Univerzitet u Beogradu
Elektrotehnički fakultet

Mirjana Ž. Stojilović

METODA PROJEKTOVANJA NAMENSKIH PROGRAMABILNIH HARDVERSKIH AKCELERATORA

Doktorska disertacija

Beograd, 2013

University of Belgrade
School of Electrical Engineering

Mirjana Ž. Stojilović

A METHOD FOR DESIGNING DOMAIN-SPECIFIC RECONFIGURABLE ARRAYS

Doctoral Dissertation

Belgrade, 2013

Mentor:

dr Lazar Saranovac, vanredni profesor,
Univerzitet u Beogradu – Elektrotehnički fakultet

Članovi komisije:

dr Lazar Saranovac, vanredni profesor,
Univerzitet u Beogradu – Elektrotehnički fakultet

dr Jelena Popović Božović, docent
Univerzitet u Beogradu – Elektrotehnički fakultet

dr Rastislav Struharik, docent
Univerzitet u Novom Sadu – Fakultet tehničkih nauka

dr Predrag Pejović, redovni profesor
Univerzitet u Beogradu – Elektrotehnički fakultet

dr Milan Ponjavić, docent
Univerzitet u Beogradu – Elektrotehnički fakultet

Datum odbrane:

To my parents, for their immense love and support...

METODA PROJEKTOVANJA NAMENSKIH PROGRAMABILNIH HARDVERSKIH AKCELERATORA

Rezime—Namenski računarski sistemi se najčesće projektuju tako da mogu da podrže izvršavanje većeg broja željenih aplikacija. Za postizanje što veće efikasnosti, preporučuje se korišćenje specijalizovanih procesora *Application Specific Instruction Set Processors–ASIPs*, na kojima se izvršavanje programskih instrukcija obavlja u za to projektovanim i nezavisnim hardverskim blokovima (akceleratorima). Glavni razlog za postojanje nezavisnih akceleratora jeste postizanje maksimalnog ubrzanja izvršavanja instrukcija. Međutim, ovakav pristup podrazumeva da je za svaki od blokova potrebno projektovati integrisano (ASIC) kolo, čime se bitno povećava ukupna površina procesora. Metod za smanjenje ukupne površine jeste primena *Datapath Merging* tehnike na dijagrame toka podataka ulaznih aplikacija. Kao rezultat, dobija se jedan programabilni hardverski akcelerator, sa mogućnosću izvršavanja svih željenih instrukcija. Međutim, ovo ima negativne posledice na efikasnost sistema.

Često se zanemaruje činjenica da, usled veoma ograničene fleksibilnosti ASIC hardverskih akceleratora, specijalizovani procesori imaju i drugih nedostataka. Naime, u slučaju izmena, ili prosto nadogradnje, specifikacije procesora u završnim fazama projektovanja, neizbežna su velika kašnjenja i dodatni troškovi promene dizajna. U ovoj tezi je pokazano da zahtevi za fleksibilnošću i efikasnošću ne moraju biti međusobno isključivi. Demonstrirano je je da je moguce uneti ograničeni nivo fleksibilnosti hardvera tokom dizajn procesa, tako da dobijeni hardverski akcelerator može da izvršava ne samo aplikacije definisane na samom početku projektovanja, već i druge aplikacije, pod uslovom da one pripadaju istom domenu. Drugim rečima, u tezi je prezentovana metoda projektovanja fleksibilnih namenskih hardverskih akceleratora. Eksperimen-

talnom evaluacijom pokazano je da su tako dobijeni akceleratori u većini slučajeva samo do 2× veće površine ili 2× većeg kašnjenja od akceleratora dobijenih primenom *Datapath Merging* metode, koja pritom ne pruža ni malo dodatne fleksibilnosti.
Ključne reči: Arhitektura procesora, CGRA, fleksibilnost, FPGA, hardverski akceleratori, rekonfigurabilnost, specijalizacija.

Naučna oblast: Tehničke nauke – elektrotehnika.

Uža naučna oblast: Elektronika.

UDK broj: 621.3.

A METHOD FOR DESIGNING DOMAIN-SPECIFIC RECONFIGURABLE ARRAYS

Abstract—Typically, embedded systems are designed to support a limited set of target applications. To efficiently execute those applications, they may employ Application Specific Instruction Set Processors (ASIPs) enriched with carefully designed Instructions Set Extension (ISEs) implemented in dedicated hardware blocks. The primary goal when designing ISEs is efficiency, i.e. the highest possible speedup, which implies synthesizing all critical computational kernels of the application dataflow graphs as an Application Specific Integrated Circuit (ASICs). Yet, this can lead to high on-chip area dedicated solely to ISEs. One existing approach to decrease this area by paying a reasonable price of decreased efficiency is to perform datapath merging on input dataflow graphs (DFGs) prior to generating the ASIC.

It is often neglected that even higher costs can be accidentally incurred due to the lack of flexibility of such ISEs. Namely, if late design changes or specification upgrades happen, significant time-to-market delays and nonrecurrent costs for redesigning the ISEs and the corresponding ASIPs become inevitable. This thesis shows that flexibility and efficiency are not mutually exclusive. It demonstrates that it is possible to introduce a limited amount of hardware flexibility during the design process, such that the resulting datapath is in fact reconfigurable and thus can execute not only the applications known at design time, but also other applications belonging to the same application-domain. In other words, it proposes a methodology for designing domain-specific reconfigurable arrays out of a limited set of input applications. The experimental results show that resulting arrays are usually around 2× larger and 2× slower than ISEs synthesized using datapath merging, which have practically null flexibility beyond the design set of DFGs.

Key words: CGRA, datapath, domain-specific customization, flexibility, FPGA routing.

Scientific area: Technical sciences, Electrical engineering.

Specific scientific area: Electronics.

UDK number: 621.3.

Contents

List of Figures

List of Tables

1 Introduction

Embedded systems often use specialized hardware accelerators to improve performance and reduce energy consumption [Smi97, IL06], especially for applications involving signal and video processing, communications, and computer vision, among others. These accelerators can be either designed and synthesized for each target application separately, or their hardware implementations can be merged into one reconfigurable accelerator to reduce total die area [BKS04, ZT09]. This so called *datapath merging* approach for creating multi-operational datapaths helps reducing the total area of the accelerator, on one side, but leads to increased latency and thus impaired accelerator performance, on the other side. Besides the requirements for low area and high performance, there is another, increasingly important design criterion—the flexibility of reconfigurable accelerators. This flexibility, or re-usability, is mandatory to accommodate late design changes or new applications in the same domain, and to avoid extremely high Nonrecurring Engineering (NRE) costs of incremental chip redesign.

For a given set of applications, the ideal accelerator minimizes difference in terms of performance, energy consumption, and area in comparison to an ASIC implementation of the accelerated functionalities. However, flexibility per se imposes some unavoidable overheads. For example, FPGAs provide high flexibility, but suffer from incredibly poor logic density, even when designers make good usage of hard DSP block, block RAMs,

and transceivers. Thus, despite many efforts, no high-volume commercially successful product, to date, has successfully embedded FPGAs into ASIC design flows. The other alternative, datapath merging, incurs limited overhead in the form of a minimal number of multiplexers inserted into the reconfigurable datapath. Hence, it offers very little, if any, flexibility beyond the ability to accelerate the applications known at the design time. Clearly, it is hard not only to define the desired form and amount of flexibility, but also to limit the unavoidable overheads to a reasonable amount.

This thesis presents a novel approach for designing domain-specific coarse-grained arrays, in a context in which only a subset of the applications that need to be accelerated are known at design-time. The approach guarantees that all DFGs in the initial set of applications can map successfully onto the array, and increases the likelihood that structurally similar DFGs from the same or similar domains can map successfully as well. Obviously, the approach exploits the fact that applications belonging to the same domain share significant amount of computational similarity. For instance, dataflow graphs representing FIR or IIR filters differ very little from DFGs representing auto-correlation application. Unlike FPGAs, which are more appropriate for applications involving bit-level logic and bit manipulations, coarse-grain datapaths are better suited for multimedia applications that require word-level processing. Additionally, the size of configuration bitstream for FPGAs is much higher, implying the configuration storage overhead and longer reconfiguration time.

The experimental evaluation demonstrates that this novel approach achieves flexibility at the reasonable area and delay overhead:

- The vast majority of DFGs can be mapped onto the domain-specific arrays created for some other DFGs in the same application domain.
- Flexible arrays are only around 2× larger and 2× slower than an accelerator synthesized using a well-known datapath merging technique [BKS04].
- At the same time, these domain-specific arrays are only about 15× larger and

2× slower than an ASIC implementation of a single DFG in isolation, and thus far more efficient than FPGAs, which are known to be 20–40× larger and 3–4× slower [KR07].

1.1 The Problem

To illustrate the problem motivating this thesis, a couple of dataflow graphs representing applications that belong to digital filtering domain are analyzed. The DFGs D_1 and D_2, shown in Figure 1.1a and 1.1b, correspond to a 16-bit complex finite impulse response (FIR) filter and a 16b least mean square adaptive filter [TI03a], respectively. The former DFG D_1 has ten input ports, two output ports, eight multipliers (MUL), two subtractors (SUB), and six adders (ADD). The latter DFG D_2 has seven input ports, one output port, four multipliers, two shift right operators (SHR), and four adders. Input and output ports serve to read data from or to write data into memory elements. Clearly, the DFGs share some computational characteristics—in both of them some multiplications are followed by additions, and the intermediate results are summed. If both applications need to be accelerated, both DFGs should be implemented as individual ASIC circuits or they should be merged first and then synthesized as a single reconfigurable ASIC circuit to conserve die area [BKS04, ZT09]. The second approach adds very little flexibility in the final datapath, as will be explained and shown now.

The state of the art datapath merging approach by Brisk et al. [BKS04] looks for the highest-area common sequences of operators in D_1 and D_2 and attempts merging the two DFGs by sharing these operators. To enable the reconfiguration, it adds multiplexers in front of the merged operators. To achieve the best area savings, the previous two steps are repeated until no merging opportunities are left. Without loss of generality, it can be assumed that the areas and delays of operators are related as follows:

$$Area(MUL) > Area(SUB) > Area(ADD) > Area(SHR), \text{ and}$$

$$Delay(MUL) > Delay(SUB) > Delay(ADD) > Delay(SHR).$$

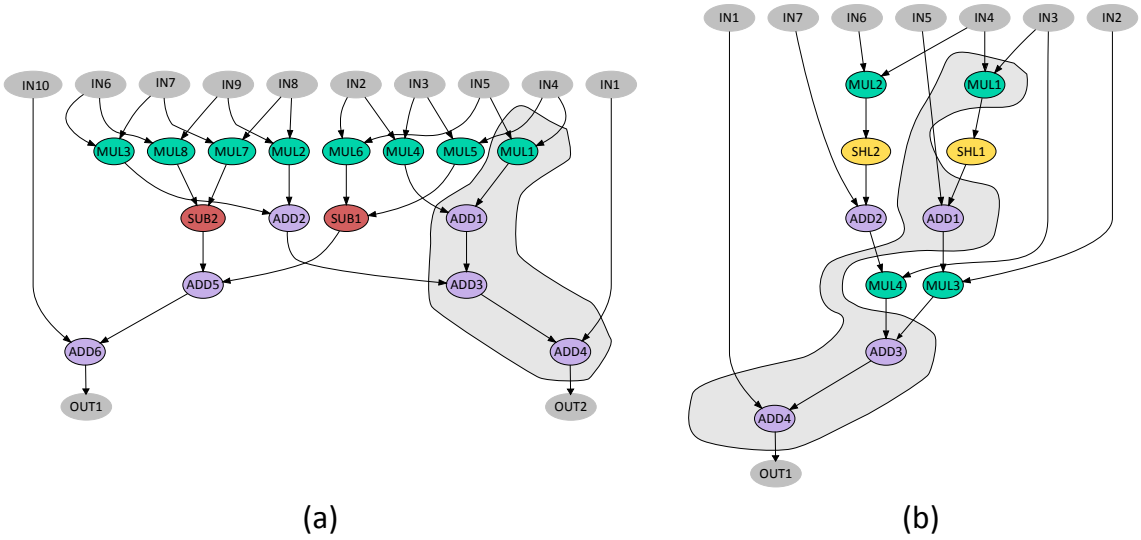


Figure 1.1: Dataflow graphs corresponding to (a) 16b complex finite impulse response filter (D_1) and (b) 16b least mean square adaptive filter (D_2) [TI03a].

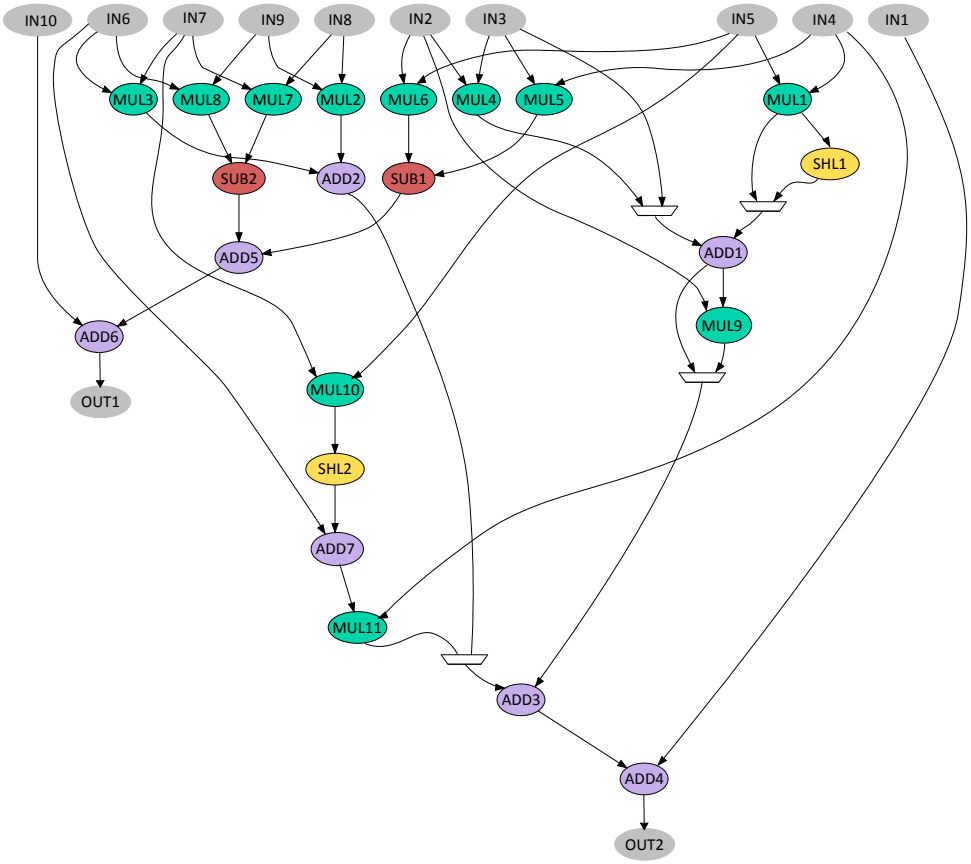
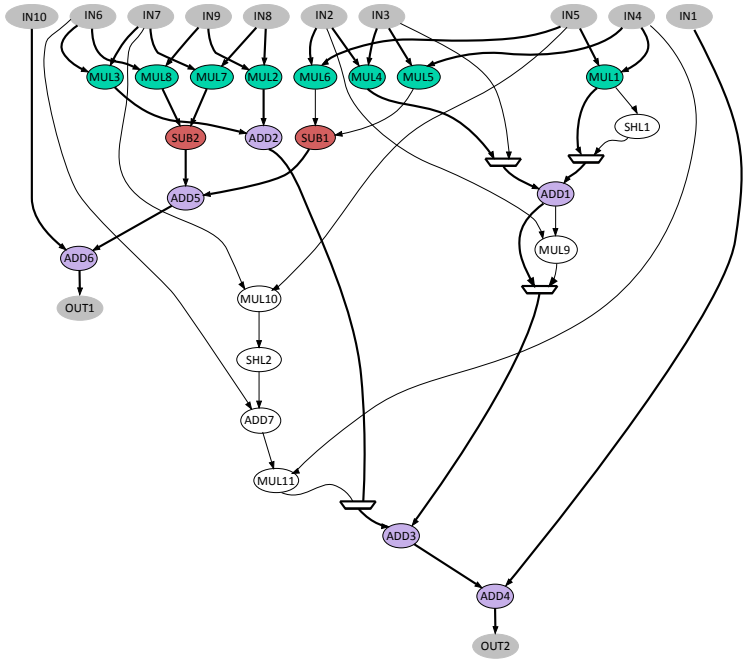
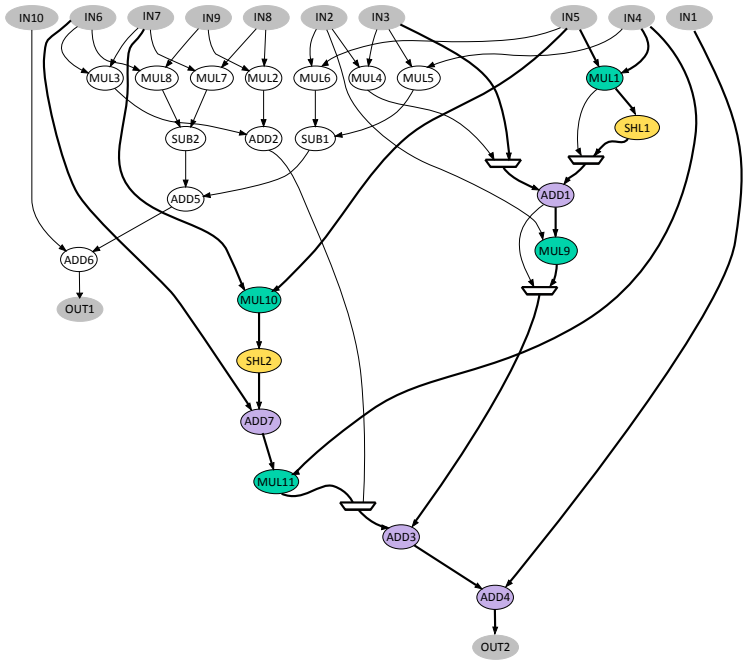


Figure 1.2: Dataflow graph $D_{1,2}$ obtained by merging D_1 and D_2 by sharing the sequence $\{MUL1, ADD1, ADD3, ADD4\}$. Four muxes for DFG reconfiguration were added.



(a)



(b)

Figure 1.3: The DFG $D_{1,2}$ with the connections and operators used by (a) D_1 and (b) D_2 shown highlighted.

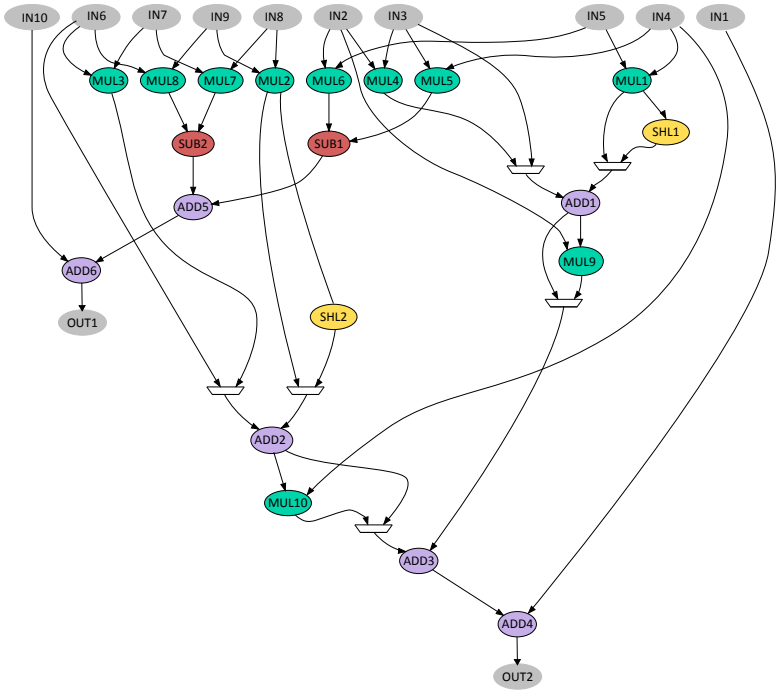
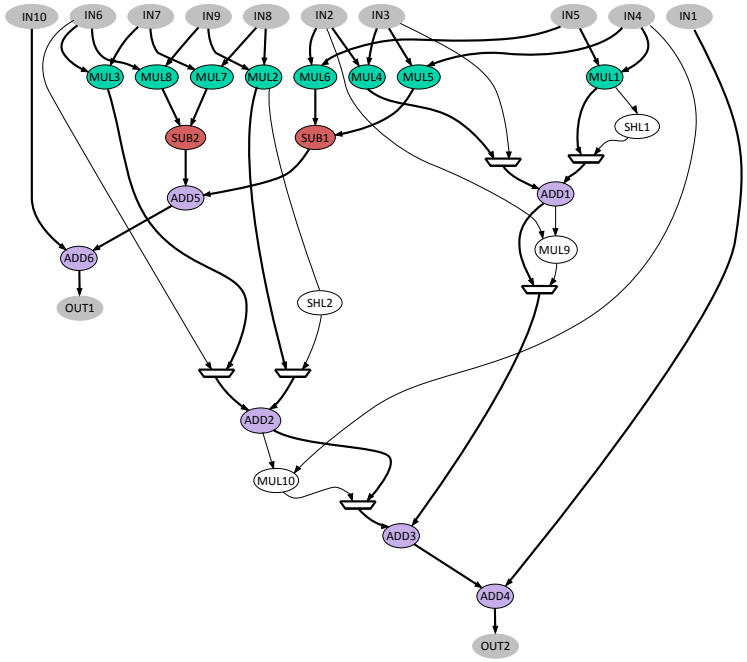


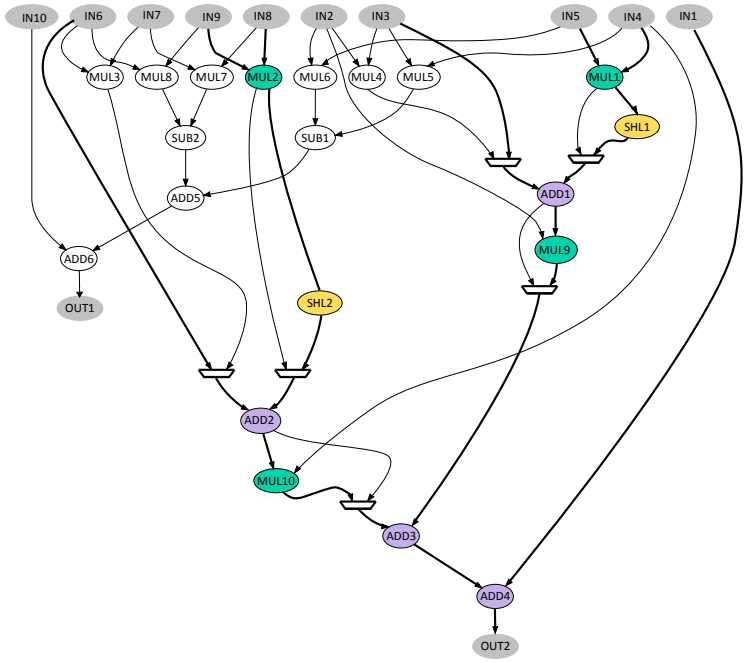
Figure 1.4: Dataflow graph $D_{1,2}$ obtained by further merging D_1 and D_2 by sharing the sequences $\{MUL2, ADD2\}$ and $\{MUL10, ADD7\}$ from Figure 1.2.

Then, the highest-area common sequence of operators shared by D_1 and D_2 is the sequence $S_1 = \{MUL1, ADD1, ADD3, ADD4\}$. The result of merging D_1 and D_2 by sharing the sequence S_1 is shown in Figure 1.2. A total of four multiplexers is inserted to enable reconfiguring the datapath to execute both DFG D_1 and DFG D_2. The connections and operators used by each DFG are shown highlighted in Figures 1.3 (a) and (b), for D_1 and D_2, respectively. In this first step, the total datapath area is reduced compared to the sum of areas of two individual ASIC circuits, but the final critical path delay is increased by the delay of two inserted multiplexers.

In the next step, D_1 and D_2 can be merged by sharing the sequences $\{MUL2, ADD2\}$ and $\{MUL10, ADD7\}$. The graph shown in Figure 1.4 is then obtained. The connections and operators used by each DFG are shown highlighted in Figures 1.5 (a) and (b) for D_1 and D_2, respectively. Two additional multiplexers are added. The final datapath area is even more reduced, while the critical path delay remains the same. Finally, there are no more opportunities for merging, because trying to share the multipliers $MUL9$ or



(a)



(b)

Figure 1.5: The DFG $D_{1,2}$ from Figure 1.4 with the connections and operators used by (a) D_1 and (b) D_2 shown highlighted.

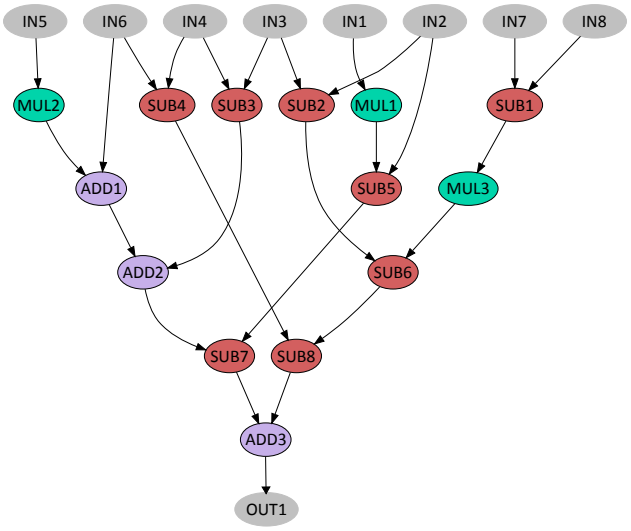


Figure 1.6: DFG D_3 corresponding to a 3×3 Sobel filter [TI03b]. Although belonging to the same application domain as D_1 and D_2, it can not be mapped onto their merged version $D_{1,2}$ shown in Figure 1.4.

$MUL10$ (Figure 1.4) with any of the $MUL3 - MUL8$ would create a cycle in the final DFG, which is not allowed [BKS04]. Hence, the final merged datapath $D_{1,2}$ is obtained.

One can assume now that in a late design stage a request to accelerate another very similar application arises. For example, it could be a request to accelerate DFG D_3 shown in Figure 1.6, corresponding to a 3×3 Sobel filter [TI03b]. It has eight input ports, two output ports, four multipliers, five adders, and two shift-right operators. This application belongs to the same domain to which D_1 and D_2 belong, and thus all three DFGs exhibit significant computational and structural similarities. Hence, it is intuitively expected that $D_{1,2}$ could be reconfigured to accelerate D_3 as well. However, this is not possible. Firstly, $D_{1,2}$ has not enough subtractors to support D_3. Even if all adders and subtractors in $D_{1,2}$ are replaced with an adder-subtractor operator, $D_{1,2}$ would not have enough of them to support D_3. Secondly, D_3 contains sequences $\{MUL, ADD, ADD, SUB, ADD\}$ and $\{SUB, MUL, SUB, SUB, ADD\}$, which are not possible to map on $D_{1,2}$. Hence, to provide some additional flexibility beyond the ability to map only the two merged DFGs, additional resources and interconnections need to be inserted into the final datapath. How to chose the types of operators and their number, and how to assemble

them into one datapath that is flexible within an application domain is the topic of this thesis. In the subsequent chapters a novel technique for generating domain-specific reconfigurable arrays will be introduced and thoroughly explained, as well as compared with the state-of-the art datapath merging methodology. In brief, this novel technique analyzes different applications input by the designer and attempts to distill the essential computational structures and connectivity in a dedicated reconfigurable array to make it possible to map new applications. Of course, the generality of the resulting datapath depends very much on how well the original applications cover the spectrum of computational structures of the target application domain.

1.2 Structure

This thesis is organized in the following way:

- **Chapter 2** provides an insight into the existing research work and comparison with the technique proposed in this thesis. At the same time, it introduces and defines the terminology to be used throughout the remaining chapters.
- **Chapter 3** presents an overview of the methodology for designing domain-specific reconfigurable arrays and briefly discusses each step in the design flow.
- **Chapter 4** focuses on the first step in designing a domain-specific array—designing the array column. This step is crucial for assuring that the array will have all operators needed to successfully map DFGs specified at the design time. Additionally, it is highly important for achieving a small area overhead, compared to existing area-efficient ASIC solutions.
- **Chapter 5** explains the column replication procedure and how the array is built.
- **Chapter 6** introduces the approach for designing highly flexible and yet efficient routing network for the domain-specific arrays under consideration. Additionally, it explains mapping and place&route procedures used by the design tool.

- **Chapter 7** provides detailed experimental evaluation on a set of applications from signal-processing domain, including the flexibility and area vs. delay results.
- Finally, **Chapter 8** presents concluding remarks.

2 Background and Related Work

This chapter provides an insight into the existing research work and comparison with the technique proposed in this thesis. At the same time, it introduces and defines the terminology to be used throughout the remaining chapters.

Section 2.1 introduces the methods for resource sharing among application dataflow graphs, used to reduce the total area needed to accelerate all target applications. The technique that will be described in Chapter 4 is motivated in part by the work of Brisk et al. [BKS04], mentioned in this section. Identifying regular patterns in application DFGs and using them for design optimizations has been a topic of extensive research work. Section 2.2 presents some of the main contributions. Another approach to improve system efficiency for a specific set of applications is to customize it. Section 2.3 presents some results in this area. A special attention is given to designing custom CGRAs, and this is discussed in Section 2.4.

2.1 Resource Sharing in Datapaths

Reconfigurable computing research has shown that a substantial performance speedup can be achieved if performance-critical subgraphs of the application dataflow graphs, most often loop kernels, are executed by especially designed hardware datapaths

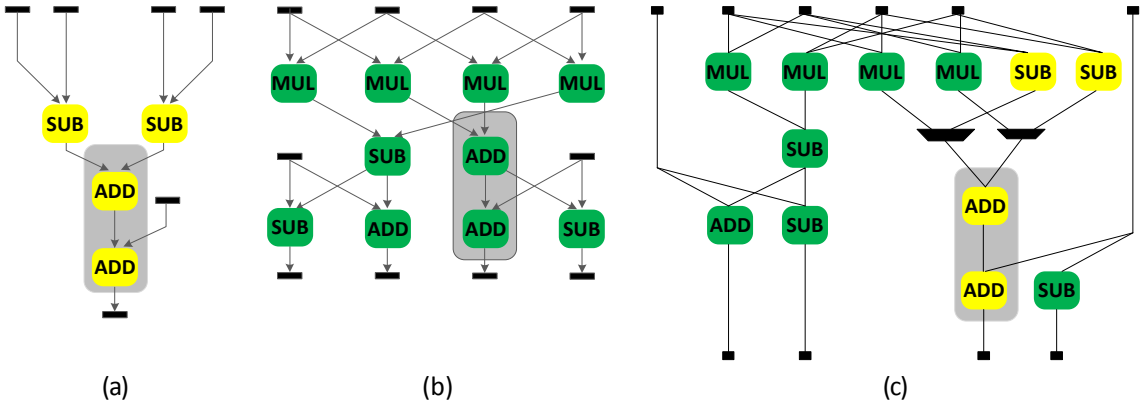


Figure 2.1: (a) Sample DFG from two-pixel sum of absolute differences. (b) Sample DFG from radix-2 FFT. (c) A typical result after merging (a) and (b). The highest area saving is achieved by sharing the sequence of two adders, marked in gray.

[CHW00]. These individual datapaths, or so called Functional Units (FUs), can be implemented as Application-Specific Integrated Circuits (ASICs), Field Programmable Gate Arrays (FPGAs), Coarse Grain Reconfigurable Arrays (CGRAs), or as hybrid solutions. A way to minimize the total area needed to implement multiple FUs is to allow hardware resource sharing among them.

