OPTIMIZED IMPLEMENTATION OF REAL-TIME IMAGE PROCESSING ALGORITHMS ON FIELD PROGRAMMABLE GATE ARRAYS

AILTON F. DIAS[1,3], CHRISTOPHE LAVARENNE[2], MOHAMED AKIL[1], YVES SOREL[2]

[1]Groupe ESIEE–Laboratoire LPSI,
B.P. 99, 93162 Noisy-le-Grand, France
diasa@esiee.fr, akilm@esiee.fr

[2]INRIA–Rocquencourt - Projet SOSSO,
B.P. 105, 78153 Le Chesnay Cedex, France
christophe.lavarenne@inria.fr, yves.sorel@inria.fr

[3]CNEN/CDTN–Divisão de Computação e Informação,
Caixa Postal 941, 30123-970 Belo Horizonte, MG, Brasil
diasaf@urano.cdtn.br

Abstract. We present the Algorithm Architecture "Adequation" methodology for the optimized implementation of real-time image processing algorithms on field programmable gate arrays. This methodology is based on a single factorized graphs model, used from the algorithm specification down to the architecture implementation, through optimizations expressed in terms of defactorization transformations. A simple image processing example is presented to illustrate the methodology.

1 Introduction

The Algorithm Architecture Adequation[1] methodology [1] helps a real-time application designer to optimize the implementation of his application algorithm on his multiprocessor target architecture. We extend this methodology to the case of multi-FPGA (Field Programmable Gate Arrays) target architectures applied to the domain of real-time image processing.

Dependence graphs are well suited to describe combinatorial circuits, registers, and their interconnections. We also use dependence graphs to specify applications algorithms. With this unified model, it is easier to transform progressively an algorithm into an architecture, and to design heuristics to perform and optimize this transformation.

The optimization consists in exploring the implementation space, seeking for the most performant solution. The implementation space is composed of all the different defactorizations of the factorized dependence graph specifying the algorithm. Each defactorization specifies a different implementation of the same algorithm, with different characteristics (FPGA area required, latency, data-rate). The optimization goal is to minimize the hardware resources (number of FPGAs cells) while satisfying the real-time constraints (upper bounds on the latency and on the data-rate).

The methodology validation has been carried out over several examples representative of low-level image processing, exhibiting data and operations potential parallelism as well as sequential data dependences. The VHDL [2] hand-coding, compilation, simulation, synthesis, netlist generation and surface-timing estimation of the examples have been carried out with Mentor Graphics [3] and Xilinx [4] software tools. The target architecture chosen for the synthesis of these examples is based on FPGAs Xilinx 4000 Series. Research in progress aims to automatically choose and apply defactorization transformations, and to automatically generate the VHDL code.

2 Factorized Graphs Model

Basically, an algorithm is modeled by a dependence graph (acyclic oriented hypergraph), where each node models an operation (arithmetic or logical or more complex), and each oriented hyperedge models a data produced as output of a node and used as input of other nodes. In image processing, dependence graphs contain lots of repeated patterns (arrays, sometimes multidimensional, of operations using and/or producing arrays of data), which may be specified factorized [5]. Reactive applications are infinitely repetitive: they are modeled by an infinitely repeated pattern, which factorization leads to what is usually called a "data-flow" graph. Graph factorization consists in replacing a repeated pattern by only one instance of the pattern, and in marking each edge crossing the pattern boundary with a special "factoring" node:

[1] "Adequation" is a french word meaning an *efficient* matching, stronger than the english word "adequacy" which involves a *sufficient* matching.

- D for "Diffusion" marks an edge which inputs the same data for every instance of the repeated pattern. An infinite diffusion models a constant, it is a graph source.
- F for "Fork" marks an edge which inputs a different item of a data array for each different instance of the pattern (it indexes the data array). An infinite fork models an infinite array of inputs from the external environment, it is a graph source denoted S for "Sensor".
- J for "Join" marks an edge which outputs a different item of a data array for each different instance of the pattern (it is the opposite of F). An infinite join models an infinite array of outputs to the external environment, it is a graph sink denoted A for "Actuator".
- I for "Iterate" marks an edge between two neighbour instances of the pattern; the first pattern instance inputs the ini data, and the last one outputs the $last$ data. An infinite iterate, also usually called "delay", has no $last$ output.

