Investigating Real-Time Validation of Real-Time Image Processing ASICs*

Ivan C. Kraljić[1], François S. Verdier[2],
Georges M. Quénot[3] and Bertrand Zavidovique[4]

[1] École Polytechnique de Montréal - CP 6079, Montréal (Québec) H3C 3A7 Canada
[2] IUT de CERGY/GRSC, Université de Cergy-Pontoise, 95014 Cergy France
[3] LIMSI-CNRS, BP133, 91403 Orsay France
[4] Institut d'Électronique Fondamentale, Université de Paris XI, 91405 Orsay France
Email: kraljic@grm94.polymtl.ca, verdier@u-cergy.fr,
quenot@limsi.fr, zavido@ief-paris-sud.fr,

Abstract

The research presented in this paper aims at designing real-time image processing Application Specific Integrated Circuits (ASICs), with emphasis on the need for correct circuits. The methodology is based on a dedicated emulator, the Data-Flow Functional Computer (DFFC), whose peak capacity is 20 million gates operating at 25 MHz. Applications are firstly validated in their target environment (real time, real-world scenes) during emulation on the DFFC. Two integration methods have been implemented: derivation and synthesis. The derivation method optimizes the architecture validated on the emulator, while the synthesis approach is not constrained by the emulator architecture, and thus allows to generate other (optimized) architectures.

1 Introduction

As long as specification and design of image processing algorithms will invoke heuristics and experimentations, ASIC validation at each step of the design flow (from algorithm down to VLSI layout) will be the main difficulty. The problem lies in the huge amount of input data (45 Mbytes/s. for color video) consumed in real-time operation. Moreover, if autonomous robot control systems are concerned (with strong dimensional constraints) the validation would be done under final environment-specific conditions. The validation of real-time image processing ASICs by simulation is then at best extremely time-consuming, at worst prohibitive.

A possible way to cope with this problem is to use an emulator [1], i.e. a set of reprogrammable hardware which can be configured by software to implement the ASIC as accurately as possible, at register-transfer (RT) or gate levels. Generic emulators are based on Field-Programmable Gate Arrays (FPGAs) using the static RAM technology (e.g. SPLASH-2 [2], PARTS [3]). The low granularity of FPGAs (only a small number of gates per logic block) limits their use for *systematic emulation* of real-time image processing applications: time consuming FPGA placement/routing, application- and implementation-dependent operating frequency, low observability and controllability. Some significant image processing applications have nonetheless been successfully implemented on FPGA-based emulators [2], [3].

Other emulators are dedicated to specific applications and hence exhibit higher performance: Grape-II [4] and the Digital Signal Multiprocessor [5] (both based on DSP processors) are dedicated to emulating real-time Digital Signal Processing systems.

This paper presents a framework for designing image processing VLSI circuits based on an emulator dedicated to processing digital pixel streams on the fly: the Data-Flow Functional Computer. All the design tools are included in the framework: behavioral specification, algorithm validation and ASIC integration. Two independent integration processes have been explored: derivation from structural (RT-level) specification and synthesis from behavioral (data-

*This work was done while authors were at Laboratoire Système de Perception - DGA/Établissement Technique Central de l'Armement - 16 bis avenue Prieur de la Côte d'Or, 94114 Arcueil France.

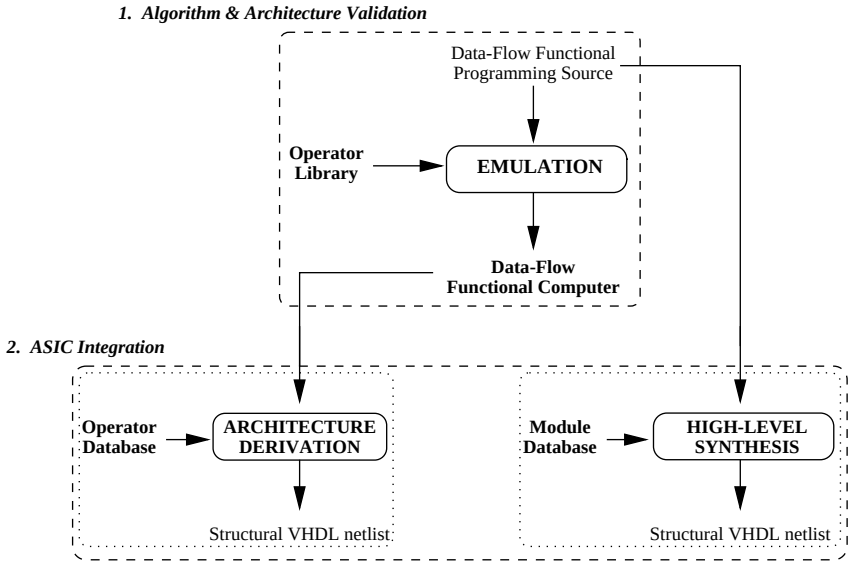


Figure 1: Design flowchart

flow graph) specification. By using the functional paradigm in specifying algorithms [6], the methodology is adapted to the intuitive and informal design of image processing applications. The design flowchart is described in figure 3. Note that the two integration processes (derivation and synthesis) are currently independent.

