

From Loop Transformation to Hardware Generation

Harald Devos, Kristof Beyls, Mark Christiaens, Jan Van Campenhout and Dirk Stroobandt
Ghent University, ELIS - PARIS
Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
{hmdevos,kbeyls,mchristi,jan.vancampenhout,dstrooba}@elis.ugent.be

Abstract—**Multimedia applications are examples of a class of algorithms that are both calculation and data intensive and have real-time requirements. As a result dedicated hardware acceleration is often needed.**

Usually the on-chip memory is not sufficient to store all data and has to be extended with external memory. The bandwidth to this memory often becomes a bottle neck. Loop transformations are needed to reduce this bandwidth, by improving the temporal and spatial data locality. They can also unveil the parallelism present in the algorithm. The polyhedral model offers a flexible program representation that allows to automate this kind of transformations. The class of applications that can be transformed with the polyhedral model fits very well into the class of applications that can benefit from hardware acceleration.

This paper describes how the existing tools, that generate software from a polyhedral program representation, have been extended to generate a VHDL description of a hardware controller. The corresponding data path is generated semi-automatically. Combining the generation of controller and data path creates a fast path to hardware. Our techniques enable an easy exploration of the design space, by generating a lot of implementation variants.

The techniques are demonstrated on an inverse discrete wavelet transform resulting in several synthesizable designs, of which one has been hand-optimized towards a FPGA implementation. The results outperform those of a commercial C-to-VHDL compiler. The generated variants run 5 to 10 times faster while consuming less resources.

Keywords— Polyhedral Model, Loop Transformation, Design Space Exploration, Hardware Synthesis, Discrete Wavelet Transform.

I. INTRODUCTION

There exists a class of applications for which a software implementation can not offer the required performance. E.g., for real-time video decoding dedicated hardware is needed. FPGAs (Field Programmable Gate Arrays), offer a high computational power combined with a huge internal bandwidth. Although the clock frequency is much lower than on a processor many applications can be accelerated thanks to the high parallelism available. All calculation and memory blocks on a FPGA can work in parallel.

The design path from a high-level algorithm description to a low-level synthesizable hardware description, is

a long, manual and error-prone process. Iterating this process, e.g., to examine the influence of design choices requires a high effort and is often not possible. Therefore automation techniques have to be developed.

Exploiting the calculative power of a FPGA is only possible when the data access rate can cope with the data process rate. Modern FPGAs contain a lot of memories and registers that are accessible in parallel and offer a huge on-chip bandwidth. However, for many applications these memories are not large enough to store all the data. The required storage can be offered by external memory, but the bandwidth to this memory will be lower and the latency higher.

This is very similar to the *memory bottle neck*, known in processor architectures with a hierarchy of main memory, caches and registers. To reach a high performance the accesses to the slower external memory have to be minimized. A lot of techniques have been developed to improve the *cache behaviour* of algorithms in software [19]. They try to bring the reuses of data elements closer together (temporal locality), or group the uses of elements that are located close to each other, (spatial locality). This is done by loop transformations that change the execution order of statement instances in a loop nest. Although these techniques are mainly developed targeting software implementations they are also useful for designing hardware.

This paper shows how a polyhedral program representation can be used to automate loop transformations and automate part of the hardware design process.

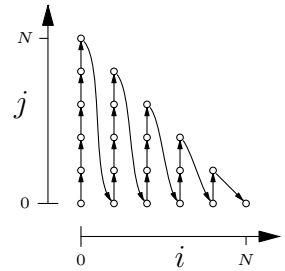
Section II explains how programs can be represented and transformed in the polyhedral model. Hardware generation from this representation is described in Sect. III. As a case study implementation variants of the Inverse Discrete Wavelet Transform (IDWT) are built in Sect. IV. Some related work is listed in Sect. V. Section VI ends the paper with conclusions and future work.

II. THE POLYHEDRAL MODEL

Compilers and refactoring tools that apply loop transformations need a way to represent loop nests and their corresponding boundaries. Usually, abstract syntax trees (AST) are used. In an AST, each loop corresponds to a node in a

Depth	Statement	Ordering	Iteration	Scheduling
0	1	2		
0: for $i = 0, N$				
0: for $j = 0, N - i$				
if $i = 0$ then				
0: $T[i + j] = 1$	S1(i, j)	(0, 0, 0)	(i, j)	(0, $i, 0, j, 0$)
end if				
1: $T[i + j] = T[i + j]I[i][j]$	S2(i, j)	(0, 0, 1)	(i, j)	(0, $i, 0, j, 1$)
1: for $k = 0, N - 1$				
0: $O[k] = T[k] + T[k + 1]$	S3(k)	(1, 0)	(k)	(1, $k, 0$)
2: PRINT("End of program.")	S4()	(2)	()	(2)

(a) Program code with ordering, iteration and scheduling vector of the statements.



