 3 and 5)

= Complexity of *Evaluate* algorithm

= (# of resource sets of G_1) * (# of nodes in G_1)

= (N) * (N)

$\simeq O(N^2)$

Step 6. Complexity of Step 6 = $O(k)$

Step 7.

Complexity of Step 7a = Complexity of Step 4 = $O(N^4)$

Complexity of Step 7b = $O(k)$

Complexity of Step 7c (using Rules 3 and 5)

= (worst case # of solutions after G_{i-1}) * (Complexity of *Evaluate* algorithm)

= (N) * (N^2)

$\simeq O(N^3)$

Complexity of Step 7d = $O(k)$

Complexity of Step 7e (using Rules 2 and 5)

= (Complexity of *Merge Sort* algorithm) + (worst case # of solutions)

= $O(NlogN) + N$

$\simeq O(NlogN)$

Complexity of Step 7 (using Rules 2, 3, and 5)

= (# of DFGs - 1) * complexity of Steps (7a + 7b + 7c + 7d + 7e)

= $(M-1)*(N^4 + k + N^3 + k + NlogN)$

$\simeq O(MN^4)$

Step 8. Calculate W_{mux} and W_{reg} using a mapping algorithm for each solution in the Pareto optimal solution set ($P_{current}$), and add them to the WSDP of each solution, resulting in a new set $P'_{current}$.

Complexity of Step 8 (using Rules 3 and 5)

= (# of solutions after Step 7) * (# of DFGs in the CDFG) *

(Complexity of the *Mapping* algorithm)

= (N) * (M) * (# of nodes in each DFG)

= (N) * (M) * (N)

$\simeq O(MN^2)$

Step 9. Retain Pareto optimal solutions from $P'_{current}$, resulting in a new set P_{final}.
Complexity of Step 9 = Complexity of Step 7e = $O(NlogN)$

Step 10. Complexity of Step 10 = $O(k)$

The total complexity of the algorithm can be calculated using Rule 2, by combining the complexities of all the steps (listed in Table 5.1) and then removing the lower order terms using Rule 5.

$\therefore$ Total Complexity of the algorithm = O(MN^4)

Table 5.1: Computation complexities of all steps.

Step	$O\Big(f\big(M,N\big)\Big)$
1	MN
2	k
3	k
4	N^4
5	N^2
6	k
7	MN^4
8	MN^2
9	$NlogN$
10	k

Chapter 6

Results

This chapter presents and analyzes the results generated by the proposed methodology in comparison with the prior work for various benchmarks taken from multiple application domains. Table 6.1 presents the test cases taken from applications including GNU scientific library (GSL), MPEG-2 video codec, MPEG-4 audio decoder, and H.264 video encoder. The control structures of these test cases are shown in fig. 6.1. Each test case represents a CDFG with BBs embedded in different types of control structures. BBs shaded in gray are composed of nontrivial DFGs (multiple nodes), which are major candidates for the architecture generation process and BBs shown in white are composed of trivial DFGs (single node). The weighting factors of the BBs are calculated wherever the information is known at compile time; otherwise random values (in the range 0 and 1.0) are assumed.

Section 6.1 discusses the accuracy of the proposed area estimation technique. Section 6.2 provides analysis of the proposed methodology. Sections 6.3 and 6.4 present a comparison with prior work including a datapath merging approach [16], and template-based algorithms [17–19]. Results are provided for architectures in terms of (i) relative area cost (in WSDP units), (ii) execution time (in clock cycles), and (iii) resource utilization (in terms of number of individual device primitives). As FPGAs are composed of four types of device primitives, the resource utilization is provided for each of the four device primitives for all the test cases. For all test cases, the Xilinx Virtex 4 family of FPGAs is considered as the target device and Microblaze (v7.10) [29] as the soft core processor wherever used. As the proposed methodology generates, a set of solutions with different execution times and area costs, during comparison, only one solution is chosen which is at least as good as or better than architectures generated by other approaches.

Table 6.1: Test cases.

	Test case	Notation	Data type	Source code	# of non-trivial DFGs
1	Absolute Difference Error	ADE	Integer	MPEG-2 encoder	3
2	2D-Inverse Discrete Cosine Transform	IDCT	Integer	MPEG-2 encoder	2
3	Search One Pixel	SOP	Integer	H.264 encoder	4
4	Sub Sampling	SS	Integer	MPEG-2 decoder	4
5	Cylindrical Bessel	CB	Floating point	GSL	5
6	Audio Synthesizer	AS	Floating point	MPEG-4 audio decoder	2
7	Wavelet Transform	WT	Floating point	GSL	3
Note: All floating point data types are single precision.					