2 Real-time emulation

The Data-Flow Functional Computer (DFFC) is the emulator dedicated to processing digital pixel streams on the fly [6]. "Functional" refers to the DFFC programming model, while "Data-Flow" refers to its execution model: wired data-flow. The data-flow model specifies that a function is executed as soon as all the data required at its inputs are present [7]. Wired data-flow architectures implement the data-flow graph physically: each node is a physical operator, each edge is a physical connection. Although limited in the algorithms they can implement (e.g. recursive functions and loops are difficult or impossible to implement), wired data-flow architectures exhibit outstanding performance in digital signal processing and image processing. Some wired data-flow architectures have been built, e.g. [8], [9].

The DFFC core is a 3D 8×8×16 array of 1024 custom Data-Flow Processors (DFPs) achieving a peak computing power of 50 giga operations per second (GOPS). The architecture of the DFFC is summarized in figure 3. Each Data-Flow Processor (see figure 2) contains a programmable 3-stage pipelined datapath containing an 8×8-bit multiplier, a 16-bit 2901-type ALU, a 256×9-bit RAM and a 16-bit counter. The datapath is controlled by a programmable state machine (a 64×32 bits program is provided for its description) and a static configuration register. Three input FIFO queues and three output queues are dedicated to data flow management. Two crossbars allow to connect a FIFO to any one of the six I/O ports.

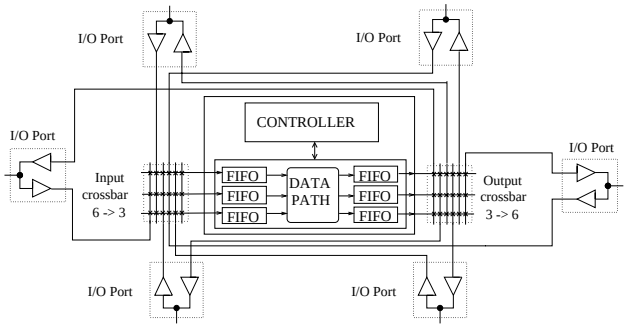


Figure 2: The Data-Flow Processor

A chip containing 2 Data-Flow Processors has been fabricated in a 1.0 μm CMOS technology. A single Data-Flow Processor can implement a wide range of low-level image processing operators, e.g. arithmetic/logic operations, counters, comparators, line/pixel shifts, 256-word FIFO, line/column sums, and histograms.

The DFFC's inputs/outputs are B/W and color cameras/monitors and low-bandwidth data-flow I/O ports. A SPARC 2 workstation is the host for the DFFC's programming environment. The computer operates with the digital video pixel clock. The 25 MHz maximum operating frequency is not a flip-flop toggle limit but the actual speed at which every part of the DFP – and the whole DFFC – can operate whatever the complexity of the operator assigned to a DFP.

The emulation flowchart is depicted in figure 3. The image processing algorithm to be validated is expressed in a functional programming language which is automatically compiled and translated into a DFP graph using operators from a database containing 200 operators, from single-DFP to multiple-DFP macro-operators (e.g. convolvers, filters). The DFP graph