The architecture is obtained by direct translation of the algorithm graph, by replacing each operation by an operator (by instanciating the corresponding component of a VHDL library), each data dependence by a communication media (VHDL signal) interconnecting operators ports, and each factoring node by the corresponding operator: D just connects its input to its output, F is implemented by a multiplexor, S by a latched analog to digital converter, J by a demultiplexor with a memory array, A by a latched digital to analog converter, and I by a register with initialization path. The F, J and I on the same pattern boundary are sequenced by a common periodic counter, driving the multiplexors selectors, and clocked by the overflow event of inner pattern boundaries counters, or by a basic (external) clock for the innermost boundary counter.

3 Mean Filtering Example

The mean filtering algorithm (3 × 3 window) illustrates the methodology. It has been purposely kept simple for the sake of this illustration.

$$A = \sum_{i=1}^{3}\sum_{j=1}^{3} S_{ij} \qquad (1)$$

In the factorized data-flow graph of figure 1, obtained from eq. (1), S delivers a 3×3 pixel window, FF_1 is the boundary of the factorization of the sum of 3 pixels in a line, FF_2 is the boundary of the factorization of the sum of the 3 previous sums, and A outputs the final sum of the 9 pixels.

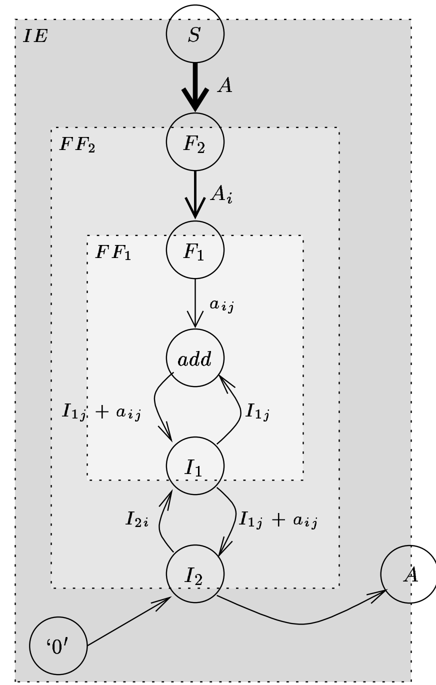

Figure 1: Factorized data-flow graph

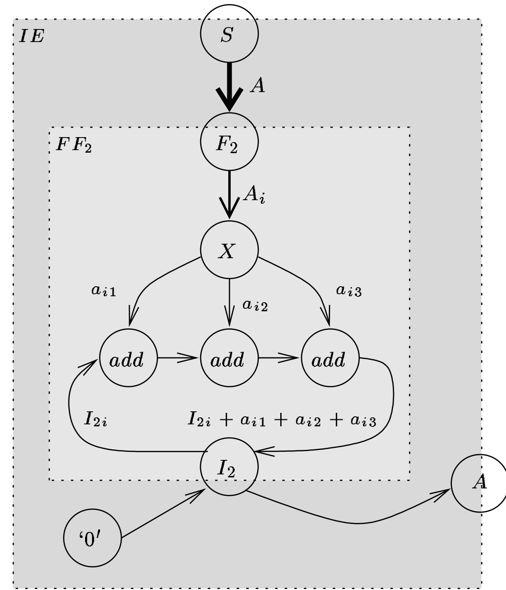

Figure 2: Defactorized by FF_1 data-flow graph

If we defactorize the FF_1 boundary, we obtain the defactorized data-flow graph shown by figure 2, where F_1 has been replaced by X (an array-decomposition operation corresponding to an operator which simply separates its array

input into its elements, one for each of its outputs), and I_1 has been replaced by the dependences between the 3 *add* operations.

We can also defactorize the FF_2 boundary. In this case, we obtain the defactorized data-flow graph shown by figure 3, where F_2 has been replaced by X (an array-decomposition operation that separates the pixel window into lines) and the FF_1 boundary has been repeated 3 times. Finally, we can defactorize both FF_1 and FF_2 boundaries to obtain a fully defactorized data-flow graph, shown by figure 4.