Huang and Malik [HM01] studied the resource sharing which could lead to reduced run-time reconfiguration overhead. Their architectural template consists of coarse grain blocks and programmable fine-grain interconnection network shared between blocks. This is similar to other reconfigurable computing projects, such as Pleiades [Wan00]. The goal of Huang and Malik was to design a single datapath such that all FUs can be mapped to it using the minimum total number of interconnects, and thus leading to reduced reconfiguration overhead. To maximize interconnection sharing between blocks they would solve a maximum bipartite matching problem at each step. Moreano et. al. [MAHM02] extended Huang and Malik's work with a technique that relies on solving the NP-Complete Maximum Clique Problem. Consequently, the quality of their results depends on the quality of the clique-finding heuristic.

The algorithm for array column generation presented in Chapter 4 of this thesis is motivated in part by the following, very important, work in datapath merging—the

merging algorithm introduced by Brisk et al. [BKS04]. Their algorithm starts from a set of Directed Acyclic Graphs (DAGs), and not from general graphs, with the goal to maximize the area reduction achieved by merging.

> *A directed acyclic graph is a dataflow graph $G = (V, E)$, where vertices V represent operations and input/output ports, directed edges E represent data dependencies between operations, and there are no cycles.*

Brisk et al. proposed a polynomial-time heuristic that combines a set of DAGs $G = \{G_1, G_2, ..., G_n\}$ into a super-graph, called a Consolidation Graph (CG). Ideally, the CG should minimize the total datapath area, but this would require solving an NP-Complete problem [BGV03]. Therefore, they proposed a heuristic based on finding longest common subsequences and substrings of two (or more) paths in DFGs.

> *A path in a DFG is a sequence of its vertices connected by edges, such that the first node in the sequence is an input port while the last node is an output port. A subsequence is a part of another sequence obtained by removing some of its nodes while keeping the order of the remaining nodes.*

A *substring* is defined to be a contiguous subsequence. A path, a subsequence, and a substring extracted from an example DFG are shown in Figure 2.2. Brisk algorithm starts by enumerating all of the paths of each DFG and continues by looking for a pair of paths that would maximize the area reduction if selected for merging. Clearly, those are the paths that share the Maximum Area Common Subsequence (MACSeq). It is only up to this point that the algorithm for array column generation proposed in Chapter 4 is similar to Brisk's algorithm. Further on, the heuristic by Brisk et al. iterates through global and local phases and merges graphs by the best candidate paths until there are no more candidate paths for merging.

Zuluaga and Topham [ZT09] continued the work by Brisk et al. They noticed that extensive resource sharing can produce a considerable increase in the application

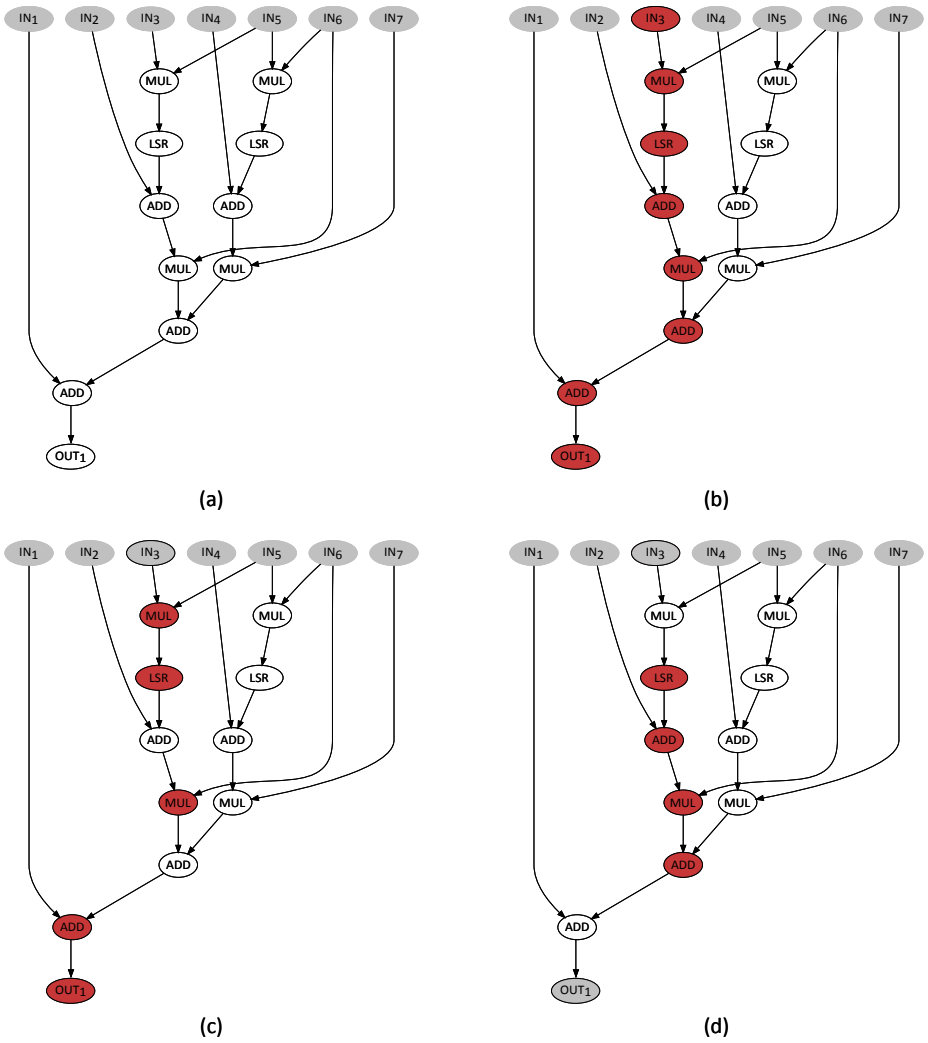


Figure 2.2: (a) An extracted dataflow graph of the 16b least mean square adaptive filtering application [TI03a]. (b) A path, (c) a subsequence, and (c) a substring of the path.

latency. Hence, they presented a heuristic to control the degree of resource sharing among given DAGs and thus achieve latency constraints.

Although datapath merging introduces a form of flexibility through addition of multiplexers and interconnects among operators, it still does not provide sufficient flexibility beyond the ability to map the dataflow graphs known at the design time. Yet, the aim of the work in this thesis is achieving significant increase in flexibility of the final datapath at a moderate cost in area and latency.

2.2 Design Optimizations by Regularity Extraction

Dataflow graphs of real applications exhibit high level of regularity, which can be exploited to improve the area and performance of the hardware layouts. This regularity implies that there exist subgraphs having multiple instances. They are referred to as *patterns* or *templates*.

Pattern recognition has been exploited in every level of the large circuit design from layout designs to high-level synthesis [Keu87, RK93, BKKS02, CKG+96, BR97, LKMM95, KS00]. Rao and Kurdahi [RK93] developed and formalized the problem of regularity extraction using a graph model, and proposed a string-matching based approach to cluster similar structures and replace the instances of a pattern with a common implementation. Their goal was to minimize the usage of distinct multiple clusters as well as the total number of clusters used. They used linear representation of directed graphs proposed by Berztiss [Ber75]. However, this linearizion process is the major drawback because selection order of nodes can dramatically affect the pattern matching result. Corazao et al. [CKG+96] tried to improve the quality of logical synthesis by considering patterns at the behavior synthesis step, where they assumed that the pattern library was given by users. Other interesting works include scheduling and binding algorithms with patterns by Bringman [BR97] and Ly [LKMM95]. They too assumed that patterns were given in advance. Another research topic that focuses on extracting regularly occurring patterns, which is orthogonal to the work presented in this thesis, is the area of creating custom hardware units to extend the computational capabilities of a processor—custom Instruction Set Extensions (ISEs) [BKKS02, CFHZ04, YM04, API03, BP07].

Chowdary et al. [CKS+99] presented an approach to extract functional regularity from datapaths described by a hardware description language (HDL). The task of regularity extraction was, again, to identify a set of templates and then to cover the given circuit by a subset of these templates. Their objective was to use large templates having large number of instances, but this involves a tradeoff; On one side, a large template usually

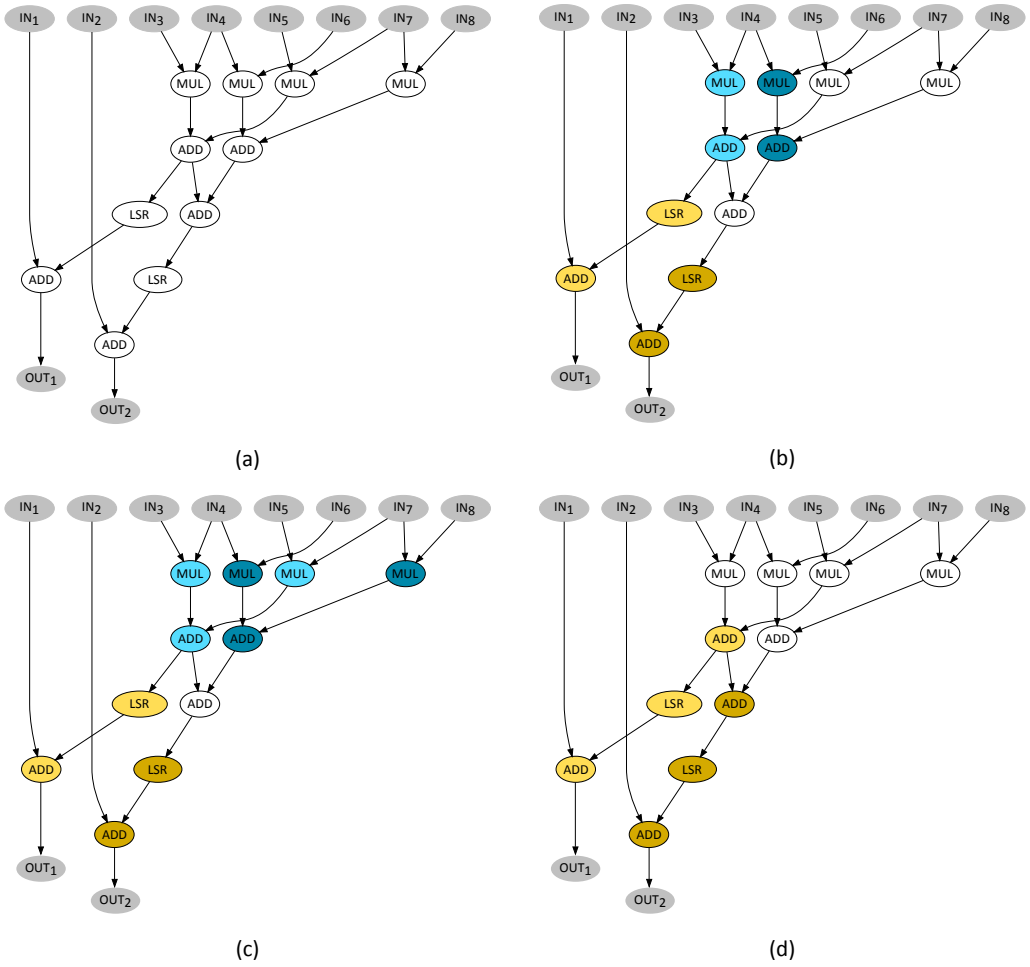


Figure 2.3: (a) An extracted dataflow graph of the IIR filtering application [TI03a]. (b) Two different patterns selected: MUL–ADD and LSR–ADD. (c) MUL–ADD pattern from (b) enlarged to contain one more multiplier. (d) LSR–ADD pattern from (a) enlarged to ADD–LSR–ADD, but without improving the coverage of the graph.

has lower number of instances than a smaller template; On the other side, a larger template implies a better area and performance optimization, while a smaller template with more instances implies less effort in synthesis and layout design phases. Examples of various extracted templates from an IIR filtering application are shown in Figure 2.3.

Cong et al. [CJ08] tried to optimize the resource usage of FPGA designs using pattern-based synthesis techniques. They presented a pattern-based behavior synthesis framework for efficient extraction of similar structures in dataflow graphs, which uses a mismatch-tolerant metric *graph edit distance*. The edit distance between two graphs

can be defined as the minimum number of vertex/edge insertion, deletion, and substitution operations to transform one graph into the other. This metric can also handle various program variations such as bit-width, structure, and port variations. Cong et al. applied this pattern-recognition framework to solve FPGA resource reduction problem. They used the fact that if all pattern instances are scheduled and bounded in a uniform way, the internal dataflows are free of multiplexors, except for those inserted due to resource sharing among nodes inside a single pattern instance. That way, the total number of multiplexors and data routing logic would decrease, as well as the total design area, delay, and power consumption. Later, Cong et al. presented an approach extended to include control-flow aware patterns and introduced a *generalized edit distance* metric for measuring variations in control-flow and dataflow graphs [CHJ10].

These works are all similar to the methodology described in thesis in that they try to reuse regularly occurring patterns to optimize the design performance. But, like other prior datapath merging techniques discussed in 2.1, these did not introduce any further generality in the final datapath.

2.3 Increasing Flexibility through DFG Generalizations

Yehia et al. [YGBT09] focused on customization in multi-core systems, as an orthogonal and complementary scalability path to parallelization. Their idea was to parallelize the application first, and then further improve its performance by customizing either the parallel sections or the remaining sequential sections. While customization can bring cost and power efficiency, it can take away some flexibility. Therefore, they proposed an approach to preserve some flexibility by automatically combining customized circuits into a larger compound unit, and thus increasing the number of applications that can benefit from a single circuit. Before creating a compound circuit of two individual circuits, they first check if one of the circuits can be mapped to another, including the dataflow and control-flow parts. If direct mapping is not possible, the framework

proposed in their paper [YGBT09] *alters* one of the two circuits by adding operators, state nodes, configuration multiplexors and interconnects, as long as mapping remains unfeasible. This process is repeated until all circuits can be mapped on one compound circuit. They observed that the compound circuit cost does not increase in proportion to the number of target applications, due to the wide range of common dataflow and control-flow patterns in programs. The methodology introduced in this thesis differs in that it does not check if a graph can be mapped to another one to build a compound solution, but it generates the output array based on analyzing all input graphs at once. Additionally, it outputs not only a compound but also a regular circuit, inherently increasing its flexibility.

Clark et al. [CZM03, CZM05] presented the design of a system to automatically identify and customize instruction set extensions. Their system uses a dataflow graph design space exploration engine to efficiently identify suitable computation subgraphs from which to create customized hardware. Additionally, it contains a subgraph matching framework to identify opportunities to exploit and *generalize* the hardware to support more than one application. To enable this more effective usage of the hardware units, they introduced three generalization techniques. The first is *subsumed subgraphs*, which uses the fact that many operators have an identity input, allowing values to pass through them unmodified. For example, if one functional unit has a sequence AND–XOR–ADD, it can execute sequences AND–ADD, AND–XOR, and XOR–ADD too. It suffices to add a multiplexor to the input of every operation to be bypassed and to connect the output of the previous operation and the identity value to the multiplexor inputs. The second generalization technique they call *wildcards*. Two subgraphs are wildcards if they are identical, except for one different operator. Combining them into one functional unit is cheap and increases flexibility, although in a limited way. Finally, the third technique is an extended version of wildcarding—*preemptive wildcarding*. This technique allows to generalize graphs by implementing multiple operations in nodes, e.g., by replacing an ADD or SUB operation with an ADD/SUB unit. Clark et al.

performed these generalization techniques to create a set of different candidates for hardware implementation. Their selection algorithm would then explore this set and decide which of the generalized subgraphs to synthesize in hardware. The similarity between their approach and the approach to achieve hardware flexibility presented in this thesis is in the use of preemptive wildcarding. However, the latter introduces a flexible routing network as a much more general alternative to the subsumed subgraphs, thus achieving significantly better flexibility results.

2.4 Domain-Specific Arrays

Ebeling et al. [ECF+97] introduced RaPiD (Reconfigurable Pipelined Datapath), a coarse-grained configurable architecture for executing regular computationally-intensive applications. RaPiD is a 1-D array of computation cells, that comprise of an integer multiplier, three integer ALUs, six general purpose registers, and three small local memories. A typical RaPiD chip would contain between 8 and 32 of these cells. A block diagram of a single cell in RaPiD architecture is shown in Figure 2.4. RaPiD limits applications to at most two reads and one write per cycle, which is significantly less than the number of memory accesses supported by the 2-D array described in this thesis. Routing in RaPiD is in the form of word-size segmented buses running parallel to the axis. Therefore, it is similar to the routing network between any pair of neighboring rows in the domain-specific array in this thesis. But, the latter offers significantly more routing opportunities due to the introduction of vertical routing channels. The RaPiD architecture was manually devised and tuned for a wide variety of circuits within the DSP domain, as well as the other relevant architectures such as PipeRench [GSM+99], Pleiades [AR96], or MorphoSys [LSL+00]. Instead, this thesis presentes a method for automated generation of configurable arrays suited to any application domain input by the designer.

Phillips et al. [PSH04] introduced a template reduction methodology to optimize reconfigurable fabric to the demands of an application domain, as part of the Totem

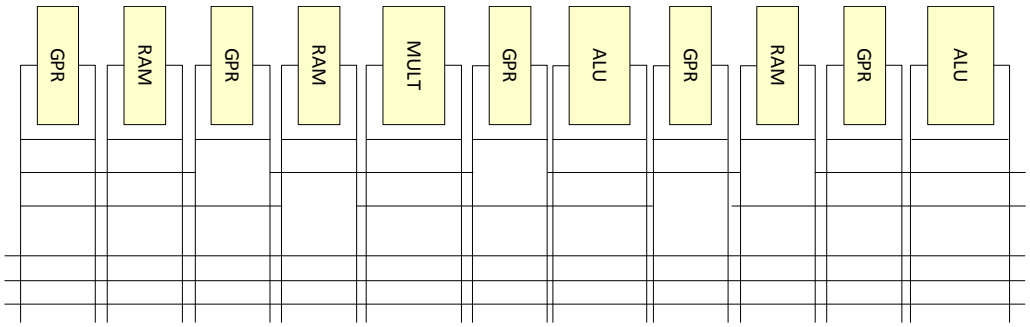


Figure 2.4: A block diagram of a single cell in RaPiD architecture. To form the full architecture, 16 cells are tied along the horizontal axis.

project [CH01]. Initially, they perform profiling to obtain a rich macro cell (template) providing a superset of all required resources. Then, in the template reduction phase, they iteratively remove the unneded routing resources and functional units. Compton and Hauck [CH08] introduced the Totem tool to generate 1-D architectures similar in style to RaPiD. To achieve a more customized design, Totem varies the number and order of word-size computation units in a RaPiD-like array, and the length and the number of tracks in the routing channel. To select architectural components, Totem takes the minimum number of each type of computation unit needed to implement all of the given circuits (one at a time). However, the tool presented in this thesis automatically infers an overhead in the number of components to accommodate larger datapaths not known at design time. Additionally, Totem constrains computation-unit types to be evenly distributed through the 1-D array, whereas the method in this thesis uses the path fusion procedure to perform the distribution of the units. Due to its 1-D nature, Totem array needs a high number of word-size tracks in the routing channel (Compton et al. reported up to 34). On the other side, a 2-D array uses less tracks per channel and provides higher routing opportunities due to the regularity of its routing network.

Ansaloni et al. [ABP08, ABP11] proposed an architectural template for design space exploration of different CGRA designs. They named this template Expression-Grained Reconfigurable Array (EGRA), due to its ability to generate complex computational cells executing expressions, rather than single operations. The basic cell of their template

is inspired by the Configurable Computation Accelerator (CCA) proposed by Clark et al. [CKP+04] and it is called a Reconfigurable ALU Cluster (RAC). The EGRA structure is organized as a mesh of RACs, memories, and multipliers. The number and placement of these elements is part of the architecture parameter space—it is decided at design time and can vary for different instances of the EGRA. Cells are connected using both nearest-neighbor connections and horizontal/vertical buses, with one such bus per column and row of the array. A RAC consists of multiple ALUs, with possibly heterogeneous arithmetic and logic capabilities. Inside the RAC, ALUs are organized into rows, and the inputs of the ALUs in subsequent rows are routed from the outputs of the previous rows, or from constant values. Four types of ALUs can be instantiated; the first one able to perform bitwise logic operations only and the other three that add a barrel shifter (with support for shifts and rotates), an adder/subtractor, and both the shifter and adder, respectively. The number of rows, the number of ALUs in each row, and the functionality of the ALUs is flexible and can be customized by the designer during the exploration phase. Therefore, the EGRA architecture can be adapted to the application domain. The approach described in this thesis is less general in that it assumes array cells are dedicated operators, rather than more general ALUs. However, this increases efficiency as such solution is closer to an ASIC implementation.

3 Design Framework Overview

The flexibility of domain-specific reconfigurable datapaths need not be absolute; to that purpose, there exist already FPGAs, which provide the highest degree of flexibility and are thus used in systems requiring diverse computations. This absolute flexibility is not always desired, because FPGAs suffer from significantly higher area overhead and critical path delay increase compared to dedicated, and thus less flexible, solutions. For a limited set of applications belonging to a single, or to multiple similar domains, it would perhaps be useful to investigate the similarities among those applications so as to design an architecture having better area/delay trade-off than FPGAs. The work presented in this thesis focuses on creating a single reconfigurable datapath (array) based on a set of input applications belonging to a same domain. This datapath is *selectively flexible*, and thus the novel methodology can be referred to as a methodology for designing selectively-flexible reconfigurable arrays.

A datapath is considered flexible if it can support the execution of a number of different applications, whereas it is selectively flexible if its flexibility is limited to a specific application domain. The computational characteristics of a domain are characterized, and thus limited, to (1) the type of operations, (2) their number, and (3) the interconnections among them. To achieve the domain-specific flexibility of the hardware datapath, the novel design technique analyzes different applications input by the designer, attempts

to distill the essential computational structures and connectivity in a dedicated reconfigurable array, and introduces selective flexibility into this array to make it possible to execute new applications from the same domain. This approach builds on:

1. the knowledge of resource sharing among datapaths (Chapter 2.1),
2. design optimizations by exploiting regularity in application DFGs belonging to a same domain (Chapter 2.2), and
3. generalizing the final datapath so as to increase flexibility (Chapter 2.3).

3.1 Design Flow

Figure 3.1 illustrates the fundamental steps of the SFRA design technique to capture the key features of a number of input applications:

- The initial step of the design methodology is to take the input applications, written in a programming language (for example in C), and to represent them in the form of Control Flow Graphs (CFGs). In a CFG each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets. This block is represented in the graph form as well, as a data-flow graph (DFG). Then, the largest or the most frequent of these blocks are selected as candidate DFGs for acceleration. This step is performed on all input applications, to generate a training set of DFGs to be used in all subsequent steps of the design flow.
- How well a domain-specific hardware represents an application domain depends heavily on the way the most relevant characteristics of that domain are identified and then used to guide the design steps. Datapath merging approaches look for the highest-area common-subsequences in input DFGs, and try merging by sharing them. This approach, in turn, looks into all paths of the input DFGs to extract the types of operations and the sequences in which they appear. Then,

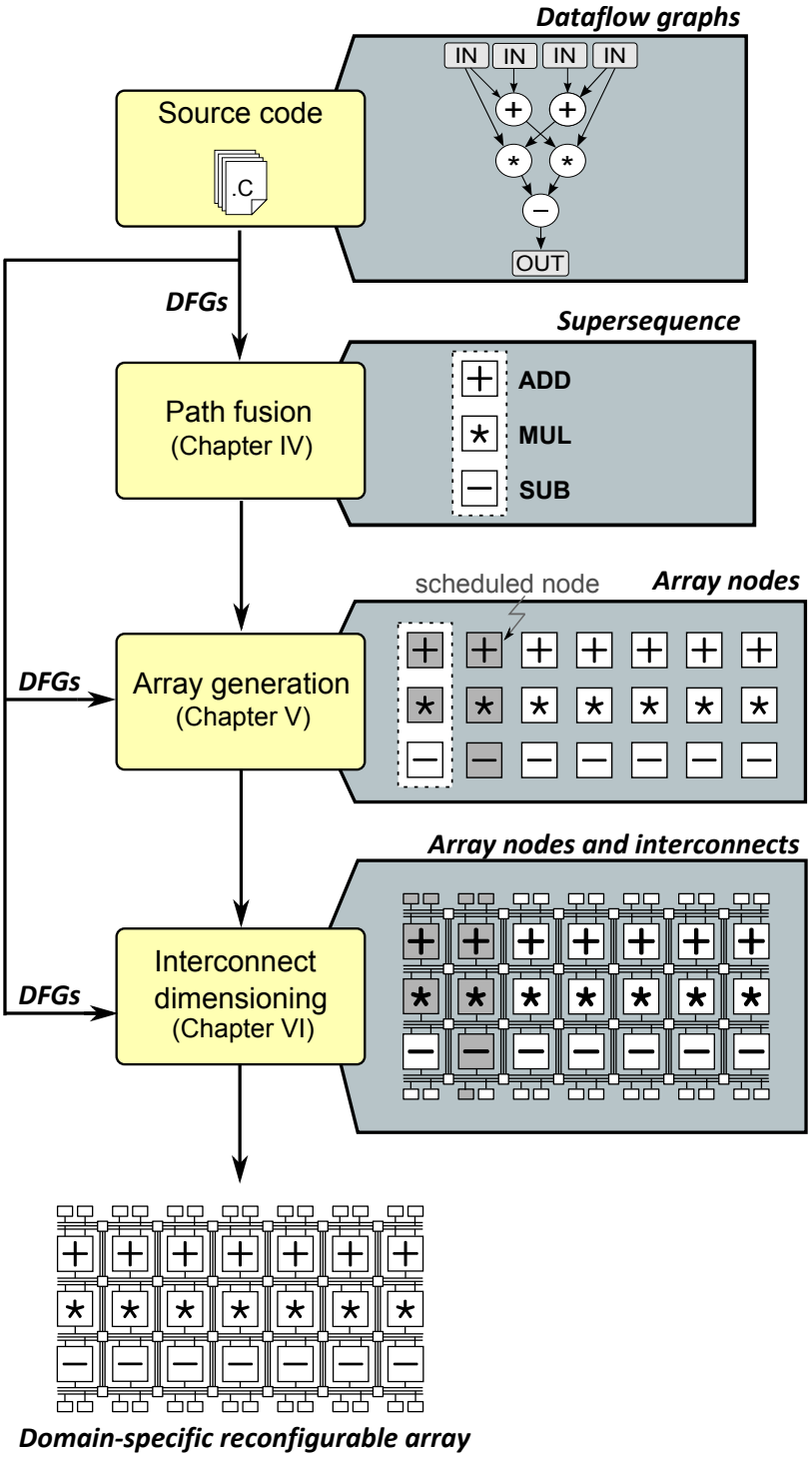


Figure 3.1: The design flow to synthesize domain-specific datapaths. Firstly, a set of candidate DFGs from input applications are generated. Then, those DFGs are analyzed to extract a column of the datapath. This column is replicated to create a regular 2-D array structure. Finally, an FPGA-like statically configured routing network is added to enable routing the DFGs.

it fuses all sequences to obtain a single *supersequence*, which defines the basic block of the array—its column. The reason behind this idea is that sequences of operators are inherent to the target domain and, once a supersequence is created, it is very likely that all sequences found in new DFGs belonging to the same domain will already be included. By definition, a supersequence is an ordered sequence of operators, which includes all the sequences of operations present in the input applications. Additionally, any two operators form a sequence if there exist a sequence of vertices connected by edges between these two operator nodes.

- To generate a 2-D array, the supersequence is replicated. Hence, the array structure is regular because every row contains only one type of operator. Assuming that the array is composed by $N_r \times N_c$ operators in total, and knowing that each column is composed by the operators in the supersequence, the number of rows N_r must equal the length of the supersequence. The number of columns, N_c, should be as large as to guarantee successful mapping of the input set of DFGs. In general, it has no upper limit—it is only the available die area that limits it. However, it should be as small as possible to achieve a proper balance between the datapath flexibility and the area overhead. Without loss of generality, it is assumed that there are two memory read (input) and memory write (output) ports available per column of the array. This assumption is due to the fact that all nodes in the array are either one-input or two-input operators.

- In typical CGRA architectures, only the nearest four or eight nodes can pass data between them. Clearly, this is very restrictive, and not a suitable solution for an architecture that needs to provide high routing opportunities. FPGAs, on the other hand, have more complex routing networks that are crucial for their successful usage for a wide range of applications. These networks consist of horizontal and vertical routing channels, which are interconnected using switch blocks, and to which every functional block, or a set of blocks, is connected in some

way. The methodology described in this thesis adopts this concept and adjusts it to the usage in CGRAs. The architecture of the routing network is essentially the same; there are vertical buses between subsequent rows and columns of the reconfigurable array, to which the array operator input and output ports are connected. However, these buses are 32-bit wide and cannot be used for bit-based routing [YR06]. This helps reducing the amount of configuration storage, because instead of using 32b to reconfigure a switch used to connect two buses, it suffices to use a single bit only. Additionally, all array nodes are coarse-grain and thus operate on words, not on individual bits.

The reconfiguration of the datapath is performed by shifting in configuration bits and storing them in configuration memory cells. It is essentially achieved as in any FPGA and in many coarser grain statically programmed arrays, and is not addressed in detail in this thesis. Applications are statically mapped on the datapath, much as in an FPGA: reconfiguration happens only before execution of one of the applications.

Such a datapath array should feature computational structures that enable a high degree of generality for a particular domain at a reasonably small overhead in the number of unused operators and redundant interconnects. In the following chapters, all mentioned steps of the technique to generate domain-specific reconfigurable arrays from a collection of DFGs are described in details.

3.2 Dataflow Graph Represenation

Dataflow graphs extracted from the corresponding application code are represented in the same textual format that is used by the software tool Clarity [BE06], originally designed for automatic identification of instruction set extensions (ISEs) and their hardware implementation. In order to process the gcc (GNU compiler collection) intermediate representation and return DFGs in Clarity intermediate representation, an additional piece of code (script) had to be developed.

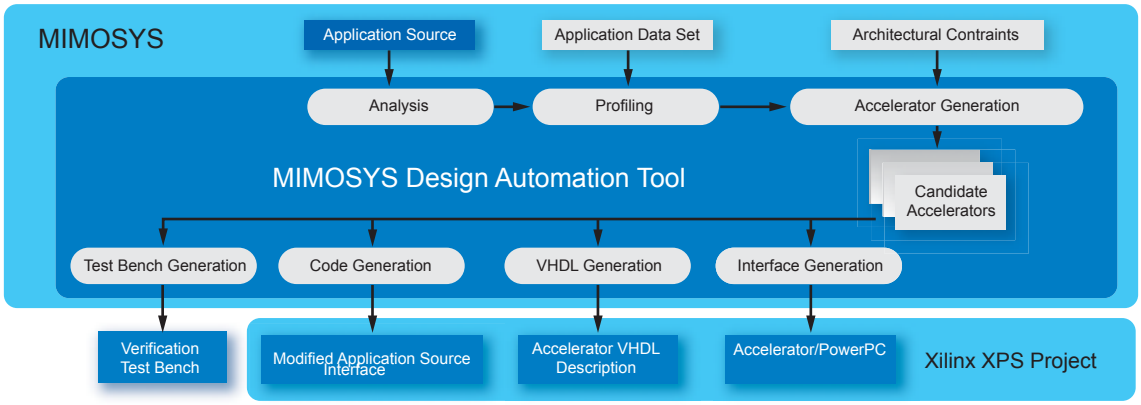


Figure 3.2: Mimosys Clarity flow; from the application source code and some constraints, one can generate a complete Xilinx Platform Studio project with accelerators [BE06].

3.2.1 The Mimosys Clarity tool

Clarity is a tool for automatic identification of ISEs for application acceleration, which was developed by the company Mimosys, in cooperation with EPFL. It is used by researchers at EPFL, and it is not publicly available. Adopting the same format of DFG representation does not constrain the features of the methodology presented in this thesis, but ensures easier integration into the Clarity tool, for potential future work.