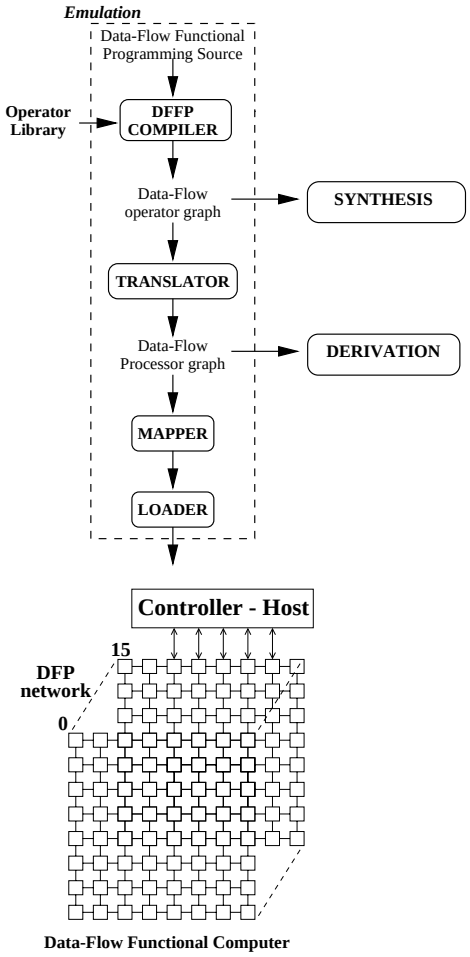


Figure 3: Emulation flowchart

is mapped into a DFFC network configuration ready to be loaded. The mapping (placement/routing of each DFP/macro-DFP operator on the 3D mesh) can be done automatically for graphs with less than 100 DFPs. Larger graphs have to be placed and routed manually using an interactive graphic tool. (The automatic tool performing the mapping is currently being upgraded so as to support larger graphs). A complete 1024 DFPs configuration is loaded in about 15 seconds. The algorithm is executed in real time using the CCD cameras as input flows and sending the output flows to video monitors. Debugging is done by executing the algorithm step-by-step using the low-bandwidth interfaces and the possibility to access any DFP register as well as the contents of the RAMs and FIFOs from the host.

An operational prototype is available very early in the design cycle. This is done thanks to: 1) the functional specification of the algorithm and, 2) to the RT-level granularity of the emulator. There is no need of a complex gate-level description during the prototyping phase.

Benefits of DFFC-based emulation include efficient prototyping and painless debugging. Algorithms are fully validated thanks to the real time execution allowing to use as many input images or sequences as desired, and to the total observability and controllability of the emulator. Furthermore, the prototype can be tested with any kind of input sequences in the target environment of the ASICs. The main limitation is that only low- to mid-level image processing applications can be targeted. Several significant applications have nonetheless been successfully developed, including connected component labeling, colored object tracking, defect detection as well as a tool for interactive satellite image analysis.

3 ASIC integration

The emulation process yields a validated multi-level description of the design: at the highest abstraction level the design is represented by a data-flow graph (DFG), then it is defined as a network of DFP finite-state machines (state transition graphs), finally it is specified by a Register-Transfer/gate level netlist. Any of these independent specifications (or a mix thereof) can be used to generate a set of ASICs. The traditional methods include retargeting from the RT/gate level netlist into a VLSI technology, and high level synthesis from the behavioral specification. The retargeting process generates an architecture equivalent to the architecture of the emulator, but in another VLSI technology. The synthesis process can generate

a whole range of architectures depending on a set of user-supplied constraints, e.g. frequency, area.

Two integration processes have been implemented in the DFFC-based framework. The *derivation from emulation results* process is similar to retargeting, it includes RT-level and DFP-level optimizations in order to reduce the area of derived chips. The *high level synthesis* process is a standard synthesis process, it includes an optimization based on the regularity of control data-flow graphs.

3.1 Derivation from emulation results

During emulation, an architecture implementing the algorithm in real time and on real-life scenes is exhibited and validated. The aim of derivation from emulation results is to exploit this validated architecture: rather than (re-)synthesizing a new architecture from the validated behavioral specification of the algorithm, *derivation analyzes and optimizes the validated structural specification at the register-transfer level* [10]. The derivation from emulation results process benefits from the coarse granularity of the emulator and the functional decomposition paradigm. The state space is restricted by choosing to target VLSI circuits architecturally equivalent to the emulator. The derivation process consists in optimizing the validated Register-Transfer level description of the design in order to obtain a reasonably cheap (i.e. small die size) integrated design in the minimum amount of time. Higher level descriptions of the design are used to implement the optimizations.

Input to the derivation process is the file describing the DFP graph. This file contains 1) the finite-state machine definition of each DFP involved and 2) the connections between the DFPs. The derivation software builds internally a Register-Transfer level netlist of the DFP graph using a generic model of the Data-Flow Processor (also at the RT-level). This netlist is not fully flattened: the intrinsic DFP-level granularity of the netlist is preserved. Optimizations are thus defined at the DFP-level and they are performed on this RT-level specification of the design. The optimizations shall fully preserve the architectural model of the Data-Flow Functional Computer (e.g. there will be no resource sharing). The derivation flowchart is presented in figure 1.

