HIGH-LEVEL ADDRESS EXPLORATION FOR AN ERROR CORRECTION OPTICAL DISK SUBSYSTEM: IMPROVING DESIGN PRODUCTIVITY

F.Sudriez¶ M.Miranda§ W.De Kimpe¶ F.Catthoor§‡ D.Verkest§ P.Vandebroeck¶

¶Philips Intnl. Technology Centre Leuven (ITCL), Belgium
§IMEC Laboratory, Leuven, Belgium
‡Also Professor at the Katholieke Universiteit Leuven, Belgium

Abstract -

In most real-time multimedia systems, major design bottlenecks are related to memory organisation and management related issues. Usually, in order to meet design constraints at low cost, designers of the memory management units directly think about low-level (register-transfer level) implementation aspects of the architecture. However, this approach typically results into time-consuming global design iterations. In contrast, we propose a high-level design exploration methodology where design bottlenecks related especially to the address generation issues are exposed as early as possible in the design trajectory, hence reducing design time considerably. The effectiveness of our approach is illustrated on the memory processor of an optical disk error correction application. Using our approach, a substantial part of the design search space has been explored in a few man-weeks instead of the months of effort required by more conventional design flows. Furthermore, not only design time is dramatically improved but also implementation cost and system performance, hence resulting in a highly productive design flow.

INTRODUCTION

In most multimedia systems, the dominant cost factor is related to the organisation of the memory subsystem and its Memory Management Unit (MMU). Usually, to meet design constraints, designers directly think about low-level (register transfer level) implementation issues of the architecture. However, the ever increasing system complexity levels, require the exploration of design alternatives to be done at a higher abstraction level for highly productive design flows.

In a bilateral research cooperation between IMEC and Philips-ITCL, a methodology for the implementation of MMUs for data intensive applications has been applied to the Memory Processor (MP) of an optical disk error correction subsystem. This processor manages, under real-time constraints, all the I/O traffic of the optical reader stream while correcting the corrupted data. The goal of the project has been to evaluate our approach with respect to more conventional RT-based design flows in terms of design effort and implementation cost.

In the conventional design flow, like the one applied at the industrial partner site, designers directly make one big jump from the system specification level to a register transfer level description without the possibility of any significant exploration of the design alternatives at the higher levels. By making such a big jump, designers hope to save design time, but they cannot evaluate the impact of all the decisions they

are implicitly making. This lack of exploration pushes most of the system problems down to the time-consuming logic synthesis phase, thus increasing significantly the search space at that level and hence, the CPU time required to perform the bit-level optimisations. This usually ends-up into very time-consuming global design iteration loops and inefficient implementations.

Using our high-level methodology, bottlenecks are exposed much earlier in the design process. This is done by spending more time than designers usually do at the high level using a systematic exploration methodology for the addressing related issues. The design flow is complemented with the use of existing behavioral synthesis design flows and tool support as synthesis back-end. To efficiently use such synthesis approaches, we generate a library of application-specific data-path resources. These resources are allocated and assigned to the complex arithmetic code as targets before the scheduling phase. In this way, the designer have full control on the degree of time multiplexing for the final architecture. Moreover, during the synthesis of the application-specific resources, the designer gets early feedback on low-level timing and cost implementation issues of the data-path architecture. These are further refined during the behavioral synthesis stage by estimations on the controller and register related cost issues. Only when the designer is fully satisfied with the architecture obtained so far, the time-consuming RT-synthesis stage can take place.

RELATED WORK

Most of the work found in literature for custom addressing focusses on bit-level optimisations for architectures combining counters and look-up tables [1, 2], with some interactive support for multiplexing HW at that level [3]. However, time-sharing at the address sequence level which we have shown [4] to be crucial to reduce the overall cost is almost not considered there. On the other hand, architecture exploration support for target styles different from counter based architectures, only have been addressed extensively in our context [5]. We have shown that both target styles can be very efficiently complemented when providing efficient selection criteria for these alternatives at early phases of the design process [6], showing that the impact on the final implementation cost can be of several orders of magnitude.