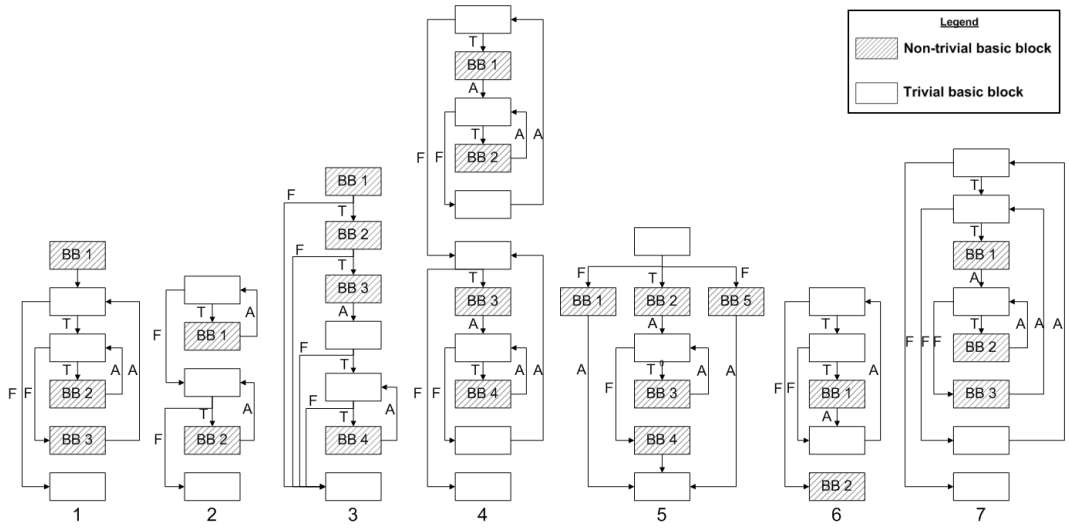


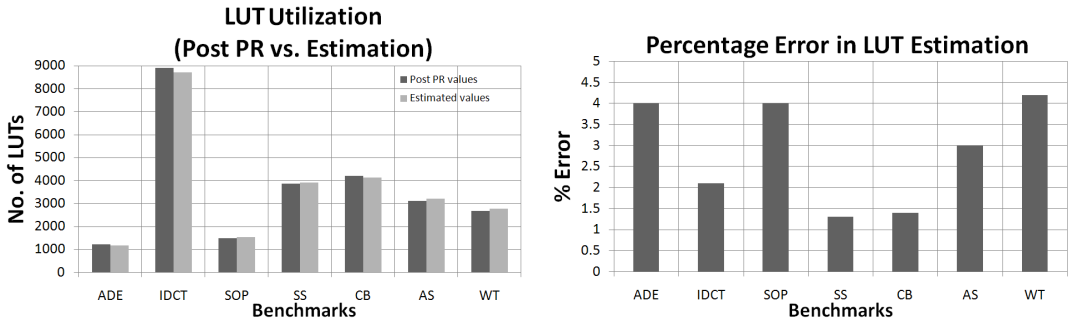
Fig. 6.1: CDFGs of test cases shown in Table 6.1. (T: true branch, F: false branch, A: always taken branch).

6.1 Accuracy of the Proposed Area Estimation Technique

The context adaptable architectures derived using the proposed methodology are implemented in Verilog, synthesized and post P&R resource utilizations are obtained using Xilinx ISE 10.1. The Xilinx Virtex-4 SX35 FPGA [30] is used as the target device to realize the hardware architectures.

An estimation technique proposed earlier in Chapter 3 is used to estimate the resource utilization of a circuit. To confirm the correctness and accuracy of this technique, the estimated values are compared against the actual post P&R resource utilization of architectures generated by the proposed methodology as shown in fig. 6.2. A comparison of LUT

utilization is shown in fig. 6.2(a). It can be observed from fig. 6.2(b) that the percentage error in LUT estimation was less than 4.5% for all the test cases. Similarly, the percentage error in FF, DSP48, and BRAM resource utilization estimation was observed to be 0.2%, 0%, and 0%, respectively. Table 6.2 shows the absolute error (in %) in area estimation of the proposed algorithm in comparison with area estiamtions used by other approaches ([22, 24, 25]). It can be observed that the absolute error of the proposed methodology is the least.