3.1.1 DFP-level optimizations

DFP-level optimizations reduce each DFP to its minimal equivalent RT/gate level implementation by analyzing the state transition graph. The knowledge

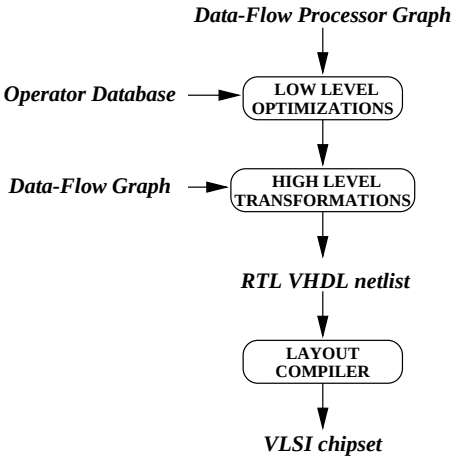


Figure 4: Derivation flowchart

required to implement the optimizations is, for each DFP operator, its state transition graph. There are four optimizations:

1. The *removal of unused datapath resources* is performed by analyzing the static programming register and the finite state machine description of the DFP operator. The analysis identifies an active sub-path in the configurable datapath (the data will flow only through this sub-path). The resources on this active sub-path are the only ones to be kept. For example, the active sub-path of the addition operator contains only an adder, two input FIFOs, and one (8-bit) or two (16-bit) output FIFOs.

2. *Controller reduction.* The derivation software generates an optimized gate-level netlist implementing the logic equations representing the state transition graph of each DFP operator.

3. *I/O ports reduction.* The I/O ports reduction consists mainly in replacing the crossbars (between FIFOs and I/O ports) by static connections implementing only the needed (application dependent) communication links.

4. *Optimization of the actually used resources.* The 16-bit wide datapath resources are reduced to a width of 8-bit if the 8 most-significant bits are not required. If necessary, the bitwidth of the whole datapath is reduced to any user-defined value (e.g. operators on binary images). The critical path of DFPs featuring the optimizations defined above is a fraction of the full DFP's one (considering a technology at least equal to that of

the 1.0 μm DFP). Hence, the DFP pipeline can be safely removed from derived DFPs.

At this point in the derivation process the design is represented by a graph of "data-flow specific processors", e.g. data-flow adders, data-flow line delays, data-flow histogramers. They still include costly flow-management resources: the I/O FIFOs account for about 50% of the total area (the area of derived I/O ports is negligible; they contain but a few gates). These resources are dealt with through DFP graph-level transformations.

3.1.2 DFP graph-level optimizations

The flow management performed by the I/O FIFOs is a characteristic of the Data-Flow Processor graph: The whole DFP graph has to be considered if one wants to optimize the flow-management resources.

The I/O FIFOs have two functions: the first one is to replicate data flows between one father and its sons so that each replicated data flow can be processed independently. This task involves the I/O ports. This is not strictly a functional task (it does not appear in the algorithm). The second task of the FIFOs is to implement data-flow delay in a DFP, i.e. to hold a piece of data for more than one cycle (pixel delay). This task is explicitly defined in the algorithms.

The I/O FIFOs that do not perform any explicit data-flow delay are automatically replaced by pipeline registers where there is no actual flow delay between neighboring DFPs. An average of 50% of all the I/O FIFOs are removed by this optimization, reducing the gate count by 20 to 40% [11]. The FIFOs that perform a true flow delay (e.g. pixel delay) cannot be removed. Their depths are nonetheless reduced to the minimum. For input FIFOs, the minimum depth is equal to the number of pixel delays plus 2. For output FIFOs, the minimum depth is 2.

3.1.3 Results

The output of derivation is a VHDL netlist at the register-transfer level which is fed into COMPASS' low-level tools (RT to gate level synthesis, floorplanning, place and route). The derived circuits have been implemented in 1.0, 0.8 or 0.5 μm CMOS standard cells technologies. The average gate count for a single derived DFP is 1K gates, for an average area of 1 mm^2 in a 1.0 μm process. Derived circuits are simulated using at most a few hundred input vectors, thanks to the fact that they are optimized versions of validated circuits.

3.2 High level synthesis

By using only the validated behavioral specification of the algorithm as input instead of exploiting the emulator architecture, the synthesis tool ALPHA [12] can be considered more general. By freeing itself from the emulator architecture, the synthesis process can generate other architectures, possibly exhibiting higher performance or lower cost.

The input representation for the synthesis process is the data flow graph translated from the functional specification.

Synthesis optimizes the potential parallelism between the operations of an algorithm and builds a VLSI circuit with a reduced amount of computing resources. The target architectural model is a fully synchronized machine with one global control part. High speed pipelined architectures as well as highly parallel non-pipelined architectures can be synthesized. In order to reach near-optimal solutions the synthesis tool implements a minimization of control complexity during the very optimization of the datapath.

Furthermore, the synthesis method is able to handle hierarchy in the sense that a complex image processing algorithm (which is expressed as a map product of simpler functions) can be synthesized hierarchically. Small parts of the final architecture are synthesized independently first, included in a global VLSI library and then used for the whole chip design as a simple operative module. The overall synthesis flow is as follows (figure 5):

1. The first step consists in reading the data-flow graph specification, then propagating data types through the graph (only primary inputs and outputs are explicitly typed in a functional program) and building the control-flow graph. The resulting hybrid representation is a Control-Data-Flow graph where each vertex represents a basic operation (logic, arithmetic) or a complete sub-graph (when hierarchy is used) and edges denotes data or control dependencies.