(b) Polyhedral representation of the iteration domain of S2. The arrows indicate the execution order.

Fig. 1. Program example to illustrate the polyhedral program representation

tree, and inner loops are represented as child nodes of an outer loop node. In contrast, our framework is based on a representation of loops in the so-called polyhedral program model [5].

A. Program Representation

A *statement* is a line of a program without control, typically an assignment with operations at the right hand side. The (lexical) *depth* of a loop or a statement is the number of loops that surround it. The program top-level (depth 0) is a sequence of loops and statements which are numbered from 0 onwards (Fig. 1(a)). Each loop in its turn also contains a sequence of statements and loops that are numbered from 0 onwards. Each statement is uniquely identified by the vector composed of the numbers telling the position in each of the surrounding loops and the top-level. This vector is called the *ordering vector*. It has dimension $D_S + 1$ with D_S the depth of statement S.

A statement is executed for a set of values of the *iteration vector*, the vector containing the iterators of the surrounding loops (dimension D_S). A single execution of a statement is called a *statement instance*. The *iteration domain* is the set of values of the iteration vector for which the statement is executed (Fig. 1(b)). The *scheduling vector* of a statement is the vector with the elements of the ordering vector as odd elements and the iterators as even elements. The dimension is $2D_S + 1$. The execution order of statement instances follows the lexicographical order of their scheduling vectors (Fig. 1(b)). The uniqueness of the ordering vectors ensures that the statement instances are strictly ordered.

We restrict ourselves to programs where the loop bounds are linear expressions (affine functions) of some parameters and the iterators of the surrounding loops. In this case

the iteration domains can be represented as parameterized integer polyhedra, hence the name *polyhedral model*.

A *parameterized integer polyhedron* P_p is defined as

$$P_p = \{x \in \mathbb{Z}^n \mid Ax \geq Bp + b\}, p \in \mathbb{Z}^m$$

where A and B are constant integer matrices, b is a constant integer vector, and p is a vector of parameters. Consider the program in Fig. 1. The iteration domains for the statements can be represented by polyhedra as follows

$$\mathcal{D}_{S1} = \left\{ \begin{bmatrix} i \\ j \end{bmatrix} \in \mathbb{Z}^2 \mid \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ -1 \end{bmatrix} \begin{bmatrix} N \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \right\} \quad (1)$$

$$\mathcal{D}_{S2} = \left\{ \begin{bmatrix} i \\ j \end{bmatrix} \in \mathbb{Z}^2 \mid \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ -1 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \geq \begin{bmatrix} 0 \\ -1 \\ 0 \\ -1 \end{bmatrix} \begin{bmatrix} N \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \right\} \quad (2)$$

$$\mathcal{D}_{S3} = \left\{ [k] \in \mathbb{Z}^1 \mid \begin{bmatrix} 1 \\ -1 \end{bmatrix} [k] \geq \begin{bmatrix} 0 \\ -1 \end{bmatrix} \begin{bmatrix} N \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\} \quad (3)$$

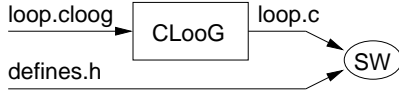


Fig. 2. The Chunky Loop Generator (CLooG) generates (control) code from a polyhedral representation (*.cloog). Together with the statement definitions (*.h) this results in executable software.