In Clarity, the identification of ISEs is performed directly from an application C source code, guided by the execution profile of the application, the number of I/O parameters available for the accelerator and the model of the execution costs for operations. The tool provides visualization of the execution profiling information in the form of control-flow graphs and corresponding data-flow graphs for each node in the CFGs. It allows ISEs to contain memory either as constant arrays only or as constant and read/write arrays. Additionally, it provides an option to select a subset of identified ISEs for hardware implementation on various hardware platforms. Clarity automatically creates test benches for each accelerator along with the necessary simulation and synthesis scripts. Within Clarity, ModelSim simulator can be invoked to verify the functionality of the accelerator. The generated synthesis scripts enable the synthesis of the accelerator on a target technology and the evaluation of the critical path and area. Among the other

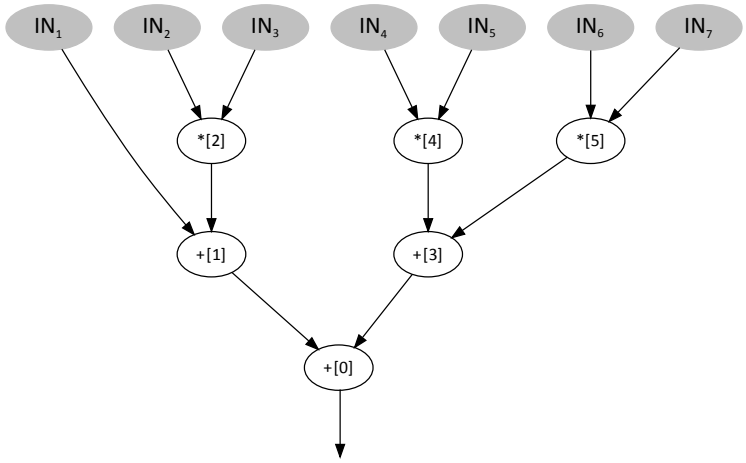


Figure 3.3: A DFG corresponding to 3 × 3 convolution. The text file used as the example in Table 3.1 depicts the connection and structure of this DFG.

features, the tool generates an intermediate textual representation of the execution profiling information, which can be used by different ISE identification algorithms. The results of ISE search can afterwards be easily passed back to Clarity. The Mimosys Clarity flow is shown graphically in Figure 3.2.

3.2.2 Dataflow Graph File Format

The format of the file describing a DFG shown in Figure 3.3 is discussed in details in the following table.

Table 3.1: Dataflow Graph File Format

Keyword	Explanation
`N M I T`	`N` is the sum of the number of DFG nodes and the number of output ports. `M` is the number of times the DFG is executed in one application run. `I` is the number of DFG input ports. `T` is the number of DFG output ports. Example: `6 1 7 1` corresponds to a graph with seven input ports, one output port, and five nodes (`N = M-T`).

`ext_in`	This line is followed by a $N \times I$ array having information on the connections between input ports and nodes: $(i, j) > 0 \Rightarrow$ node i is connected to the input port j, $i = 0..N-1$, $j = 1..I$, e.g.:
	`ext_in`
	`0 0 0 0 0 0 0`
	`1 0 0 0 0 0 0`
	`0 1 2 0 0 0 0`
	`0 0 0 0 0 0 0`
	`0 0 0 1 2 0 0`
	`0 0 0 0 0 1 2`
	The first line explains that the node 0 has no connections with the input ports. The second line shows that the node number $i = 1$ is connected to the input port IN_1 ($j = 1$). Based on the third line, the node number $i = 2$ is connected to two input ports IN_2 ($j = 2$) and IN_3 ($j = 3$), and so on. The values within rows always increase by a factor of 2×, in the direction of increasing column number j.
	Continued on next page

Keyword	**Explanation** *(Continued from previous page)*

`ext_out`	This line is followed by `T` (`T` = 1 lines in the form x y, where x denotes the node number, and $(y-1)$ is the output outpu of the node that is a DFG output port at the same time. For nodes having only one output, y equals zero, e.g.:
	`ext_out`
	`0 0`
	This means that the output of the node 0 is the DFG output port at the same time.

`hw_sync_lat`
`hw_lat`
`sw_lat`
	This line is followed by `N` lines representing the latencies of nodes. The value -9999.0 stands for a forbidden node type. Otherwise, latencies are zeros.

`dest`	This line is followed by `N` lines, each one giving the total number of outputs of the corresponding node that are driving other nodes, e.g.:
	`dest`
	`0`
	`1`
	`1`
	`1`
	`1`
	`1`
	This means that the node 0 is not driving other nodes, while all other nodes have one output that is connected to one, or more, nodes in the DFG.

	Continued on next page

Keyword	**Explanation** *(Continued from previous page)*

`adj_list`	This line is followed by `N` lines, each one giving the out-degree of the corresponding node. The out-degree might be different from the total number of outputs, since some outputs might be left floating. The sum of all values in these lines equals `E`—the total number of internal edges in the graph. The subsequent `E` lines in the form $x\,y$ give information about edges. The source of each edge is known—it is the node having a non-zero value in the previous `N` lines. Thus, the only missing information is the destination, which is given in the value x. The value y is the index of the input of the destination node, as a node may have several inputs. For two input nodes, the left input is numbered as 1, while the right input is numbered as 2. E.g.:
	`adj_list`
	`0`
	`1`
	`1`
	`1`
	`1`
	`0 1`
	`1 2`
	`0 2`
	`3 1`
	`3 2`
	Continued on next page

Keyword	**Explanation** *(Continued from previous page)*
	This means that all nodes but the node zero are sources of one edge. The first node is connected to the left input of the node zero. The second node is connected to the right input of the node number one. The third node is connected to the right input of the node zero, and so on.
`memory_deps`	Unused.
`opcodes`	The next `N` lines give the internal operation codes for each node in the DFG. If opcode equals 1, the corresponding node is a forbidden node. The encoding of the operation codes is as follows: bits 58–63 = output bitwidth–1 (range 1–64); bits 55–57 = (number of memory ports–1), if bits 39–50 are not zero (range 1–8); bits 51–54 = type: 0/1 = signed/unsigned integer, 2/3 = float/double;bits 39–50 = memory table if allowed, if not zero; bits 32–38 = opcode; bits 0–31 = literals, etc.
`bitwidths`	The next `N` lines contain bitwidths of input operands for every node. If the node is forbidden, bitwidth is set to -1. Otherwise it is calculated by (i) concatenating the following set of six bits, in this very order: bitwidth(op 0) – 1, bitwidth(op 1) – 1, bitwidth(op 2) – 1, etc., and then (ii) converting this binary number into a decimal value. It is this decimal value that is stored as `<bitwidths>` parameter.
`end`	The end of file.

4 Array Column Generation

The key idea behind the methodology introduced in this thesis is in the way the array column is generated. Unlike in prior research, here the array column is observed as the key element that must capture the computational characteristics that are representative of an application domain. These characteristics are (i) the types of operators needed to execute applications belonging to that domain and (ii) the sequencing of operators corresponding to the flow of data in application DFGs. In this work it is assumed that mapping DFGs onto domain-specific CGRAs is performed following a top-down approach, and thus preserving the natural data-flow between operators (discussed in more details in Chapter 6).

The process of defining the number of operators and the way they are sequenced within the array column will be refered here as *path fusion*. The name of the process suggests that all DFG paths need to be enumerated, analyzed, and somehow combined. Figure 4.1 shows two sample dataflow graphs from (a) a two pixels sum of absolute differences (SAD) and (b) radix-2 FFT butterfly. As defined in Section 2.1, a path in a graph is a sequence of vertices connected by edges, such that the first node in the sequence is an input port (memory read) while the last node is an output port (memory write). Hence, Figure 4.1c shows all distinct paths of the two DFGs. The outcome of the path fusion process is a single sequence, a *supersequence (SSeq)* of nodes, such that all enumerated

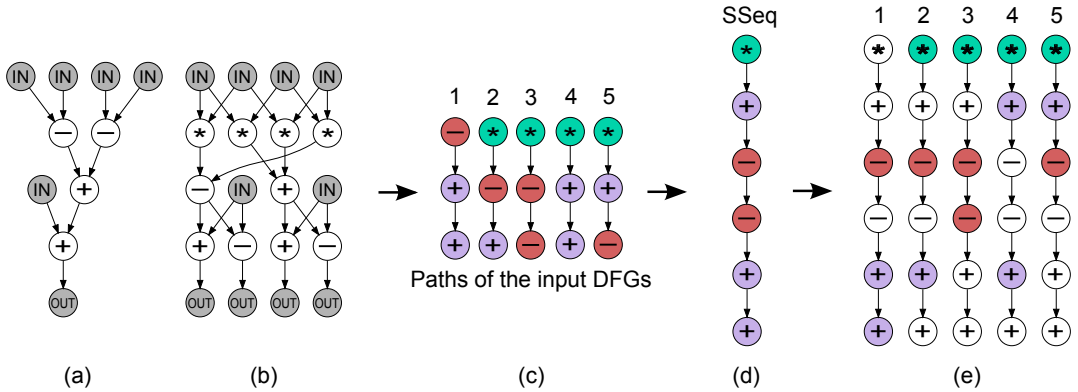


Figure 4.1: Two sample dataflow graphs from (a) a two pixels sum of absolute differences and (b) radix-2 FFT butterfly. (c) All paths identified in these two DFGs. Memory read and write nodes are not included. (d) An example supersequence. There are many possible solutions. (e) Every DFG path is a subsequence of the supersequence, i.e. it can be obtained by removing some elements while preserving the order of the remaining elements.

paths are subsequences of it. Clearly, there exist many such sequences. Figure 4.1d shows only one possible solution, and (e) illustrates how all paths are subsequences of the sequence in (d), and can thus be derived from it by deleting some elements, without changing the order of the remaining elements. An array created by replicating the SSeq sufficient number of times certainly provides all necessary operators to enable executing the DFGs in (a) and (b), because every path is contained within the array column. This is a very important feature of SSeq.

By creating a supersequence of all paths found in input DFGs and using it as the array column, not only that all needed resources are provided, but also additional resources are introduced, which leads to increased flexibility of the array. This flexibility is specific for the application domain to which input DFGs belong and, clearly, it comes at a cost of increased die-area. To achieve a reasonable area overhead, the supersequence area should be minimized. The problem of finding a minimum area SSeq is similar to one of the classical problems in computer science and genetics—finding the Shortest Common Supersequence (SCS). That is an NP complete problem, for which various approximation algorithms exist. One of the most known is the algorithm by Branke et al. [BMS98].

In Section 4.1, a modified algorithm by Branke et al. is explained and implemented to generate one possible SSeq. Then, a novel heuristic for finding the minimum area supersequence is developed and presented in Section 4.2. This heuristic is based on reusing the Maximum Area Common Subsequence (MACSeq) metric of existing graph-merging algorithms [BKS04]. Finally, the algorithm complexity analysis is given in Section 4.3.

4.1 Creating Shortest Common Supersequences

Branke et al. [BMS98] formulated the problem of finding the shortest common supersequence (SCS) as follows:

> *Given a finite set of strings L over an alphabet* Σ*, find a string of minimal length that is a supersequence of each string in L.*

By definition, a string is a finite sequence of symbols chosen from an alphabet, or, in the context of application DFGs, a path in the dataflow graph. One of the very well known heuristics for finding a SCS is the Majority Merge (MM) [JL95], which builds the supersequence starting from an empty string in the following way: it looks at the first symbol in each string in L and chooses the most frequent one, removes it from the strings where it has been found as the first symbol, and appends it to the supersequence. The process is repeated until all strings are emptied. Figure 4.2 illustrates the algorithm steps on the example set of paths extracted from DFGs shown in Figure 4.1.

If used to find the shortest supersequence of a set of strings, majority merge achieves poor results, because it does not take into account that strings might have different lengths. To overcome that, Branke et al. [BMS98] proposed new heuristics, one of which is the Weighted Majority Merge (WMM). That algorithm selects the candidate symbol with the maximum sum of weights of its occurrences at the front of the strings. The weight of a symbol is assumed to be the length of the string suffix, excluding the

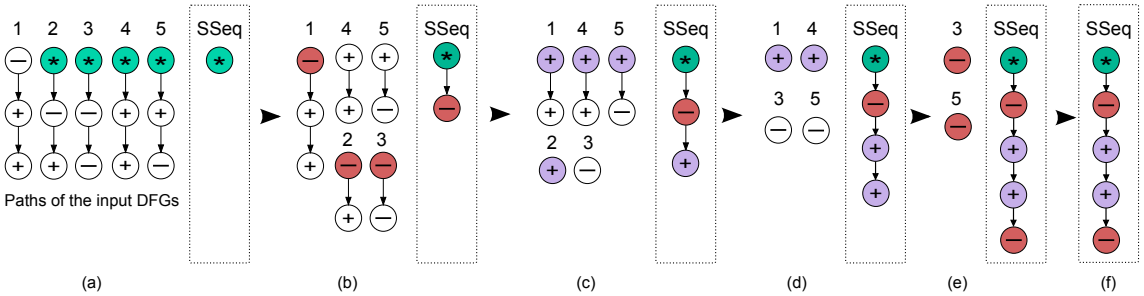


Figure 4.2: An example illustrating the steps of the algorithm by Jiang et al. [JL95] to find the shortest common supersequence (SSeq) based on Majority Merge (MM) heuristic. (a) The most frequent operator at the beginning of the paths is the multiplier. It is thus selected to start the SSeq. (b) The multiplier is removed from the beginning of the paths, and the next most frequent operator, the subtractor, is selected and appended to the SSeq. (c) The subtractor is removed from the paths. The next candidate operator is the adder, which is then appended to the SSeq. (d) Two adders and two subtractors remained. Assuming that the ties are broken so that an adder is selected next, then (e) the only remaining operator, the subtractor, is selected and the final SSeq in (f) is obtained.

symbol itself. However, this approach is not suitable for solving the problem of finding a supersequence of all paths enumerated from application DFGs. This is because operators in these paths differ not only by type but also by area they occupy. Thus, the shortest sequence is not necessarily the smallest area one. For example, the sequence containing two multipliers and one adder might occupy more area than the sequence of one multiplier and three adders, although the latter is longer.

This section introduces a novel algorithm to find a minimum area supersequence rather than the minimum length one, based on the WMM heuristic. This algorithm calculates the weight of the symbol as the sum of the area of the operator implementing that symbol and of the areas of all following symbols in the same path. If there are multiple candidates, the algorithm takes the one with the longer string suffix. The algorithm steps are as follows:

1. Initialize the supersequence to the empty sequence and enumerate all paths of each DFG.
2. Choose the candidate operator *op* by computing the sum of weights for all opera-

tors at the front of the paths and by finding the one that maximizes this sum, and append *op* to the supersequence.

3. Update all paths by removing *op* from the front.
4. Repeat steps (2) and (3) until all paths are empty.

Figure 4.3 illustrates the process of finding the supersequence (SSeq) for the DFGs shown in Figures 4.1 (a) and (b). Excluding for simplicity one-node paths, the first DFG has one path $P_1 = \{S, A, A\}$, while the second DFG has four different paths, $P_2 = \{M, S, A\}$, $P_3 = \{M, S, S\}$, $P_4 = \{M, A, A\}$ and $P_5 = \{M, A, S\}$. Here, S represents subtraction, A addition, and M multiplication. Assuming that the areas of different operators are related as $Area(M) > Area(S) > Area(A)$, the maximum weighted sum is found for the multiplier M:

$$Weight(M) = 4 \times (Area(M) + Area(S) + Area(A)), \tag{4.1}$$

which is then inserted into the SSeq and removed from the paths $P_{2,3,4,5}$. Next, the maximum weighted sum is found for the subtractor, which is then appended to the SSeq and removed from $P_{1,2,3}$. Then, the adder is selected, appended to the SSeq and removed from $P_{1,2,4,5}$. Next, between an adder and a subtractor, the subtractor is selected due to its higher area ($Area(S) > Area(A)$). Finally, only the adder remains, so the final supersequence generated by the algorithm becomes: $SS_{\text{eq}} = \{M, S, A, S, A\}$.

4.2 Creating Minimum Area Supersequences

Besides the algorithm presented in the previous subsection, an additional novel heuristic is proposed in this section. It is based on reusing the Maximum Area Common Subsequence (MACSeq) metric of other existing graph-merging algorithms [BKS04]. This way, the algorithm gives priority in the path fusion to those paths that share the MACSeq. The idea is the following: if the paths P_i and P_j sharing the MACSEq are

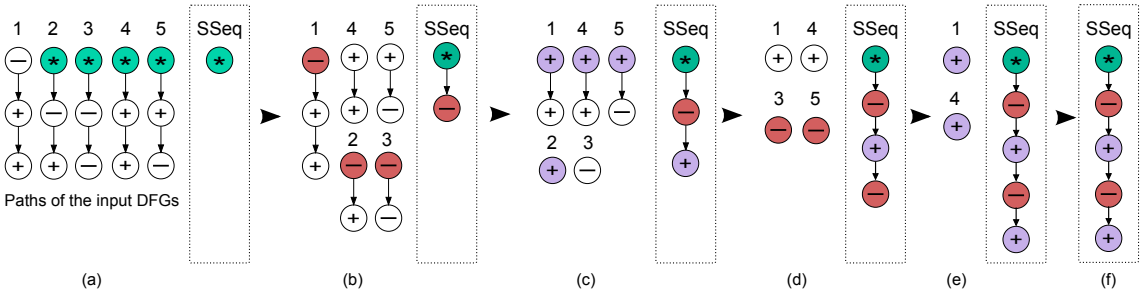


Figure 4.3: Steps of the novel algorithm for finding the shortest common supersequence based on weighted majority merge (WMM) heuristic [BMS98]. (a)The multiplier (M) is the operator occurring at the beginning of the paths and having the highest weight (Equation 4.1). It is thus selected to start the SSeq. (b) The multiplier is removed from the beginning of the paths, and the next operator at occurring at the beginning of the paths and having the highest weight, the subtractor (S), is selected. (c) The subtractor is removed from the paths. The next candidate operator is the adder (A), which is then appended to the SSeq. (d) Two adders and two subtractors remain. Assuming that the ties are broken so that an adder is selected next, the only remaining operator, the subtractor, is selected (e) and the final SSeq is obtained (f).

combined into a single path $P_{i,j}$ first, then the remaining paths in the input set (i) will be either completely contained in the path $P_{i,j}$, or (ii) they will likely share more operators with $P_{i,j}$ than with another path created by fusing any two paths from the input set. This is a slightly different approach towards minimizing the total area of the supersequence than the approach described in the previous section.

The algorithm steps are as follows:

1. Enumerate all paths of each input DFG and group them into multiple sets depending on their length (number of nodes).

2. Starting from the set having the longest paths, perform pairwise search for the MACSeq between paths, using the algorithm for finding the Longest Common Subsequence (LCS) [CZ09] and calculating the LCS area as the sum of areas of its operators. Since the result of the LCS search algorithm depends on the order of input paths P_i and P_j, $i \neq j$, it is run for both (P_i, P_j) and (P_j, P_i) as inputs. If there are multiple pairs of paths sharing the MACSeq, this step reports the pair

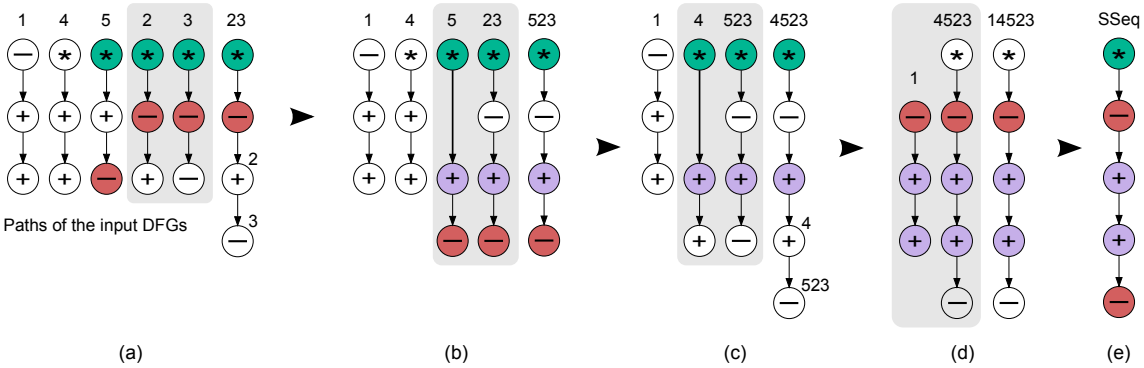


Figure 4.4: Steps of the algorithm based on reusing MACSeq metric of graph-merging algorithms. The paths highlighted in gray are selected for merging. The colored nodes form MACSeq.

that was found first. Additionally, it always reports the order of paths P_i and P_j used by LCS search algorithm for which the MACSeq was found.

3. Perform path fusion on the ordered pair of paths (P_i, P_j) sharing the MACSeq (the order is repored by the previous step). Fusing two paths, begins by aligning all their respective nodes belonging to the MACSeq (see Figure 4.4). Then, the fused path is initialized to the MACSeq, aligned as in paths P_i and P_j. Finally, all nodes of P_i and P_j not belonging to the MACSeq are added to the fused path. In order to do so, all nodes are added in the same relative position with respect to the MACSeq as in the original path. First are added the nodes of P_i and, after them, those of P_j. Once the fused path is obtained, it replaces P_i and P_j in the set of paths yet to fuse.

4. Repeat (2) and (3) until only one path is left in the set.

5. Move the resulting path to the next set containing the paths of smaller length and repeat (2) to (5) until only one path is left: the supersequence.

In practice, the fused path converges quickly into the supersequence, as the shorter paths are most likely already contained in previously fused longer paths.

Figure 4.4 describes how this algorithm is implemented on the same DFGs used for

Figure 4.1: As explained in Section 4.1, the following five paths can be extracted from the two DFGs: $P_1 = \{S, A, A\}$, $P_2 = \{M, S, A\}$, $P_3 = \{M, S, S\}$, $P_4 = \{M, A, A\}$ and $P_5 = \{M, A, S\}$. In this case, the algorithm groups all the paths in the same set because they all have the same length. To find the MACSeq, it can be assumed again that the areas of the different operators are related as $Area(M) > Area(S) > Area(A)$. Accordingly, the MACSeq corresponds to $\{M, S\}$, which is contained in P_2, P_3, and P_5. The first pair that reports MACSeq is (P_2, P_3). Therefore, P_2 is fused with P_3, resulting in $P_{23} = \{M, S, A, S\}$. The next found MACSeq is $\{M, A, S\}$ found for the pair of paths (P_5, P_{23}), resulting in fused path $P_{523} = \{M, S, A, S\}$. Similarly, the next path selected for fusing is P_4, so the fused path becomes $P_{4523} = \{M, S, A, A, S\}$. Finally, the only remaining path, P_1, is selected. This path is already included in P_{4523}, so the supersequence SSeq becomes $P_{14523} = \{M, S, A, A, S\}$.

Clearly, the two heuristics may give different results. For the example DFGs in Figure 4.1, the total area of the SSeq created by modified weighted majority merge algorithm and the algorithm based on finding MACSeqs is the same, but that might not always be the case. Thus, the performances of the two algorithms will be thoroughly analyzed and compared in the experimental part of this work.

4.3 Algorithm Complexity

The algorithms for finding a supersequence involve enumerating all paths of the input DFGs. In the worst-case, this may lead to an exponential number of paths, and thus reaching exponential complexity. Yet, most DFGs derived from real applications do not exhibit this property, despite having a relatively large number of nodes.

Even if the number of paths in a graph were to be exponential, one could design a heuristic to limit the number of enumerated paths. For example, one could enumerate only the unique paths in a graph using a straightforward application of topological sort, and then insert a thresholding mechanism to decide between proceeding with

exhaustive or limited enumeration, depending on the number of paths found. As will be shown in the experimental chapter, in the experiments presented in this thesis a relatively low (in the order of a millisecond) execution time of both supersequence generation algorithms is observed, regardless of the size of the input DFG. Hence, the number of enumerated paths was not constrained.

5 Array Generation

Once a supersequence capturing the characteristics of an application domain is generated, the array column is completely defined and as such used to create a regular domain-specific array. The array generation is performed by replicating the column so that each row in the array is composed of the same type of operators. Hence, the array is homogeneous in one dimension. The value chosen for the number of array columns, N_c, affects the total area of the array as well as its generality. The methods used to determine the optimal value of the parameter N_c are the topic of this chapter.

First, the key steps for finding N_c are elaborated in Section 5.1. Since one of these steps implies mapping application DFGs onto CGRAs, a detailed review of the state-of-the-art in this area is provided in Section 5.2. Then, a novel approach for mapping DFGs is introduced and explained in details in Section 5.3. This approach is essential not only for finding the optimal N_c value, but also for achieving the generality of the final reconfigurable array. Once N_c is chosen, the definition of the array size is complete. However, to provide generality beyond the size of the input set of DFGs, the algorithm for finding N_c can apply an oversizing factor to enlarge the minimum value of N_c. The amount of oversizing can either be provided by the circuit designer or devised automatically, based on the characteristics of the input DFGs. The automated approach for finding a good oversizing factor is described in Section 5.4.

5.1 Method for Determining the Array Size

The optimal number of array columns, N_c, can be defined as the value that (i) leads to a minimum-size array, while (ii) ensuring that all application DFGs known at the design time can fit on this minimum-size array. In other words, it must be possible to map all DFGs from the input set of applications on the $N_r \times N_c$ array, where N_r (the number of rows) equals the length of the supersequence and N_c equals the number of columns.

Assuming that a designer wants to create a single array for accelerating a set of N applications belonging to a same domain, the dataflow graphs are first extracted from the application kernels and then put together in a set. This set will be referred to as the set G of the size N, containing DFGs from D_1 to D_N. The algorithm to devise the minimum value of N_c for the set G starts by creating a supersequence from all DFGs in the set G and building an array by replicating this supersequence a large number of times (Algorithm 1). Without loss of generality, two input and two output ports per column of the array are inserted. Then, a DFG D_i, where $1 \leq i \leq N$, is mapped onto this array. The number of occupied operators per each row of the array is calculated, and the maximum found value assigned to $N_c[i]$. The last two steps are repeated for all DFGs in G. The maximum of all $N_c[i]$ values is N_c, the minimum required number of array columns ensuring that all input DFGs can fit.

The most important step of this algorithm is the implementation of the function for mapping an application DFG onto a reconfigurable array. Mapping a DFG onto a CGRA translates to finding a suitable distribution of DFG nodes among the corresponding array operators. The way how it is done affects not only the minimum needed array size, but also its ability to capture and preserve domain-specific generality.

To leverage on the topological regularity existing in DFGs belonging to the same domain, a mapping algorithm that mimics effective graph drawing algorithms is chosen. Moreover, since graph drawing algorithms tend to keep graph edges as short as possible, to minimize the number of edge crossings and emphasize symmetries, the require-

Algorithm 1: An algorithm to estimate the minimum number of array columns N_c.

```
/* Create a supersequence from all DFGs in the set G.                    */
    arrayColumn = createSupersequence(G);

/* Using the previously found supersequence, create an array             */
/* having a maximum allowable, MAX_INT, number of columns.               */
    Array_MAX_INT = createArray(array_column, MAX_INT);

/* Mapping the DFGs D_i (1 ≤ i ≤ N) on an array having                   */
/* unlimited number of columns (Array_MAX_INT).                          */
    for 1 ≤ i ≤ N do
        Array[i] = map(D_i, Array_MAX_INT);
        N_c[i] = maxNumberOfOccupiedNodesPerRow(Array[i]);

/* Reporting the minimum required size of the array                      */
/* required for D_i to fit in it.                                        */
    N_c = MAX_{1≤i≤N}(N_c[i]);
```

ments imposed on the routing network are alleviated. Such features are particularly important for a domain-specific array introduced in this work. Several researchers have used the same idea of implementing a graph-layout-based application mapping onto CGRAs. The relevant related work is discussed in the following Section. The details of the algorithm used for mapping DFGs in this work are elaborated afterwards.

5.2 Related Work in Graph-Based Application-Mapping

Compilation for CGRAs has traditionally been focused on two issues [AYP+06], [YSP+08], [PFM+08], [Rau94], [LBF+98], [VNK+01], [MVV+02], [LB03], [eLCD03]: (i) placing operations (arithmetic, logic, multiplication, and load/store) of a loop kernel onto the array operators, and (ii) assuring the flow of data (routing) among operators using the existing, usually sparse, routing resources. The loop kernel can then be transformed into a pipeline on a CGRA, completing one iteration every cycle or every *II* cycles, where *II* is the initiation interval of the pipeline [Rau94].

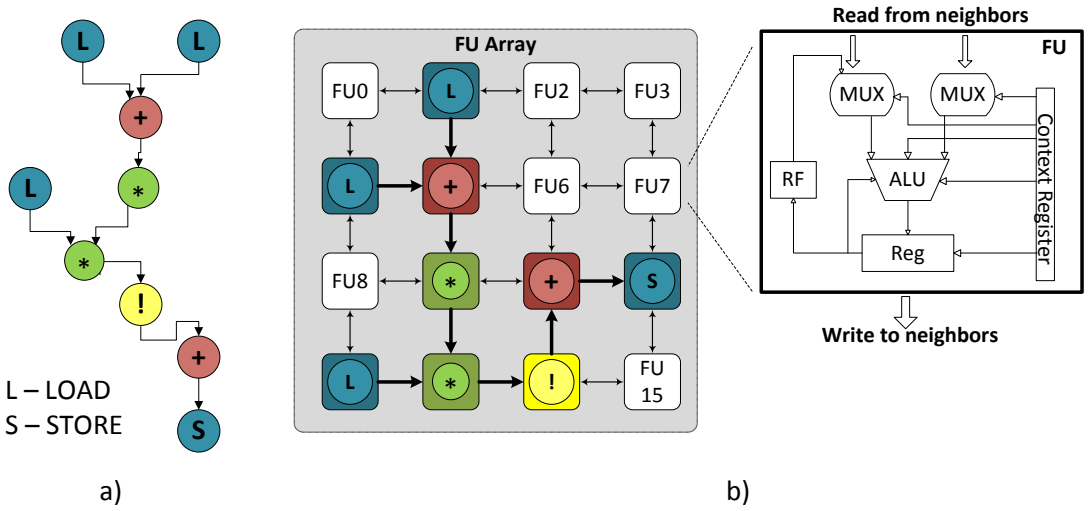


Figure 5.1: (a) An example of the application kernel. (b) One possible mapping of the kernel in a) onto a 4×4 CGRA. It is assumed that FUs are identical, composed of an ALU, multiplexers at the ALU inputs, and register files. Additionally, only connections between four neighboring FUs are provided.

Mapping an application DFG to a regular CGRA structure can be thought of as finding a suitable transformation between the DFG and an array of interconnected functional units (FUs). Figure 5.1 shows an example DFG and one possible mapping of that DFG onto a 4×4 CGRA. It is assumed that each FU is composed of an arithmetic and logic unit (ALU), multiplexers at the ALU inputs, and register files (RFs), and that FUs can communicate only with immediate neighboring FUs. There are many possible ways in which this mapping can be done. In this Section, several relevant papers are discussed:

- First, the spatial mapping algorithm published by Ahn et al. [AYP+06], which uses Sugiyama method [STT81] for drawing layered graphs to find the node placement.
- The algorithm by Yoon et al. [YSP+08] based on Split & Push algorithm [BPV00].
- The edge-centric approach by Park et al. [PFM+08], in which mapping is guided by the *affinity cost* function, directly related to the proximity of DFG nodes.
- Finally, the most recently published graph-minor approach by Chen et al. [CM12].

5.2.1 Spatial Mapping Algorithm for Heterogeneous CGRAs

Ahn et al. [AYP+06] analyzed the problem of automatically mapping applications onto Multiple Instruction Multiple Data (MIMD) heterogeneous CGRAs (HCGRAs). In HCGRAs each functional unit can be configured separately. Hence, the overall performance depends mainly on the application mapping, which should exploit the parallelism embedded in an application and the computational resources of the hardware simultaneously. In the even earlier works several attempts of mapping applications to CGRA were made, but with some differences and limitations. Kim et al. performed manual mapping [KKP+05], Mei et al [MVV+02] provided no support for sharing of common resources among FUs [MVV+02], Lee et al. assumed homogeneous FUs [LB03], while Venkatarani et al. used single instruction multiple data (SIMD) CGRA [VNK+01].