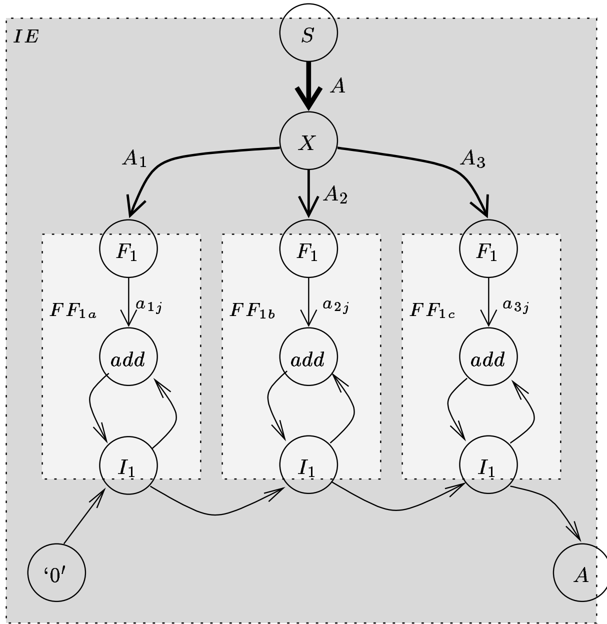

Figure 3: Defactorized by FF_2 data-flow graph

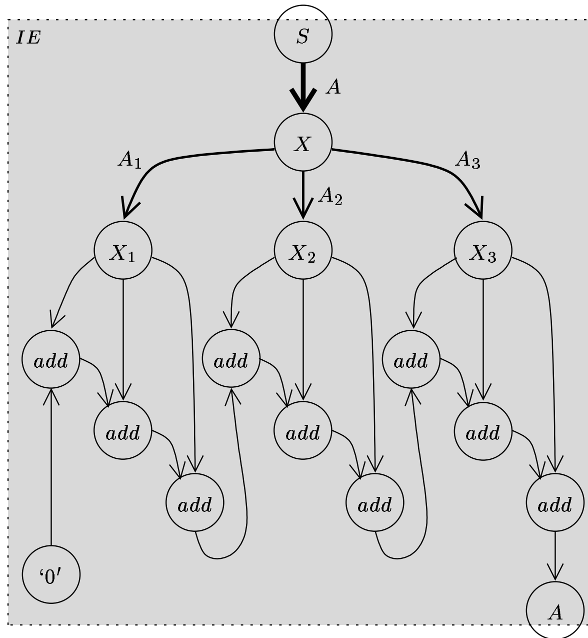

Figure 4: Fully defactorized data-flow graph

Table 1 shows the implementation results of a mean filtering using a 3×3 window with pixels coded on 8 bits. The algorithm graph was implemented (simulated and synthesized) on 4025ePG299-3 Xilinx FPGAs by using the following tools: *QuickVHDL* simulator and *Xilinx Design Manager*, version M1.3.7. Implementation results are given for the area (number of FPGA CLBs - Configurable Logic Blocks), the number of cycles required for the algorithm execution, the maximum operation frequency and data latency.

Table 1: Implementation results

Implementation	*Area (CLB)*	*Nb. Cyc*	*Freq. (MHz)*	*Lat. (ns)*
Specification	128	21	13.8	1524
FF_1 defact.	89	5	11,6	433
FF_2 defact.	137	15	18,1	830
Fully defact.	47	1	11,0	91

Figure 5 shows the hardware resources (number of used CLBs) plotted versus the *latency*. Units are relative to the initial data-flow factorized graph specification, that corresponds to 100% hardware resources, 100% latency. For a real-time constraint bounding the latency to 50%, the solution is the full defactorization which satisfies the real-time constraint and minimizes the hardware resources required.