(a) (b)

Fig. 6.2: (a) Comparing the estimated against the actual post P&R LUT utilization, (b) Percentage error in the estimated LUT utilization when compared against post P&R values.

Table 6.2: Comparison of absolute error in area estimations.

Authors	Estimator Input	Absolute Error (%)	Handle Memory units?
Nayak *et al.* [24]	MATLAB	16	No
Kulkarni *et al.* [25]	SA-C	6.3	No
Bilavarn *et al.* [22]	CDFG	20	Yes
Proposed Algorithm	CDFG	**4.5**	Yes

6.2 Analysis of Proposed Methodology

6.2.1 Updating Available Device Primitives

In the proposed methodology, scheduling and resource selection are performed simultaneously. For each additional resource required by the resource set of a scheduled DFG, WSDPs of all implementations are calculated and the option with the least WSDP is selected and the number of available device primitives is updated. The advantage of evaluating WSDPs of implementations, after allocation of each resource, is discussed here. Consider a scheduled DFG (say G_1), which requires three floating-point multiplication units and two floating-point addition units. Assume that device primitives available for mapping this DFG are: 1500 LUTs, 1710 FFs, and 18 DSP48s (note that the available device primitives for mapping a DFG/CDFG need not be the total number of device primitives present on the un-configured FPGA). Routing resources are neglected, for sake of discussion. Table 6.3 shows the different implementations of the multiplier and adder, and their corresponding area costs (in WSDPs) are presented for different sets of available device primitives.

For the initial set of available device primitives, resources FMUL-2 and FADD-2 have the least area cost among other implementations of the same type. If these two implementations are chosen for multipliers (all three) and adders (both) for DFG G_1, then a total of 1148 LUTs, 1588 FFs, and 20 DSP48s are consumed. As there are only 18 DSP48s available, this is an invalid solution. Therefore, a new approach of selecting implementations for one functional unit at a time and then recomputing the area costs of different implementations (as shown in columns 5 through 9 and the last row) is proposed. In column 5, as FMUL-2 has the least area cost (shown in bold text) among all multiplier implementations, therefore resources are allocated for it, available device primitives are updated, and area costs are recomputed in column 6. Again FMUL-2 has the least area cost. However, in column 8 we see that FADD-1 has the least area cost, but when the device primitives were updated in column 9, FADD-2 has the least area cost. Using this approach, the final circuit consumes 1394 LUTs, 1710 FFs, and 16 DSP48s, which is a valid solution.

Table 6.3: Illustration of updating available device primitives and WSDPs.

Resource Type	Required device primitives			Available device primitives in WSDPs for (LUT/FF/DSP48)				
	LUT	FF	DSP48	1500/1710/18	1348/1504/14	1196/1298/10	1044/1092/6	452/485/6
FMUL-1	653	703	0	0.85	0.95	1.09	1.27	2.89
FMUL-2	152	206	4	**0.44**	**0.54**	**0.69**	1	1.43
FMUL-3	121	185	5	0.47	0.57	0.74	1.12	1.48
FADD-1	592	607	0	0.75	0.84	0.96	**1.12**	2.56
FADD-2	346	485	4	0.74	0.86	1.06	1.44	**2.43**
Allocated resources				1^{st} FMUL	2^{nd} FMUL	3^{rd} FMUL	1^{st} FADD	2^{nd} FADD

6.2.2 Optimal Resource Selection

In the following sub-section, solutions generated by the proposed heuristic resource selection algorithm are compared with optimal solutions. The optimal solutions are generated by exhaustively listing all possible combinations of resources and their implementations. To reduce the complexity of testing, the following assumptions were made: (i) there is only one operation type in the entire CDFG, (ii) each DFG has only one schedule (say, critical path schedule), and (iii) resources available for mapping operations are implemented using only LUTs and/or DSP48s (FFs and BRAMs are not considered). Note that these assumptions are not limitations of the algorithm itself. As each DFG has only one schedule, there exists only one final solution for comparison, rather than a set of solutions as explained in the $CAAE$ algorithm. Also, the execution time of the CDFG remains the same for the proposed heuristic resource selection and optimal resource selection. We have generated five synthetic test cases, each with different number of DFGs and operation types as shown in Table 6.4. For each test case, the input available device primitives (LUTs and DSP48s only) are varied resulting in large number of test points as shown in column 3 of Table 6.4. Each test point represents one combination of device primitives. The $CAAE$ algorithm proposed in this thesis is run for each test point and the resource utilization is noted. Also, the optimal resource utilization is determined by exhaustively evaluating all the resource combinations. Both the values are then compared and the number of test points for which my algorithm generates optimal solutions is tabulated in column 4. The maximum and average area cost overhead of the solutions generated using the proposed heuristic resource

Table 6.4: Percentage area overhead of the proposed approach compared to optimal solution.