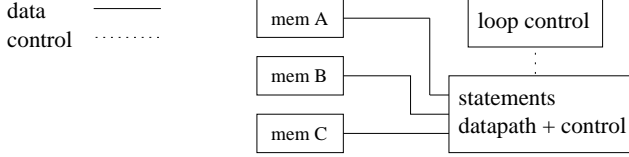


Fig. 3. Architecture with separate memory for each array.

or, using a more compact notation:

$$\begin{aligned} \mathcal{D}_{S1} &= \{(i, j) \in \mathbb{Z}^2 \mid i = 0 \wedge 0 \leq j \leq N\} \\ \mathcal{D}_{S2} &= \{(i, j) \in \mathbb{Z}^2 \mid 0 \leq i \leq N \wedge 0 \leq j \leq N - i\} \\ \mathcal{D}_{S3} &= \{(k) \in \mathbb{Z}^1 \mid 0 \leq k \leq N - 1\}. \end{aligned}$$

If the array indices are linear expressions of the iterators and parameters, the dependences between the statement instances can also be represented as polyhedra or unions of polyhedra, which are called *dependence domains* [6].

B. Describing Loop Transformations

In the polyhedral model, a program is represented as a set of statements. For each statement, the values of the surrounding loop iterators for which it is executed, is represented by a number of matrices, e.g., (1)-(3).

All loop transformations are performed by transforming the matrices and scheduling vectors of the statements, that are contained in the transformed loops. The statement definitions, i.e., the operations that are executed by a statement in function of the iterators, remain untouched. After a transformation has been applied, the code is still represented by a set of statements with associated matrices. As a result, loop transformations can easily be combined by combining the corresponding matrix operations.

The polyhedral representation can then be translated back into an AST to obtain an executable specification, e.g., by CLooG (Chunky Loop Generator) [4] (Fig. 2). The exact representation of transformations, and their practical implementation in a tool called URUK, is presented in detail in [9] and [12].

III. HARDWARE GENERATION

As explained in Sect. II, a program definition can be split into statement definitions and a representation of the iteration domains and ordering. This separation can be

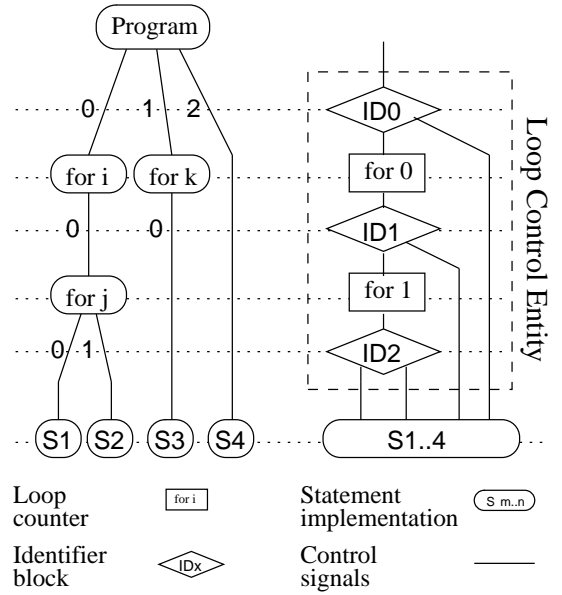


Fig. 4. Abstract syntax tree of the program in Fig. 1(a) and corresponding hardware architecture of the loop controller. A single hardware block implements all statements S1..4. The block *for 0* implements the loops at depth 0: *for i* and *for k*. Which of the two is executed depends on the value received from *ID0*.

used to create a hardware architecture composed of two parts, a *loop control entity* that drives the iterators and statement instances and a *statements entity* that implements the statement operations for the iterator values received from the control entity (Fig. 3 and 4).

This section describes the (semi-)automatically generated architecture of these blocks. Until now only a sequential execution of the statement instances is supported. This means that only the parallelism within one statement can be exploited.

A. Loop Control

The loop controller consists of a set of communicating automata, a so called factorized implementation [2]. Experiments showed that the proposed factorized implementation consumes less area and reaches a higher clock frequency than a monolithic control block.

In the abstract syntax tree (AST) on Fig. 4, the numbers and iterators on the path between the top-node *program* and a statement node correspond to the elements of the scheduling vector of that statement in Fig. 1(a).

We propose a controller composed of automata, each corresponding with one dimension of the scheduling vector. This results in two types of automata. A first type, the loop counters, is responsible for the iterators, e.g., *for 0* in Fig. 4 drives *i* and *k*. The other type, the identifier blocks, corresponds to the elements of the ordering vectors, e.g., *ID 0* in Fig. 4, The loop counter blocks calcu-

late the loop bounds and stride in function of the parameters, the iterators of surrounding loops and the more significant elements of the ordering vector.¹ The identifier blocks count from zero onwards to enumerate the different statements and loops at the level below.