Application mapping can be classified into two categories: temporal and spatial mapping. In temporal mapping, necessary configurations are all stored in the configuration cache and the configuration of each FU is dynamically changed with time. In spatial mapping, each FU has a fixed configuration, and the data to be processed are routed through FUs. Temporal mapping may reduce the number of FUs required for mapping in comparison with spatial mapping. In spatial mapping, the mapping is limited by the topology and size of the reconfigurable array, but it has no configuration overhead and thus reduces the configuration storage. Therefore, the spatial mapping strategy is in some cases more effective for embedded applications, and it is applied in this work.

Ahn et al. [AYP+06] proposed an algorithm for spatial mapping using methods for drawing hierarchical graphs. They first analyzed the application code to detect loop kernels and represent them in a tree form, called the kernel tree. In a kernel tree, each node is an atomic operation, such as addition or multiplication, while an edge represents a data dependence between operators. An example of the application code extracted from complex update application from DSPStone benchmarks [ŽVSM97] is shown in the following algorithm.

Algorithm 2: An example code extracted from DSPStone benchmarks [ŽVSM97].

for $(i = 0; i < N; i+ = 2)$ **do**
    $temp_1 = c[i] + a[i] * b[i]$;
    $d[i] = temp_1 – a[i+1] * b[i+1]$;
    $temp_2 = c[i+1] + a[\mathrm{i}+1] * b[i]$;
    $d[i+1] = temp_2 + a[i] * b[i+1]$

Figure 5.2a shows the kernel tree extracted from this code. Ahn et al. [AYP+06] assumed that FUs are composed of one ALU, preceded by multiplexers at its every input and followed by a shifter and pipelining register. By changing the configuration of the FU, various combinations of operations can be executed on it, e.g., loading operators on both inputs followed by multiplication, performing ALU operation followed by shifting, or loading only one operator and shifting. Several kernel operations can be mapped on one FU, if the corresponding configuration exists. Using the set of possible configurations, Ahn et al. transform the kernel tree to a configuration tree in which each node represents a configuration for each FU, possibly covering, and thus executing, more than one operation of the kernel tree.

Ahn et al. divided the algorithm for spatial mapping into three phases: covering, partitioning, and layout. The covering problem is essentially analogous to instruction selection problem [API03]. To solve it, they implemented a compiler that takes as input the kernel code, computes the covers for nodes in the kernel tree as shown in Figure 5.2b, and produces as output the configuration tree in Figures 5.2c. Several nodes in the kernel tree can be replaced by one node in the configuration tree if there is a configuration of FU in the CGRA that supports their execution on a single FU. In the partitioning phase, they partition the nodes of the configuration tree into different clusters, each scheduled later to each column of the PE array. Later, in the laying-out phase, these partitions are input to their integer linear programming (ILP) solver for finding vertical assignment of nodes in the CGRA. The algorithm always tries to assign two partitions with heavy data traffic as close as possible, which usually leads to minimized number of FUs that are used as route-through only.

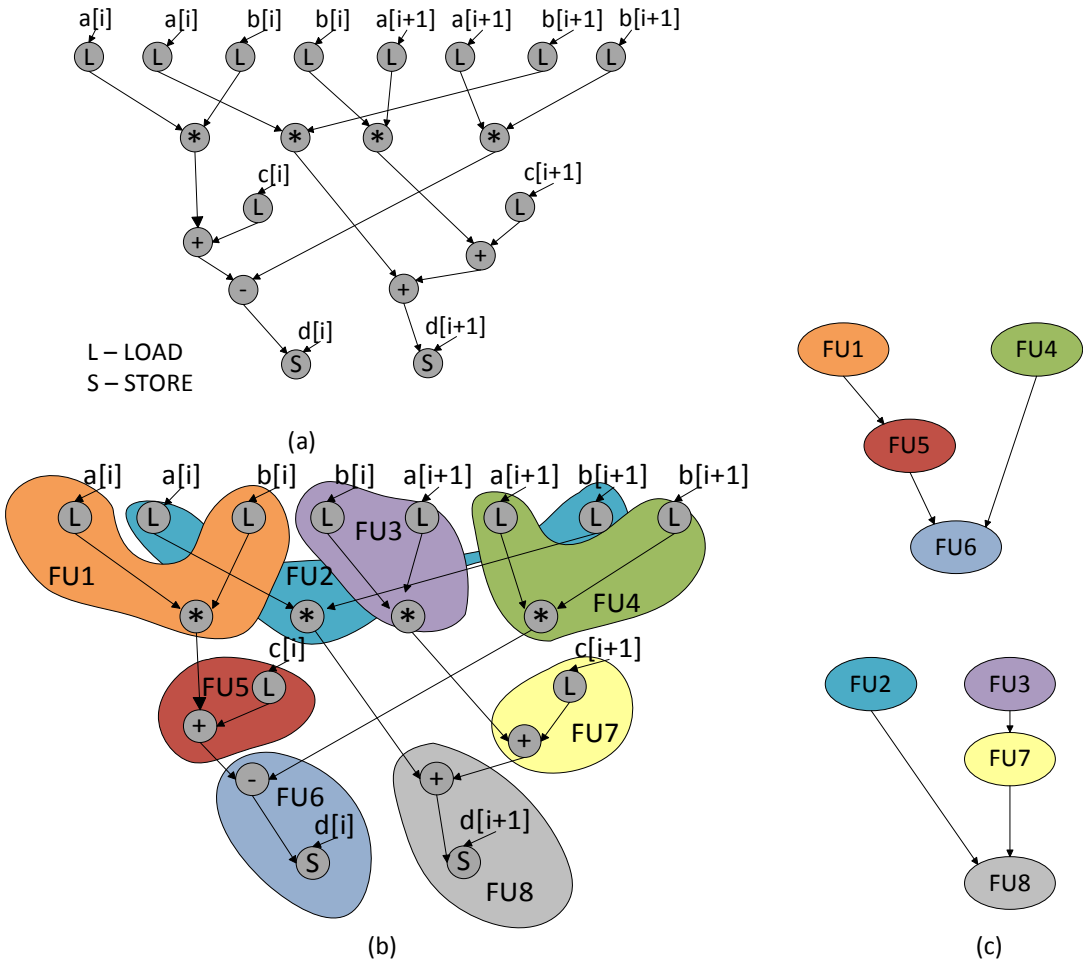


Figure 5.2: (a) The kernel tree of the complex update application from DSPStone benchmarks [ŽVSM97]. (b) Kernel tree after covering. (c) Configuration tree.

After vertical assignment, Ahn et al. use the Sugiyama method [STT81] for drawing layered graphs to find the node positions that

- minimize edge crossings and
- keep adjacent the node pairs that exchange data.

The same idea is used in this thesis.

Experimental evaluation [AYP+06] has shown that this algorithm does not always produce optimal solutions, although in many cases its performance is comparable to those obtained with hand optimizations.

5.2.2 Split & Push Kernel Mapping Algorithm

Yoon et al. [YSP+08] focused on solving the problem of mapping a kernel of a given loop onto a large-scale CGRA, while simultaneously minimizing the number of resources required. They proposed a graph-drawing based approach called split-push kernel mapping (SPKM). It is based on the Split & Push algorithm used in the graph-drawing area [BPV00]. In Figure 5.3 an example of mapping a four-operation kernel graph onto a 2 × 2 CGRA is shown. The algorithm starts with all nodes of a kernel graph located at the same coordinate Figure 5.3a. Then, it uses cuts to split vertices into two distinct groups. A cut is a plane orthogonal to one of the axes (shown by dotted lines). After the node separation, the vertices in one of the two groups are pushed to a new coordinate. Figure 5.3b shows the result of Split & Push along the horizontal dotted line. This procedure is repeated until every vertex has distinct coordinate, as shown in Figure 5.3c.

To minimize the number of used rows in the mapping, Yoon et al. proposed a three-stage heuristic. The first stage is a column-wise scattering, in which vertices are distributed to the minimum number of utilized rows in the same column. The second stage is the routing FU insertion, in which routing FUs are generated and connected with existing vertices. The third stage is a row-wise scattering, in which they try to avoid diagonal edges and edge crossings by placing the nodes that have connections between different rows in the same column.

The authors compared the SPKM with the approach in [AYP+06] on a set of kernel graphs from benchmarks such as Livermore loops, MultiMedia and DSPStone, and on a set of randomly generated kernel graphs. The results showed that SPKM can on average map 4.5 × more applications than the approach in [AYP+06]. SPKM is also able to generate mappings requiring smaller number of rows than the approach [AYP+06] in 62% of the applications. Additionally, SPKM has only 5% overhead in mapping time, while both approaches are significantly faster than ILP.

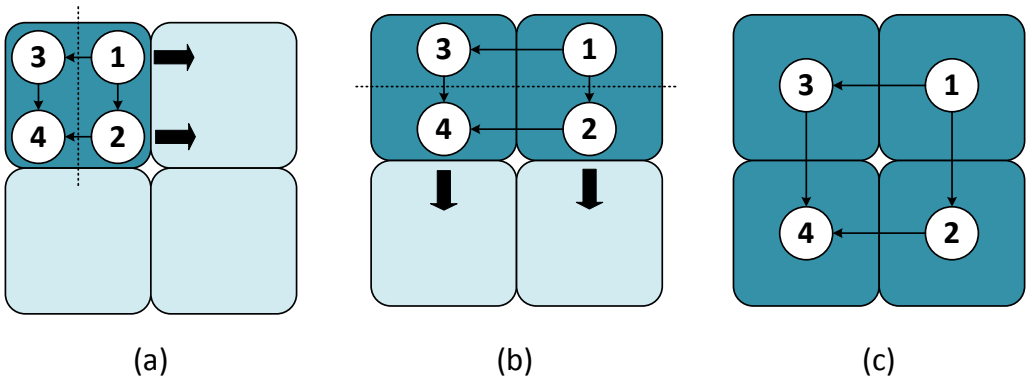


Figure 5.3: (a) The kernel tree of the complex update application from DSPStone benchmarks [23]. (b) Kernel tree after covering. (c) Configuration tree.

5.2.3 Edge-Centric Modulo Scheduling

Traditional schedulers [MVV+03, PFKM06] address the scheduling task in a node-centric manner by focusing on assigning DFG nodes to FUs. Park et al. [PFM+08] argue that selecting intelligent paths from producing to consuming FUs, which do not block other operand paths is crucial for achieving higher throughput schedules. Hence, they propose an edge-centric modification of modulo scheduling, in which the scheduler focuses primarily on routing, while placement is a by-product of the routing process. Modulo scheduling is a software pipelining technique that exposes parallelism by overlapping successive loop iterations to find a valid schedule that will minimize the interval between successive iterations (initiation interval, or *II*) [Rau94].

In an edge-centric approach, the scheduler does not place operations up front. Instead, it selects an edge from the already placed producers or consumers of the operator and attempts routing that edge. To do that, the router searches for an empty slot capable of executing the target operation. In a node-centric approach, the router would instead route towards a placed operation. In the edge-centric approach, once a compatible slot is found, the target operation is placed in it and the scheduler continues routing DFG edges to remaining nodes. Minimization of the number of the routing resources used is achieved by assigning a statically determined fixed cost to the routing resources, thus forcing the router to look for a path minimizing the total cost. The authors propose

propose using *affinity cost* [PFKM06]. The affinity cost of a pair of operations reflects their proximity in the DFG. Hence, it forces the router to place operators near their producers and consumers whenever possible, and thus reduce the number of used routing resources. To avoid routing failure, occupancy probability is associated to all scheduling slots to find which resources are likely to be used by other edges in the future.

To evaluate the performance of their approach, they selected a set of loops from typical media applications, such as H.264 decoder, 3D graphics, AAC decoder, MP3 decoder, and others, to map onto various CGRA configurations. The results showed that the edge-centric approach improved performance by 25% over the traditional modulo scheduling and achieved 85-98% of the performance compared to the state-of-the-art simulated technique DRESC [MVV+03] with compilation time reduced by 18×. However, the drawback of this approach is that its performance strongly depends on the characteristics of DFG structures and the underlying CGRA architectures.

5.2.4 Graph-Minor Approach

Chen et al. [CM12] noticed that none of the previous approaches attempted to share routes corresponding to different graph edges having the same source node. Additionally, most of the existing techniques did not even model explicit routing through register files. Most approaches implicitly assumed the availability of a sufficient number of registers and interconnections between them as well as functional units, even though the register files consume significant amount of area and substantially impact CGRA performance [MVV+03].

The authors introduced another approach that integrates register allocation with scheduling through explicit modeling of the register files and their connectivity with the functional units. Essentially, they transformed a CGRA mapping problem with route sharing into a graph-minor problem and proposed a framework based on graph mapping for solving this problem. To model the register files and their connectivity,

Chen et al. introduced some modifications to Modulo Routing Resource Graph (MRRG). Initially, MRRG was proposed by Mei et al. [MVV+03] as a resource-management graph that captures interconnections among FUs and register files. Park et al. [PFKM06] introduced a slightly modified version of MRRG, that was used by Chen et al. According to them, MRRG is a directed graph whose nodes represent an FU or a register file, while edges represent the connectivity between nodes in a time-space view.

Chen et al. used the graph-minor [RS99] based formulation of the application mapping problem on CGRA with route sharing. An undirected graph $G\prime$ is called a minor of the graph G if $G\prime$ is isomorphic to a graph that can be obtained by edge contractions on a subgraph of G. An edge contraction is an operation that removes an edge from a graph while simultaneously merging together the two vertices it used to connect. Since the definition of the graph-minor is restricted to undirected graphs, they extended it by defining the edge contraction operation for directed graphs.

The algorithm for CGRA with route sharing by Chen et al. is as follows. First the minimum possible *II* is computed. Then, the modulo routing resource graph corresponding to the CGRA architecture and the minimum *II* is created. Further on, a subgraph $G\prime$ in G, such that the application dataflow graph is a restricted minor of $G\prime$, is searched for. If such graph exists and can be found, then the DFG can be mapped to the CGRA with the initiation interval *II*. Otherwise, *II* is incremented by one, and the previously mentioned steps are performed again. The whole process is repeated until a MRRG with sufficiently large *II* is generated and the DFG can satisfy the graph-minor test.

To evaluate the performance of the algorithm, Chen et al. used a set of kernels from standard benchmarking suites and three different register file configurations: one with no register files, one with local shared register files, and one with a central shared register file. The target CGRA architecture was a 4×4 array with possibly heterogeneous units that can be found in ADRES [BBKG07], MorphoSys [LSL+00], and other known CGRAs. They measured the achieved performance for different CGRA configurations (different number of memory units, different register file configurations). The results

have shown that adding registers may not necessarily improve *II*, contrary to the conclusions published by Kwok et al. [KW05], who recommended a global register file with a large number of registers. The reason was that the algorithm for register allocation may end up with a schedule that uses a large number of registers, while sharing routes helps reducing register file pressure and thus achieves a valid schedule using smaller number of registers. Compared to DRESC [MVV+02], their algorithm yields improvement in the compilation time more than an order of magnitude, along with increased average resource utilization (62% compared to 54% for DRESC).

5.3 DFG Placement onto Domain-Specific Arrays

As seen in the previous section, graph-based approaches for application mapping onto CGRAs have been used extensively. When input applications belong to a single domain, their DFGs exhibit topological regularity, which should be exploited to devise an efficient mapping algorithm. One way to leverage on this regularity is to use a placement algorithm that mimics effective drawing algorithms. Those algorithms tend to produce graph layouts with graph edges as short as possible, minimized number of edge crossings, and emphasized symmetries. For example, related work by Ahn et al. [AYP+06] has relied on the Sugiyama method [STT81] for drawing layered graphs to define the horizontal node positions, for a fixed vertical node assignment. A similar approach is implemented here, using the well-known publicly-available package for manipulating graphs and their drawings—Graphviz [GN00].

Besides path fusion, the way application mapping is done is a very important step towards achieving the goal of this work—automatically creating domain-specific reconfigurable arrays. Preserving the topological regularity of DFGs even after their placement on the array, increases the probability that this array could also accelerate other applications sharing similar computational structures, and thus belonging to the same or a similar domain.

Three subsection follow. Subsection 5.3.1 describes the way Graphviz tool is invoked and the algorithm it uses to draw hierarchical graphs. Subsection 5.3.2 presents a method for guiding the graph-drawing tool to place DFG nodes on specific horizontal coordinates, resembling array rows. Finally, subsection 5.3.3 describes methods for snapping nodes to array columns and for tuning the horizontal node positions, when the suggested placement by the tool cannot be implemented on a real array as is.

5.3.1 Laying Out Graphs with `dot`

Graphviz toolkit contains two libraries, Libgraph and Dynagraph. Libgraph supports reading, writing, and manipulating graph abstractions, allowing fine-tuning of performance critical code. Dynagraph is layered on top of Libgraph and realizes a framework for displaying incrementally changing graphs. Both share a common graph specification language. Libgraph embodies a common attributed graph data language for graph manipulation tools. Embedding tool-specific data and command syntax in graph descriptions makes it difficult to write compatible graph filters. By delegating graph file I/O to Libgraph, graph tools are syntactically compatible by default. The Libgraph language is conventionally known as the `dot` format, after its best-known application. The `dot` language provides syntax for defining graphs, nodes and edges, plus the ability to attach string-valued name-attribute pairs to graph components.

`dot` draws directed graphs. It reads graph text files with the extension `.dot` and writes drawings, either as graph files or in a graphics format such as GIF, PNG, SVG, or PostScript. Graph drawing is done in four main phases [GKN06]:

- Since the layout procedure used by `dot` relies on the graph being acyclic, the first step is to break any cycles which occur in the input graph by reversing the internal direction of certain cyclic edges.
- The next step assigns nodes to discrete *ranks* or *levels*. In a top-to-bottom drawing, ranks determine y coordinates. Edges that span more than one rank are broken

into chains of "virtual" nodes and unit-length edges.

- The third step orders nodes within ranks to avoid crossings.
- The fourth step sets x coordinates of nodes to keep edges short, and the final step routes edge splines; In mathematics, a spline is a sufficiently smooth polynomial function that is piecewise-defined, and possesses a high degree of smoothness at the places where the polynomial pieces connect [Che09].

This is the same general approach used by the majority of hierarchical graph drawing programs, which are based on the work of Warfield [War77], Carpano [Car80], and Sugiyama [STT81]. `dot` accepts input described using `DOT` language. This language describes three kinds of objects: graphs, nodes, and edges. Besides `dot`, which makes layouts of directed graphs, there is one more layout utility in Graphviz package, which accepts the same input and draws undirected graphs. It is called `neato` [Nor04]).

Figure 5.4 shows an example graph text file in the `dot` language corresponding to a 3 × 3 convolution application DFG shown in Figure 5.5. The first line gives the graph name, `G`, and type, `digraph`. The lines that follow create nodes, edges, or subgraphs, and set attributes. Names (labels) of all these objects may be C identifiers, numbers, or quoted C strings. A node is created when its name first appears in the file. An edge is created when nodes are joined by the edge operator "− >". Running dot on this file (for example called `graph1.dot`) is done using the following command:

```
$dot -Tps graph1.dot -o graph1.ps,
```

and yields the drawing of Figure 5.5, a basic block extracted from 3 × 3 convolution.

It is often needed to adjust the representation or placement of nodes and edges in the layout. This is done by setting attributes of nodes, edges, or subgraphs in the input file. Attributes are *name-value* pairs of character strings. Node or edge attributes are set off in square brackets. Nodes are labeled by the node name and drawn, by default,

```
shape=ellipse, width=.75, height=.5,
```

where dimensions are in inches. Other common shapes are `box`, `circle`, `record`, and

```
digraph G {

        /* Labeling graph nodes */
        1 [label="1 (IN)"];
        2 [label="2 (IN)"];
        3 [label="3 (IN)"];
        4 [label="4 (IN)"];
        5 [label="5 (IN)"];
        6 [label="6 (IN)"];
        7 [label="7 (IN)"];
        8 [label="8 (MUL)", color = gray, style = filled];
        9 [label="9 (MUL)", color = gray, style = filled];
        10 [label="10 (MUL)", color = gray, style = filled];
        11 [label="11 (ADD/SUB)", color = gray, style = filled];
        12 [label="12 (ADD/SUB)", color = gray, style = filled];
        13 [label="13 (ADD/SUB)", color = gray, style = filled];
        14 [label="14 (OUT)"];

        /* Drawing graph edges */
        1->11;
        8->11;
        2->8;
        3->8;
        4->9;
        5->9;
        6->10;
        7->10;
        9->12;
        10->12;
        11->13;
        12->13;
        13->14;
}
```

Figure 5.4: Graph text file example.

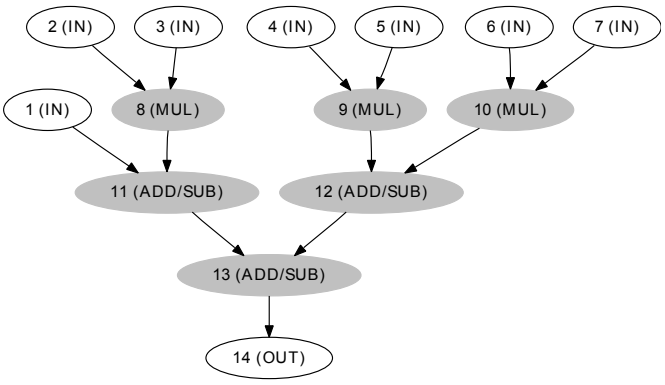


Figure 5.5: A DFG corresponding to 3 × 3 convolution. The graph text file in Figure 5.4 corresponds to this graph, drawn using `dot`.

plaintext. Nodes and edges can specify a color attribute, with black the default. This is the color used to draw the node's shape or the edge.

Two attributes that play an important role in determining the size of a dot drawing are nodesep and ranksep. The former specifies the minimum distance, in inches, between two adjacent nodes on the same rank. The second deals with rank separation, which is the minimum vertical space between the bottoms of nodes in one rank and the tops of nodes in the next. This attribute sets the rank separation, in inches. Alternatively, one can set ranksep=equally, which guarantees that all ranks are equally spaced, as measured from the centers of nodes on adjacent ranks. In this case, the rank separation between two ranks is at least the default rank separation. For example:

```
ranksep="1.0 equally"
```

causes ranks to be equally spaced, with a minimum rank separation of 1 inch.

In graphs with time-lines, or in drawings that emphasize source and sink nodes, one has to constrain rank assignments in reference with the minimum rank, which occurs at the top of the drawing. The rank of a subgraph may be set to same, min, source, max, or sink. The value same causes all nodes in the subgraph to occur on the same rank. If set to min, all nodes in the subgraph are guaranteed to be on a rank at least as small as any other node in the layout. This can be made strict by setting rank=source, which forces the nodes in the subgraph to be on some rank strictly smaller than the rank of any other nodes (except those also specified by min or source subgraphs). The values max or sink play an analogous role for the maximum rank. These constraints induce *equivalence classes* of nodes, and are key for making dot graph drawing algorithm suitable for mapping application DFGs on domain-specific coarse-grained arrays. Nodes belonging to the same rank can be thought of as the nodes to be placed in the same array row. Since the DFGs are to be layed out in top-down fashion, nodes having different ranks in dot correspond to nodes having different depth in a graph.

> *A depth of a node is the maximum length of a path from input ports to the node, among all paths containing that node.*

5.3.2 Assigning Nodes to Rows

The procedure of laying out a dataflow graph starts by marking each `dot` rank to the corresponding row in the array. Assuming that the supersequence found in the path fusion step equals the sequence $SSeq = \{OpRow_1, OpRow_2, ..., OpRow_{N_r}\}$, where N_r equals the number of rows in the array and $OpRow_i$ marks the type of the operator in the i-th row, the following piece of text in `dot` input file defines all possible ranks that DFG nodes can take (note that `dot` enumerates ranks in decreasing order):

```
{    /* the operators */
   node [shape=plaintext];
   "IN" → "OpRow_r" → "OpRow_{r-1}" →...→ "OpRow_1" → "OUT";
};
```

The next step is to write out DFG nodes, divided into groups based on their ranks (rows to be placed in). To decide the row in which to place a node, nodes are first grouped according to their depths. Since nodes, in general, can belong to multiple paths in the graph, the depth of a node is defined as the maximum length of a path from input ports to the node, among all paths containing that node. For example, node 11, an adder/subtractor belonging to the DFG shown in Figure 5.5, is contained by two partly overlapping paths: P_1 = {8 (MUL), 11 (ADD/SUB), 13 (ADD/SUB)} and P_2 = {11 (ADD/SUB), 13 (ADD/SUB)}. In the path P_1, the depth of this node equals two, while in the path P_2 its depth equals one. Thus, the final depth of the node 11 is the larger of the two values and thus equals two.

However, there is one drawback of this approach. In particular, in DSP applications it is often the case that nodes belong to a binary tree of identical operators, due to accumulation of partial results. Thus, if nodes are to be distributed in different rows according to their depth only, the overall row utilization would be very low. To improve it, a *binary tree optimization* procedure is proposed and implemented:

Binary tree optimization: The graph nodes that form a binary tree are repeatedly assigned to rows with as high rank as appropriate, as long as there are free operators in those rows.

The procedure for assigning nodes to array rows can be summarized in the following *top-down* approach:

1. Group nodes by depth.
2. Assign all inputs to the row containing input ports.
3. Start from the group of operators having the minimum depth d.
4. For all nodes in the selected group, do the following:
   - If the node is a part of a binary tree assign it to the row with the same operator type and the rank equal or lower than that of the predecessor node(s).
   - If the node is not a part of a binary tree, assign it to the row with the correct operators having lower rank than that of the predecessor node(s).
5. Move to the group of nodes having depth $d+1$.
6. Repeat steps (4) and (5) until all nodes are assigned to a row.

Figure 5.6 shows how the optimization of row utilization is achieved for binary trees: (a) illustrates a chain of operators due to accumulation of multiple partial results, frequently found in DSP application DFGs; (b) shows how this chain is transformed into a binary tree; (c) suggests that the row utilization can be improved by assigning the tree nodes to one of the predecessor node rows.

Two important facts should be noticed. First, by construction of the supersequence and of the array, all nodes can always be placed greedily on the rows of the array. However, due to the binary tree optimization, some rows in the datapath may never be used by any of the input DFGs. To optimize total array area, these rows are automatically

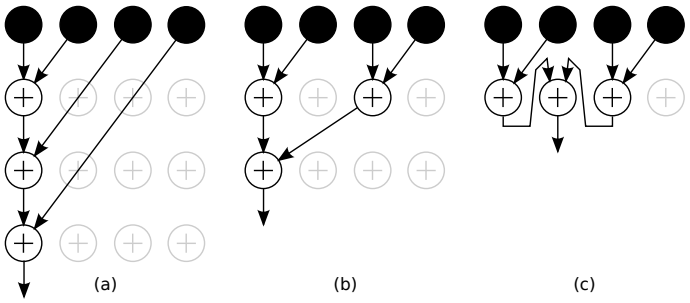


Figure 5.6: Optimization of row utilization for binary trees. (a) Part of a typical DFG with accumulation of multiple partial results. (b) The chain of operators transformed into a binary tree. (c) Tree nodes assigned to one of the previous rows to minimize the tree height and thus improve row utilization.

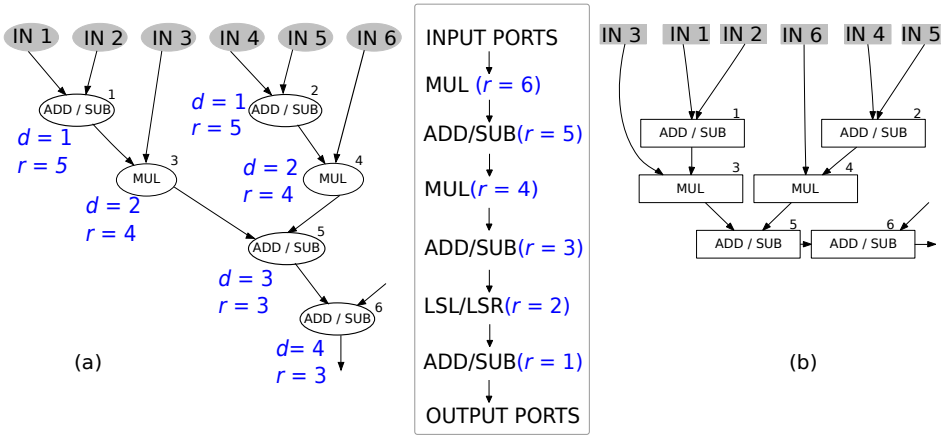


Figure 5.7: Assigning nodes of a subgraph for placement. (a) Subgraph with node depths d marked. Nodes having the same depth and performing the same operation are assigned to the same row (rank r). Nodes are always assigned lower rank than their predecessors, unless they are a part of a binary tree (Node 6). (b) The same subgraph after placement by `dot`.

removed from the array. The array in Figure 5.6c uses one row less than the array in Figure 5.6b. Second, removing rows from the array this way has no effect on its ability to support new sequences of operators belonging to other applications, as long as the resources available are sufficient.

In Figure 5.7, a subgraph of DFG D_7 used in the experimental evaluation (Chapter 8) is shown. Nodes 1, 2, 5, and 6 perform addition/subtraction. Nodes 3 and 4 perform multiplication. The supersequence obtained by merging D_7 with other DFGs is the sequence $\{M, AS, M, AS, L, AS\}$ where M stands for multiplication, AS for addition/subtraction,

and *L* for left/right shift. The first node in the supersequence has the rank 6 (there are 6 rows in the datapath), while the last node has the rank 1. Node depths are marked on the subgraph shown left: nodes 1 and 2 have the lowest depth, and perform operation *AS*. They are assigned the highest ranked row of adders/subtractors, the row with the rank $r = 5$. Nodes 3 and 4 have depth $d = 2$. They can be assigned to the row with the rank $r = 6$ or the row with the rank $r = 4$; however, following the top-down approach, they are assigned to the row with rank $r = 4$ because it is lower than the rank of the rows of predecessor nodes 1 and 2. Similarly, node 5 is assigned the row with rank $r = 3$. Node 6 belongs to a binary tree and thus its predecessor, node 5, performs the same operation. Therefore, to minimize the tree height, node 6 is assigned the row with rank $r = 3$, the same as the predecessor.

Once the supersequence and DFG nodes with their ranks assigned are passed to `dot`, the tool is forced to place operators only within the rows, without the possibility to move any operator from one row to another.

5.3.3 Assigning Nodes to Columns

Besides drawing graph layouts, `dot` outputs layout information in a textual file. The default output format is the attributed `dot` format (`Tdot`) [GKN06], which reproduces the input graph description, along with layout information, such as coordinates of nodes and edges, total graph size, and node dimensions. Coordinate values increase up and to the right. Positions are given by two integers separated by a comma, representing the X and Y coordinates of the location specified in points (1/72 of an inch). A position refers to the center of its associated object (node, edge, label).

To map data-flow graphs as if only a discrete set of values of node and port coordinates is allowed, the `dot` is guided in the following way:

- Nodes and ports are defined as rectangular boxes.
- The width of nodes is set to 1.8 in.