Number of DFGs	Operation type	Number of total test points	Number of test points with optimal solution	Max. area overhead over optimal solution (in %)	Avg. area overhead over optimal solution (in %)
5	FMUL	960	679	10.87	1.14
4	FMUL	1440	664	15.40	2.72
3	FMUL	1440	820	12.42	1.83
6	IMUL	720	720	0	0
4	IMUL	1300	1151	19.40	1.38

selection algorithm when compared to the optimal solutions is listed in columns 5 and 6 respectively. As we can see, though the maximum area overhead is approximately 20%, the average area overhead was not more that 3%. However, it is important to note that, as the proposed algorithm is a heuristic, these values are greatly dependent on the amount of available device primitives at the beginning of the algorithm. The computation complexity of the proposed resource selection algorithm is $O(nk)$ as opposed to $O(n^k)$ for the exhaustive search algorithm, where n is the maximum number of resources of any type and k is the maximum number of implementations for any resource.

6.3 Comparison with Datapath Merging Approach

This section evaluates the proposed architecture against the architectures generated using the Datapath Merging (DM) approach proposed by Moreano *et al.* [16]. For the DM approach, only DFGs present inside loops are processed in the decreasing order of number of nodes. The remaining DFGs (not present inside loops, but capable of hardware acceleration) are instead mapped onto a soft-processor. The area of the final merged architecture is calculated using

$$Area_{DM} = Area_{merged} + Area_{static}, \tag{6.1}$$

$$Area_{merged} = \sum_{j=1}^{V_R-1} C_j R_j + Area_{multiplexers}, \tag{6.2}$$

where $Area_{static}$ is the area cost associated with a soft-processor (Microblaze on Xilinx FPGAs), which has a constant value depending on the type of operations it has to execute as shown in Table 6.5, and the number of available device primitives (using (3.1)). In (6.2), C_j represents the total number of operations of type j present in the merged graph, V_R is the number of distinct operations, and R_j is the area cost (WSDP) of a resource that can implement operation j. $Area_{multiplexers}$ is calculated using the pseudo code shown in fig. 4.3.

The DM approach is applied for the seven test cases and a comparison is provided in fig. 6.3 against the proposed method. Depending on the size of the test case, different sets of available device primitives are fed to the $CAAE$ algorithm, as shown in Table 6.6. It can be seen that the proposed method generates a faster and smaller circuit. Figure 6.3(c) shows the final resource utilization for each of the individual device primitives for all the seven test cases. It can be observed that, though the proposed method consumes more DSP48s, it results in a lower final area cost (fig. 6.3(a)) of the circuit by balancing out the device primitives. Note that DM approach failed to generate a circuit that can fit the target FPGA for one test case (IDCT) where the number of data nodes is large (∼80).

6.4 Comparison with Template-Based Approaches (Prior Work)

The algorithms presented by Guo *et al.* [17], Kastner *et al.* [18], and Cong *et al.* [19] propose to accelerate applications by extracting hardware templates for configurable processors. Hence, we extend their methodology for applications involving control flow

Table 6.5: Three configurations of the Microblaze (v7.10) processor in Xilinx Virtex 4 FPGAs.

Microblaze type	LUT	FF	DSP48	BRAM
Integer units only	1955	1163	3	32
Integer + FP† unit (Basic)	2936	1611	7	32
Integer + FP† unit (Extended§)	3483	1926	7	32
†FP: Floating Point (single precision) §Extended: Support for square root operations				