Compared to other arithmetic operations present in the initial code, different opportunities are also suited and available to exploit i.e., heavy system-level multiplexing based on (data-flow) similarity and (control-flow) regularity in the addressing. Typically, the amount of arithmetic involved is much bigger (from hundreds to thousands of address expressions) than in other application domains where more generic HW synthesis techniques are applied (e.g., DSP filter synthesis). In this paper we show that this complexity can be considerably reduced by performing global-scope algebraic transformations to exploit factorisation and common-subexpression opportunities amongst all Address Expressions (AEs) before the architecture mapping. Additionally, specific synthesis techniques aiming at the time-shared cluster-level (based on our Cathedral-3 methodology for lowly multiplexed data-path synthesis [7, 8]) can be used to trade off more efficiently cycle, area and/or power during the architecture exploration step, instead of going down directly to more traditional operation-level scheduling and allocation. These system-level opportunities are not fully exploited by more traditional HW synthesis techniques [9, 10] and

which should be complemented by them to efficiently bridge the abstraction level from the system down to the operation-level.

To demonstrate the feasibility of design support for this crucial task, a systematic methodology has been developed at IMEC, partly supported by prototype tools in the Adopt environment [11]. This is targeted to optimally transform the index operations into arithmetic operations which are then efficiently mapped onto custom processors. To assist the user in applying this methodology, the most error-prone and tedious design tasks need to be supported with robust tools. The usefulness and feasibility for partial tool support has been demonstrated in experiments for several industrial parties using prototype software combined with user interaction. An example of such is the experience described in this paper. The main contribution of the paper is the additional non-trivial steps required to go from an industrial application to the final design based on the principles of our initially proposed research techniques. Also, the lessons learned from performing this design are included.

SYSTEM DESCRIPTION OF THE MEMORY PROCESSOR

This section describes how the group of data at the input of our demonstrator application is formated, namely in the PHysical Cluster, hereafter called PHC. The input of the complete system is a periodical stream of blocks. Each block contains video/audio data and Error Correction Code (ECC). To increase robustness against burst errors, the bytes of the PHC are interleaved both horizontally (see left side of Figure 1) and vertically.

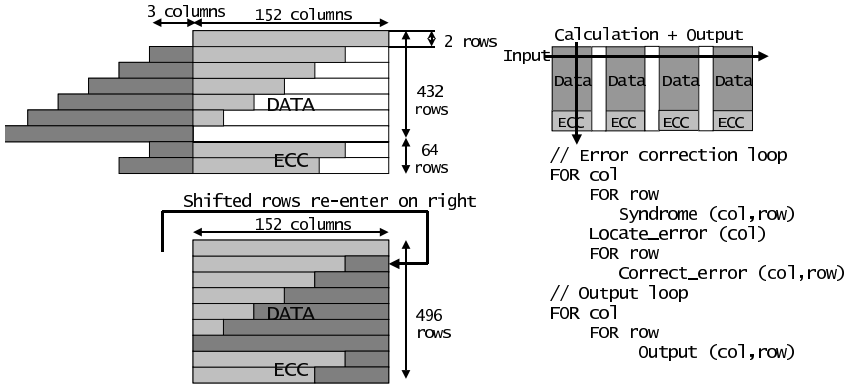

Figure 1: The construction of the physical cluster

The basis of the error correction algorithm [12] is shown on the right side of Figure 1. Although, in principle the details of the algorithm are not related to memory, it still has influence on the data organisation issues. To avoid the addressing overhead associated to the interleaved format, we store the data in memory in a de-interleaved manner. Hence, we can imagine the memory as a block containing the de-interleaved format of the physical cluster. The de-interleaved data format is the original block organisation. This de-interleaving stage introduces quite complex address expressions to the initial specification that we will show it can be efficiently removed during our address exploration stage.

IMPROVING DESIGN PRODUCTIVITY

The MP application is clearly throughput constrained. By adding one pipeline stage at the system level, the cycle requirements have been estimated to be a 0.3% lower than the available budget. Since the calculations made so far are just a lower bound estimations of the required cycle budget (we haven't taken into account yet lower-level timing issues like whether each address expression can indeed be computed in just one clock cycle or similar), it is still too risky to finish the architecture exploration phase at this point. The problem is that we can only know after behavioral synthesis whether the introduction of the system level pipeline is sufficient to meet the required timing. Otherwise, we would create a long design iteration loop that we want to avoid. Instead, further architecture exploration is still possible as shown in the next subsection.

