High-Level Synthesis by Systematic Derivation of Vision Automata from Emulation Results

Ivan C. Kraljić, Georges M. Quénot* and Bertrand Zavidovique
Laboratoire Système de Perception
DGA/Etablissement Technique Central de l'Armement
16 bis avenue Prieur de la Côte d'Or, 94114 ARCUEIL Cedex FRANCE
Email: ik@etca.fr

Abstract

We present a high-level synthesis method called derivation from emulation results. A vision algorithm is first emulated in real time on a general image processing computer, then a chipset implementing the algorithm is automatically derived from the computer's configuration through multiple optimizations. The optimizations' correctness guarantees that the chipset will perform exactly as during emulation. A first version of the derivation software implementing low level optimizations has been developed; its output is a structural VHDL description. We present our first results and discuss our current research.

1 Introduction

High-level synthesis takes as input an algorithm expressed in a high-level language and generates an RTL netlist implementing the algorithm according to explicit constraints. The resulting netlist must be thoroughly simulated in order to validate its behavior. The increasing interest in rapid prototyping, formal verification or FPGA-based emulation indicates that simulation does not cover all the aspects of design validation. When real-time vision applications are concerned, the specification of the application is vague and imprecise and the enormous amount of input data requires weeks of traditional simulation. Furthermore, the application should be validated in its environment (a problem seldom addressed by simulation).

We propose an approach called *derivation from emulation results* where the validation step precedes the synthesis step: the algorithm is firstly emulated in its environment (with input from video cameras) on a general image processing computer (the Data-Flow Functional Computer), then the actual computer configuration is analyzed, optimized and finally implemented as a VLSI chipset. The RTL netlist implementing the algorithm is thus directly derived from the hardware which emulated the algorithm. Contrary to high-level synthesis where the synthesis process is separated from the validation, *emulation provides simultaneously the synthesis of an RTL netlist and its validation.* Advantages are twofold:

- Total equivalence of the derived RTL netlist's behavior with the results obtained during emulation (functional behavior and performance).

*G.M. Quénot is now with Hyperparallel Technologies, Ecole Polytechnique, X-Pole, 91128 Palaiseau Cedex, FRANCE. Email: quenot@hyperparallel.polytechnique.fr

- The post-emulation "synthesis" step is reduced to the optimization of the emulator's active resources.

The emulation step as well as the Data-Flow Functional Computer (DFFC) emulator have already been extensively described [5, 8]. Figure 1 shows the emulation flow. A vision algorithm is