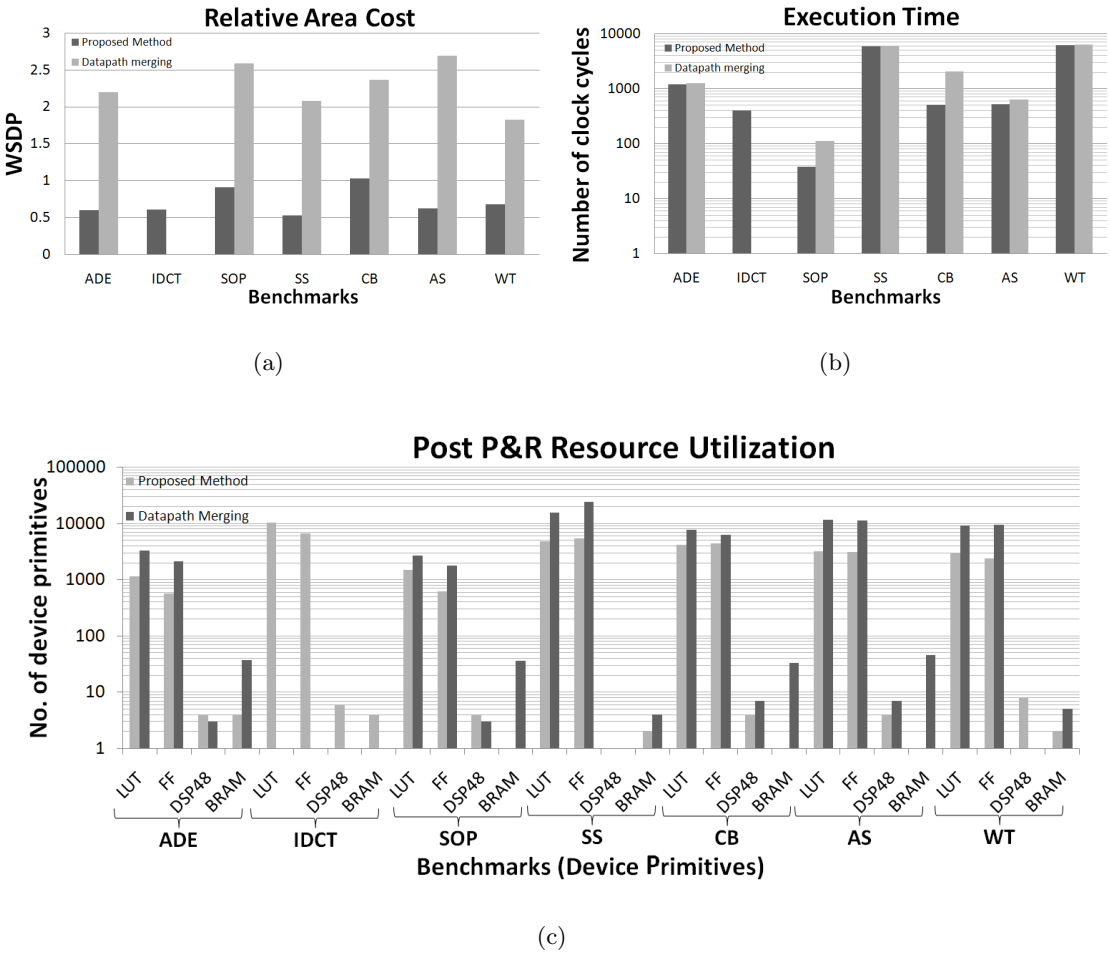


Fig. 6.3: Comparison with datapath merging approach: (a) Relative area cost, (b) CDFG execution time, and (c) Resource utilization for individual FPGA device primitives.

Table 6.6: Available device primitives used to obtain results shown in fig. 6.3.

	Test case	LUT	FF	DSP	BRAM
1	ADE	4000	3000	50	60
2	IDCT	30720	30720	50	50
3	SOP	3000	2000	50	50
4	SS	17000	26000	20	20
5	CB	9000	9000	50	50
6	AS	12000	12000	50	70
7	WT	12000	12000	50	20

and compare with the proposed approach. For each test case, templates are identified using these three approaches and are used to cover the DFGs of a CDFG, individually. These templates are potential candidates for hardware acceleration, and are hence termed as nontrivial templates (NT). The uncovered operations of all DFGs are identified as trivial templates (single node) and are executed on a software processor (Microblaze). As there is no mention on the number of instances of NT templates, only one instance of each template is assumed to be present in the circuit. Unlike Guo *et al.* [17] and Kastner *et al.* [18], the hardware templates used in Cong *et al.* [19] are assumed to have nonoverlapped execution, because only one application specific instruction corresponding to a template can be executed at any given time. For every NT template, the corresponding DFG is scheduled using FDS and a set of number of resources $\{n_r \mid r \in R\}$ is obtained. The area cost of a single NT template k is calculated using

$$Area(NT_k) = \sum_{r \in R} n_r W_r, \tag{6.3}$$

where W_r is the relative area cost (WSDP) of resource r. The overall area cost is computed as summation of the area costs of all the m NT templates and n T templates is calculated using

$$Area_{prior} = \begin{cases} \left(\sum_{k=0}^{m} Area(NT_k)\right) & if\ n\ =\ 0 \\ \left(\sum_{k=0}^{m} Area(NT_k)\right) + Area_{static} & if\ n\ \neq\ 0 \end{cases}. \quad (6.4)$$

In the presence of trivial templates, area cost of the Microblaze ($Area_{static}$) is added.