2. The main RT-level synthesis step involves two stochastic optimization algorithms (Simulated Annealing): scheduling and binding. The scheduling step optimizes the allocation of a time stamp for each operation in the graph. Module selection is achieved simultaneously with scheduling. Moreover, by using a specially-designed quality measure in the cost function (the CDFG regularity), the control part is also optimized. The second algorithm performs simultaneously oper-

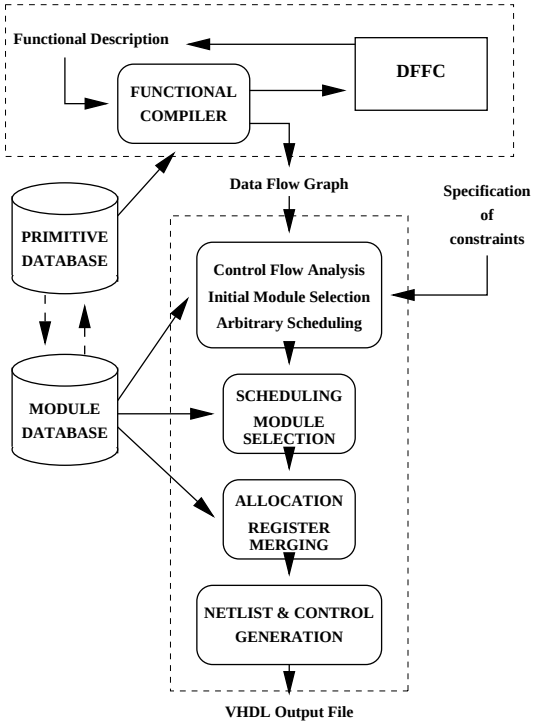


Figure 5: The ALPHA synthesis system

ator binding (allocation of functional unit instances to operations in the graph) and transfer resources minimization (registers and multiplexors).

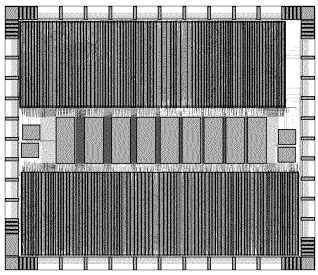
The last step consists in creating the VHDL netlist of RTL components and generating the signal level description of the finite state machine controlling the architecture. These two VHDL descriptions are then fed to COMPASS' gate-level and layout synthesis tools.

4 Results and discussion

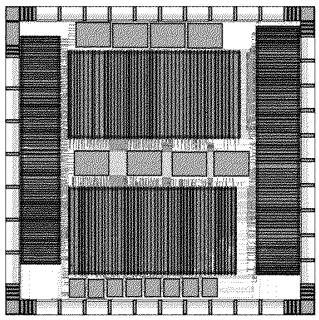
This section presents and discusses results. The functional programming sources of the algorithms are presented in appendix A.

4.1 Derived and synthesized circuits

Table 1 presents derived circuits for three benchmark applications. The "Labeler" (13 instructions/pixel) is the first pass of connected component labeling (i.e. label generation without resolution of equivalent labels). The "Target tracker" (45 instructions/pixel) implements a simple target tracking algorithm based on pixel count in a tracking window. The two-chip "Defect detector" (100 instructions/pixel) identifies and locates defective regions in strongly patterned images (e.g. wafers) by considering edge directions. The layouts of the two-chip detector are presented in figure 6.



(a) Chip 1



(b) Chip 2

Figure 6: Layouts of the derived defect detector (different scales)

Table 2 presents synthesized circuits for three benchmark applications. The "Edge detector" (15 instructions/pixel) implements a minimal edge detection based on an L filter. The "Labeler" is the same as the one presented for derivation but the synthesized version does not contain the line delay operator. The "Defect detector" is a modified version of the derived one. The layout of the defect detector is presented in figure 7.

4.2 Discussion

The two integration methodologies share the same emulation platform, but they do not exploit the emulation results in the same way. Synthesis' input space is the set of data-flow graphs, and it generates finite state machine plus datapath architectures. Derivation's input space is the set of data-flow processor graphs, and it generates wired data-flow architectures.