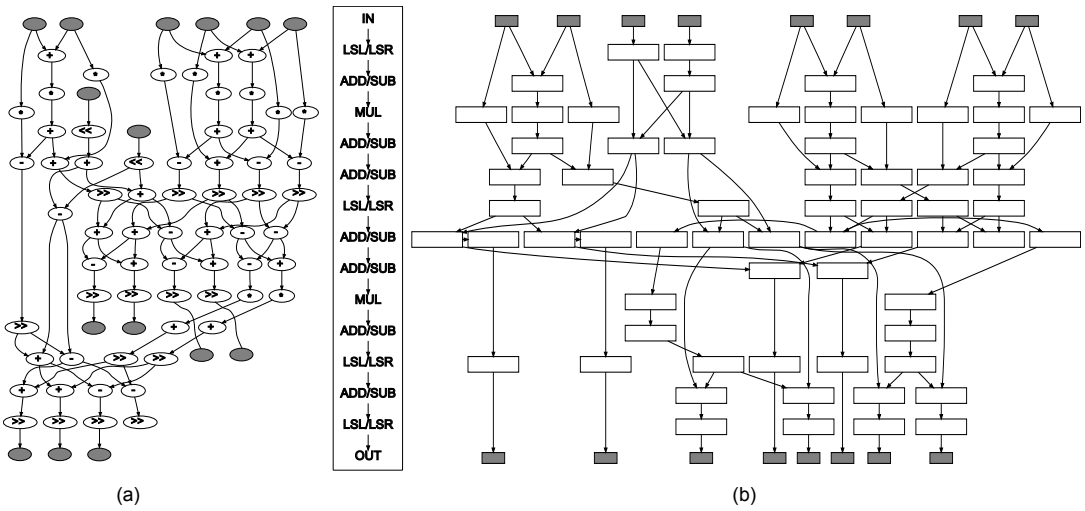


Figure 5.8: The placement process. (a) A basic block extracted from 32b Inverse Two-dimensional DCT (Table 7.1). (b) The DFG from (a) is laid out using `dot` and appropriate constraints and parameters to suggest a detailed placement on the array.

- The width of ports is set to 0.8 in.
- The separation between nodes is set to 0.2 in.

Hence, the centers of any two nodes in the same row (with the same vertical coordinate) cannot be separated by less than 1 in. Consequently, if rounded to the nearest integer, the node centers are guaranteed to be non-overlapping. Additionally, not more than two input (output) ports can be layed out next to one another above (below) a node to which they are connected. An example `dot` layout of a DFG extracted from 32b inverse two-dimensional DCT is shown in Figure 5.8.

When designing the array, the graph layout as suggested by `dot` should be preserved because `dot` layouts emphasize regular patterns that are characteristic to a domain. Therefore, the array size should be sufficiently flexible not to constrain the DFG placement. For each of the DFGs that are input to the analysis, the minimal number of array columns $N_{c,min}$ needed to place them, while completely preserving `dot` layout, can be devised from the `dot` output file:

$$N_{c,min} = \frac{\text{MAX horizontal distance between any pair of node centers}}{\text{MIN horizontal distance between any pair of node centers}}. \tag{5.1}$$

The final array size should be sufficiently large to accommodate every input DFG. However, when a designer wants to map a DFG that has not been available during the design time, he has to find the correct placement using an array of fixed size. If, after laying out the DFG by `dot`, the DFG nodes are too distant and thus cannot fit the array, a rescaling of the coordinates suggested by `dot` is initiated. First, all coordinates are simply scaled horizontaly. The scaling factor f_s is chosen to be as large as possible, to preserve the most of the layout suggested by `dot`. Thus, the scaling factor equals the ratio of the number of columns required to place the DFG and the number of columns available in the array, and is less than one:

$$f_s = \frac{\text{The number of columns required by } \texttt{dot} \text{ placement.}}{\text{The number of columns available in the array.}} \leq 1. \tag{5.2}$$

Scaling is done by multiplying the node center coordinates (output by `dot`) with the scaling factor f_s. Since the scaling factor is a real number, rounding the scaled coordinates to obtain node positions respective to array columns may cause some nodes that are in the same row to overlap, i.e. desire to occupy a single column. To avoid this, an algorithm to redistribute nodes within columns is proposed and implemented.

- First, all nodes belonging to a row are sorted in increasing order of their horizontal coordinate (recall here that in the `dot` outputs the coordinates increasing up and to the right).
- Starting from the node with the lowest horizontal coordinate value, the algorithm proceeds by attempting to place it in the column closest to the rounded and scaled coordinate value.
- If the node desires to be placed in the already occupied slot, the algorithm first checks if there are empty slots in one of the columns on the left side of the candidate one.
  - If there are empty slots, the algorithm shifts previously placed nodes towards the left. This shifting is done carefully to keep the overall geometrical relation

among nodes as to resemble those suggested by `dot`. To guide the shifting a special function calculating the cost of displacing already placed nodes is introduced. This function first identifies the first empty slot on the left. Then it estimates the cost of shifting all nodes that are on the right side of this empty slot to the left for a single column. The following *displacement cost function* is used to estimate the cost of shifting nodes:

$$f_{displacement\_cost} = \sum_{N_{\text{empty\_cell}} < i < N_{\text{candidate\_cell}}} (x_{i,prev} - x_{i,new})^2. \tag{5.3}$$

The cost is calculated as the sum of squared differences between the node previous coordinate $x_{i,prev}$ and the node coordinate that would be obtained after shifting in the left $x_{i,new}$, for all nodes i placed on the right side of the empty cell, i.e. between columns $N_{\text{empty\_cell}}$ and $N_{\text{candidate\_cell}}$. If the estimated cost is less than the cost of placing the conflicting node in the first available cell on the right, than the nodes are shifted, and the node is placed in the candidate cell. Otherwise, the node is placed in the first free cell on the right of the candidate cell.

– If there are no empty slots on the left side of the candidate cell, then the shifting cannot be performed, and the new node has to placed in the first empty slot on the right side of the candidate cell.

Algorithm 3 gives the details of the procedure for redistributing nodes within rows. Figure 5.9 shows the DFG from Figure 5.8a placed on a reconfigurable array after rounding and scaling the node coordinates.

5.4 Oversizing The Number of Columns

To provide generality beyond the size of the input set of DFGs, the array generation methodology can apply an oversizing factor to provide for more than just a minimum

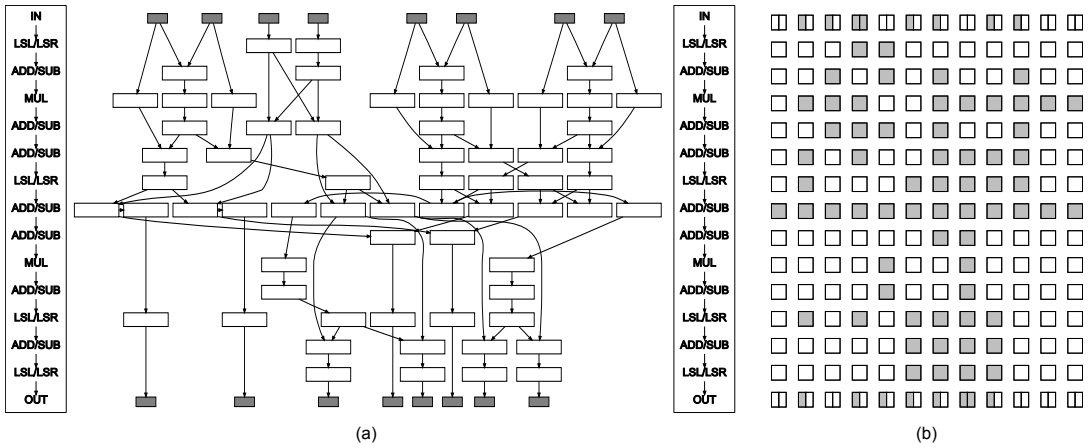


Figure 5.9: (a) The basic block extracted from 32b Inverse Two-dimensional DCT (Table 7.1) is laid out using `dot` and appropriate constraints and parameters to suggest a detailed placement on the array. (b) The suggested placement of the DFG in (a) on a reconfigurable array, after rounding and scaling the node coordinates suggested by `dot`.

number of columns $N_{\text{c,min}}$. Due to the array architecture, the number of input/output ports is directly related with the number of columns. Hence, the oversizing factor $F_{\text{col,oversize}}$ influences both the final number of columns and the final number of array ports. The value of the oversizing factor can be either chosen arbitrarily (estimated by a designer) or devised automatically. The final number of columns in the array N_c would then equal the minimum number of columns needed to support placing all input DFGs increased for the value of the oversizing factor:

$$N_{\text{c}} = N_{\text{c,min}} + F_{\text{col,oversize}}, \text{ where } F_{\text{col,oversize}} \geq 0. \tag{5.4}$$

The algorithm for analyzing input DFGs and estimating the oversizing factor is is based on measuring the minimum oversizing which would have been necessary to implement any of the given DFGs if it were not part of the initial set. For example, one can assume that the designer has provided a set G containing N DFGs. For each DFG D_i ($1 \leq i \leq N$) from G, a set G_i containing all DFGs except D_i is created, and path fusion is used to find the supersequences $SSeq_i$. Further, two arrays are created. The first is the array created by replicating $SSeq_i$ as many times as needed to assure all DFGs in G_i can be

Algorithm 3: The algorithm to redistribute nodes within a row.

```
for 1 ≤ r ≤ N_r do
    for each node n assigned to this row do
        candidateColumn = scaled&rounded horizontal coordinate of the node n;
        if candidateColumn > totalNumberOfColumns then
            Report insufficient number of columns in the array and return;
        if theLastOccupiedColumn < candidateColumn then
            placeNode(n, candidateColumn);
            theLastOccupiedColumn = candidateColumn; Continue;
        if theLastOccupiedColumn == candidateColumn then
            if theLastOccupiedColumn ≠ theRightMostColumn then
                if theFreeOnTheLeftCost < theDisplacementCost then
                    FreeLastSlot(theLastOccupiedColumn);
                    placeNode(n, theLastOccupiedColumn); Continue;
                else
                    placeNode(n, theLastOccupiedColumn + 1); Continue;
            else
                FreeLastSlot(theLastOccupiedColumn);
                placeNode(n, theLastOccupiedColumn); Continue;
        if theLastOccupiedColumn > candidateColumn then
            if theLastOccupiedColumn ≠ theRightMostColumn then
                Keep moving already placed nodes until the cost of moving is balanced
                with the displacement cost of the node;
                Place the node n in the next column on the right; Continue;
            else
                Free the slot in the last occupied column and place the node n in the
                next column on the right; Continue;
```

placed onto it. The number of columns in this array is denoted as $N_c(G_i)$. The second is the array created by replicating the same supersequence, but as many times as needed to place D_i. The number of columns in this second array is denoted as $N_c(D_i)$. The oversizing factor equals:

$$F_i = \begin{cases} N_c(D_i) - N_c(G_i) & \text{if } N_c(D_i) - N_c(G_i) \geq 0, \\ 0 & \text{otherwise.} \end{cases} \tag{5.5}$$

The maximum of all candidate F_i oversize factors, where i is in the range $1 \leq i \leq N$, is the final oversizing factor $F_{\text{col,oversize}}$:

$$F_{\text{col,oversize}} = \max_{1 \leq i \leq N} \{F_i\}. \tag{5.6}$$

6 Routing Network Design

Once the appropriate operators are arranged in a 2D array, routing resources need to be added. As much as the flexibility of the array depends on the choice of the operator types and their distribution, it also depends on the characteristics of the routing network, which should reflect the characteristics of a target application domain.

Typically, CGRA routing networks are regular and sparse, composed of short connections between neighboring nodes only, as shown in Figure 6.1a. These connections are buses composed of multiple one-bit routing tracks, because the operators perform word-level manipulations. Although area efficient, such networks have several drawbacks:

- They are of limited flexibility.
- They are not tuned to the requirements of an application domain.
- For long connections, the operators have to be used to route data, and thus can not be used for their basic functionality.

The most flexible routing networks can be found in FPGAs, because they are designed with the goal to support variety of applications. An FPGA consists of a large number of small configurable logic blocks (CLBs) arranged in a 2D array and a programmable routing network composed of a number of independent one-bit tracks to enable CLB

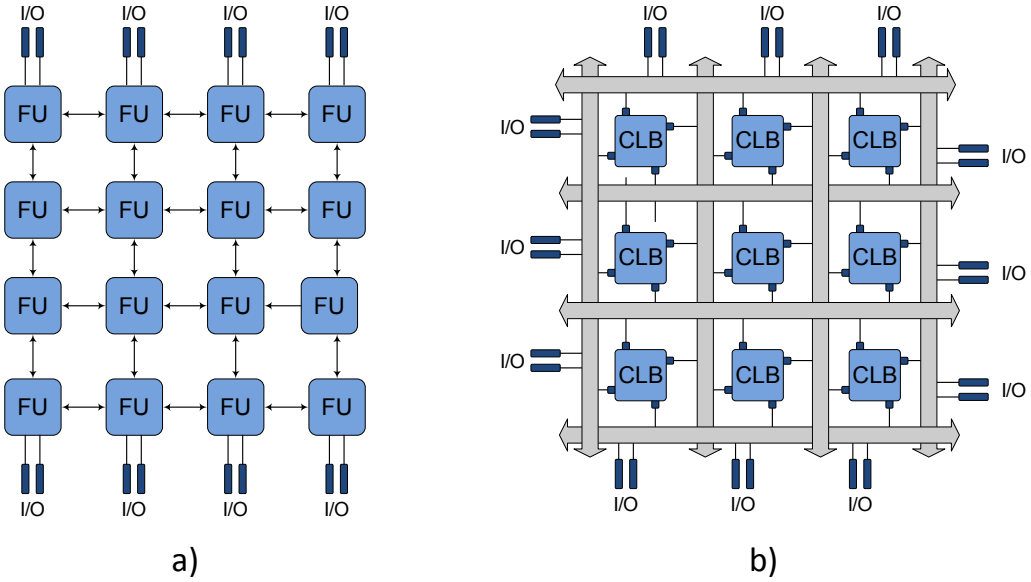


Figure 6.1: (a) A typical CGRA routing network architecture. (b) An island-style FPGA: 2D array of configurable logic blocks with routing channels between rows and columns and input/output ports on the sides.

inputs and outputs to connect and form a complex circuit. To achieve high flexibility, these routing networks are very segmented and thus many routing switches are provided to support connecting different routing segments. Therefore, the most of FPGA die-area is consumed by routing resources [FK08]. Additionally, the delay of a circuit mapped onto an FPGA is due mostly to the delay induced by the transistors in the routing switches, rather than the CLBs that perform computation.

A state-of-the-art routing architecture is the *island-style* FPGA routing architecture, employed by all major FPGA vendors. The island-style FPGA architecture uses a symmetric structure in which CLBs are laid out in an array of *islands*. A simplified view of such a routing architecture is shown in Figure 6.1b. The CLBs are surrounded by routing channels composed of multiple routing tracks. At the periphery of the array are input and/or output ports for connecting CLBs to external devices.

If this kind of routing architecture is to be implemented on a CGRA, the die-area consumed by routing resources would certainly be higher than that of a non-flexible routing network found in typical CGRAs. However, if looked into closely, this routing architec-

ture offers significant opportunities for designing a domain-specific CGRA:

- It offers high flexibility that is crucial for achieving hardware generality beyond the ability to support the input set of applications only.
- Although routing network segmentation and switch insertion causes considerable signal-propagation delay, in CGRAs this delay is not as dominant as in FPGAs, because FUs in CGRAs exhibit much higher delay than CLBs in FPGAs.
- Configuration overhead for routing switches in CGRAs is significantly smaller than in FPGAs, because routing tracks in CGRAs are essentially 16-bit or 32-bit buses and not one-bit wires found in FPGAs. Hence, only one configuration bit per a wide multiplexer in routing switches suffices for CGRAs, whereas FPGAs need one configuration bit per each single wire entering a routing switch.
- By varying the parameters of the routing network, which are introduced and discussed in more details in the following section, it is shown that it is possible to tune the routing network performance to fit the requirements of an application domain and save some of the die-area consumed by it. However, this is not possible with fixed and short interconnections found in typical CGRAs.

For the listed reasons, this type of routing network architecture is selected for further analysis and incorporation into an array of operators described in Chapter 5, and thus complete the design of a domain-specific CGRA. Without loss of generality, it is initially assumed that each routing segment spans only one functional unit and that all vertical and horizontal routing channels contain the same number of 32-bit buses. Hence, the main challenge is to find a minimum number of buses per channel that will provide successful mapping of all application DFGs known at the design time and guarantee high generality of the array.

The organization of this Chapter is as follows. In Section 6.1 an island-style FPGA architecture and its routing network are described in details. Then, Section 6.2 explains the

algorithm to devise a minimum number of buses per channel that provides successful mapping of all input DFGs and guarantees high generality. To place and route DFGs, an existing open-source tool called Versatile Place and Route (VPR) is used. This tool was developed at the University of Toronto [BR00] and has been widely adopted by researchers in the domain of FPGA architectures and reconfigurable hardware architectures in general. Section 6.3 explains in details the process of placing a DFG using VPR, including the format of netlist description file (Subsection 6.3.1) and the format of reconfigurable datapath architecture (Subsection 6.3.2). Section 6.4 explains the format of circuit placement description file (Subsection 6.3.3), which is used by VPR algorithms for DFG routing. Finally, a way to increase the routing channel width beyond the minimum is the topic of Section 6.5.

6.1 Island-Style FPGA Architecture

Island-style FPGA architecture is the most common FPGA architecture. It is comprised of an array of configurable logic blocks (CLBs). On the sides of the array are input and/or output ports for connecting FPGA internal signals with the external devices. There are typically two I/O ports per row or column of the array. Between I/O ports and CLBs, as well as between every consecutive pair of CLB rows or columns, are routing channels. These channels are composed of a set of wiring tracks, that can be of different length. A detailed drawing of an island-style FPGA is shown in Figure 6.2.

Routing channels are defined by the channel width W, which equals the number of routing tracks inside a channel. Usually, W is constant throughout the CLB array. However, in the case of directionally-biased FPGAs [BR96], vertical channels have more routing tracks than horizontal channels. Additionally, in the case of non-uniform FPGAs [BR96], the center region has higher channel width than outer regions. Other layouts also exist, depending on where the manufacturer expects the greatest routing congestion. The parameter specifying how many CLBs the track is spanning is called

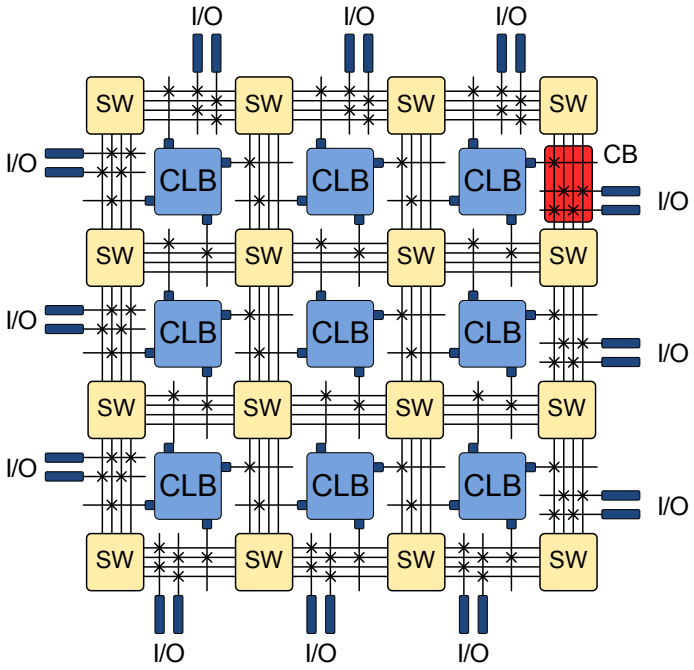


Figure 6.2: An island-style FPGA architecture shown in details. The FPGA is composed of a 2D array of configurable logic blocks with routing channels between consecutive rows and columns and input/output ports on the sides. Inputs and outputs of logic blocks are connected with the neighboring routing channels via connection blocks (CBs). Routing wires in different channels are connected via switch boxes (SBs).

the segment length L. In commercial FPGAs the routing channels most often contain tracks of different segment length, because that may improve the routing area and the critical path delay. Connection boxes (CBs in Figure 6.2) connect the routing tracks inside channels with the input and output pins of the CLBs. These pins can be on any side of CLB (up/down/left/right). A connection block is described using two parameters. The input flexibility F_{cin}, indicates how many tracks per channel can be connected to each CLB input. The output flexibility F_{cout} indicates how many tracks per channel are connected to each CLB output.

Figure 6.3a shows a routing channel of the width $W = 3$, which contains tracks of the length $L = 1$, $L = 2$, and $L = 3$. Figure 6.3b shows three input connection boxes, at the top, the left, and the right side of the logic block, respectively. Their flexibility equals $F_{\text{cin}} = 0.25$ because only one out of four ($W = 4$) wires from the routing channels can be

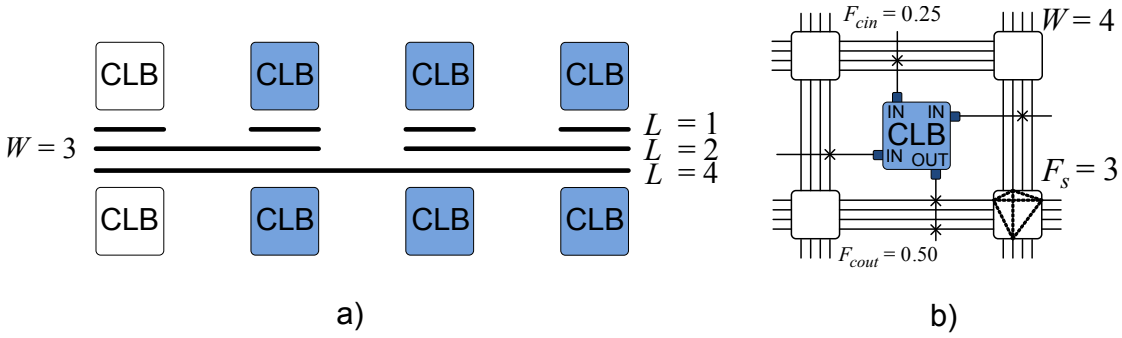


Figure 6.3: A routing channel can contain tracks of different lengths L. The number of tracks per channel equals the channel capacity W. The output connection block flexibility F_{cout} indicates how many tracks per channel are connected to each CLB output. The switch block flexibility F_s is the number of possible connections a wire segment can make to other wire segments inside a switch block.

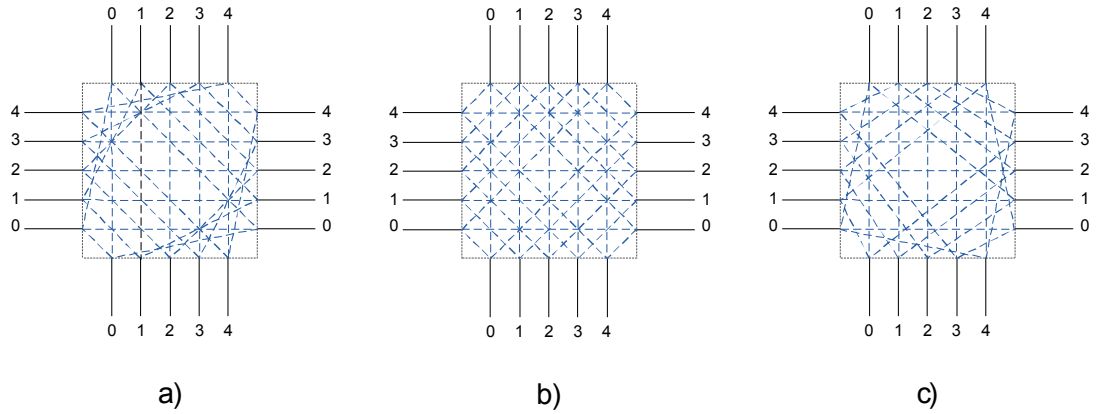


Figure 6.4: FPGA switch block topologies. (a) Disjoint switch block[LB93]. (b) Universal switch block [CWW96]. (c) Wilton switch block [Wil97].

connected to any of the input ports of the CLB. Similarly, there is one output connection block at the bottom side of the CLB, with the flexibility $F_{\text{cin}} = 0.50$.

A Switch Block (SB) is used to make connections between tracks in adjacent routing channels. Its flexibility, F_s, defines for a wiring segment entering the switch block the number of other wiring segments it can be connected to [LKJ+09]. Hence, the switch block in Figure 6.3b has $F_s = 3$. The most used topologies of switch blocks are the following: the Disjoint Switch Block [LB93], the Universal Switch Block [CWW96], and the Wilton Switch Block [Wil97]. They are shown in Figure 6.4. In the Disjoint block, when a routing wire is implemented using track i, $1 \leq i \leq W$, all segments that

constitute that wire are restricted to the track i. Hence, all tracks are partitioned into W subsets, and this reduces the overall routability, compared to other switch blocks. In the Universal switch block, the focus is on maximizing the number of simultaneous connections that can be made. The Wilton switch block is very similar to the Disjoint block, except that each diagonal connection is rotated by one track. This results in increased number of routing choices for each connection. The Wilton block is the most used one by the commercial FPGAs.

The choice of a switch block is the key to the overall flexibility of an FPGA architecture. Besides, it affects the design speed, since the transistors in SBs add capacitance and resistance loading to each routing track. Nevertheless, since a large portion of an FPGA is devoted to the routing resources, the die area required by each switch block determines the overall logic density of the device.

6.2 Method for Determining the Channel Width

The algorithm for calculating the minimum channel width needed to support mapping a set of DFGs (Algorithm 4) starts by creating an array of operators following the methods described in the previous Chapters. Then, for every DFGs D_i in a group G, a netlist, placement, and architecture description files are created and passed to VPR tool. VPR performs circuit placement and routing, and reports the minimum channel width W_i needed for successful routing. The final array channel width W is the maximum of all reported W_i values, as it is the value that ensures successful routing of all input DFGs.

6.3 DFG Placement Using VPR

To place and route a DFG onto a reconfigurable array, an open source tool called VPR (Versatile Place and Route) is used. This tool has been developed at the University of Toronto [BR00] and has been used extensively by researchers in reconfigurable hardware

Algorithm 4: An algorithm to estimate the minimum channel width W.

```
/* Creating an array of operators from all DFGs in the set G,      */
/* following the procedures described so far.                      */
```

arrayInit = createArray(G);

```
/* Placing and routing the DFGs D_i (1 ≤ i ≤ N) on the arrayInit   */
/* and reporting the minimum channel width necessary to            */
/* successfully route every single one, individually.              */
```

for $1 \leq i \leq N$ **do**
    array[i] = map(D_i, arrayInit);
    netlist = createNetlistFile(D_i, array[i]);
    placement = createPlacementFile(array[i]);
    architecture = createArchitectureFile(array[i]);
    createEmptyRoutingFile(array[i]);
    VPRoutput = runVPR(netlist, architecture, placement, routing);
    channelWidth[i] = minChannelWidth(VPRoutput);

```
/* Result.                                                         */
```

W = MAX$_{1 \leq i \leq N}$(channelWidth[i]);

architectures. Inputs to VPR consist of a technology mapped netlist and a text file describing the target architecture. VPR can place a circuit, or a pre-existing placement can be read in. Then, VPR can perform either a global routing or a combined global and detailed routing of the placed benchmark circuit. Global routing balances the densities of all routing channels, while detailed routing assigns specific wiring segments for each connection [BFRV92, HA96]. The place and route results consist of node coordinates, as well as statistics, such as routed wire-length, track count, maximum net length, and area consumed by routing resources and others. This statistics is crucial for evaluating the utility of the architecture under test.

The tool is invoked by typing:

```
> vpr netlist.net architecture.xml placement.p routing.r [-options],
```

where:

- The `netlist.net` is the netlist file describing a DFG.
- The `architecture.xml` file describes the architecture of an FPGA-like hardware on which the DFG should be realized.
- If VPR is placing a circuit, it writes out the node coordinates into the `placement.p` file. If VPR is routing a previously placed circuit, it reads in the node coordinates from the `placement.p` file.
- The routing result is written into the file `routing.r`.
- VPR has a lot of optional command line options (`-options`). For example, one can enable/disable graphical outputs (`-nodisp`), timing analysis (`timing_analysis {on|off}`), printing some extra statistics (`-full_stats`), or he can choose whether VPR should place or not the input DFG (`-route_only`).

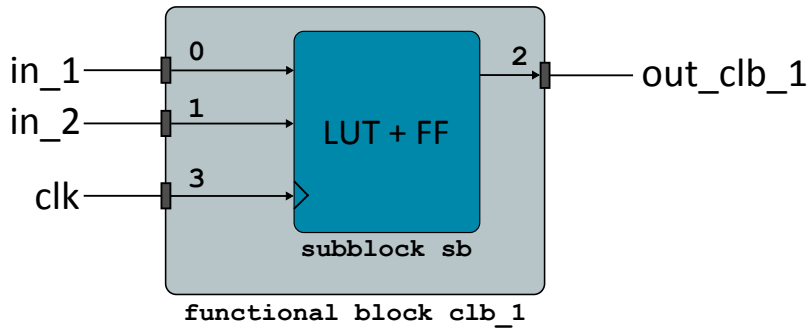
6.3.1 Circuit Netlist (`.net`) Format

VPR allows for three circuit elements: input pads, output pads, and functional blocks (FBs or CLBs). Input and output pads are specified using the element type keywords `.input` and `.output`, respectively. Functional blocks are specified by `.[name]`. The `name` is a user defined type of the FB, which must correspond with the type specified in the architecture file (Section 6.3.2). For example, a functional block of the type `.clb` in the netlist should be specified as the `.clb` in the `architecture.xml` file as well.

The format of describing a functional block in the `netlist.net` is the following:

```
element_type_keyword block_name
    pinlist:  net_a net_b net_c ...
    subblock:  subblock_name pin_num1 pin_num2 ...
```

After defining the type and the name of the functional block, a list of nets connected to each pin of the functional block is given in the `pinlist`. The first listed net connects to the pin 0 of a functional block, and so on. If any FB pin is to be left unconnected,



a)

```
.clb clb_1
   pinlist: in_1 in_2 out_clb_1 clk
   subblock: sb 0 1 2 3
```

b)

Figure 6.5: (a) The content of a functional block and (b) its description in the netlist file.

the corresponding entry in the pinlist should be the reserved word `open`. The content of the functional block is described within the `subblock` lines. Each functional block must have at least one subblock line, and can have up to $max\_subblocks$ (a user defined attribute) subblocks per block. A subblock is a K-input O-output boolean logic element (LE) and a flip flop, as shown in Figure 6.5. The parameter K is set via the $max\_subblock\_inputs$ attribute, while O is set via the $max\_subblock\_outputs$ attribute in the `architecture.xml` file. Each `subblock` line should contain the name of the subblock and the FB pins, or the subblock output pins, to which LE pins are connected. If an LE pin is unconnected, the corresponding pin entry is marked as `open`. The order in which the LE pins are specified is the following: first the input pins, then the output pins, and the last the clock input. If the FB is a combinatorial circuit, the clock input should be left open.

Input and output pads have only one pin. The name of a net connected to the pad is given after the `pinlist` keyword. The format of describing pads is the following:

```
    .input pad_name_a
       pinlist:  signal_name_a

    .output pad_name_b
       pinlist:  signal_name_b
```

The following lines give a complete netlist description of the 3 × 3 convolution DFG shown in Figure 5.4.

```
# INPUT ports
    .input in_1
       pinlist:  in_1
    .input in_2
       pinlist:  in_2
    .input in_3
       pinlist:  in_3
    .input in_4
       pinlist:  in_4
    .input in_5
       pinlist:  in_5
    .input in_6
       pinlist:  in_6
    .input in_7
       pinlist:  in_7
# OUTPUT ports
    .output out_14
       pinlist:  out_clb_13
# CONFIGURABLE LOGIC BLOCKS
    .clb clb_8
       pinlist:  in_2 in_3 out_clb_8 open
```

```
    subblock:  sb 0 1 2 open
.clb clb_9
    pinlist:  in_2 in_3 out_clb_9 open
    subblock:  sb 0 1 2 open
.clb clb_10
    pinlist:  in_2 in_3 out_clb_10 open
    subblock:  sb 0 1 2 open
.clb clb_11
    pinlist:  in_2 in_3 out_clb_11 open
    subblock:  sb 0 1 2 open
.clb clb_12
    pinlist:  in_2 in_3 out_clb_12 open
    subblock:  sb 0 1 2 open
.clb clb_13
    pinlist:  in_2 in_3 out_clb_13 open
    subblock:  sb 0 1 2 open
```

6.3.2 Reconfigurable Datapath Architecture (`.xml`) Format

The area and performance of a reconfigurable array depend on the type and implementation of its architecture. VPR can output area and delay measurements to evaluate the datapath performance, provided that the area and delay of all basic components are specified. To simplify the use of VPR, a collection of compatible architecture files [KR08a, KR08b] containing accurate area and delay measurements is provided at The intelligent FPGA Architecture Repository (iFAR) website www.eecg.toronto.edu/vpr/architectures/. These island-style architectures differ in logic block parameters, such as LUT size, and routing parameters, such as segment lengths. Additionally, careful transistor sizing of each architecture is performed, for different technologies ranging from 22 nm to 180 nm

CMOS. The architecture description that fits well the expected architecture of the reconfigurable CGRA is given in the *N10K04L01.FC20FO10.AREA1DELAY1.CMOS65NM.BPTM* architecture file. This file contains all the area and delay parameters of buffers, multiplexers, and wire segments, estimated for 65 nm CMOS technology. It is the architecture description with the highest available number of tracks per channel, such that the track length equals one. This is important, because it implies that one switch block is instantiated in every crossing between a horizontal and a vertical routing channel. All attribute values that will be mentioned in this Chapter are adopted from this specification.