Figure 6.4 shows the relative area cost and execution time of architectures generated by the proposed approach and the three template based approaches, for all the test cases. For each test case the available device primitives used at the beginning of the algorithm are shown in Table 6.7. From fig. 6.4(a) it can be seen that the architectures generated by the proposed methodology have the least relative area cost for all the test cases except for IDCT, when compared with Cong *et al.* [19]. This is because the number of distinct NT templates present in the graph is very small (though the number of templates required to cover the graph is large) and we assumed only one instance of each template to be present in the hardware which requires less circuit. Note that the area cost for the proposed architecture includes the resources for data routing (i.e., delay registers and multiplexers), whereas, the area cost for the architectures generated using other approaches include only the area cost of arithmetic/logic resources. However, the penalty can be observed in the execution time difference. From fig. 6.4(b) it can be observed that the execution time for architecture generated for IDCT using Cong *et al.*'s method is 5072 cycles, whereas for the architecture generated using the proposed algorithm is only 600 cycles. That is to say, when compared with Cong *et al.*'s method, though the proposed algorithm has an area overhead of 5%, savings of 88% can be observed in execution time.

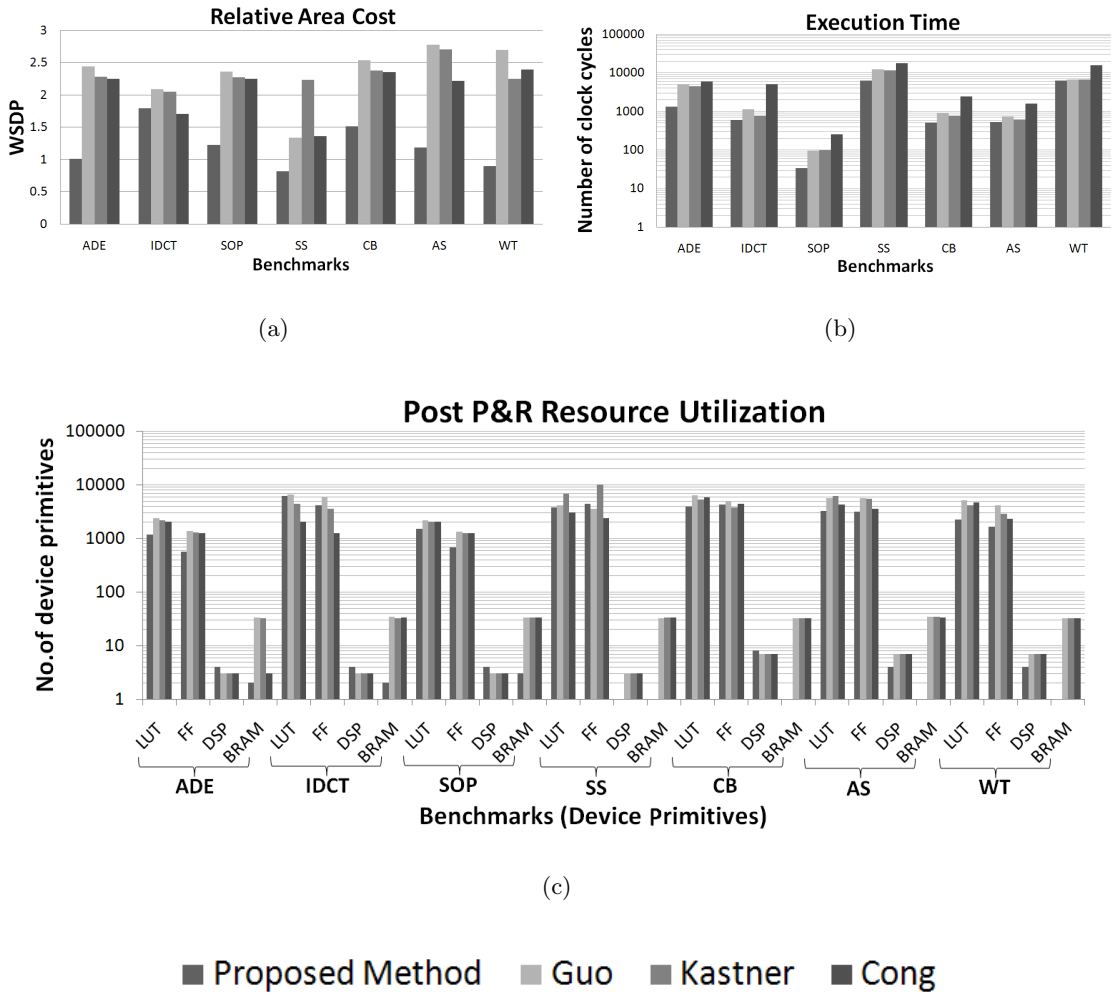


Fig. 6.4: Comparison of proposed architectures and architectures generated using Guo *et al.*, Kastner *et al.*, and Cong *et al.*: (a) Relative area cost, (b) CDFG execution time, and (c) Resource utilization for individual FPGA device primitives.

Table 6.7: Available device primitives used to obtain results shown in fig. 6.4.

	Test case	LUT	FF	DSP	BRAM
1	ADE	3000	2000	10	50
2	IDCT	6700	6050	20	50
3	SOP	3000	2000	10	50
4	SS	9000	11000	20	50
5	CB	7000	6000	20	50
6	AS	7000	6000	20	50
7	WT	6000	5000	20	50

Chapter 7

Conclusion and Future Work

This thesis presents a methodology to derive context adaptable architectures for FPGA that can support multiple DFGs contained in the CDFG of an application. An area metric (WSDP) that is based on the heterogeneous mixture of device primitives in an FPGA is presented and is used to guide resource selection, when multiple implementations for a particular resource type are available. A context adaptable architecture (CAA) template is presented and a CAA exploration (CAAE) algorithm, which includes heuristic-based scheduling, resource selection, and mapping algorithms, is described in detail.