The tool CLoogVHDL was built on top of the CLoog-library, to generate VHDL-code of this hardware structure starting from a polyhedral description in a *.cloog file. The execution time of a statement instance does not have to be known at compile time and may vary during execution or between variants of the statement implementations. A simple handshake between controller and statements makes this possible without losing clock cycles.

B. Statement Control and Data Path

All statements are implemented as a single VHDL process. This is only possible since no two statements operate at the same time. It allows the synthesis tools to “see” all statements during optimization and results in hardware being shared between statements. The generation of the statements entity is not fully automated yet. This is not a large problem since a loop transformation only influences the controller entity and therefore the statements entity can be reused for several loop transformation variants.

The statement definitions are split into operations and array accesses. The former are translated into a VHDL-syntax and the latter are translated into memory transactions, assuming one memory for each array and an access time of two cycles (Fig. 3). An intermediate file, *steps.vhd* on Fig. 5, contains the actions per cycle for each statement. In this file it is possible to do some scheduling optimizations by hand. (Scheduling techniques such as in [3] could automate this step.) From here the path to synthesizable VHDL is automated (Steps2process).

IV. CASE STUDY: THE 2-D IDWT

A. The Basic Algorithm and Variants

The Discrete Wavelet Transform (DWT) and its inverse (IDWT) have become popular for image processing and compression, e.g., JPEG-2000. A lot of optimized implementations have been published [15]. Fig. 6 shows the algorithm, using a 9/7 bi-orthogonal filter pair [16]. Each level of transformation (iterator l), consists of a vertical and horizontal filtering step. This results in a bad data locality while the input is first scanned column by column and then row by row and this for each level. This ex-

¹The generated VHDL expressions for the loop bounds are equivalent to the expressions generated by CLoog. As a result operators as `mod` and `div` are synthesizable only for powers of 2. Techniques as those presented in [20] could extend this synthesizable subset.

```

for  $l = K-1, 0$ 
   $S = R / 2^{l+1}$ 
  for  $j = 0, (C / 2^l) - 1$  // Vertical filtering.
    for  $i = 0, S - 1$ 
       $B_{2i,j} = A_{i-1..i+1,j} \cdot H_o + A_{S+i-2..S+i+1,j} \cdot G_o$ 
       $B_{2i+1,j} = A_{i-1..i+2,j} \cdot H_e + A_{S+i-2..S+i+2,j} \cdot G_e$ 
     $S = C / 2^{l+1}$ 
  for  $i = 0, (R / 2^l) - 1$  // Horizontal filtering.
    for  $j = 0, S - 1$ 
       $A_{i,2j} = B_{i,j-1..j+1} \cdot H_o + B_{i,S+j-2..S+j+1} \cdot G_o$ 
       $A_{i,2j+1} = B_{i,j-1..j+2} \cdot H_e + B_{i,S+j-2..S+j+2} \cdot G_e$ 

```

Fig. 6. Simplified representation of the IDWT basic algorithm (Row-Column-based). R and C are parameters representing the number of rows and columns of the image that is transformed over K levels. A and B are two dimensional arrays and G_o, G_e, H_o and H_e are vectors containing the odd and even elements of the wavelet filters G and H with lengths 9 and 7 (a 9/7 bi-orthogonal filter pair).

plains the name of this variant: Row-Column-based level-by-level.

Several loop transformations, found by manual analysis or suggested by tools like SLO [7], can improve the locality. After interchanging the i and j iterator of the vertical filtering step, the two filtering steps can be fused what brings the production and consumption of the elements of array B close to each other. The input is now scanned line by line, hence the name line-based level-by-level. Tiling [19] allows to interleave the operations of the different transformation levels. More details on the construction of the transformation sequences can be found in [10].

As can be seen in Fig. 6, some indices get values outside the array (image) boundaries. Therefore the image has to be extended by mirroring the pixels near the border. This makes the code more complex than shown here.

B. Generating Variants without Memory Hierarchy

Some preprocessing was needed before the loop transformations were possible. First the code on Fig. 6 was converted to a dynamic single assignment form to eliminate false dependences. Secondly, the outer loop was unrolled to remove the exponential expressions in the loop boundaries of the i and j loops so that the program could be represented in the polyhedral model.