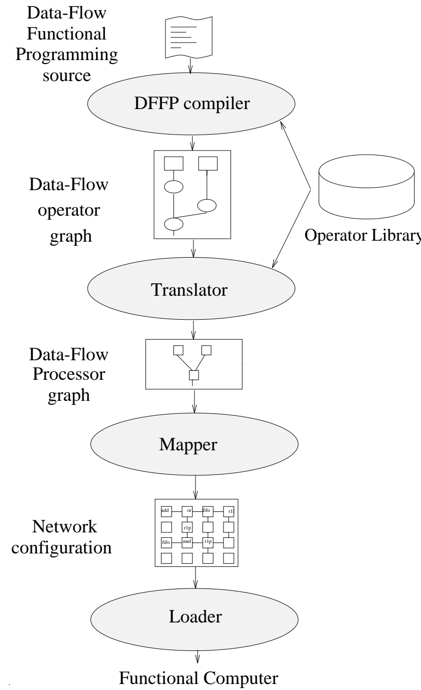


Figure 1: Emulation flow

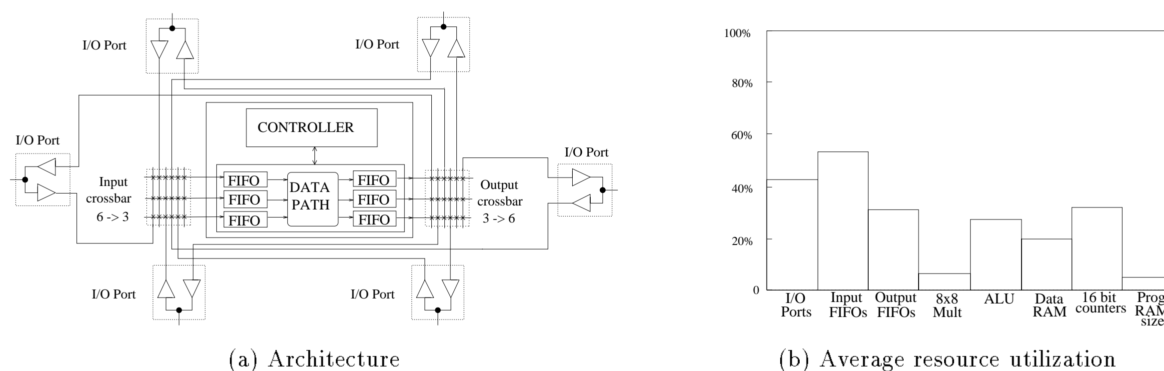
first expressed in a functional programming (FP) language, then the FP source is compiled into a data-flow graph (DFG). Each DFG node is translated into a corresponding (correctly configured) Data-Flow Processor (DFP) taken from an operator database (200 elements). The DFP graph is finally mapped on the DFFC which is an $8 \times 8 \times 16$ array of 1024 DFPs [7]. The algorithm is executed in real time on the DFFC; input flows come from B/W and color cameras, and output flows are sent to monitors. Figure 2a summarizes the Data-Flow Processor's architecture [6]. The datapath is a programmable 3-stages pipelined datapath containing 3 input FIFO queues and 3 output queues, an 8×8-bit multiplier, a 16-bit 2901-type ALU, a 256×9 bits RAM and a 16-bit counter. The datapath is configured through a programmable controller (a 64×32 bits program is provided for its description). The DFP architecture has been designed to implement efficiently a wide range of low-level image processing operators (arithmetic/logic operations, 8/16-bit histogrammer, line/pixel delays, line/column sums).

2 The derivation concept

The actual DFFC configuration implementing the algorithm is used as an input to the derivation process. A first approach is to replace the processors used only for routing by direct connections and the processors used as FIFOs by commercial FIFOs, and to keep other operating processors. This could result in an automaton implementable as a board or Multi-Chip Module (MCM). A second approach is based on a processor-level analysis: identification of the effectively used resources and optimization of these resources. The resulting automaton can be synthesized as a

VLSI chipset. This paper describes the second approach: our aim is to synthesize a VLSI chipset implementing a vision algorithm from the results of its emulation on a vision computer. The chips architecture is pipelined FSMD (Finite State Machine with Datapath).

The VLSI approach to derivation is highly facilitated by 1) the granularity of the Data-Flow Processor which is at the 8/16-bit operator level and 2) the relative simplicity of the DFP architecture which allows a deterministic knowledge of whether a resource is used or not. Figure 2b shows the average (static) resource utilization for several non-trivial algorithms (3×3 convolution, Nagao-like filter, erosion, default detector...). While it could seem at first sight that one could not reduce a graph of several hundreds processors into a few chips, figure 2b shows clearly that the resource utilization of the presented algorithms is low. This means that a simple resource identification (i.e. removal of the unused resources) would bring dramatic size reduction of the RTL netlist. Further improvements will be achieved through optimizations of this RTL netlist. Derivation is similar to the turn of an FPGA implementation into an ASIC (retargeting): the conversion of the design to a different technology (often a gate-array type) results in a smaller and faster implementation. Thanks to the coarse grain architecture of the Data-Flow Processor derivation is far more easily automated.



(a) Architecture

(b) Average resource utilization

Figure 2: The Data-Flow Processor

In accordance to the emulation flow steps, 3 different representations for an algorithm are available (from high to low level of abstraction): 1) data-flow graph, 2) Finite State Machine (FSM) network and 3) RTL netlist. Whereas most traditional high-level synthesis systems use the data-flow representation (e.g., [9]), derivation will use the other 2 representations: low level optimizations are performed on the RTL netlist and high level transformations are performed on the FSM representation. (Note that more and more high-level synthesis systems perform post-synthesis optimizations on the synthesized RTL netlist [2, 4]). The block diagram of the derivation process is shown in figure 3.

3 The low level optimizations

These optimizations are performed on the RTL netlist with input from the Finite State Machine definition of each processor. They are applied to each processor and to connections between adjacent processors. The relative areas of the Data-Flow Processor's main architectural elements are