Distribution of the storage cycle budget across loops

Three different loops exist in our application code: input_deinterleaver, error_correction and output. All these loops have to be executed within the given cycle budget. To arrive to an architecture fairly meeting the timing constraints, we have the possibility to parallelise some of these loops, thus exploiting concurrency at the loop level. After fast exploration of memory organisation alternatives[13], we deduce a pipeline architecture where both output and error_correction loops are merged in the same task. A double buffer (ping-pong) architecture is therefore allocated between the input_deinterleaver loop and the two merged loops to keep 'alive' versions of the data that need to be simultaneously accessed by the surrounding tasks.

By merging both error_correction and output loops the resulting data from the first of the loops can be accessed by the second loop as soon as one full data column has been processed. Consequently, we just need to keep one column alive during the error_correction loop so that this column can be accessed by the output loop. We store this column in a register file in the functional units of the MP.

The two merged loops require about a 75% of the cycle budget while the input_deinterleaver loop consumes the remaining 25%. Note that the required cycle budget to execute both loops is not the sum of the cycle budgets of each of them but the maximum of each. This lower cycle budget allows to reduce power consumption by relaxing the clock frequency requirements with limited memory overhead. Further optimisation of this overhead requires detailed exploration of the memory organisation related issues [13]. Alternatively, it can be also used to reuse the same MP implementation for faster generations of the optical disk readers.

High-level address exploration and optimisation

To achieve real time constraints, the arithmetic linked to the addressing code needs to be evaluated under tight timing constraints (one cycle per address expression). Two different ACU target architecture styles are usually available to speed up address expression evaluation at a reasonable cost. These can be based on customised data-path units or customised counter organisations. Techniques to decide the best suited style for each address (sub)expression can be found in [6]. However, before the addressing code is actually mapped onto a custom processor (or eventually also onto an instruction-set processor [14]), it is best to reduce the cost associated to

the initial addressing behaviour by a combination of high-level transformations [11]. This high-level address transformation stage aims to be platform independent and its goal is to optimise all three related design cost aspects: timing, area and, energy associated to the initial code before actual synthesis/compilation [14]. Only afterwards, during the architectural exploration phase, the different time-multiplexing trades-off can be explored : area versus cycle budget or area versus power.

For the MP the design script applied consists of two main steps described in the following two subsections.

Address expression extraction, splitting/clustering and target selection: each address expressions present in the MP are not directly mapped to an ACU. Instead, by splitting the address expression into smaller arithmetic clusters, we create the possibility to test on each different architectural styles. The goal of the step is to define for each cluster, the architecture that will have the largest impact on the resulting cost figure. Techniques for deciding how to select the clustering process and to select the best suited target style can be found in [11]

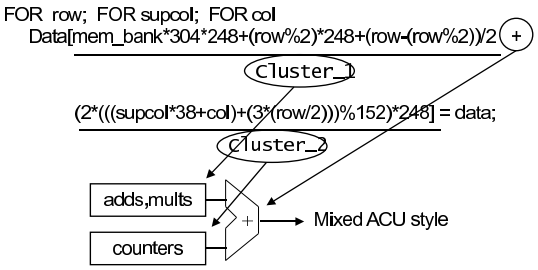

Figure 2: Address expression associated to the de-interleaving loop

An example of this stage is the address expression associated to the input_deinterleaver loop (see Figure 2). In this loop we have to generate an arithmetic expression containing a modulo operation. Following our systematic approach, we focus on this modulo operation because its non-linearity. We have analysed the sequence of addresses generated by the cluster containing the modulo operation in order to look for sequence periodicity properties as suggested in [11].

The full sequence (see main-seq in Figure 3) is composed by subsequences of numbers ranging between 0 and 151. These subsequences are generated by a circular binary counter and it is activated by the evolution of the innermost loop. This inner loop reach its reset state after 152 iterations. During this reset state the counter is initialised with a different value each time. This initial value is supplied by another counter activated by the surrounding loops and it can be derived from two other periodical subsequences. The first periodical subsequence (subseq1) defines the start points of the main sequence (main-seq). The subseq1 values range also between 0 and 151. The next value of subseq1 is incremented by three with every two iterations of the outermost loop. The resulting sequence is: 0,0,3,3,6,6,. . .. The second periodical subsequence (subseq2) defines the starting point of the first periodical subsequence (subseq1) when this one reaches the value 151. The subseq2 is very simple: 0,1,2,0,. . ..