Architectures generated by the proposed methodology are compared against those generated using other published techniques. The test cases used for benchmarking are obtained from multiple applications domains. Overall WSDP, execution times, and resource utilization (in terms of number of device primitives) are used as metrics for comparison. The proposed methodology outperformed the other published techniques by generating smaller (an average savings of 46% in WSDP) and faster (an average savings of 46% in execution times) architectures.

Future research can be carried out in the following directions:

(1) Resource implementations for multiple latencies can be explored, which will have an impact on the execution time of the CDFG, FPGA resource utilization, and maximum clock frequency of the generated architecture.

(2) Maximum clock frequency of the circuit can be studied by determining the absolute time delays of all the components of the architecture and identifying the critical path. The clock frequency can be improved by inserting registers at appropriate places in the architecture.

(3) Power consumed by the architecture can be anayzed and find ways to reduce it.

(4) Come up with an efficient way to map arrays present in the CDFG to the FPGA on-chip memory.

(5) Perform partial or full unrolling of loops to decrease the execution time of the CDFG.

References

[1] Xilinx, "Virtex-4 fpga user guide," [http://www.xilinx.com/support/documentation/user_guides/ug070.pdf], 2008.

[2] Altera, "Stratix device handbook," [http://www.altera.com/literature/hb/stx/stratix_handbook.pdf], 2005.

[3] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, "A formal approach to the scheduling problem in high level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 4, pp. 464–475, Apr. 1991.

[4] C.-J. Tseng and D. Siewiorek, "Automated synthesis of data paths in digital systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 5, no. 3, pp. 379–395, July 1986.

[5] S. Y. Kung, H. J. Whitehouse, and T. Kailath, *VLSI and Modern Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1985.

[6] B. Pangrle and D. Gajski, "Design tools for intelligent silicon compilation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, no. 6, pp. 1098–1112, Nov. 1987.

[7] P. G. Paulin and J. P. Knight, "Force-directed scheduling in automatic data path synthesis," in *DAC '87: Proceedings of the 24th ACM/IEEE Design Automation Conference*, pp. 195–202. New York: Association for Computing Machinery, 1987.

[8] P. Paulin and J. Knight, "Force-directed scheduling for the behavioral synthesis of asics," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 6, pp. 661–679, June 1989.