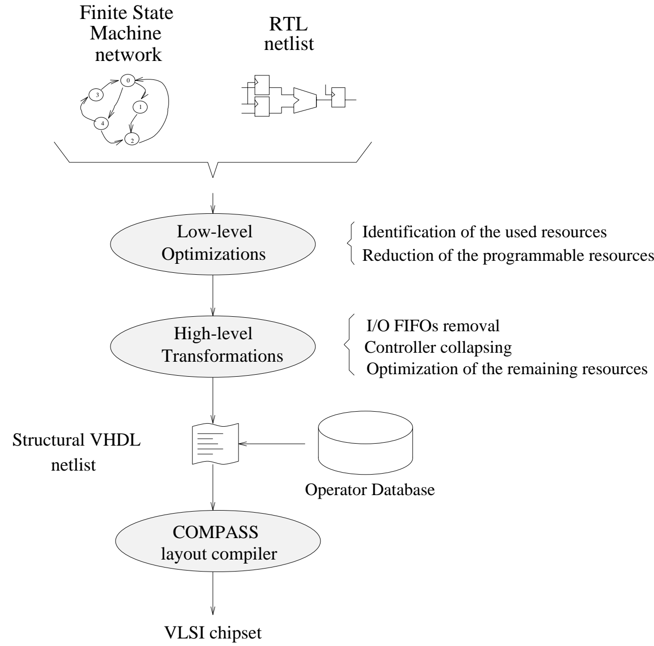


Figure 3: Derivation flow

displayed in table 1. Obviously the programmable elements (I/O ports, crossbar and controller) are *extremely costly*; section 3.2 explains how we deal with them.

DFP element	Relative area
I/O ports and crossbar	26.2%
Controller	24.0%
Datapath	13.6%
I/O FIFOs	10.5%
DATA RAM	4.5%

Table 1: Relative areas of the DFP's main architectural elements

3.1 Removal of the unused resources

An analysis of the static and dynamic (FSM) definitions of the DFP operator determines precisely whether a datapath resource (ALU, MUX, FIFO...) is used or not. A resource is unused if it produces no output flow or if its output flow is equal to its input flow or if it produces a constant flow. Unused resources are removed or replaced by buses while used resources are left untouched.

3.2 Reduction of the programmable resources

Reducing a programmable resource (controller, crossbar or I/O port) means "freezing" it to its specific function in each DFP operator. The controller which was implemented with a 64×32 program RAM can be implemented with a ROM, PLA or in standard cells. The I/O ports of a DFP were connected to its FIFOs through a crossbar; in a derived DFP the (fixed) connections are implemented statically. This immediate optimization step brings huge area reduction.

4 The high level transformations

Our current research aims at further optimizing the network of derived DFPs through high level transformations. Our goal is to transform the network from a local control (to each DFP) to a global control. Intuitively a global control is less costly than a local control, especially when arithmetic operators (adders, shifters...) are concerned.

4.1 Internal I/O FIFOs removal

Apart from the immediate reduction in area, this transformation has other purposes: 1) it facilitates the controller collapsing and 2) it allows a final datapath optimization. The goal is to replace the I/O FIFOs between adjacent DFPs by registers; this could be done by synchronizing the DFP Finite State Machines.

4.2 Controller collapsing

The controller collapsing will be done by the well-known technique of generating the cartesian product (with removal of redundant steps) of the processor's finite state machines (see for example [10]). All the controllers will not necessarily be collapsed, the state explosion must obviously be avoided. Furthermore it has been shown that single FSMs are not necessarily less costly than their multiple-FSM equivalent [1].

4.3 Optimization of the remaining resources

This step will implement an automatic operator selection for each datapath element in order to minimize the cost (mainly area) of the pipelined design. A database of datapath elements (slow, fast...) will be provided.

5 First results

We have implemented a first version of the derivation software performing the first 2 low level optimizations. We use the COMPASS tools to synthesize the chips. Each derived circuit is composed of "mini" specific DFPs: adders, FIFOs, delays... which function exactly as a normal DFP. Results of derived DFPs are shown in table 2. Currently only the I/O FIFOs are implemented using COMPASS's Datapath Compiler, while the rest is implemented in standard cells. Subsequent use of COMPASS's Datapath Compiler will further reduce the datapath's area [3]. As expected, the I/O FIFOs occupy a huge part (about 43%) of the derived processors area. With the 1μm technology used, we can pack about 20 derived processors on a chip. A 0.6μm technology would raise this number to 50.

Operator	Initial DFP	abs	add	and	max	256-word FIFO	Pixel delay	Line delay	Histogrammer
Size (mm^2)	35.64	2.6	4.8	3.6	3.9	4.9	4.1	3.9	7.1
Controller (mm^2)	8.54	0.18	0.27	0.16	0.17	0.08	0.55	0.46	0.54
Datapath (mm^2)	4.85	0.76	1.30	0.94	1.38	1.09	1.25	1.00	1.94
Controller/Datapath	1.76	0.24	0.21	0.17	0.12	0.07	0.43	0.47	0.28
FIFO/Operator	0.11	0.48	0.52	0.50	0.47	0.25	0.45	0.47	0.26

Table 2: Initial and derived Data-Flow Processors in a 1μm technology

An edge detector (figure 4) with 11 processors has been successfully implemented in a 1μm technology. The chip contains about 23,000 gates in 73.5 mm^2 (core size 55.7mm^2). This is obviously larger than a handmade implementation, however we insist that *no actual optimization has been involved in the derivation process as yet.* We expect further optimizations to reduce the edge detector's size by at least 60 %.