Once the primitive periodical sequences are identified we can now deduce an opti-

mal organisation of counters with flexible initialisatio and reset states. Two counters are needed to implement the target sequence. The first counter (counter1) gets incremented in every column and it resets in every row. This first counter generates the main-seq and it can be implemented as a free-run circular counter. A second counter (counter2) provides the start counting value to counter1, according to the evolution of the sequences subseq1 and subseq2.

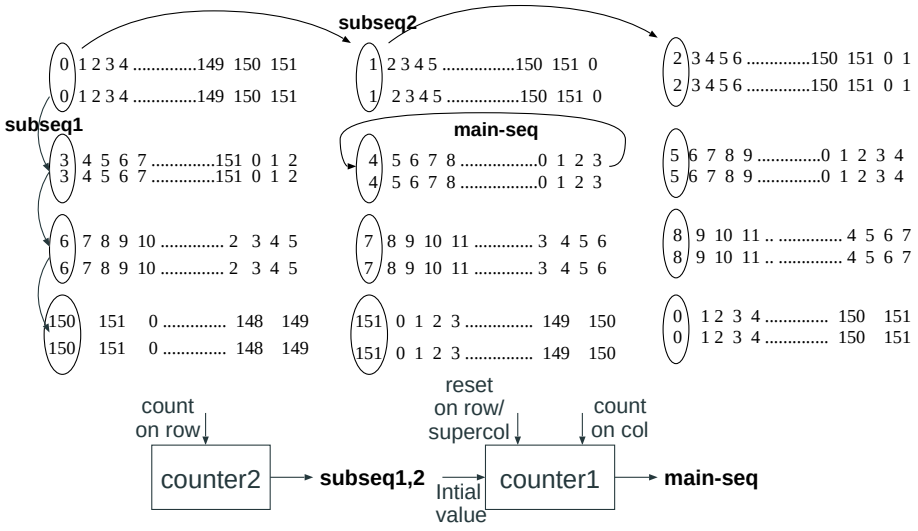

Figure 3: The address sequence for the modulo-152 address (sub)expression

Global algebraic transformations and loop-invariant code motion: this step consists in performing algebraic transformation on address expressions [15, 16]. These transformations results in different factorisations possibilities that leads to optimal common subexpression elimination and loop invariance. The algebraic transformation are done at the word level instead of the bit level as opposed to what happens during RT logic synthesis. Due to this step, we have successfully reduced the amount of address arithmetic operators of the input_deinterleaver loop in a factor of 1.6. For the error_correction&output loop the reduction reaches almost a factor 5.

Algebraic transformations are complemented by loop-invariant code motion, which consists in detecting the loop invariant part of an address expression and move it out of the actual loop scope to place it in the highest possible loop scope. Induction variable analysis has not been explored for the MP design. However, it can be very useful to increase the locality of the computation (due to the recursive nature of the resulting code) and when complemented with loop-invariant code motion results in code with reduced activity and hence energy.

All the high-level exploration decision have been expressed in an optimised C description of the MP and verified using simulations.

ELABORATION OF HDL DESCRIPTION AND SYNTHESIS

For behavioral synthesis we have divided the code into two different HDL files, one for each of the parallel loops obtained during exploration of the memory organi-

sation. Each loop can be synthesised and analysed separately since data consistency is guaranteed by the double buffer (ping-pong) memory architecture used.

After high-level exploration, the design flow proceeds by controlling the scheduling of the code using a mixture of traditional high-level synthesis methodologies and tool support (Synopsys-BC^{TM}) in our project) and our specific design-flow on top of it. To do this, we pre-synthesise at the RT-level design-encapsulated objects deduced from the improved code and corresponding to the different address (sub)expressions. These objects are actually Application Specific Units (ASUs) [7] which are modelled as Synopsys' Design-Ware components. They make the link for behavioral synthesis between the operation level and the address expression level. After RT synthesis of the Design-Ware components, and before scheduling, the designer can already obtain an accurate estimation of the area and critical path for each ASU. Once these constraints are satisfied, the Design-Ware components are then used as resources for high-level synthesis by performing allocation and assignment for the complex arithmetic code before scheduling.