[9] W. F. Verhaegh, E. H. Aarts, J. H. Korst, and P. E. Lippens, "Improved force-directed scheduling," in *EURO-DAC '91: Proceedings of the Conference on European Design Automation*, pp. 430–435. Los Alamitos: IEEE Computer Society Press, 1991.

[10] J. Siddhiwala and L.-F. Chao, "Scheduling conditional data-flow graphs with resource sharing," in *Proceedings of the Fifth Great Lakes Symposium on VLSI (GLSVLSI'95)*, pp. 94–97. Washington, DC: IEEE Computer Society, 1995.

[11] R. Camposano, "Path-based scheduling for synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 1, pp. 85–93, Jan. 1991.

[12] H. Al-Sukhni, H. Youssef, S. Sait, and M. Benten, "Loop based scheduling for high level synthesis," in *IEEE International Conference on Computers and Communications*, pp. 76–81, Mar. 1995.

[13] R. Bergamaschi, S. Raje, I. Nair, and L. Trevillyan, "Control-flow versus data-flow-based scheduling: combining both approaches in an adaptive scheduling system," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 5, no. 1, pp. 82–100, Mar. 1997.

[14] T. Kim, J. Liu, and C. Liu, "A scheduling algorithm for conditional resource sharing," in *IEEE International Conference on Computer-Aided Design*, pp. 84–87, Nov. 1991.

[15] G. Lakshminarayana, A. Raghunathan, and N. Jha, "Incorporating speculative execution into scheduling of control-flow-intensive designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 3, pp. 308–324, Mar. 2000.

[16] N. Moreano, E. Borin, C. de Souza, and G. Araujo, "Efficient datapath merging for partially reconfigurable architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 7, pp. 969–980, July 2005.

[17] Y. Guo, G. Smit, H. Broersma, and P. Heysters, "Template generation and selection algorithms," in *The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications*, pp. 2–5, June 2003.

[18] R. Kastner, A. Kaplan, S. O. Memik, and E. Bozorgzadeh, "Instruction generation for hybrid reconfigurable systems," *Association for Computing Machinery Transactions on Design Automation of Electronic Systems*, vol. 7, no. 4, pp. 605–627, 2002.

[19] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," in *FPGA '04: Proceedings of the ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, pp. 183–189. Association for Computing Machinery, Jan. 2004.

[20] M. T. Goodrich and R. Tamassia, *Algorithm Design: Foundations, Analysis and Internet Examples*. New York: John Wiley & Sons, Inc., 2009.

[21] J. Phillips, A. Sudarsanam, H. Samala, R. Kallam, J. Carver, and A. Dasu, "Methodology to derive context adaptable architectures for fpgas," *Institution of Engineering and Technology - Computers & Digital Techniques*, vol. 3, no. 1, pp. 124–141, Jan. 2009.

[22] S. Bilavarn, G. Gogniat, J.-L. Philippe, and L. Bossuet, "Design space pruning through early estimations of area/delay tradeoffs for fpga implementations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1950–1968, Oct. 2006.

[23] S. A. Blythe and R. A. Walker, "Efficient optimal design space characterization methodologies," *Association for Computing Machinery Transactions on Design Automation of Electronic Systems*, vol. 5, no. 3, pp. 322–336, 2000.

[24] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee, "Accurate area and delay estimators for fpgas," in *Proceedings on Design, Automation and Test in Europe Conference and Exhibition*, pp. 862–869, 2002.

[25] D. Kulkarni, W. Najjar, R. Rinker, and F. Kurdahi, "Fast area estimation to support compiler optimizations in fpga-based reconfigurable systems," in *10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 239–247, 2002.

[26] J. D. Phillips, *A C to RTL Algorithm Using Structured Circuit Templates: A Case Study with Simulated Annealing*. Ph.D. dissertation, Utah State University, Logan, UT, Dec. 2008.

[27] D. Chen, J. Cong, Y. Fan, and Z. Zhang, "High-level power estimation and low-power design space exploration for fpgas," in *Asia and South Pacific Design Automation Conference*, pp. 529–534, Jan. 2007.

[28] S. Cromar, J. Lee, and D. Chen, "Fpga-targeted high-level binding algorithm for power and area reduction with glitch-estimation," in *46th ACM/IEEE Design Automation Conference*, pp. 838–843, July 2009.

[29] Xilinx, "Microblaze soft processor core," `[http://www.xilinx.com/tools/microblaze.htm]`, 2009.

[30] Xilinx, "Virtex-4 family overview," `[http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf]`, 2007.