To investigate the influence of the complexity due to the mirroring at the borders two versions of the code were made. One that does not calculate the pixels near the borders and one that does by using predicated statements. The predicates are used to distinguish between the execution at the center and near the borders of the image. This dis-

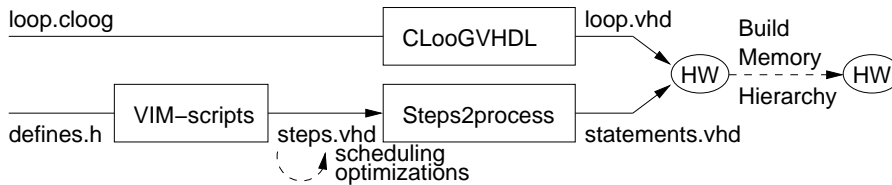


Fig. 5. CLoogVHDL extends the CLoog software generation process (Fig. 2) to create hardware.

tion is done within the statement and not within the loop control structure. This makes the loop transformations and control complexity similar for both cases and will only have an impact on the statements' implementation.

The WRaP-IT/URUK tool set [9], [12] performs a sequence of loop transformations specified in a script. Several scripts were written to generate variants: fused (line-based), fused-and-tiled, with and without borders. For each variant a hardware controller was generated by CLoogVHDL and combined with a statement block, with or without manual optimizations. Table II gives an overview of all the different implementations. They are compared with three designs generated by Impulse C (version 1.22) [1], a commercial tool for automatic synthesis of stream-based applications, and a manual design [11], that aimed at minimal area while transforming 45 CIF frames/s.

The inclusion of the calculations near the borders adds a lot of complexity to the design. If a processor is available in the system it might be beneficial to put the border calculations on the processor and run only the regular non-border operations on the FPGA.

Impulse C has the worst synthesis results. It aims at compiling a larger class of C programs than those representable in the polyhedral model. Therefore it is more conservative in its approach to hardware generation, while CLoogVHDL has a more specialized implementation strategy. A single large automaton is generated instead of a factorization into smaller automata. In some places 32 bit data types are used where smaller word lengths suffice, even if shorter data types are used in the source code. Newer versions of this tool are likely to give better results.

The frame rate printed in the last column assumes a calculation limited design. In practice the accesses to the external memory can slow down the design. Table I shows the dataflow and burst usage for transforming one frame, assuming the data locality is exploited, e.g., by a memory hierarchy as in Fig. 7. This illustrates the effectiveness of the loop transformations.

C. Building a Memory Hierarchy

The architecture constructed until now (Fig. 3), uses one memory for each array and does not yet exploit the data

TABLE I

DATA FLOW (IN PIXELS) TO/FROM THE MAIN MEMORY (LEFT SIDE OF FIG. 7) FOR DIFFERENT IDWT VARIANTS.

	Data flow		Burst usage
		$K = 3$	
RC	$\frac{16}{3}RC(1 - 1/4^K)$	$5.25RC$	50%
Fused	$\frac{8}{3}RC(1 - 1/4^K)$	$2.625RC$	100%
"+"Tiled	$2RC$	$2RC$	100%

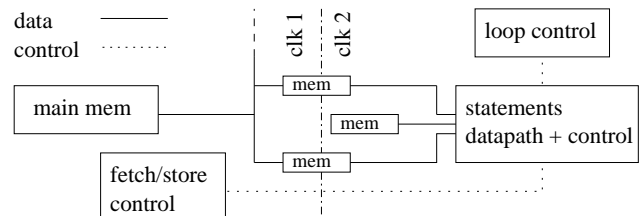


Fig. 7. Architecture with memory hierarchy. Fetching and storing data is done in parallel with the operations of the IDWT. Execution speed is determined by the slowest of these two processes. Multiple local buffers allow parallel memory accesses.

locality by using on-chip buffers. One promising variant, fused with borders and manual optimizations, was selected to be extended with a memory hierarchy as in Fig. 7.

The large memories were replaced by small buffers large enough to contain the working data set. By using 2-port memories the fetching and storing can be done in parallel with the statement execution. Since the bus connected to the main memory is likely to be shared with other cores the timing of data transfers is not deterministic. Synchronization points delay operations if data is not available in the buffers on time.

A queue of prefetch and store requests is kept in the *fetch/store control block*. A new request is added each time space becomes available in the buffers and not just before the data is needed. Therefore, in a system that is not bandwidth limited, only in the beginning time is wasted waiting for data.