Since the architecture is specified in an `.xml` file, the description is composed of a hierarchy of start and end tags with optional attributes and content inside each tag providing additional information. The following lines show the opening and closure of the outermost (`<architecture>`) tag.

```
<!- VPR Architecture Specification File ->
<architecture>
   #Architecture description
</architecture>
```

This tag contains five other tags: `<layout>`, `<device>`, `<switchlist>`, `<segmentlist>`, and `<typelist>`.

The `<layout>` tag specifies the size and shape of the 2-D array. The size can either be explicitly given as the size in the x-direction (width) followed by the size in the y-direction (height). Here is an example of a 16 × 7 array:

```
<layout width="16" height="7"/>
```

Otherwise, the size can be chosen automatically to be the minimal that fits the given circuit, using the keyword `auto`. The aspect ratio of the array is given after the auto keyword and is the ratio of width and height. Here is an example of an array having equal number of columns and rows:

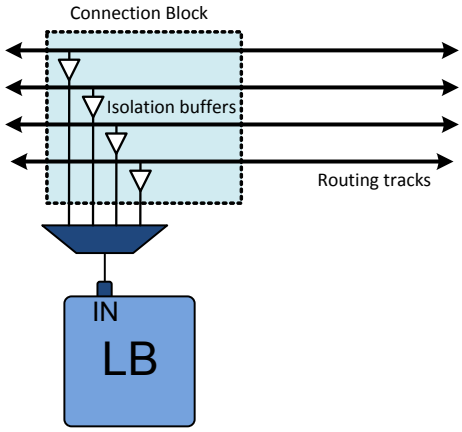


Figure 6.6: The connection between routing channels and logic block input pins.

```
<layout auto="1.0"/>
```

The content inside `<device>` tag specifies device information and contains the tags `<sizing>`, `<timing>`, `<area>`, `<chan_width_distr>`, and `<switch_block>`. The following example illustrates a device having the same number of tracks in horizontal and vertical routing channels, which are connected using Wilton switch blocks. When bidirectional segments are used, each wire segment can connect to three other wire segments in the switch block (`fs="3"`). Parameters `R_minW_nmos` (the resistance of minimum-width NMOS transistor), `R_minW_pmos` (the resistance of minimum-width PMOS transistor), and `ipin_mux_trans_size` (the size of transistors in multiplexers at the logic block inputs, given in the minimum transistor units) specify parameters used by the area model built into VPR. The multiplexers at the inputs of logic blocks, shown in Figure 6.6, are implemented as two-level multiplexers.

```
<device>
    <sizing R_minW_nmos="4502.470215"
            R_minW_pmos="12028.500000"
            ipin_mux_trans_size="1.203190"/>
    <timing C_ipin_cblock="0.000000e+00"
            T_ipin_cblock="6.753000e-11"/>
    <area grid_logic_tile_area="7206.319824"/>
```

```
    <chan_width_distr>
       <io width="1.000000"/>
       <x distr="uniform" peak="1.000000"/>
       <y distr="uniform" peak="1.000000"/>
    </chan_width_distr>
    <switch_block type="wilton" fs="3"/>
</device>
```

The <switchlist> tag contains a group of <switch> tags, which specify the types of switches and their properties. The following example defines a switch of a multiplexer type and specifies the delay through the switch (Tdel), the output (Cout) and the input capacitance of the switch (Cin), its resistance (R), and a unique alphanumeric string that matches the segment definition (name). The attributes buf_size and mux_trans_size are needed for directional switches, and specify the area of the buffer in minimum-width transistor area units and the size of each transistor in the mux in the switch, respectively. The structure of a directional switch used inside Wilton switch block is shown in Figure 6.7.

```
<switchlist>
    <switch type="mux"
            buf_size="11.698900" mux_trans_size="1.989870"
            Tdel="5.344000e-11" Cout="0.000000e+00"
            Cin="0.000000e+00" R="0.000000" name="0"/>
</switchlist><
```

The <segmentlist> tag encompasses a group of tags that define the types of wire segments inside routing channels and their properties. The following example describes unidirectional segments of length one (spanning only one logical block). Since the most of the delay in the routing network is due to the delay in switch blocks, the wire capacitance and resistance per unit length are here neglected and set to zero. The type attributes describe the depopulation of a switch block (sb type) and connection

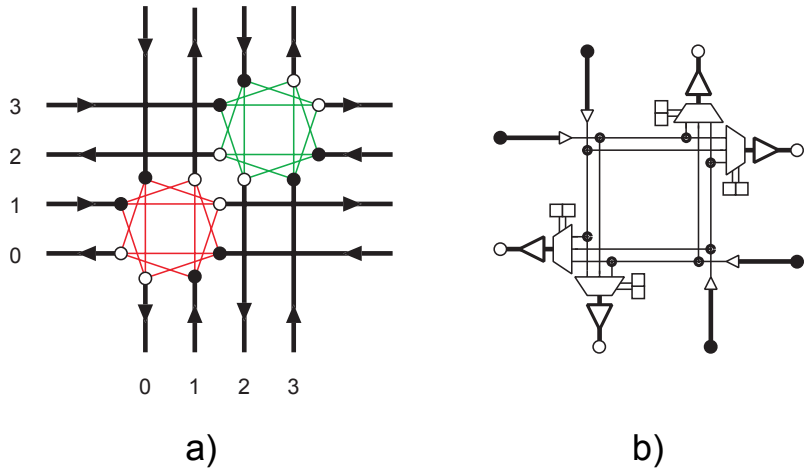


Figure 6.7: Directional switch block [GEMA04]. (a) The Wilton switch block pattern for a channel having the width $W = 4$. Tracks 0 and 2 pass data towards left and down, while tracks 1 and 3 pass data towards right and up. (b) The realization of the directional switch between tracks 0 and 1 and between tracks 2 and 3.

block (`cb type`) respectively. In this example, since the wire length equals one, there is one switch block on each end of the wire, and thus the switch block depopulation pattern equals `1 1`. Similarly, there is only one connection block per a unit-length wire, and thus the connection block depopulation pattern equals `1`.

```
<segmentlist>
    <segment length="1" type="unidir"
             Rmetal="0.000000" Cmetal="0.000000e+00">
        <mux name="0"/>
        <sb type="pattern">1 1</sb>
        <cb type="pattern">1</cb>
    </segment>
</segmentlist>
```

The characteristics of input/output ports and functional blocks are defined within the `<typelist>` tag. In the following example, `io_capacity` is set to two, meaning that there are two input and output ports per row and column of the array and some estimates on the input and output pad delay are given. Additionally, it is set that an input

pin of functional blocks can be connected to no more than four wires in the routing channel bordering the pin (`fc_in_type`), while an output pin can be connected to any of the wires in the channels bordering the pin (`fc_out_type = "full"`).

```
<io capacity="2"
    t_inpad="5.624000e-11" t_outpad="1.729000e-11">
    <fc_in type="abs">4</fc_in>
    <fc_out type="full"/>
 </io>
```

After the definition of input and output ports, configurable logic blocks are defined. Firstly the name of a functional block is specified. This name must correspond exactly with the name for the block in the netlist. The name format is `.[name]` (e.g. `.clb`). It is possible to define the height of a functional block, in case it spans more than one tile. However, that is not the case for the configurable array that is the topic of this work. Tags `fc_in` and `fc_out` set the number of tracks to which each logic block input pin connects in each channel bordering the pin. The value used is always the minimum between the specified value and the channel width. The `type` attribute indicates whether the specified value should be interpreted as the absolute number of tracks to which each pin connects (`abs`), or as the fraction of tracks in a channel to which each pin connects (`fractional`). Within the `<pinclasses>` tag, the pins of the functional block are specified. In this specific example, two input, one output, and one global input pin are defined. Moreover, this example shows how to restrict all logic blocks to 2-input 1-output operators, while allowing an additional input for the global clock signal. Also, the input pins are positioned on the top side, while the output pins are positioned on the bottom side of the functional block (Figure 6.6). The line `<loc type="fill" priority="1"/>` specifies that the array of logic blocks is homogeneous, composed of this particular CLB type only. Finally, VPR allows setting the values of (i) the delay from the output of the subblock to the logic block output pin (`T_sblk_opin_to_sblk_opin`), (ii) the delay from the output of a subblock to the

input of another subblock within the same clb (T_sblk_opin_to_sblk_ipin), and (iii) the delay from an input pin of a clb to an input pin of a subblock within that clb (T_fb_ipin_to_sblk_ipin). In the case when the operator delays are significantly higher than these delays, they can be set to zero.

```
<type name=".clb">
    <fc_in type="abs">4</fc_in>
    <fc_out type="full"/>
    <pinclasses>
       <class type="in">0 1</class>
       <class type="out">2</class>
       <class type="global">3</class>
    </pinclasses>
    <pinlocations>
       <loc side="top">0 1 3</loc>
       <loc side="bottom">2</loc>
    </pinlocations>
    <gridlocations>
       <loc type="fill" priority="1"/>
    </gridlocations>
    <timing>
       <tedge type="T_sblk_opin_to_sblk_ipin">0.000000e+00</tedge>
       <tedge type="T_fb_ipin_to_sblk_ipin">0.000000e+00</tedge>
       <tedge type="T_sblk_opin_to_fb_opin">0.000000e+00</tedge>
    </timing>
    <subblocks max_subblocks="1" max_subblock_inputs="2">
       <timing>
          <T_comb>
             <trow>0.000000e+00</trow>
```

```
            <trow>0.000000e+000</trow>
         </T_comb>
         <T_seq_in>
            <trow>0.000000e+00</trow>
         </T_seq_in>
         <T_seq_out>
            <trow>0.000000e+00</trow>
         </T_seq_out>
      </timing>
   </subblocks>
</type>
```

6.3.3 Circuit Placement (`.p`) Format

The first line in the placement file specifies the name of the circuit netlist file and the architecture file that are used to create this placement. These names have to match the names used in the command line invocation of the tool. The next line gives the size of the logic block array used by the placement file for domain-specific arrays:

`Array size:` N_r `x` N_c `logic blocks,`

where N_r and N_c equal the number of rows and columns in the array, respectively. The remaining lines are in the format:

```
block_name x y subblock_number,
```

where the `block_name` corresponds with the name of the same block in the netlist file. The coordinates `x` and `y` correspond to the row and column of the array in which the block is placed, respectively. The subblock number specifies which of the several possible input/output pad locations in the row `x` and the column `y` contain this pad, and it is used if there is more than one pad per row/column in the array. The logic

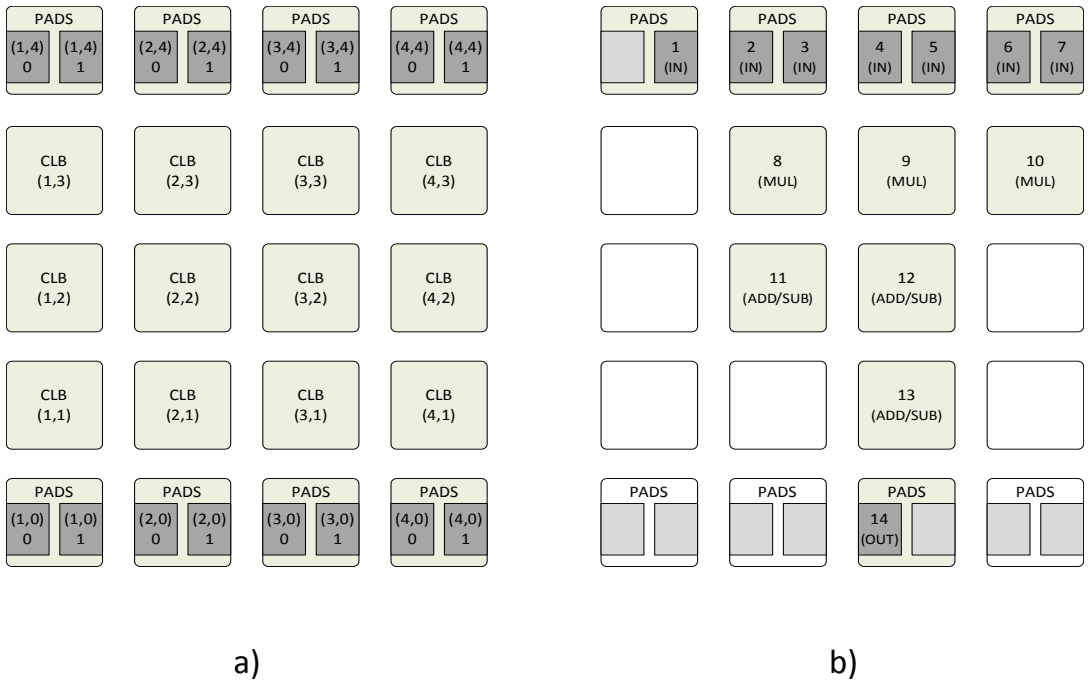


Figure 6.8: An example of the placement the 3 × 3 convolution DFG shown in Figure 5.4 that corresponds to the placement file given in the text. Since the data is transferred in a top-down approach, the arrays are designed to have input ports on the top side and output ports on the bottom side.(a) The coordinate system used by VPR. (b) The placement of nodes and ports.

bock x and y coordinates lie in the range 1..N_r and 1..N_c respectively. Since the pads are placed on the sides of the array, their x (y) coordinates equal either zero or $N_r + 1$ ($N_c + 1$). Since the data is transferred in a top-down approach, the arrays are designed to have input ports on the top side and output ports on the bottom side.

```
# Summary:  A complete placement description of
# the 3 × 3 convolution DFG (Figure 5.4) shown placed in Figure 6.8b.
```

```
# block name x y subblk block number
# ---------- - - ------ ------------
   in_1 1 4 1 #0
   in_2 2 4 0 #1
   in_3 2 4 1 #2
```

```
in_4 3 4 0 #3
in_5 3 4 1 #4
in_6 4 4 0 #5
in_7 4 4 1 #6
out_14 3 0 0 #7
clb_8 2 3 0 #8
clb_9 3 3 0 #9
clb_10 4 3 0 #10
clb_11 2 2 0 #11
clb_12 3 2 0 #12
clb_13 3 1 0 #13
```

6.4 DFG Routing Using VPR

VPR can be run in one of two basic modes. In its default mode, VPR places a circuit and then repeatedly attempts to route it in order to find the minimum number of tracks required by the specified architecture to successfully route this circuit. If routing is unsuccessful, VPR increases the number of tracks in each routing channel and attepts routing again; if routing is successful, VPR decreases the number of tracks before trying to route it again. Once the minimum number of tracks required to route the circuit is found, VPR exits. The second mode of VPR is invoked when a user specifies a fixed channel width for routing. In that case, VPR places a circuit and attempts to route it only once, with the specified channel width. If the circuit will not route at the specified channel width, VPR reports that it is unroutable. To specify the channel width the attribute `-route_chan_width` is used.

Routing is an NP complete problem [GV04] that is generally separated in two phases using the divide and conquer paradigm [AD04]:

- a global routing that balances the densities of all routing channels, and
- a detailed routing [BFRV92, HA96] that assigns specific wiring segments for each connection.

VPR can perform either global or combined global and detailed routing. This is set using the attribute

`-route_type {global|detailed}`.

The global router performs a coarse route to determine, for each connection, the minimum distance path through routing channels that it has to go through. If the net to be routed has more than two terminals the global router will break the net into a set of two-terminal connections and route each set independently. The global router considers for each connection multiple ways of routing it and chooses the one that passes through the least congested routing channels. By keeping track of the usage of each routing channel, congestion is avoided; and the principal objective of the global router, balancing the usage of the routing channels, is achieved. Once all connections have been coarse routed, the solution is optimized by ripping up and rerouting each connection a small number of times. After that, the final solution is passed to the detailed router.

The detailed router determines for each two point connection the specific wiring segments to use in the routing channel assigned by the global router. To do this, detailed routing algorithms construct a directed graph from the routing resources to represent the available connection between wires, connection blocks, switch blocks and logic blocks. The search performed on this directed graph is usually based on Dijkstra's algorithm to find the shortest path between two nodes. The paths are labeled according to a cost function that takes into account the usage of each wire segment and the distance of the interconnecting points. The distance is estimated by calculating the wire length in the bounding box of the interconnecting points using a Manhattan metric. Most of the routers relax the bounding box constraints and allow searching for possible solutions in the surrounding routing channels of the bounding box. This is done to avoid subsequent

iterations of ripping out and re-routing if the solution lies on the near outside of the bounding box. The selection of the routing algorithm used by VPR is done using the following attribute:

`-router_algorithm {breadth_first|timing_driven|directed_search}`.

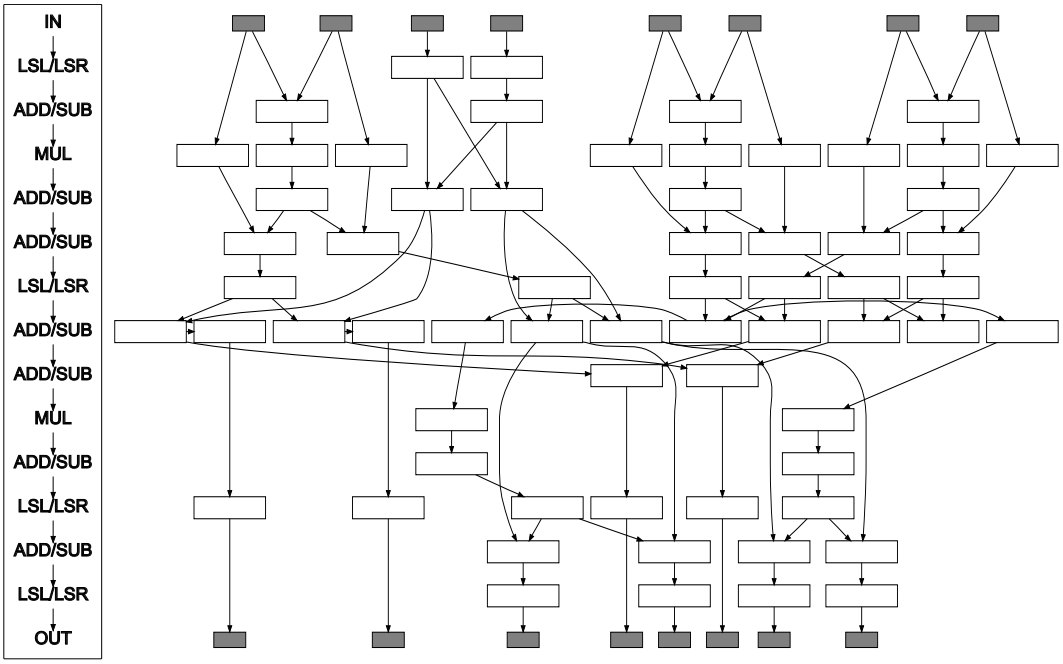
The default routing algorithm used by VPR is the timing-driven one, which focuses on both achieving a successful route and achieving good circuit speed. The breadth-first router focuses solely on routing a design successfully, and it is is capable of routing a design using slightly fewer tracks than the timing-driving router. The directed-search router is routability-driven and uses an A* heuristic to improve runtime over breadth-first.

Since the most important feature of a domain-specific array is its generality, it is crucial to achieve high probability of successful routing of various circuits. Additionally, since each routing track is a complete 32-bit bus, it is desirable to minimize the channel width. For these reasons, breadth-first routing algorithm was selected.

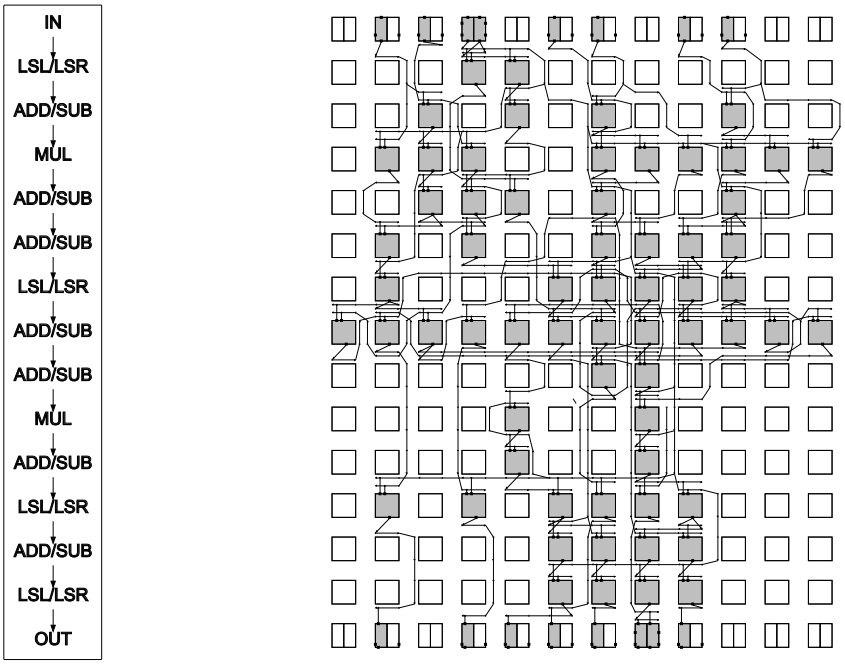
6.5 Oversizing The Routing Channels

Similarly to the column oversizing described in Chapter 5, designer has the possibility to define a channel-width oversizing factor to increase the routability of the final datapath beyond the minimum. It suffices to specify the channel width higher than the minimum needed. Clearly, the cost is in additional die-area consumed by the routing resources.

An example of the circuit routed by VPR is shown in Figure 6.9. First, the basic block extracted from 32-bit Inverse Two-dimensional DCT (Table 7.1) is laid out using `dot` and appropriate constraints and parameters to suggest a detailed placement on the array. Then, this placement information is used to run VPR routing algorithm, the result of which is shown in Figure 6.9b.



(a)



(b)

Figure 6.9: (a) The basic block extracted from 32b Inverse Two-dimensional DCT (Table 7.1) is laid out using `dot` and appropriate constraints and parameters to suggest a detailed placement on the array. (b) The DFG placed and routed on a reconfigurable array using VPR.

7 Experimental Evaluation

This chapter focuses on assessing the performance of the novel method to design domain-specific reconfigurable arrays. At first, the two algorithms for designing an area efficient supersequence, described in Sections 4.1 and 4.2, are compared. Then, a generality of domain-specific arrays is defined and measured, and an insight into the drawbacks of the current implementation is provided. Additionally, the effects of grouping various domains on generality and total area of the arrays are analyzed. Finally, the areas of domain-specific arrays and critical path delays of applications when placed and routed onto them are compared with respect to ASIC, FPGA, and a well known datapath merging technique by Brisk et al. [BKS04].

Although the experimental evaluation of energy consumption of the domain-specific arrays is not performed, it is clear that the methodology itself inherently promises good performance and low energy consumption due to two factors. Firstly, at the level of the computational units—they are implemented using standard cells and thus they are as high performance and power efficient as they can be in a semi-custom design flow, and certainly significantly better than in FPGAs. Secondly, at the level of interconnections among units—the methodology strives to achieve short and regular routing, therefore helping shortening the critical path and keeping power consumption to a minimum, while using minimal resources.

Table 7.1: Data-flow graphs covering classical signal and image processing computations [TI03a, TI03b, TI10, EEM06, Exp].

DFG	Name	Description
D_1	DSPLIB_C64_autocor_UNROLL12_BB_2	Autocorrelation
D_2	DSPLIB_C67_dotp_cplx_UNROLL3_BB_1	Complex Dot Product
D_3	IMGLIB_C64_corr3x3_UNROLL3_BB_2	3x3 Correlation for 8b Data
D_4	IMGLIB_C64_sobel3x3_UNROLL2_BB_1	16b Sobel 3x3
D_5	DSPLIB_C64_fir_cplx_UNROLL4_BB_2	16b Complex FIR
D_6	DSPLIB_C64_fir_lms_UNROLL16_BB_1	16b LMS Adaptive Filter
D_7	DSPLIB_C64_fir_sym_UNROLL8_BB_2	16b Symmetric FIR Filter
D_8	DSPLIB_C64_iir_UNROLL4_BB_1	IIR Filter
D_9	DSPLIB_C64_fftr4_UNROLL2_BB_3	16x16 Radix 4 DIF FFT
D_{10}	DSPLIB_C67_fftmix_BB_2	Forward FFT with Mixed Radix
D_{11}	DSPLIB_C67_fftr2_UNROLL4_BB_3	Forward FFT with Radix 2
D_{12}	EEMBC_dct_BB_1	DCT h264 Encoder Library
D_{13}	EEMBC_idct_BB_1	IDCT h264 Encoder Library
D_{14}	ExPRESS_mpeg_idct, BB_1	32b Inverse 2-D DCT
D_{15}	ExPRESS_mpeg_idct, BB_4	32b Inverse 2-D DCT
D_{16}	IMGLIB_C64_dct_BB_2	8x8 Block FDCT with Rounding
D_{17}	IMGLIB_C64_dct_BB_8	8x8 Block FDCT with Rounding
D_{18}	IMGLIB_C64_idct_BB_2	IDCT on 8x8 DCT Coef. Blocks
D_{19}	IMGLIB_C64_idct_BB_8	IDCT on 8x8 DCT Coef. Blocks

7.1 Experimental Setup

To estimate the performance of domain-specific arrays, nineteen different DFGs from applications available in benchmarks and commercial libraries, such as TMS320C64x DSP Library [TI03a], TMS320C64x Image/Video Processing Library [TI03b], TMS320C67x DSP Library [TI10], MPEG-2 Decode EEMBC benchmark [EEM06], and ExpressDFG Instruction Scheduling Benchmarks [Exp] have been selected. These DFGs cover various classic signal and image processing computations, such as FFT, DCT, IDCT, FIR, IIR, and autocorrelation. To increase the number of nodes in DFGs and thus the overall speedup, loop unrolling using different unrolling factors is applied. Loop unrolling is a compiler optimization technique applied to application kernels, which are usually loops, to reduce the frequency of branches and loop maintenance instructions. Loop unrolling replicates the code inside the loop body a number of times, where this number

Table 7.2: Loop unrolling factors and total number of DFG nodes.

DFG	Loop Unrolling factor	Number of nodes
D_1	12	24
D_2	3	24
D_3	3	18
D_4	2	28
D_5	4	32
D_6	16	40
D_7	8	24
D_8	4	22
D_9	2	80
D_{10}	1	40
D_{11}	4	40
D_{12}	1	42
D_{13}	1	42
D_{14}	1	55
D_{15}	1	64
D_{16}	1	60
D_{17}	1	61
D_{18}	1	59
D_{19}	1	77

of copies is called the loop unrolling factor. Hence, the total number of loop iterations becomes the initial number of iterations divided by the loop unrolling factor. To achieve the best array utilization, DFGs are simultaneously unrolled until all of them require a similar number of columns in the array for high area operators, such as multipliers. In other words, they are unrolled until the maximum number of multipliers per depth, defined in Section 5.3.2, for every DFG is similar. Table 7.1 lists all 19 DFGs along with their descriptions, while Table 7.2 gives the loop unrolling factors and the final number of DFG nodes.

Further on, all DFGs are divided into various groups. G_{1A}, G_{1B}, G_{1C}, and G_{1D} include DFGs belonging to similar computational domains. Group G_{1A} contains all correlations, G_{1B} FIR and IIR filters, G_{1C} all FFTs, and G_{1D} all DCTs/IDCTs. Groups G_{2A} to G_{2F} comprise all combinations of any two domains, while groups G_{3A} to G_{3D} comprise all

Table 7.3: DFGs distributed in groups of different size. Group G_{1A} contains all correlations, G_{1B} FIR and IIR filters, G_{1C} all FFTs, and G_{1D} all DCTs/IDCTs. Groups G_{2A} to G_{2F} comprise all combinations of any two domains, while groups G_{3A} to G_{3D} comprise all combinations of any three domains. Finally, G_{4A} comprises all four domains.

Group name	Group size [DFG]	Group contents
G_{1A}	4	$D_1 - D_4$
G_{1B}	4	$D_5 - D_8$
G_{1C}	3	$D_9 - D_{11}$
G_{1D}	8	$D_{12} - D_{19}$
G_{2A}	8	$D_1 - D_8$
G_{2B}	7	$D_1 - D_4, D_9 - D_{11}$
G_{2C}	12	$D_1 - D_4, D_{12} - D_{19}$
G_{2D}	7	$D_5 - D_8, D_9 - D_{11}$
G_{2E}	12	$D_5 - D_8, D_{12} - D_{19}$
G_{2F}	11	$D_9 - D_{19}$
G_{3A}	11	$D_1 - D_{11}$
G_{3B}	16	$D_1 - D_8, D_{12} - D_{19}$
G_{3C}	15	$D_1 - D_4, D_9 - D_{19}$
G_{3D}	15	$D_5 - D_{19}$
G_{4A}	19	$D_1 - D_{19}$

combinations of any three domains. Finally, group G_{4A} comprises all four domains, and thus, all the DFGs. Table 7.3 shows the distribution of DFGs D_1 to D_{19}, described in Table 7.1, among the groups.

7.2 Comparison of Path Fusion Algorithms

To compare the two algorithms for finding the area-efficient supersequences, described in Sections 4.1 and 4.2, they are run on all sets of DFGs in Table 7.3. Then, the ratios between the area of the supersequence generated using the algorithm based on reusing the MACSeq and the area of the supersequence generated using the modified weighted majority merge algorithm are measured. To estimate the area of operators, every one of them is synthesized, placed, and routed using a 65nm standard cell library. The total area of the supersequence is the sum of areas of its nodes (operators). The ratios between the area of the supersequence generated using the algorithm based on reusing

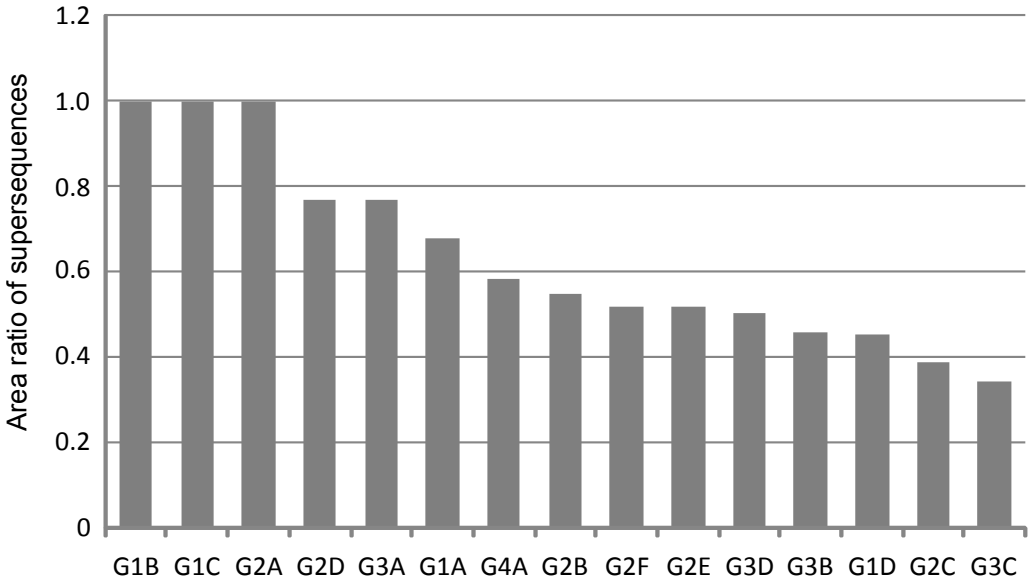


Figure 7.1: The ratio between the area of the supersequence generated using the algorithm based on reusing the MACSeq and the area of the supersequence generated using the modified WMM algorithm. The former algorithm achieves superior results, and is thus used in the rest of the experimental evaluation of the methodology.

the MACSeq and the area of the supersequence generated using the modified WMM algorithm are sorted in descending order and shown in Figure 7.1.