Chip	Labeler	Target tracker	Defect detector chip 1	Defect detector chip 2
Technology	0.5 μm	0.8 μm	1.0 μm	1.0 μm
# processors	17	46	40	53
# gates	7.1K	33.6K	41K	37K
RAM (bits)	0	512	20K	0
ROM (bits)	0	0	20K	0
# Mult. 8×8	0	0	0	8
Area mm^2	4.7	46.2	108.5	110.9
Pixel freq. MHz	70	17	12	12

Table 1: Derived circuits

Chip	Edge detector	Labeler	Defect detector
Technology	1.0 μm	1.0 μm	1.0 μm
# processors	12	13	108
# gates	6.3K	5K	36K
RAM (bits)	0	0	18 432
ROM (bits)	0	0	22 528
# Mult. 8×8	0	0	12
Area mm^2	14.6	11.0	83.0
Pixel freq. MHz	2.8	7.6	1.3

Table 2: Synthesized circuits

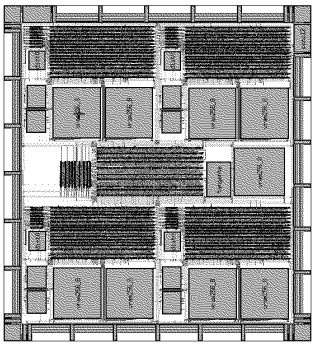


Figure 7: Layout of the synthesized defect detector

On average, derivation generates architectures with 1,000 gates per data-flow graph node. As far as synthesis is concerned, the gate count estimation is not immediate since synthesis can generate an infinite number of architectures implementing the same algorithm depending on frequency/area constraints. The synthesized circuits presented in section 4.1 exhibit on average 400 gates per data-flow node. However these circuits do not operate in real time (i.e. processing on the fly). It is estimated that (pipelined) circuits implementing the same algorithms and able to operate in real time would exhibit about 500 gates per data-flow node.

The difference in gate count is attributed to the extra material cost of the wired data-flow architectural model. The I/O FIFOs required to manage the data flows between operators are directly responsible for the higher cost of derived circuits. Consider a one-pixel delay operator: it is synthesized as a register with some control gates, but it is derived in the worst case as a full wired data-flow operator including one input FIFO and one output FIFO. On algorithms that do not feature many delays, derived and synthesized circuits exhibit about the same gate count. (Most I/O FIFOs are replaced by registers in derived circuits, and these registers are also found in synthesized circuits as pipeline registers).

The higher material cost of derived circuits is balanced by the straightforwardness of derivation. The algorithmic complexity of derivation's optimizations is lower than the complexity of synthesis. The validation of derived circuits is easier since they are optimized versions of already validated circuits. Furthermore, any new data-flow operator (not already described physically in the module database) must be manually designed and validated before it can be used in synthesis. This is not the case with derivation: any new DFP operator is optimized by the derivation software without any additional user effort.

A possible area of research is to build an integration environment on top of derivation and synthesis. It would generate circuits containing both derived and synthesized parts cooperating on the same chip. The input specification would include the behavioral (data-flow graph) and structural (Data-Flow Processor graph) descriptions of the design. The en-

vironment would partition manually or automatically the input specification according to user-specified constraints (frequency, area); each part would then be either derived or synthesized. The environment would finally generate the logic needed to control the circuit. The environment should allow the user to evaluate different partitions in terms of frequency and area. (The combining of different synthesis tools into a single environment is an area being currently researched, see e.g. [13], [14]).

5 Conclusion

The purpose of this research is to develop a coherent design framework for description, validation and integration of real-time image processing ASICs. The first two steps of the design flow (i.e. functional specification and emulation) are well-adapted to image processing applications. Using the DFFC emulator dedicated to image processing brings many advantages in terms of validation (systematic real-time performance, validation in the target environment).

Two different integration methods have been explored and applied on several realistic applications. The results seem to indicate that derivation and synthesis should be strongly coupled into a single integration package within the whole design framework. For each specific application, the integrated ASICs would contain both derived and synthesized parts according to the efficiency of each method.

A Algorithm functional programming sources

This appendix presents the functional programming source files of the applications described in section 4.

Edge detection

```
//-----------------------
// Edge detection
//-----------------------

// PD(k):   k-pixel delay
// LD(k):   k-line delay
// SHFT(k): shift right by k
// THR:     threshold

MAIN [
        VIDEO INPUT image:FRAME(PIXEL);
        VIDEO INPUT threshold:PIXEL;
        VIDEO OUTPUT edge:FRAME(PIXEL);
]

DEF dx = SUB.[image, PD(1).[image, image]];
DEF dy = SUB.[image, LD(1).[image, FIFO.image]];

DEF dx2 = SHFT(2).dx;
DEF dx2 = SHFT(2).dx;

DEF adx = ABS.dx2;
DEF ady = ABS.dy2;

DEF adx2 = SHFT(4).adx;
DEF ady2 = SHFT(4).ady;

DEF edge = THR.[MAX.[adx2, ady2], threshold];

END
```