Some buffers are split into parallel accessible line buffers to increase the on-chip bandwidth. As a result the

frame rate increases to more than 45 frames/s and is only smaller than the manual design due to the lower clock frequency. The number of cycles is roughly the same for both designs.

These two designs were tested on an Altera PCI Development Board with a Stratix EP1S60F1020C6 FPGA and DDR SDRAM memory. An Avalon fabric is used to connect the hardware blocks to the external memory, running at 50 MHz. The generated design transforms up to 53 CIF frames/s (synthesis with Quartus II v5.1). The manually made design reached only 11.3 CIF frames/s due to the large amount of transactions that could not be done in bursts.

A line-based software implementation reached 43.5 CIF frames/s on an AMD Athlon XP 2500+ running at 1830 MHz. This corresponds to 42 million cycles for one CIF frame, 50 times more than the optimized FPGA implementations.

V. RELATED WORK

Hardware generation from high-level languages is the target of many projects. In MMAAlpha [13], loop nests are represented in a functional, dynamic single assignment language. The code is mapped onto a systolic array. The PICO project [17] also translates loop nests into systolic arrays and runs the left-over code on a specialized EPIC (Explicitly Parallel Instruction Computing) or VLIW (Very Long Instruction Word) processor. PARO [14] maps Piecewise Regular Algorithms (PRA) onto a configurable processor array. These projects all handle a subset of the programs we can handle.

The Compaan/Laura tool suite [18], [20] translates polyhedral loop nests into Kahn Process Networks (KPN), by eliminating global memory and global control. Laura translates the KPNs into VHDL.

The Cameron Project has created a high-level algorithmic language, named SA-C [8], for expressing image processing applications. Compilation to FPGAs is done using data flow graphs. Impulse C [1] uses a subset of C extended with IO-macros. Translation is done by constructing one large finite automaton where the states relate to the control flow graph of the program.

These projects all focus more on exploiting parallelism than on improving bandwidth aspects. The inputs for the algorithms range from PRAs (MMAAlpha, PICO, PARO), over polyhedral programs (Compaan, SA-C, this work), to more general constructs (Impulse C). This results in different trade-offs between the set of algorithms handled and the efficiency of the resulting hardware.

VI. CONCLUSIONS AND FUTURE WORK

Hardware acceleration on a FPGA does not only require to have enough parallelism in an application but also a good data locality. Loop transformations can be used to improve the data locality and are easily performed using the polyhedral model. Automating the hardware generation from this model allows to compare implementation variants. Although a manual implementation can be more efficient, generated designs may outperform it due to bandwidth requirements.

Future work may include the automation of the scheduling operations and the construction of a memory hierarchy. The hardware architecture will have to be adapted to allow parallel execution of statement instances. Isolating complex code, e.g., due to border extension, and running it on a processor might improve the performance.

VII. ACKNOWLEDGEMENTS

The authors would like to thank A. Cohen, S. Girbal and N. Vasilache for providing access to the URUK tool and giving support. We would like to thank Altera for donating FPGA boards and tools. This research is supported by I.W.T. grant 020174, F.W.O. grant G.0021.03 and by GOA project 12.51B.02 of Ghent University. Harald Devos is supported by the F.W.O. (Research Foundation - Flanders).

REFERENCES

- [1] <http://www.impulsec.com/>.
- [2] P. Ashar, S. Devadas, and A. R. Newton. *Sequential logic synthesis*. Kluwer Academic Publishers, 1992.
- [3] I. Augé, F. Pétrot, F. Donnet, and P. Gomez. Platform-based design from parallel C specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(12):1811–1826, December 2005.
- [4] C. Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC'2 IEEE International Symposium on Parallel and Distributed Computing*, pages 23–30, Ljubljana, october 2003.
- [5] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In *LCPC'16 International Workshop on Languages and Compilers for Parallel Computing, LNCS 2958*, pages 209–225, College Station, october 2003.
- [6] C. Bastoul and P. Feautrier. More legal transformations for locality. In *EURO-PAR Parallel Processing, Lecture Notes in Computer Science*, volume 3149, pages 272–283. Springer-Verlag Berlin, 2004.
- [7] K. Beyls and E. H. D'Hollander. Intermediately executed code is the key to find refactorings that improve temporal data locality. In *CF '06: Proceedings of the 3rd conference on Computing Frontiers*, pages 373–382, New York, NY, USA, May 2006. ACM Press.
- [8] W. Bohm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar. Mapping a single assignment programming language to reconfigurable systems. *Journal of Supercomputing*, 21(2):117–130, February 2002.