Clearly, the algorithm based on reusing the MACSeq metric is superior, since the created supersequences are at least as area-efficient as those generated by the modified WMM algorithm, for all groups. This may be attributed to the fact that MACSeq heuristic is most directly related to the goal of minimizing area because it performs a greedy selection based on the actual richest mergeable sequences, while WMM bases the same selection on the available opportunity (cumulative area of the units after the merged nodes) without really assessing whether this opportunity will translate in actual merging. Finally, that is why MACSeq heuristic is chosen for the rest of the experimental evaluation of the methodology.

To estimate the overhead in the column length compared to the longest path in dataflow graphs, the following experiment is performed. For each of the groups G_{1A} to G_{4A}, a domain-specific array using the novel methodology and the MACSeq algorithm is

Table 7.4: Supersequence length compared to the length of the longest path in a graph.

Group Name	Maximum Path Length	SuperSeq Length	Max / SuperSeq Length
G_{1A}	6	3	0.50
G_{1B}	9	7	0.78
G_{1C}	5	5	1.00
G_{1D}	12	13	1.08
G_{2A}	9	7	0.78
G_{2B}	6	5	0.83
G_{2C}	12	13	1.08
G_{2D}	9	8	0.89
G_{2E}	12	15	1.25
G_{2F}	12	13	1.08
G_{3A}	9	8	0.89
G_{3B}	12	15	1.25
G_{3C}	12	13	1.08
G_{3D}	12	15	1.25
G_{4A}	12	15	1.25

created. Then, the supersequence length is measured and compared to the length of the longest path found in DFGs belonging to the group. The results are given in Table 7.4. In 6 out of 15 groups (40% of all cases) the supersequence is actually shorter than the longest path by at least 11% up to 50%. Therefore, MACSeq algorithm finds supersequences that are not significantly longer than the longest path in the considered DFGs. Additionally, the tree height minimization procedure is the reason why the array column length can be shorter than the longest path. Namely, at the end of the tree hight minimization phase, al unused rows of the array are removed. In the remaining 9 cases, the supersequence was longer than the longest path, but only up to 25%.

7.3 Array Generality Estimation

For each of groups G_{1A} to G_{4A}, the *generality* of the created reconfigurable array can be estimated in the following way: if N is the number of DFGs in group G, each DFG $D_i \in G$ is removed in turn from G itself and an array from the remaining $N-1$ DFGs is created.

Then, the tool attempts to map D_i onto the array following the same place&route flow used for the array generation (Section 3.1), with the exception that the channel width is in this case known and fixed. The generality for group G is then defined as the ratio of the number of successfully mapped excluded DFGs in the N experiments to the total number of DFGs N:

$$Generality(G) = \frac{\text{The number of successfully mapped excluded DFGs.}}{\text{The total number of DFGs } N.}. \quad (7.1)$$

The algorithm to estimate generality is as follows.

Algorithm 5: An algorithm to estimate the array generality.

```
/* From each group G one DFG is removed, and an array is created   */
/* from the remaining DFGs.  Then, the algorithm tries to map      */
/* the removed DFG onto the new array.                             */
/* Generality increases only if mapping is feasible.               */
```

for *All groups G* **do**
    $numMappedDFGs[G] = 0$;
    for $1 \leq i \leq size(G)$ **do**
        $array[i] = createArray(G - D_i)$;
        $flag = mappedSuccessfully(D_i, array[i])$;
        if *flag* **then**
            $numMappedDFGs[G]++$;
    $generality[G] = numMappedDFGs[G]/size[G]$;

The results are shown in Table 7.5: generality is higher than 75% in most of the cases. The possible reasons a DFGs can fail mapping are the following: insufficient number of columns or ports in the array, failed routing due to insufficient channel width, or limited generality of the supersequence leading to insufficient number of rows in the array for successful mapping. The reasons that caused mapping failures in the experimental

Table 7.5: Generality for various groups of benchmarks.

Group name	Generality [%]	Generality′ [%]	Generality′′ [%]
G_{1A}	50	50	75
G_{1B}	50	50	50
G_{1C}	67	67	67
G_{1D}	75	88	75
G_{2A}	75	75	75
G_{2B}	86	86	86
G_{2C}	83	83	83
G_{2D}	71	71	71
G_{2E}	83	83	92
G_{2F}	82	82	82
G_{3A}	82	82	82
G_{3B}	88	88	94
G_{3C}	87	87	87
G_{3D}	87	93	93
G_{4A}	89	95	95

setup are discussed in the following subsections.

Limited generality of the supersequence

In each group at least one DFG has failed the top-down placement, described in Section 5.3, due to lack of rows in the datapath. What causes this is the fact that the generality of the supersequence highly depends on how well the input application dataflow graphs capture the main characteristics of the domain.

Insufficient channel width

In group G_{4A}, DFG D_6 fails to route. The channel width in the datapath has been set initially to four buses per horizontal/vertical routing channel, which is the minimum needed to successfully route the remaining DFGs in the group. As shown in Section 5.4,

it is possible for the designer to define the channel width oversizing factor, and thus increase the routability at the expense of higher area allocated for routing resources. Due to the regularity of the routing network, a minimum increase in channel width provides high increase in the probability for successful routing. After increasing the channel width for additional two buses beyond the minimum, which is conservative because VPR restricts the number of buses to an even number, DFG D_6 can be successfully routed and the generality for group G_{4A} increases from 89% to 95%.

To estimate how often constraining the channel width to a minimum needed value leads to routing failures, generality is measured also when the channel width is not limited. The column labeled Generality′ in Table 7.5 summarizes the results and shows that the generality increases in 20% of all tests.

Insufficient number of columns or ports

Since the array size is fixed fairly tightly based on the input DFGs, if the excluded DFG needs even a little more space to fit into the reconfigurable datapath, mapping may be impossible. One way to avoid this problem is to use the automatic increase in number of columns, described in Section 5.4, or to manually define the total number of columns of the final array. By increasing the number of columns beyond the minimum one gets more computational and routing resources, and thus an increased generality, at the cost of increasing the array size. Additionally, increased array size allows DFG mappings to resemble more the `dot` mapping, which is usually scaled horizontally to fit narrow arrays. The problem of insufficient number of ports is analogous, because it is assumed to have two input/output ports per column of the array. To estimate how often constraining the number of columns and ports to a minimum needed value leads to mapping failures, generality is measured also when the array size is not constrained in that way. The column labeled Generality″ in Table 7.5 summarizes the results and shows that the generality increases in 33% of all tests.

7.4 Array Dimensions and Utilization

For each group in Table 7.3, a reconfigurable array is designed following the novel methodology. Table 7.6 shows the obtained array dimensions (the number of rows N_r × the number of columns N_c) and the minimum channel width. The minimum array size is achieved for group G_{1A}, storing the DFGs belonging to the same domain: 3 rows × 15 columns. Predictably, the maximum array size is obtained for group G_{4A}, storing all domains at once: 15 rows × 24 columns. The number of columns in the array is the minimum needed to enable successful mapping of all DFGs in the domain. Therefore, when two or more domains are joined, the new array has the number of columns equal to the maximum of all values N_c found for each domain separately. For example, groups G_{1B} and G_{1C} need 15, or 24 columns respectively, for successful mapping of all their DFGs. Hence, the union of those two groups G_{2D} needs at least 24 columns in the array.

To find the array area utilization for every group, both the area of the array and the area occupied by DFGs when mapped onto it are evaluated. For that purpose, synthesis, placement, and routing of all the operations found in the DFGs is performed using the gate implementations of a 65nm standard cell library. For each operation the possibility to use either a direct output or a registered output is provided. Added pipelining registers are organized as bypassable registers placed after every functional unit, as in FPGAs. To estimate the routing area, VPR is used. Since VPR does not natively support bus-based connections, an appropriate technology configuration file along with the real number of wires is prepared. This approach conservatively overestimates the routing area because VPR assumes that each wire can be routed independently, whereas the reconfigurable array uses bus-based interconnects. The maximum and average area utilization per groups are shown in Table 7.6. Maximum area utilization ranges from 30% for the array generated for all domains at once (G_{4A}), up to 78% for the array generated for group G_{1C}, storing the DFGs belonging to the same domain. Highest values of average area utilizations are found for groups G_{1A} and G_{1C}, each storing the DFGs belonging to a

Table 7.6: Array size, channel width, and area utilizations for various benchmarks.

Group Name	Array Size $N_r \times N_c$	Channel Width	Max Area Utilization [%]	Average Area Utilization [%]	Routing Area / Array Area [%]
G_{1A}	3 × 15	4	58	51	28
G_{1B}	7 × 19	6	71	30	40
G_{1C}	5 × 24	6	78	54	44
G_{1D}	13 × 12	4	48	45	38
G_{2A}	7 × 19	6	71	29	40
G_{2B}	5 × 24	6	78	39	44
G_{2C}	13 × 15	4	39	26	38
G_{2D}	8 × 24	6	52	28	41
G_{2E}	15 × 19	6	38	19	45
G_{2F}	13 × 24	4	33	20	38
G_{3A}	8 × 24	6	52	24	41
G_{3B}	15 × 19	6	38	17	45
G_{3C}	13 × 24	4	33	18	38
G_{3D}	15 × 24	4	30	16	35
G_{4A}	15 × 24	4	30	15	35

particular domain. The more domains get mixed within a group, the more average area utilization decreases, indicating that this methodology is optimized to provide area-efficient domain-specific arrays.

7.5 Routing Network Characteristics

The routing network is comprised of wiring segments of length one, distributed among vertical and horizontal routing channels, which all have constant channel width. The ratio of the area dedicated for routing resources to the total array area is shown in Table 7.6, and it is in the range 28–45%, where higher ratios are found for the arrays having higher channel width. Interestingly, these results are significantly better than what is reported in programmable logic devices—according to the paper by Feng and

Kaptanogly from Actel Corporation [FK08] up to 90% of a Programmable Logic Device chip is occupied by the programmable interconnect, including wires, switches and configuration bits.

Since the array uses only four or six buses per routing channel (Table 7.6), it is not immediately clear if having different segment lengths in routing channels (as in FPGAs) might help increasing the generality and/or decreasing the array area. To get a sense, experiments are repeated, but with a routing network using both segments of the length one and of the length two (note that the regular and ordered placement requires less long-distance communication). The results have shown that adding longer segments seems to never bring additional generality with exactly the same (or less) routing resources. When using mixed segment lengths the array requires channel width six to successfully route all DFGs (four is no longer enough). Consequently, increased routing opportunities lead to increased generality (in 2 groups out of 15), but also to a slight increase in the array area of 5–7% (in 7 out of 15 groups). Additionally, various segment lengths in some cases lead to reduced routing opportunities and thus decreased generality (in two groups out of 15). Only for those arrays where channel width was not affected by introducing segments of the length two the total area of the array is decreased for about 10% (in 8 out of 15 groups). Therefore, it seems that there is no true and clear superiority of using segments of different length.

7.6 Effects of Domain Grouping on Generality and Area

The technique presented in this thesis aims at designing domain-specific arrays. To verify that it is indeed well tailored for that purpose, the following tests are performed. First, reconfigurable arrays are generated for each individual domain, for all combinations of two or three domains, and for all domains at once. Then, for each of these arrays their generality and area are measured. The results are plotted in Figure 7.2 and Figure 7.3.

Figure 7.2 shows that increasing the number of domains at the input leads to increased generality, due to larger and thus richer input set of DFGs. Twelve out of fifteen groupings achieve generality higher than 70%. Only two groups with four DFGs, G_{1A} and G_{1B} (Table 7.3), achieve generality equal to 50%, due to a very small set of input DFGs.

Figure 7.3 shows the array areas normalized to the area of the array generated for all domains at once. Clearly, increasing the number of domains systematically leads to increased area. However, when generated for individual domains, the array is considerably smaller, proving that this methodology is indeed optimized to provide domain-specific arrays.

If input DFGs belong to various domains, one may wonder whether it is more area efficient to generate a single reconfigurable array out of all DFGs, or it is better to divide the input set of DFGs into two or more disjoint sets and generate the same number of arrays. To answer this question new experiments are performed, relying on the nature of groups shown in Table 7.3. For example, group G_{4A} storing all DFGs can be represented as a union of the following two disjoint sets: $G_{1A} \cup G_{3D}$, or $G_{2A} \cup G_{2F}$, etc. Additionally, it can be represented as a union of the following three disjoint sets: $G_{1B} \cup G_{1D} \cup G_{2B}$, or $G_{1C} \cup G_{1D} \cup G_{2A}$, etc. Hence, the input set of four domains can be divided into two or three sets, and the sum of areas of the arrays generated for those sets can be measured. Then, the results can be normalized to the area of the array generated for all domains at once.

The effects of dividing the input set of DFGs into two, three, or four sets on the total area needed to accommodate generated arrays are shown in Figure 7.4. The results indicate that it is more area efficient to create one instead of multiple domain-specific arrays (one per each domain). This evidences the fact that even though the original DFGs belong to four different domains, they actually share many common computation characteristics that can still be exploited to build a rather efficient array.

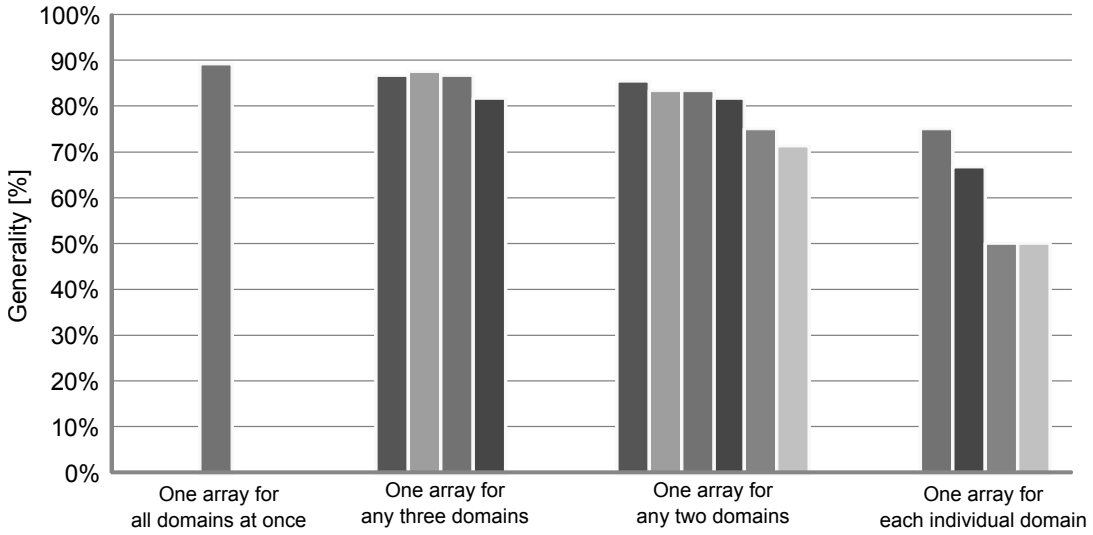


Figure 7.2: Generality of the array for different combinations of domains. Increasing the number of domains leads to increased generality, due to increased input set size. However, even for single domains the generality is reasonably high, at least 50%.

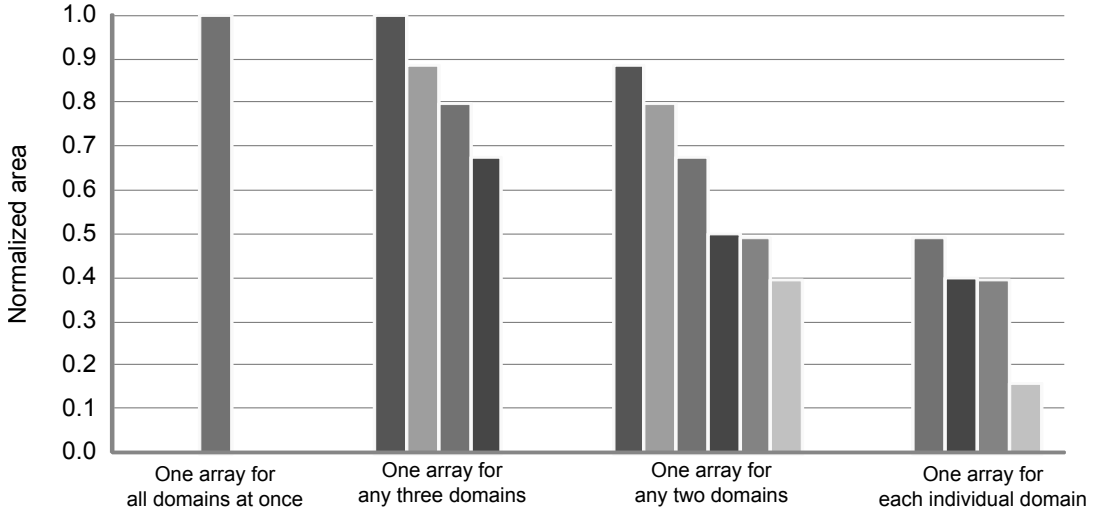


Figure 7.3: The area of the array generated for each individual domain and the combinations of any two, three, or four domains, normalized to the area of the array created for all domains at once. Increasing the number of domains per group leads to increased area. But, when generated for individual domains, the array is considerably smaller and thus area efficient, proving that this novel methodology effectively tailors the array to the domain.

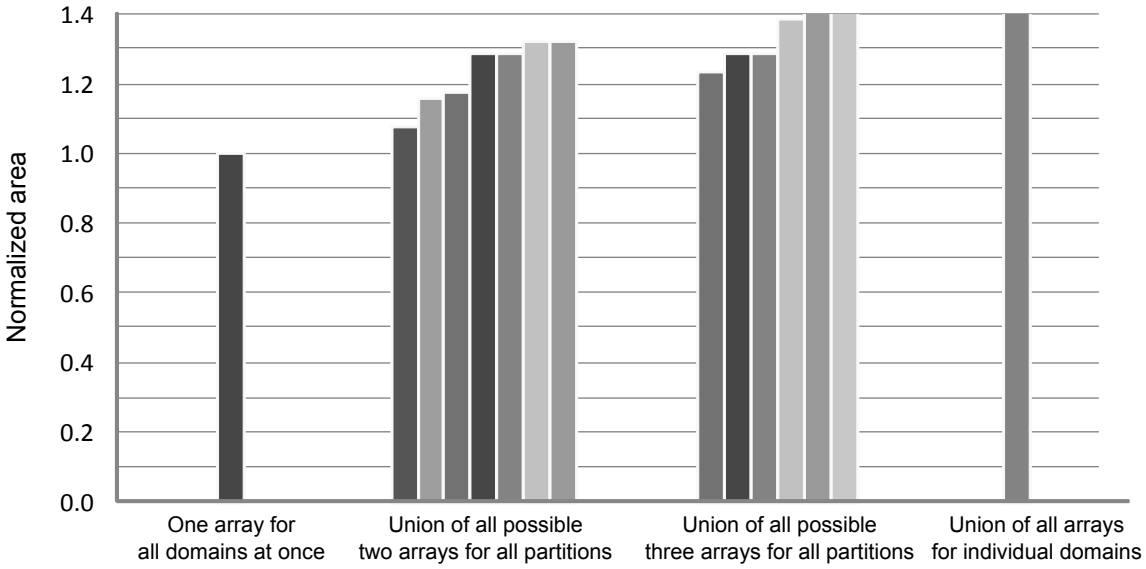


Figure 7.4: The effect of dividing the input set of DFGs into two, three, or four sets on the total area needed to accommodate the arrays. The group G_{4A} storing all DFGs was divided into two, three, or four disjoint groups and the sum of areas of generated arrays was measured. This sum was then normalized to the area of the array created for all four domains at once. The results indicate that the single array generated for all domains at once is the most area efficient solution compared to any multiple-array solution.

7.7 Area/Delay Oversize Compared to ASIC

Next, the reconfigurable array area and delay are estimated and compared with a 65nm standard cell ASIC implementation. For all DFGs D_1 to D_{19} it is assumed, conservatively, that their ASIC implementations require no routing area besides the area required for the operators and pipeline registers, and that all shifts are by a constant value and can be implemented via wiring in the ASIC. Thus, the area of a DFG implemented as ASIC $AreaAsic(G, D_i)$ is the sum of areas of individual operators, including the pipeline registers. To find the area and delay of operators, they are synthesized, placed, and routed using a 65nm standard cell library. Again conservatively, it is assumed that the delay of the ASIC implementation equals only the delay through the critical path of the components of the DFG; routing delays are ignored. To find a critical path delay $DelayAsic(G, D_i)$, the algorithm for finding all paths in a graph, mentioned in Section 4.2, was adapted.

To estimate the critical path delay of a DFG mapped onto a domain-specific array and the total area of the array, VPR is used. VPR outputs a detailed report containing the information on the array area, the area occupied by routing network, as well as critical path delay of placed and routed DFGs. Yet, VPR imposes one constraint—it assumes all operators are identical. To circumvent this constraint, all operator areas and delays are set to zero in the input architectural file. Hence, VPR can report correctly the area and delay used by the routing network only. Area used by routing resources is then added to the sum of areas of individual operators to find the total array area. In the same way, the critical path delay reported by VPR is added to the critical path delay of the DFG (in which routing delays are ignored) to obtain the final critical path delay.

The following experiment is run on all individual groups (domains) from Table 7.3, as well as for all combinations of two and three domains, and for all domains at once. Assuming that DFGs input to the experiment comprise a set called G, for each DFG D_i in G, the algorithm creates a domain-specific array A_i from all the remaining DFGs $G - D_i$. Then, it tries placing and routing the DFG D_i on the array A_i. If it succeeds, the area of A_i ($Area(A_i)$) and the critical path delay of D_i ($Delay(A_i, D_i)$) are evaluated. Finally, the area ratio equals

$$ArrayRatio(G, D_i) = \frac{Area(A_i)}{AreaAsic(G, D_i)}, \tag{7.2}$$

while the delay ratio equals

$$DelayRatio(G, D_i) = \frac{Delay(A_i, D_i)}{DelayAsic(G, D_i)}. \tag{7.3}$$

The array ratio gives a sense of the cost in die-area to execute an application DFG on a highly flexible domain-specific array, compared with the most area-efficient alternative (ASIC), which, on the other hand, has no flexibility at all. Similarly, the delay ratio shows the decrease in the application execution-speed when its DFG is placed and routed on a flexible array, instead of being implemented as the most efficient ASIC alternative.

The array ratio and delay ratio pairs found for all groups G comprising of individual domains, all combinations of two or three domains, and all DFGs at once, are plotted in Figure 7.5. Area ratios are given at x-axis, while the delay ratios are given at y-axis. Since these values are scaled with respect to ASIC areas and delays, the point at coordinates (1,1) corresponds to ASIC implementations of all DFGs. The gray area marked as FPGA represents the area/delay space where results would be expected if DFGs were to be mapped on an FPGA, achieving perfect generality if enough LUTs are present in the architecture. The boundaries of the FPGA area roughly correspond to the data published by Kuon and Rose [KR07]. Their study provides detailed experimental measurements of the differences between FPGAs and ASICs in terms of logic density, circuits speed, and power consumption for core logic. Kuon and Rose's results [KR07] show that for circuits containing only look-up table-based logic and flip-flops, the ratio of silicon area required to implement them in FPGAs and ASICs is on average 32× when hard DSP blocks are not used, whereas it decreases to 24× when these blocks are used. These numbers present an optimistic lower bound on the area gap because they assume that all logic array blocks can be fully utilized. Additionally, they report the critical path delay ratio to be on average 3.4× when hard DSP blocks are used, and even a slightly higher ratio, around 3.5×, when these blocks are used in the design.

The results from Figure 7.5 show that the majority of the DFGs result in arrays with an area up to 15× larger than the corresponding ASIC area, and thus significantly more area-efficient than FPGAs with DSP blocks, due to the usage of specialized coarse grain operators. Additionally, the average delay increase compared to ASICs is less than 2×, which is again superior to FPGAs with DSP blocks, due to the efficient word-based communication network. Hence, the novel methodology succeeds in populating the area-delay design space that currently separates ASIC from FPGA implementations while still providing a high generality. However, there are two DFGs that have high area ratio compared to their ASIC implementation—DFGs D_{12} and D_{13}. They do not contain high-area operators, such as multipliers. Consequently, the results in Figure 7.5 appear

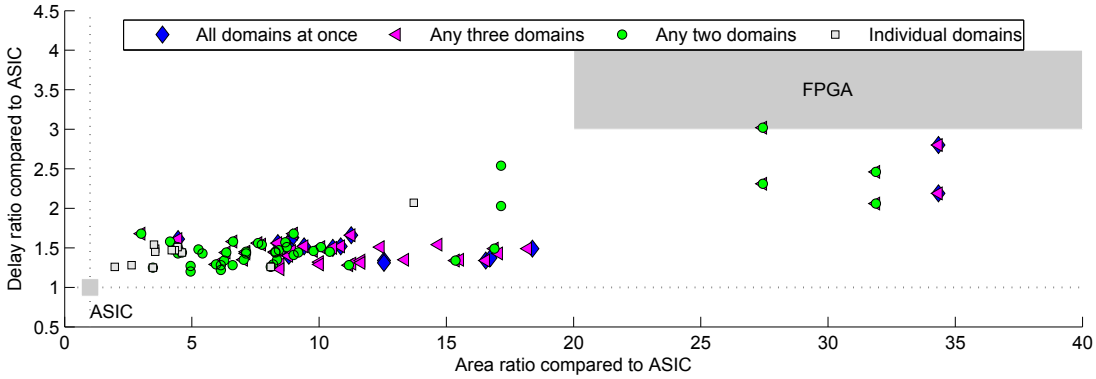


Figure 7.5: Area/delay ratio of the arrays generated from all DFGs in the group except the removed DFG, with respect to an ASIC design of the DFG removed from the group. The datapath is usually up to 20× larger and up to 2× slower than the corresponding ASIC design (with some deviations in extreme cases). The results are clustered by the number of domains in the group. The shaded FPGA zone is as reported in prior studies [KR07].

skewed, but these data points appear as outliers.

Clearly, the idea of using custom-designed coarse-grained operands as basic building blocks results in both reduced area and improved critical path delay compared to using fine-grained FPGA fabric. Additionally, the amount of configuration storage for both operands and routing network decreases significantly due to the bus-based connectivity, where wires do not need to be configured independently.

7.8 Area/Delay Oversize Compared to Datapath Merging

To estimate the area and delay oversize of the domain-specific reconfigurable arrays compared with the datapath-merging methodology, the algorithm introduced by Brisk et al. [BKS04] is implemented entirely. Another, more recent datapath merging algorithm presented by Zuluaga and Topham [ZT09] is not selected because it is based on the algorithm by Brisk et al. and because it introduces new features that are not directly relevant to this work. Namely, they introduce latency constraints in the merging process to explore the space of possible implementation alternatives instead of trying to find a unique solution, while the datapath merging by Brisk is focusing on maximizing the

area savings. However, that algorithm is not very efficient for complex DFGs as those used in this work. Hence it had to be modified to improve the algorithm runtime.

Datapath merging algorithm [BKS04] assumes as an input a set of directed acyclic graphs (DAGs) $G = \{G_1, G_2, ..., G_n\}$. It has two phases, global and local, that repeat and alternate until all DAGs are not merged, or until there are no more candidates for merging.

The *global phase* starts with decomposing each DAG $G_i \in G$ into a set of input-to-output paths P_i, where the set $P = \{P_1, P_2, ..., P_n\}$ stores the sets of paths corresponding to each DAG. Then, it looks for the candidate DAGs G_i and G_j to merge, by finding the pair of paths p_x and p_y, $1 \leq x \leq |P_i|$, $1 \leq y \leq |P_j|$, $1 \leq i, j \leq n$, $i \neq j$, such that they share the maximum-area common-subsequence MACSeq. Then, it merges G_i and G_j by sharing the nodes in MACSeq and inserting multiplexers that enable configuring the datapath to execute either G_i ir G_j. Finally, G_i and G_j are replaced by their merged version G'.

The *local phase* begins with new DAG G', and continues merging nodes inside G', trying to avoid creating cycles in G'. To accommodate large graphs, Brisk et al. [BS06] recommend to replace the enumeration of all paths by a pruning heuristic that limits the set P to a reasonable size. Hence, the implementation of the algorithm in this thesis includes one such heuristic:

- First, the size of the sets P_i, $1 \leq i \leq n$ is limited to 100 different paths per set, as estimated based on the size of the input DFGs (Table 7.2) and frequent overlaps among paths in the graphs.
- Then, to select good candidates for P_i while enumerating the paths, the algorithm checks if a newly found path is a subsequence of the path already present in P_i. If yes, the algorithm ignores it and continues enumerating. This way P_i will contain the paths offering various maximum-area common-subsequences.

To estimate the datapath area, multiplexers of various size were synthesized, placed, and routed using the same 65nm standard cell library. These multiplexers were inserted

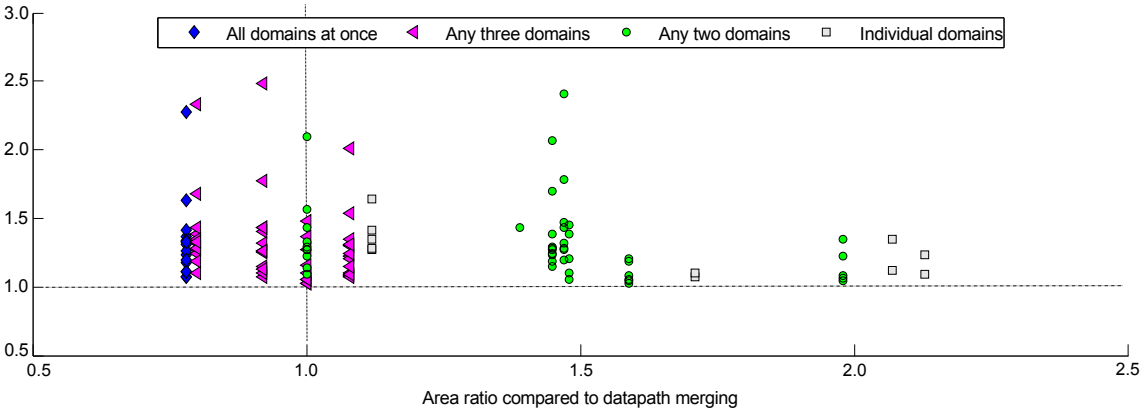


Figure 7.6: Area/delay ratio of the arrays generated from all DFGs in the group, with respect to those of the datapath obtained by merging the same DFGs. The array is usually around 2× larger except extreme cases, while the critical path delay of applications mapped onto it is in most cases up to 2× higher than the corresponding delay of the applications when run on merged datapath. The results are clustered by the number of domains in the group.

in the datapath while merging. The areas of operators have been already calculated for the previous experiments. The area of a merged datapath approximatelly equals the sum of areas of its operators and inserted multiplexers. Conservatively, the area used for routing is neglected. To estimate the routing delay and the area of the array, the methodology described in previous section is used.