Connected component labeling

```
//-----------------------------------------
// Connected component labeling - First pass
//-----------------------------------------

// PD(k):   k-pixel delay
// LD(k):   k-line delay

MAIN [
        VIDEO INPUT image:FRAME(PIXEL);
        VIDEO INPUT labeld:PIXEL;
        VIDEO INPUT Regl:FRAME(PIXEL);
        VIDEO OUTPUT Regw:FRAME(PIXEL);
        VIDEO OUTPUT label:PIXEL;
]
// labeld: image of labels delayed by one line

DEF ImPD = PD(1).[image, image];
DEF ImLD = LD(1).[image, FIFO.image];
DEF ImPLD = PD(1).[ImLD, ImLD];

DEF r1 = OR.[Regl, AND.[ImPD, ImPLD]];
DEF r2 = AND.[Regl, ImPD];
DEF regw = OR.[r1, r2];

DEF nc = NAND.[regw, regw];
DEF c1 = AND.[nc, AND.[image, ImLD]];
DEF c2 = AND.[AND.[nc, ImPD], NAND.[image, image]];

DEF label = OR.[AND.[labeld, c1], 1.COUNT.c2];

DEF Regw = regw;

END
```

Target tracking

```
//-----------------
// Target tracking
//-----------------

// NTH(i,j): element i of a j-uple
// CL:       line counter
// CP:       pixel counter

MAIN
```

```
 [
   ASYNC INPUT Param ;
   ASYNC OUTPUT histox ;
   ASYNC OUTPUT histoy ;

   VIDEO INPUT Camera ;
   VIDEO OUTPUT display ;
   VIDEO OUTPUT image ;
 ]

// Binarization
DEF Image = AND.[LEQ.[Camera,NTH(1,5).Param] ,
                 GEQ.[Camera,NTH(2,5).Param]] ;

// Window
DEF line   = CL.Camera ;
DEF column = CP.Camera ;
DEF pline   = NTH(3,5).Param ;
DEF pcolumn = NTH(4,5).Param ;
DEF size = NTH(5,5).Param ;
DEF size2 = MUL(2).size ;
DEF m1 = LEQ.[column, ADD.[size2, pcolumn]] ;
DEF m2 = GEQ.[column, pcolumn] ;
DEF m3 = LEQ.[line, ADD.[size2, pline]] ;
DEF m4 = GEQ.[line, pline] ;
DEF Mask = AND.[AND.[m1,m2] , AND.[m3,m4]] ;

// Count
DEF SsImage = AND.[Image , Mask] ;

// Display of a reticle
DEF aff = MUL(255).SsImage ;

DEF retic1 = EQ.[ADD.[pcolumn, size], column]
DEF retic2 = EQ.[ADD.[pline, size], line];
DEF reticle = MUL(255).OR.[retic1, retic2] ;

DEF dis = ADD.[MUL(128).Image, MUL(127).Masque];
DEF display = XOR.[dis, reticle] ;
DEF image = XOR.[XOR.[aff,Camera], reticle] ;

// Histogramme de la position
DEF mline = AND.[aff,line] ;
DEF histoy = DECIM.HISTO.mline ;
DEF mcolumn = AND.[aff,column] ;
DEF histox = DECIM.HISTO.mcolumn ;
END
```