TABLE II

COMPARISON OF DIFFERENT IMPLEMENTATIONS OF THE IDWT. SYNTHESIS RESULTS WITHOUT MEMORIES (FIG. 3 OR THE RIGHT SIDE (CLK 2) OF FIG. 7) WERE OBTAINED USING ALTERA QUARTUSII v4.2 FOR THE STRATIX FPGA FAMILY. THE FRAME RATE IS NORMALIZED TO CIF FRAMES (288×352) WITH BORDERS. (* = NUMBER OF CYCLES FOR CIF RESOLUTION INSTEAD OF 72×88 PIXELS)

Tool	Transform	Borders	LE	DSP blocks (#Mul)	f_{max} (MHz)	Cycles (72×88)	Frames/s (288×352)
CLooGVHDL	none	yes	3561	18(9)	46.72	214269	16.99
		no	1691	18(9)	52.16	187350	20.15
	fuse	yes	3336	18(9)	49.24	215848	17.78
		no	1712	18(9)	53.66	200138	19.41
	fuse + tile	yes	3895	18(9)	39.68	215881	14.33
		no	2575	18(9)	46.20	200497	16.68
CLooGVHDL + Manual opt.	none	no	1495	18(9)	58.16	129821	32.43
	fuse	yes	4525	18(9)	40.47	161037	19.59
		no	1622	18(9)	57.32	142570	29.11
	fuse + tile	no	2420	18(9)	49.45	142929	25.05
+ Mem hier.	fuse	yes	17645	18(9)	47.55	* 868917	45.72
Impulse C	none	yes	37127	144(18)	24.13	697431	2.70
		no	13146	80(10)	30.27	605588	3.38
	fuse	yes	23283	144(18)	34.70	508116	5.32
		no	1738	10(5)	68.91	* 869530	79.25
+ Mem hier.	yes	2184	13(8)				

- [9] A. Cohen, S. Girbal, D. Parelo, M. Sigler, O. Temam, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ACM Int. Conf. on Supercomputing (ICS'05)*, Boston, Massachusetts., June 2005.
- [10] H. Devos, K. Beyls, M. Christiaens, J. Van Campenhout, E. H. D'Hollander, and D. Stroobandt. Finding and Applying Loop Transformations for Generating Optimized FPGA Implementations. *Transactions on High Performance Embedded Architectures and Compilers*, 2006. To be published.
- [11] H. Devos, H. Eeckhaut, B. Schrauwen, M. Christiaens, and D. Stroobandt. Ever considered SystemC? In *Proceedings of the 15th ProRISC Workshop*, pages 358–363, Veldhoven, November 2004.
- [12] S. Girbal. *Optimisations d'applications - Composition de transformations de programme: modèle et utils*. PhD thesis, Université de Paris-Sud, 2005.
- [13] A. C. Guillou, P. Quinton, and T. Risset. Hardware synthesis for systems of recurrence equations with multi-dimensional schedule. *International Journal of Embedded Systems*, 2005. To be published.
- [14] F. Hannig, H. Dutta, and J. Teich. Mapping a Class of Dependence Algorithms to Coarse-grained Reconfigurable Arrays: Architectural Parameters and Methodology. *International Journal of Embedded Systems*, 2(1/2):114–127, 2006.
- [15] C.-T. Huang, P.-C. Tseng, and L.-G. Chen. Analysis and VLSI architecture for 1-D and 2-D discrete wavelet transform. *IEEE Transactions on Signal Processing*, 53(4):1575–1586, April 2005.
- [16] S. G. Mallat. A theory for multiresolution signal decomposition: the wavelet representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(7):674–693, July 1989.
- [17] B. R. Rau and M. S. Schlansker. Embedded computer architecture and automation. *IEEE Computer*, 34(4):75–83, April 2001.
- [18] A. Turjan, B. Kienhuis, and E. Deprettere. Translating affine nested-loop programs to process networks. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 220–229, New York, NY, USA, 2004. ACM Press.
- [19] M. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley, 1996.
- [20] C. Zissulescu, B. Kienhuis, and E. Deprettere. Expression synthesis in process networks generated by LAURA. In *16th IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP 2005)*, pages 15–21, July 2005.