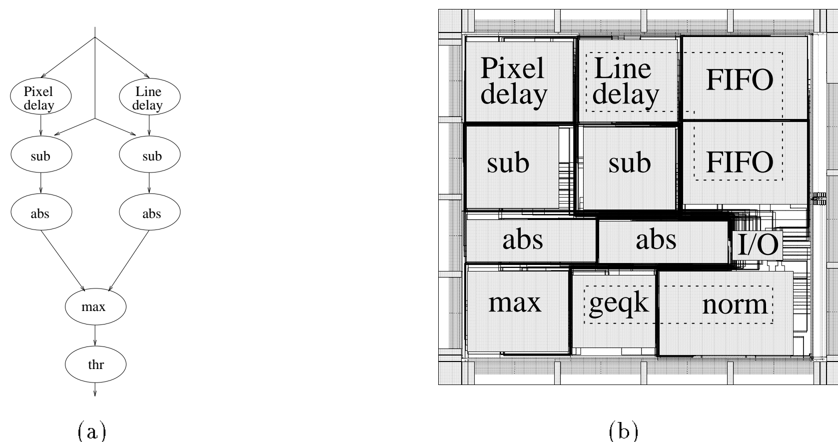


Figure 4: The data-flow graph of the edge detector (a) and its floorplan (b)

6 Conclusion

We have presented an approach to high-level synthesis that solves the design validation issue. Our method called emulation-derivation exploits the results of the emulation of an algorithm in real time and in its environment. The emulation step provides simultaneously the synthesis of an RTL netlist (the emulator's active resources) and its validation; then the derivation step optimizes this RTL netlist and a commercial layout compiler is used to synthesize a chipset implementing the algorithm.

We have implemented the derivation concept for real-time vision applications: a general vision computer has been built (the 1024-processor Data-Flow Functional Computer), the emulation software takes as input a functional programming language source file and maps it on the Computer. A first version of the derivation software implementing low level optimizations has been developed. Our first results are promising (a reduction of more than 90% in area), we expect a further gain of at least 50%.

References

[1] P. Ashar, S. Devadas, and A.R. Newton. Optimum and Heuristic Algorithms for an Approach to Finite State Machine Decomposition. *IEEE Transactions on Computer-Aided Design*, 10(3):296–310, March 1991.

[2] Hans Eveking and Stefan Höreth. Optimization and Resynthesis of Complex Data-Paths. In *Proc. of the 30th ACM/IEEE Design Automation Conference*, pages 637–641, 1993.

[3] R. Leveugle and C Safinia. Generation of optimized datapaths: bit-slice versus standard cells. In G. Saucier and J. Trilhe, editors, *Synthesis for Control Dominated Circuits (IFIP Transactions A-22)*, pages 153–166. Elsevier, 1993.

[4] Christos Papachristou, Haidar Harmanani, and Mehrdad Nourani. An Approach for Redesigning in Data Path Synthesis. In *Proc. of the 30th ACM/IEEE Design Automation Conference*, pages 419–423, 1993.

[5] G.M. Quénot, I.C. Kraljić, J. Sérot, and B. Zavidovique. A Reconfigurable Compute Engine for Real-Time Vision Automata Prototyping. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 91–100, April 1994. Napa, CA, USA.

[6] G.M. Quénot and B. Zavidovique. A data-flow processor for low-level real-time image processing. In *IEEE Custom Integrated Circuit Conference*, May 1991. San Diego, CA, USA.

[7] G.M. Quénot and B. Zavidovique. The ETCA massively parallel data-flow functional computer for real-time image processing. In *IEEE International Conference on Computer Design*, Oct 1992. Cambridge, MA.

[8] J. Sérot, G.M. Quénot, and B. Zavidovique. Functional Programming on a Data-flow Architecture: Applications in Real-Time Image Processing. *International Journal of Machine Vision and Applications*, pages 44–56, May 1993.

[9] F. Verdier and B. Zavidovique. A Complete Environment for Global Architecture Synthesis. In *Proceedings of Computer Architectures for Machine Perception, New Orleans, USA*, pages 77–81, December 1993.

[10] Wayne Wolf. An Algorithm for Nearly-Minimal Collapsing of Finite-State Machine Networks. In *Proc. of ICCAD*, pages 80–83, 1990.