Defect detection

```
//------------------
// Defect detection
//------------------

//---------------------------
// Macro - dirk (direction k)
//---------------------------
BEGIN dirk [
        INPUT ddirection;
        INPUT dedge;
        INPUT dcount;
        OUTPUT dcontribk; ]

// ADD1P2P: add(1-pix delay, 2-pix delay)
// ADD1L2L: add(1-line delay, 2-line delay)

DEF  I  =  MUX(0).[RAM.ddirection, dedge];
DEF  CH = 1.ADD.[ADD1P2P.[I,I,I], I];

DEF 11d = FIFO.CH;
DEF CV    = ADD1L2L.[11d, FIFO.11d, CH];

DEF IF   = MUX(0).[CV, dedge];

DEF LST  = PD(1).[IF, IF];
DEF XEQ  = XOR(255).EQ.[LST, IF],
DEF LESS =  AND.[LEQ.[LST, IF], XEQ];
DEF RES0 = COUNT.LESS;

DEF o1   = 1.RES0;

DEF INV  = RAM.dcount;
DEF MI = MUL.[INV, o1];
DEF MULH = 2.MI;
DEF MULL = 1.MI;
DEF MOYT = ADD.[MULL, MULH];

DEF MOY  = 1.MOYT;

DEF INV2 = RAM.MOY;
DEF resT = MUL.[INV2, IF];
DEF res  = resT;
DEF dcontribk = res;

END

//--------------------
// Macro - extraction
//--------------------

// LPD(1,1): 1-line 1-pixel delay
// SHFT(k):  shift right by k
// XORSGN:   sign of the XOR
// ATAN:     arctangent
// THR:      threshold

BEGIN extraction [
        VIDEO INPUT image:FRAME(PIXEL);
        VIDEO INPUT threshold:PIXEL;
        VIDEO OUTPUT direction:FRAME(PIXEL);
        VIDEO OUTPUT edge:FRAME(PIXEL);
        VIDEO OUTPUT count:PIXEL;
]
DEF ret111p = LPD(1,1).[image, FIFO.image, image];

DEF 1pd = 1.1pld;
DEF 11d = 2.1pld;

DEF  dx = SUB.[image, 1pd];
DEF  dy = SUB.[image, 11d];

DEF dx2 = SHFT(2).dx;
DEF dx2 = SHFT(2).dx;

DEF angle = XORSGN.[dx2, d2];

DEF adx = ABS.dx;
DEF ady = ABS.dy;
DEF adx2 = SHFT(4).adx;
DEF adx2 = SHFT(4).adx;

DEF atan = ATAN.[adx2, ady2];
```

```
DEF direction = ADD(254).[atan, angle];

DEF edge1 = THR.[MAX.[adx, ady], threshold];

DEF count = COUNT.edge1;
DEF edge = edge1;

END


MAIN  [
        VIDEO INPUT image0:FRAME(PIXEL);
        VIDEO INPUT threshold0:PIXEL;
        VIDEO OUTPUT defect0:FRAME(PIXEL);
         ]

DEF extract   = extraction : [image0, threshold0];
DEF direction0 = 1.extract;
DEF edge0   = 2.extract;
DEF count0    = 3.extract;

DEF dir0 = dirk : [direction0, edge0, count0];
DEF dir1 = dirk : [direction0, edge0, count0];
DEF dir2 = dirk : [direction0, edge0, count0];
DEF dir3 = dirk : [direction0, edge0, count0];

DEF max0 = MAX.[dir0, dir1];
DEF max1 = MAX.[dir2, dir3];
DEF defect0 = MAX.[max0, max1];

END
```

References

[1] N. Zafar. Using emulation to cut ASIC and system verification time. *ASIC Design*, Apr. 1994.

[2] P.M. Athanas and A.L. Abbott. Real-Time Image Processing on a Custom Computing Platform. *Computer*, 28(2):16–24, Feb. 1995.

[3] J. Woodfill and B. Von Herzen. Real-Time Stereo Vision on the PARTS Reconfigurable Computer. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 242–250, 1997.

[4] R. Lauwereins, M. Engels, M. Adé, and J.A. Peperstraete. Grape-II: A System-Level Prototyping Environment for DSP Applications. *Computer*, 28(2):35–43, Feb. 1995.

[5] B.A. Curtis and V.K. Madisetti. Rapid Prototyping on the Georgia Tech Digital Signal Multiprocessor. *IEEE Trans. on Signal Processing*, 42(3):649–62, March 1994.

[6] J. Sérot, G. Quénot, and B. Zavidovique. Functional programming on a dataflow architecture: applications in real-time image processing. *Machine Vision and Applications*, 7(1):44–56, 1993.

[7] J.B. Dennis. Data flow supercomputers. *Computer*, (13):48–56, 1980.

[8] I. Koren and G.B. Silberman. A Direct Mapping of Algorithms onto VLSI Processor Arrays Based on the DataFlow Approach. In *Proceedings of the International Conference on Parallel Processing*, pages 335–7, Aug. 1983.

[9] S.Y. Kung, S.C. Lo, S.N. Jean, and J.N. Hwang. Wavefront Array Processors: Concept to Implementation. *IEEE Computer*, 20(11):18–33, July 1987.

[10] I.C. Kraljić, G.M. Quénot, and B. Zavidovique. From Real-Time Emulation to ASIC Integration for Image Processing Applications. *IEEE Transactions on VLSI Systems*, 4(3):391–404, Sept. 1996.

[11] I.C. Kraljic. *Dérivation systématique d'automates de vision à partir de résultats d'émulation.* PhD thesis, Université de Paris-Sud Centre d'Orsay, France, 1996. In French.

[12] F.S. Verdier and B. Zavidovique. High Level Synthesis of a Defect Detector. In *Third International Workshop on Computer Architecture and Machine Perception (CAMP'95)*, 1995.

[13] P. Gupta, C.-T. Chen, J.C. DeSouza-Batista, and A.C. Parker. Experience with image compression chip design using Unified System Construction Tools. In *Proceedings of the* 31^{st} *ACM/IEEE Design Automation Conference, CD-ROM Publication, San Diego, CA, USA*, 1994.

[14] I. Verbauwhede and J.M. Rabaey. Synthesis for Real Time Systems: Solutions and Challenges. *Journal of VLSI Signal Processing*, (9):67–88, 1995.