Ideally, if DFGs within a group are perfectly merged, one would expect that the merged datapath is as large as the DFG reporting the maximum area utilization in Table 7.6, with the addition of the area of the inserted multiplexers, but reduced for the area used for routing in the reconfigurable array. Since the area used for multiplexers is certainly less than the total area used for FPGA-like routing network in domain-specific reconfigurable arrays, it can be expected that for individual domains the area ratio should be less than $100/48 \simeq 2.08$, for any two domains less than $100/33 \simeq 3.03$, and for any three domains and all domains at once less than $100/30 \simeq 3.33$ (Table 7.6). The experimental results presented in Figure 7.6 show that the array is up to 3× larger than the merged datapath, while for majority of the groupings this ratio is only up to 2.2×. For two groups, G_{2F} and G_{3C}, area of the routing network was considerably higher than

the area of the multiplexers in the merged datapath. To understand why, one should look at the size of the array generated for G_{2F}: it equals 13 rows × 24 columns. Group G_{2F} is the union of groups G_{1C} and G_{1D} (Table 7.3), where G_{1C} needs an array of the size 5×24 and G_{1D} of the size 13×12. Obviously, the array created for G_{2F} introduces 50% of unused routing resources, so the area ratio is somewhat higher in this particular case. Figure 7.6 also shows that the delay ratio is up to 2.5×, in most of the cases up to 2×.

In total, the results indicate that the novel method for designing domain-specific arrays succeeds in generating datapaths with a reasonable level of generality at speeds comparable to those of datapaths created by merging the DFGs—designs which have, arguably, practically no generality.

8 Conclusions

Semiconductor technology keeps following Moore's law—transistor density doubles roughly every 18 months. Yet, an improvement in one aspect is often accompanied by increasing constraints in other aspects, which need to be carefully managed; With transistors going into deep submicron scales, chip power consumption increases, manufacturing cost rises, variability increases, and reliability decreases. To reduce energy consumption and improve performance, embedded systems use specialized hardware accelerators [Smi97, IL06], especially for applications involving signal and video processing, communications, and computer vision.

Specialization is the key to efficiency. It can be achieved by designing and synthesizing ASIC accelerators for each target application separately, but this approach is not very area-efficient. A better way would be to merge these accelerators into a single reconfigurable datapath of smaller die-area, as proposed by Brisk et al. [BKS04] and Zuluaga et al. [ZT09]. However, this improvement comes at the cost of increased latency and thus impaired accelerator performance.

The flexibility of such ASIC accelerators is very limited—they can be used to execute only those applications that are known at the design time. Yet, providing more flexibility is necessary to accommodate late design changes or new applications in the same domain, in order to avoid the extremely high nonrecurring engineering costs of incremental

chip redesign. On the other side, the circuits that provide the highest flexibility, FPGAs, suffer from incredibly poor logic density, even when system designers make good usage of DSP Blocks, block RAMs, and transceivers. Additionally, the fine-grained nature of FPGAs is particularly nonoptimized for digital signal processing applications, which utilize common operations such as multiplications and additions, and thus benefit more from efficient coarse-grained components. Hence, a number of reconfigurable systems with a coarser-grain structure (CGRAs) has been designed. Usually, they are not specialized to fit the characteristics of a specific application domain, but to a wider range of applications.

This thesis presented a novel approach in designing coarse-grain reconfigurable arrays; this technique is different in several aspects:

- Instead of designing the array in an intuitive way and then checking how well it fits for an input set of applications, the array design process is automated and guided by a special algorithm for analyzing the characteristics of those applications.
- A limited amount of flexibility is inserted in the arrays in a controlled manner, so that it is very likely that arrays will not only run the input applications, but also many of the computationally similar applications.
- The resulting arrays are domain-specific, i.e., they are tailored to an application domain, represented by the input set of applications.
- The resulting arrays present a good compromise between absolutely flexible FPGA alternatives and almost completely inflexible ASIC alternatives, and are well suited to digital signal processing domains, due to their coarse-grained nature.

The novel design method is composed of four main phases. Firstly, a set of candidate DFGs from the input applications is generated. Then, those DFGs are analyzed to extract the column of the datapath. This column is replicated to create a regular 2D array structure. Finally, an FPGA-like statically configured routing network is added to

enable routing the DFGs. All these phases are implemented in a standard programming language to build a complete tool for designing domain-specific reconfigurable arrays. This tool can be used in multiple ways. For example, chip developers can use it to automatically design the architecture of the arrays to be incorporated in larger VLSI circuits, or to perform a detailed experimental evaluation of the benefits and drawbacks of the proposed methodology, or to compare the array performance with the ASIC and FPGA alternatives.

The related work in designing domain-specific CGRAs focused on (i) ways of tuning the characteristics of the operator that is replicated throughout the array [ABP08, ABP11, PSH04] and thus exploring different CGRA configurations, or on (ii) ways of choosing operators for 1D arrays having very small number of input/output ports and limited interconnectivity [CH08]. This work is different in several aspects:

- Here the arrays are two-dimensional and built by replicating the column throughout the array. Hence, each array row is homogeneous and composed of a single operator type.
- Then, the allowed number of I/O ports is considerably higher—two input and two output ports per column of the array are provided.
- Finally, a graph-drawing approach is used to map DFGs in a top-down fashion, where data is routed from the input towards the output ports, and to design an efficient routing network with short connections and minimal number of edge crossings. Employing a graph-drawing approach is the key for replicating the regularity of computational patterns found in application DFGs onto the array using the array operators and routing network resources.

The experimental evaluation shows that array generality is on average higher than 80% and sometimes reach even 95%. This means that the achieved probability to successfully execute applications that belong to the same domain, but which are not

known at the design time, is very high. Hence, resulting domain-specific CGRAs are indeed significantly more flexible than ASIC accelerators.

The achieved generality comes at the cost of increased array area. Namely, the measured maximum area utilization varies between 30% up to 78%, where it was low for a mix of applications belonging to different domains and high for applications belonging to a single domain. This indicates that the novel methodology is indeed optimized to provide area-efficient domain-specific arrays.

The ratio of the area dedicated for routing resources to the total array area is in the range 28–45%, which is significantly better than what is reported in programmable logic devices—according to the paper by Feng and Kaptanogly from Actel Corporation [FK08] up to 90% of a Programmable Logic Device chip is occupied by the programmable interconnect, including wires, switches and configuration bits.

The majority of arrays have the area up to 15× larger than the ASIC area of a single DFG in isolation, and are thus significantly more area efficient than FPGAs with DSP blocks. This is due to the usage of specialized, area efficient, coarse grain operators. Additionally, the ratio of the delay of a DFG mapped on the array compared to the delay of the same DFG implemented as an ASIC circuit is on average smaller than 2×. This is again superior to FPGAs with DSP blocks, due to the usage of an efficient word-based communication network. Therefore, the novel methodology succeeds in populating the area-delay design space that currently separates ASIC from FPGA implementations, while providing a high generality.

Compared with the state-of-the-art datapath merging approach, the arrays were up to 3× larger than the merged datapath, while for the majority of the DFG groups under test this ratio was up to 2.2× only. Additionally, the delay ratio was in the most of the cases up to 2× only. This shows that the new method succeeds in creating arrays with a significant level of generality at speeds comparable to those of datapaths created by merging the DFGs. Merged DFGs, on the other side, have practically no generality.

There are several avenues for future work, such as specializing the bitwidth of the operators, and composing multiple limited-precision operators to form higher-precision operators. Another possibility would be to introduce flexible arithmetic components, e.g., multipliers that can be configured to perform addition/subtraction as well. Finally, to improve array utilization the rectangular shape of the array could be customized to better fit the domain, as some classes of DFGs, especially instruction set extensions, often have the general shape of inverted cones [CFHZ04].

All in all, this thesis explores a new direction of significant importance in a world where heterogeneous spatial systems are likely to emerge as a dominant form of computation, especially for code acceleration in domain-specific embedded systems.

Bibliography

[ABP08] Giovanni Ansaloni, Paolo Bonzini, and Laura Pozzi. Design and architectural exploration of expression-grained reconfigurable arrays. In *Proceedings of the 6th IEEE Symposium on Application Specific Processors*, pages 26–33, Anaheim, Calif., June 2008.

[ABP11] Giovanni Ansaloni, Paolo Bonzini, and Laura Pozzi. Egra: A coarse grained reconfigurable architectural template. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, VLSI-19(6):1062–1074, June 2011.

[AD04] H. Arslan and S. Dutt. Acm great lakes symposium on vlsi. In *Proceedings of the 14th ACM Great Lakes Symposium on VLSI*, pages 208–213, Boston, Massachusetts, April 2004.

[API03] Kubilay Atasu, Laura Pozzi, and Paolo Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. *International Journal of Parallel Programming*, 31(6):411–28, December 2003.

[AR96] Arthur Abnous and Jan Rabaey. Ultra-low-power domain-specific multimedia processors. In *Proceedings of the 9th Workshop on VLSI Signal Processing*, pages 461–70, San Francisco, Calif., October 1996.

[AYP+06] M. Ahn, J.W. Yoon, Y. Paek, Y. Kim, M. Kiemb, and K. Choi. A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures.

In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 363–68, Munich, March 2006.

[BBKG07] F. Bouwens, M. Berekovic, A. Kanstein, and G. Gaydadjiev. Architectural exploration of the ADRES coarse-grained reconfigurable array. In *Reconfigurable Computing: Architectures, Tools and Applications*, volume 4419 of *Lecture Notes in Computer Science*, pages 1–13. Springer, Berlin, June 2007.

[BE06] Jason Brown and Marc Epalza. Automatically identifying and creating accelerators directly from c code. *Xcell Journal*, pages 58–60, July 2006.

[Ber75] Alfs T. Berztiss. *Data Structures: Theory and Practice*. Academic Press, New York, second edition, 1975.

[BFRV92] Stephen D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *Field-Programmable Gate Arrays*. Kluwer Academic Publishers, Boston, Mass., 1992.

[BGV03] H. Bunke, G. Guidobaldi, and M. Vento. Weighted minimum common supergraph for cluster representation. In *International Conference on Image Processing*, pages II – 25–8, September 2003.

[BKKS02] Philip Brisk, Adam Kaplan, Ryan Kastner, and Majid Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 262–69, Grenoble, France, October 2002.

[BKS04] Philip Brisk, Adam Kaplan, and Majis Sarrafzadeh. Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. In *Proceedings of the 41st Design Automation Conference*, pages 395–400, San Diego, Calif., June 2004.

[BMS98] Jürgen Branke, Martin Middendorf, and Frerk Schneider. Improved heuristics and a genetic algorithm for finding short supersequences. *OR Spectrum*, 20(1):39–45, February 1998.

[BP07] Paolo Bonzini and Laura Pozzi. Polynomial-time subgraph enumeration for automated instruction set extension. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Nice, April 2007.

[BPV00] Giuseppe Di Battista, Maurizio Patrignani, and Francesco Vargiu. A split & push approach to 3d orthogonal drawing. *Journal of Graph Algorithms and Applications*, 4(3):105–133, April 2000.

[BR96] V. Betz and J. Rose. Directional bias and non-uniformity in fpga global routing architectures. In *Proceedings of the International Conference on Computer Aided Design*, pages 625–659, San Jose, Calif., November 1996.

[BR97] O. Bringmann and W. Rosenstiel. Resource sharing in hierarchical synthesis. In *Proceedings of the International Conference on Computer Aided Design*, pages 318–325, San Jose, Calif., November 1997.

[BR00] Vaughn Betz and Jonathan Rose. Automatic generation of FPGA routing architectures from high-level descriptions. In *Proceedings of the 8th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 175–84, Monterey, Calif., February 2000.

[BS06] Philip Brisk and Majid Sarrafzadeh. Datapath synthesis. In Paolo Ienne and Rainer Leupers, editors, *Customizable Embedded Processors—Design Technologies and Applications*, Systems on Silicon Series, chapter 10, pages 233–55. Morgan Kaufmann, San Mateo, Calif., 2006.

[Car80] Marie-José Carpano. Automatic display of hierarchized graphs for computer aided decision analysis. *IEEE Transactions on Systems, Man, and Cybernetics*, 10(11):705–715, 1980.

[CFHZ04] Jason Cong, Yiping Fan, Guoling Han, and Zhiru Zhang. Application-specific instruction generation for configurable processor architectures. In *Proceedings of the 12th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 183–89, Monterey, Calif., February 2004.

[CH01] Katherine Compton and Scott Hauck. Totem: custom reconfigurable array generation. In *Proceedings of the 9th IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, Calif., April 2001.

[CH08] Katherine Compton and Scott Hauck. Automatic design of reconfigurable domain-specific flexible cores. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, VLSI-16(5):493–503, May 2008.

[Che09] Wai-Kai Chen. *Feedback, Nonlinear, and Distributed Circuits*. CRC Press, third edition, 2009.

[CHJ10] Jason Cong, Hui Huang, and Wei Jiang. A generalized control-flow-aware pattern recognition algorithm for behavioral synthesis. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 1255–1260, Dresden, Germany, March 2010.

[CHW00] Timothy J. Callahan, John R. Hauser, and John Wawrzynek. The Garp architecture and C compiler. *Computer*, 33(4):62–69, April 2000.

[CJ08] Jason Cong and Wei Jiang. Pattern-based behavior synthesis for FPGA resource reduction. In *Proceedings of the 16th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 107–16, Monterey, Calif., February 2008.

[CKG+96] Miguel R. Corazao, Marwan A. Khalaf, Lisa M. Guerra, Miodrag Potkonjak, and Jan M. Rabaey. Performance optimization using template mapping for datapath-intensive high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(8):877–888, August 1996.

[CKP+04] Nathan Clark, Manjunath Kudlur, Hyunchul Park, Scott Mahlke, and Krisztian Flautner. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Proceedings of the 37th Annual International Symposium on Microarchitecture*, pages 30–40, Portland, Oreg., December 2004.

[CKS+99] Amit Chowdhary, Sudhakar Kale, Phani K. Saripella, Naresh K. Sehgal, and Rajesh K. Gupta. Extraction of functional regularity in datapath circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(9):1279–1296, September 1999.

[CM12] Liang Chen and Tulika Mitra. Graph minor approach for application mapping on cgras. In *Proceedings of the IEEE International Conference on Field Programmable Technology*, pages 285–292, Seul, June 2012.

[CWW96] Yao-Wen Chang, D.F. Wong, and C. K. Wong. Universal switch modules for fpga design. *ACM Transactions on Design Automation of Electronic Systems*, 1:80–101, 1996.

[CZ09] Kun-Mao Chao and Luoxin Zhang. *Sequence Comparison: Theory and Methods*. Computational Biology. Springer, London, 2009.

[CZM03] Nathan Clark, Hongtao Zhong, and Scott Mahlke. Processor acceleration through automated instruction set customisation. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 129–40, San Diego, Calif., December 2003.

[CZM05] Nathan T. Clark, Hongtao Zhong, and Scott A. Mahlke. Automated custom instruction generation for domain-specific processor acceleration. *IEEE Transactions on Computers*, C-54(10):1258–70, October 2005.

[ECF+97] Carl Ebeling, Darren C. Cronquist, Paul Franklin, Jason Secosky, and Stefan G. Berg. Mapping applications to the RaPiD configurable architecture.

In *Proceedings of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 106–15, Napa Valley, Calif., April 1997.

[EEM06] EEMBC Consortium. *DENBench Version 1.0, Benchmark Name: MPEG-2 Decode*, February 2006. `http://www.eembc.org/`.

[eLCD03] Jong eun Lee, Kiyoung Choi, and Nikil D. Dutt. Compilation approach for coarse-grained reconfigurable architectures. *IEEE Design and Test of Computers*, 20(1):26–33, February 2003.

[Exp] University of California, Santa Barbara, Calif. *ExpressDFG—Instruction Scheduling Benchmarks*. `http://express.ece.ucsb.edu/benchmark/`.

[FK08] Wenyi Feng and Sinan Kaptanoglu. Designing efficient input interconnect blocks for LUT clusters using counting and entropy. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 1(1):6:1–6:28, March 2008.

[GEMA04] Lemieux G., Lee E., Tom M., and Yu A. Directional and single-driver wires in fpga interconnect. In *Proceedings of the IEEE International Conference on Field Programmable Technology*, pages 41–48, Brisbane, Australia, December 2004.

[GKN06] Emden Gansner, Eleftherios Koutsofios, and Stephen North. *Drawing graphs with* dot, January 2006. `http://www.graphviz.org/Documentation/dotguide.pdf`.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software—Practice and Experience*, 30(11):1203–1233, 2000.

[GSM+99] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. PipeRench: a co-processor for streaming multimedia acceleration. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 28–39, Atlanta, Ga., May 1999.

[GV04] V. Gudise and G. Venayagamoorthy. Fpga placement and routing using particle swarm optimization. In *IEEE Computer Society Annual Symposium on VLSI*, pages 307–308, Tampa, Florida, February 2004.

[HA96] Scott Hauck and Anant Agarwal. Software technologies for reconfigurable systems. *IEEE Transactions on Computers*, pages 1–40, 1996.

[HM01] Zhining Huang and Sharad Malik. Managing dynamic reconfiguration overhead in systems-on-a-chip design using reconfigurable datapaths and optimized interconnection networks. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 735–740, Munich, Germany, March 2001.

[IL06] Paolo Ienne and Rainer Leupers, editors. *Customizable Embedded Processors—Design Technologies and Applications*. Systems on Silicon Series. Morgan Kaufmann, San Mateo, Calif., 2006.

[JL95] Tao Jiang and Ming Li. On the approximation of shortest common supersequences and longest common subsequences. *SIAM Journal on Computing*, 24(5):1122–39, October 1995.

[Keu87] Kurt Keutzer. Dagon: Technology binding and local optimization by dag matching. In *Proceedings of the 24th Design Automation Conference*, pages 341–347, Florida, USA, March 1987.

[KKP+05] Yoonjin Kim, Mary Kiemb, Chulsoo Park, Jinyong Jung, and Kiyoung Choi. Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 12–17, Munich, Germany, March 2005.

[KR07] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-26(2):203–15, February 2007.

[KR08a] Ian Kuon and Jonathan Rose. Area and delay trade-offs in the circuit and architecture design of FPGAs. In *Proceedings of the 16th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 149–58, Monterey, Calif., February 2008.

[KR08b] Ian Kuon and Jonathan Rose. Automated transistor sizing for fpga architecture exploration. In *Proceedings of the 45th Design Automation Conference*, pages 792–795, Anaheim, Calif., June 2008.

[KS00] Thomas Kutzschebauch and Leon Stok. Regularity driven logic synthesis. In *Proceedings of the International Conference on Computer Aided Design*, pages 439–446, San Jose, Calif., November 2000.

[KW05] Z. Kwok and S.J.E. Wilton. Register file architecture optimization in a coarse-grained reconfigurable architecture. In *Proceedings of the 13th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 35–44, Napa Valley, Calif., April 2005.

[LB93] Guy G. Lemieux and Stephen D. Brown. A detailed routing algorithm for allocating wire segments in field-programmable gate arrays. In *Proceedings of the ACM/SIGDA Physical Design Workshop*, pages 215–226, San Francisco, Calif., April 1993.

[LB03] Tien-Lung Lee and Neil W. Bergmann. An interface methodology for retargetable FPGA peripherals. In *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 167–173, Las Vegas, Nev., June 2003.

[LBF+98] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman P. Amarasinghe. Space-time scheduling of instruction-level parallelism on a Raw machine. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, San Jose, Calif., October 1998.

[LKJ+09] Jason Luu, Ian Kuon, Peter Jamieson, Ted Campbell, Andy Ye, Wei Mark Fang, and Jonathan Rose. Vpr 5.0: Fpga cad and architecture exploration tools with single-driver routing, heterogeneity and process scaling. In *Proceedings of the 17th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 133–142, Monterey, Calif., February 2009.

[LKMM95] Tai Ly, David Knapp, Ron Miller, and Don MacMillen. Scheduling using behavioral templates. In *Proceedings of the 32nd Design Automation Conference*, pages 101–106, New York, NY, June 1995.

[LSL+00] Ming-Hau Lee, Hartej Singh, Guangming Lu, Nader Bagherzadeh, Fadi J. Kurdahi, Eliseu M. C. Filho, and Vladimir Castro Alves. Design and implementation of the MorphoSys reconfigurable computing processor. *Journal of VLSI Signal Processing Systems*, 24(2–3):147–64, March 2000.

[MAHM02] Nahri Moreano, Guido Araujo, Zhining Huang, and Sharad Malik. 15th international symposium on system synthesis. In *Proceedings of the 15th International Symposium on System Synthesis*, pages 38–43, Kyoto, Japan, October 2002.

[MVV+02] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. DRESC: A retargetable compiler for coarse-grained reconfigurable architectures. In *Proceedings of the IEEE International Conference on Field-Programmable Technology*, pages 166–73, December 2002.

[MVV+03] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 296–301, Munich, Germany, March 2003.

[Nor04] Stephen C. North. *Drawing graphs with* neato, April 2004. `http://www.graphviz.org/pdf/neatoguide.pdf`.

[PFKM06] Hyunchul Park, Kevin Fan, Manjunath Kudlur, and Scott Mahlke. Modulo graph embedding: mapping applications onto coarse-grained reconfigurable architectures. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 136–146, Seoul, Korea, October 2006.

[PFM+08] Hyunchul Park, Kevin Fan, Scott A. Mahlke, Taewook Oh, Heeseok Kim, and Honh-seok Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th International Conference on Parallel Architecture and Compilation Techniques*, pages 166–76, Toronto, October 2008.

[PSH04] Shawn Phillips, Akshay Sharma, and Scott Hauck. Automating the layout of reconfigurable systems via template reduction. In *Proceedings of the 12th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 340–341, Napa Valley, Calif., April 2004.

[Rau94] B. Ramakrishna Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, November 1994.

[RK93] D. Sreenisava Rao and Fadi J. Kurdahi. On clustering for maximal regularity extraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(8):1198–1208, August 1993.

[RS99] N. Robertson and P. D. Seymour. Graph minors. *Journal of Combinatorial Theory*, 77(1):162–210, 1999.

[Smi97] Michael J. S. Smith. *Application-Specific Integrated Circuits*. Addison-Wesley, Boston, Mass., 1997.

[STT81] Kozo Sigiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, February 1981.

[TI03a] Texas Instruments. *TMS320C64x DSP Library Programmer's Reference*, October 2003. Lit. no. SPRU565B.

[TI03b] Texas Instruments. *TMS320C64x Image/Video Processing Library Programmer's Reference*, October 2003. Lit. no. SPRU023B.

[TI10] Texas Instruments. *TMS320C67x DSP Library Programmer's Reference*, January 2010. Lit. no. SPRU657C.

[VNK+01] Girish Venkataramani, Walid Najjar, Fadi Kurdahi, Nader Bagherzadeh, and Wim Bohm. A compiler framework for mapping applications to a coarse-grained reconfigurable computer architecture. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 116–125, Atlanta, Ga., November 2001.

[War77] John Warfield. Crossing theory and hierarchy mapping. *IEEE Transactions on Systems, Man, and Cybernetics*, 7(7):505–523, 1977.

[Wil97] Steven J. E. Wilton. *Architecture and Algorithms for Field Programmable Gate Arrays with Embedded Memory*. Ph.D. thesis, University of Toronto, 1997.

[YGBT09] Sami Yehia, Sylvain Girbal, Hugues Berry, and Olivier Temam. Reconciling specialization and flexibility through compound circuits. In *Proceedings of*

the 15th International Symposium on High-Performance Computer Architecture, pages 277–88, Raleigh, N.C., February 2009.

[YM04] Pan Yu and Tulika Mitra. Scalable custom instructions identification for instruction set extensible processors. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 69–78, Washington, D.C., September 2004.

[YR06] Andy Ye and Jonathan Rose. Using bus-based connections to improve field-programmable gate-array density for implementing datapath circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(5):462–73, May 2006.

[YSP+08] Jonghee W. Yoon, Aviral Shrivastava, Sanghyun Park, Minwook Ahn, Reiley Jeyapaul, and Yunheung Paek. SPKM: A novel graph-drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 776–82, Seoul, Korea, January 2008.

[ZT09] Marcela Zuluaga and Nigel Topham. Design-space exploration of resource-sharing solutions for custom instruction set extensions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-28(12):1788–1801, December 2009.

[ŽVSM97] Vojin Živojnovic, Juan Martínez Velarde, Christian Schläger, and Heinrich Meyr. Dspstone: A dsp-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing Applications and Technology (ICSPAT)*, 1997.

Biography

Mirjana Stojilović was born on 3rd January 1983 in Zemun, Republic of Serbia. She finished the elementary school in Zemun as the best scholar in her class. Then, she finished the Mathematical Grammar School in Belgrade, specialized for students talented in mathematics, physics, and computer science, as a recipient of the “Vuk Karadžić” award. In parallel, she finished a two-year Primary Music School in solo singing, in the class of professor Sonja Gligorić. During her elementary and high education, she was very active in mathematics and physics contests. The most success she had in the physics contests, winning top prizes at the national competition level. She took part in solo singing, poetry writing, and recitation contests as well. Ms. Stojilović entered the School of Electrical Engineering in Belgrade in the school year 2002/2003. She graduated from the department of Electronics in 2006, one year in advance and with the GPA 9.9/10. Professor Lazar Saranovac was the mentor of her diploma work entitled "Transmission of video signals over low-voltage network". This work summarized the results of her research performed at the "Elsys Eastern Europe" company in Belgrade.

Ms. Stojilović entered the PhD program of the School of Electrical Engineering in Belgrade, at the department of Electronics, in the school year 2007/2008. During the studies, she passed all the required exams with GPA 10/10. She published one paper in

the prestigious international journal IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, and presented 16 papers at conferences. One of the conference papers related to the topic of her PhD thesis, she presented at one of the top two conferences in the domain—Design Automation and Test in Europe (DATE). In 2012, she was awarded the first prize at the "Western Balkan Countries' ICT Idea Competition", organized by FP7-funded projects ICT-WEB-PROMS and WINS-ICT. In 2012, she was a recipient of the Young Author Best Paper Award for the paper "Design of antenna system for short range wireless sensor network", presented at the 19^{th} Telecommunications forum TELFOR 2011 in Belgrade. Ms. Stojilović is an IEEE member and serves as a Reviewer of ACM Transactions on Design Automation of Electronic Systems Journal and Design Automation Conference (DAC).

From January 2007 until May 2013, she worked as an Embedded System Developer with the Institute Mihailo Pupin in Belgrade. In the scope of the project "Advancing Embedded System Research in Serbia" she was cooperating with the Processor Architecture Laboratory of the Swiss Federal Institute of Technology in Lausanne (EPFL), visiting periodically as a guest researcher. She now lives in Switzerland and works as a scientific collaborator at the University of Applied Sciences and Arts Western Switzerland.

Biografija

Mirjana Ž. Stojilović je rođena u Zemunu, Republika Srbija, 3. januara 1983. godine. Osnovnu školu je završila u Zemunu, kao đak generacije. Potom je završila Matematičku gimnaziju u Beogradu, kao nosilac diplome „Vuk Karadžić“. Tokom gimnazijskog školovanja osvajala je nagrade iz fizike na svim nivoima takmičenja u zemlji. Pored toga, završila je i Nižu muzičku školu Kosta Manojlović u Zemunu, odsek solo pevanja, u klasi prof. Gligorić. Školske 2002/03. godine upisala je Elektrotehnički fakultet Univerziteta u Beogradu. Diplomirala je na smeru za Elektroniku, pre roka, decembra 2006. godine, sa prosečnom ocenom 9,90/10, i diplomskim radom na temu „Prenos video signala preko niskonaponske mreže“. Mentor diplomskog rada je bio dr Lazar Saranovac, docent. Diplomski rad je bio rezultat stručne prakse u kompaniji „Elsys“ u Beogradu.

Doktorske studije na Elektrotehničkom fakultetu Univerziteta u Beogradu, smer Elektronika, upisala je školske 2007/08. godine. Na studijama je položila sve ispite sa prosečnom ocenom 10,00/10. Tokom studija objavila je jedan rad u međunarodnom časopisu, prikazala je dvanaest radova na međunarodnim konferencijama i četiri rada na domaćim konferencijama. Od tih radova, pet radova na konferenciji Telfor proizašlo je iz istraživanja na predmetima koje je polagala na doktorskim studijama. Za rad

„Design of antenna system for short-range wireless sensor network“, prikazan na konferenciji Telfor 2011, dobila je nagradu „Blažo Mirčevski“ za najbolji rad mladog autora. U neposrednoj vezi sa doktorskom disertacijom su četiri rada iz oblasti projektovanja namenskih programabilnih hardverskih akceleratora: jedan rad objavljen u međunarodnom časopisu i tri rada prikazana na međunarodnim konferencijama, od čega je jedan rad prikazan na jednoj od dve najznačajnije konferencije u ovoj oblasti u svetu.

Član je IEEE udruženja u statusu punopravnog člana. Recenzent je međunarodnog časopisa „ACM Transactions on Design Automation of Electronic Systems“, međunarodne konferencije „Design Automation Conference“ iz iste oblasti kao i doktorska disertacija, kao i konferencije Telfor. Od januara 2007. godine do aprila 2013. godine radila je u Institutu Mihajlo Pupin, na poziciji istraživača i projektanta namenskih računarskih sistema, gde je, između ostalog, učestvovala na međunarodnom istraživačkom projektu „Advancing embedded system research in Serbia“ zajedno sa EPFL u Lozani, Švajcarska. Od maja 2013. godine zaposlena je na University of Applied Sciences and Arts Western Switzerland, u Švajcarskoj, kao naučna saradnica na FP-7 projektu STRUCTURES.

Прилог 1.

Изјава о ауторству

Потписани-а Мирјана Стојиловић

број уписа 5024/07

Изјављујем

да је докторска дисертација под насловом

Метода пројектовања наменских програмабилних хардверских акцелератора

- резултат сопственог истраживачког рада,
- да предложена дисертација у целини ни у деловима није била предложена за добијање било које дипломе према студијским програмима других високошколских установа,
- да су резултати коректно наведени и
- да нисам кршио/ла ауторска права и користио/ла интелектуалну својину других лица.

Потпис докторанта

У Београду, 12.08.2013.

Прилог 2.

Изјава о истоветности штампане и електронске верзије докторског рада

Име и презиме аутора Мирјана Стојиловић

Број уписа 5024/07

Студијски програм ЕЛЕКТРОТЕХНИКА И РАЧУНАРСТВО

Наслов рада Метода пројектовања наменских програмабилних хардверских акцелератора

Ментор др. Лазар Сарановац, ванредни професор

Потписани Стојиловић Мирјана

изјављујем да је штампана верзија мог докторског рада истоветна електронској верзији коју сам предао/ла за објављивање на порталу **Дигиталног репозиторијума Универзитета у Београду**.

Дозвољавам да се објаве моји лични подаци везани за добијање академског звања Доктора наука, као што су име и презиме, година и место рођења и датум одбране рада. Ови лични подаци могу се објавити на мрежним страницама дигиталне библиотеке, у електронском каталогу и у публикацијама Универзитета у Београду.

Потпис докторанта

У Београду, 12.03.2013.

Прилог 3.

Изјава о коришћењу

Овлашћујем Универзитетску библиотеку „Светозар Марковић“ да у Дигитални репозиторијум Универзитета у Београду унесе моју докторску дисертацију под насловом:

Метода пројектовања ~~ћелијских~~ програмабилних хардверских акцелератора

која је моје ауторско дело.

Дисертацију са свим прилозима предао/ла сам у електронском формату погодном за трајно архивирање.

Моју докторску дисертацију похрањену у Дигитални репозиторијум Универзитета у Београду могу да користе сви који поштују одредбе садржане у одабраном типу лиценце Креативне заједнице (Creative Commons) за коју сам се одлучио/ала.

1. Ауторство
2. Ауторство – некомерцијално
3. Ауторство – некомерцијално – без прераде
4. Ауторство – некомерцијално – делити под истим условима
5. Ауторство – без прераде
6. Ауторство – делити под истим условима

(Молимо да заокружите само једну од шест понуђених лиценци, кратак опис лиценци дат је на полеђини листа.)

Потпис докторанта

У Београду, 12.08.2013.