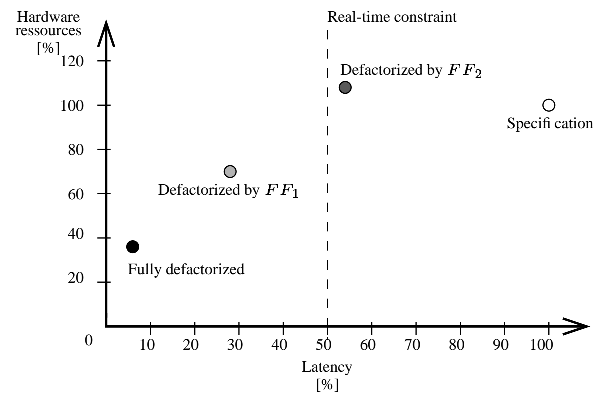

Figure 5: Number of CLBs versus latency

In this example, hardware resources required by the repetitive pattern (an adder) are small compared with the hardware resources required by the special "factoring" nodes (Fork, Iterate). Then, we observe that two defactorized implementations (FF_1 and fully defactorized) require less hardware resources than the initial specification. If we add a multiplier to the repetitive pattern to multiply each pixel value by a constant, we observe that defactorized solutions reduce the latency, but all of them increase the number of hardware resources, because the multiplier requires more hardware resources than the special "factoring" nodes.

4 Conclusions

All graph transformations were done by hand, it means that we started from a factorized graph specification and we transformed it by hand to obtain different defactorized graphs. Hardware graphs were obtained by simple substitution of operation nodes in the algorithm graph by the equivalent operator nodes. Each hardware graph operator corresponds to a VHDL library component. A specific VHDL library has been developped for the factoring nodes.

The defactorization of an algorithm specification allows us to move inside the implementation space. When we defactorize a graph, we expect to reduce the latency by increasing the number of hardware resources, but there is a tradeoff between hardware resources required by computing nodes and special "factoring" nodes. If the hardware resources required by computing nodes belonging to a repetitive pattern are smaller than the hardware resources required by the special "factoring" nodes, we can obtain defactorized solutions that reduce simultaneously hardware resources and latency. We validated the methodology on different simple algorithms (scalar product, vector product, matrix-vector product) in order to study all factorization possibilities.

5 Future work

The main goal of this research is to obtain, from a single factorized specification, an optimized FPGA implementation, by exploring only a small subset of all possible transformations into the implementation space. An implementation is obtained by successive defactorizations of the factorized specification. From our experience of manual transformations, we intend to automate the choice and the processing of these transformations.

This automation will be based on heuristics using a cost function to compare the performances of different defactorizations of the specification. This cost function must include the evaluation of at least three characteristics of an implementation: hardware resources required (number of CLBs), latency and data-rate. Hence, each operator must be characterized in function of the above parameters, for each different target architecture (FPGA model or series). By composition of these operator characteristics, both the time and area performances of an implementation may be estimated.

In the mean filtering example, boundary are totally defactorized because the number of pattern repetitions is too small. Applications with much more pattern repetitions may also be defactorized only partially. Transformation heuristics must be able to manage partial defactorizations.

Finally, we intend to develop a tool for FPGA based architectures, offering automatic partial defactorizations driven by optimization heuristics, and generating VHDL to code the resulting implementation.

6 Acknowledgements

This research is supported by *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior*–CAPES (grant 0100-95/13) and *Comissão Nacional de Energia Nuclear*–CNEN.

References

[1] Y. Sorel. *Massively Parallel Computing Systems with Real-Time Constraints: the "Algorithm Architecture Adequation" methodology.* Proc. of Massively Parallel Computing Systems, Ischia Italy, May 1994.

[2] R. Airiau, J.-M. Bergé, V. Olive, J. Rouillard. *VHDL du langage à la modélisation.* Lausanne : Presses polytechniques et universitaires romandes, 1990.

[3] Mentor Graphics. *QuickVHDL user's and reference manual - software version 8.4_4.* Mentor Graphics Co., 1995.

[4] Xilinx. *The programmable logic data book.* San Jose : Xilinx, Inc., 1994.

[5] C. Lavarenne, Y. Sorel. Modèle unifié pour la conception conjointe logiciel-matériel. *Traitement du Signal*, vol. 14, n. 6, 1997.