SYNTHESIS RESULTS

We show in this section the results obtained after the scheduling and logic synthesis phases when comparing a traditional high-level synthesis approach [9, 10] with our high-level exploration phase and improved synthesis strategy. In order to perform the comparison under exactly the same timing constraints, the behavioral code used in both cases includes all task and system-level pipeline decision explained so far. Also, since Synopsys-BC does not have components in their generic library to map modulo operations, we have included in both versions the hierarchical counter organisation obtained to map the address (sub)expression affected by the modulo-152 operation.

Complexity	Input loop				Calc_out loop			
	Direct use of BC		ADOPT approach		Direct use of BC		ADOPT approach	
	no. operations				no. operations			
Arithmetic	29		18		43		9	
Scheduling	65		47		101		67	
	no. lines				no. lines			
Behv-HDL	233		341		263		370	
RT-HDL	1447		960		1434		1211	
Mapping	hierch	flat	hierch	flat	hierch	flat	hierch	flat
Gate-HDL	4838	3897	1350	1078	2768	2278	1997	1623

Table 1: Complexity figures for the MP loops: using traditional high-level synthesis and complemented by Adopt

We've performed several behavioral synthesis runs to detect the minium clock period at which the scheduling failed in meeting the cycle budget constraints. Using our synthesis strategy, the clock period resulted a factor of 2 smaller that the one needed using the traditional approach before the scheduling failed.

For logic synthesis, two different scripts have been evaluated. In both scripts, retiming of registers at the logic level is performed to improved the delay of the critical

path. The first of the scripts, namely **flattened** synthesis, consists in compiling and re-timing a previously flattened design. The second script, namely **hierarchical** synthesis, consists in first compiling hierarchically the design and then performing the re-timing step. Flattened synthesis has the advantage of exploiting global scope optimisations at the logic level, however its synthesis time is typically un-affordable for large complexity designs. Hierarchical synthesis is however much faster and therefore can cope with bigger complexity designs at the cost of local (hence suboptimal) logic level optimisations. All these synthesis strategies are supported by Synopsys' Design Compiler environment for logic synthesis [17].

CPU time (minutes)	Input loop				Calc_out loop			
	Direct use of BC		ADOPT approach		Direct use of BC		ADOPT approach	
Schedule	4		4		7		6	
Mapping	hierch	flat	hierch	flat	hierch	flat	hierch	flat
RT Synth.	60	67	17	20	84	80	23	30
Total	64	71	21	24	91	87	29	36

Table 2: CPU time (in minutes) for the MP loops: using traditional high-level synthesis and complemented by Adopt

The results are gathered in tables to show the difference between both synthesis approaches. The figures show the results for complexity, synthesis time, normalised timing, and area for each loop. The first main column of each table (Direct use of BC) presents synthesis results using a traditional high-level synthesis. The second column (ADOPT approach) presents the synthesis results when optimising the address arithmetic at the word-level algebraic and our improved synthesis strategy.

Table 1 shows the evolution of the complexity of the design for two different aspects. Firstly, in terms of arithmetic operations. Here, we do not include the shifting operations and divisions by power-of-two values because such operations are mapped as interconnections. Secondly, in terms of scheduling operations which includes ASU arithmetic, loop boundary and memory access operations.

Timing	Input loop				Calc_out loop			
	Direct use of BC		ADOPT approach		Direct use of BC		ADOPT approach	
Mapping	hierch	flat	hierch	flat	hierch	flat	hierch	flat
Delay/Max.Delay (ns)	.72	.63	.51	.46	1.0	.87	.65	.50
Rate/Max.Rate (MBs)	.57	.76	.91	1.0	.16	.19	.25	.33

Table 3: Normalised timing (critical-path and maximum throughput) for the MP loops: using traditional high-level synthesis and complemented by Adopt

The second issue we want to highlight in Table 1 is the evolution of the lines of code between the traditional approach and our approach. Whereas with our approach the number of lines between the behavioral code and the gate level code increases by a factor of 3-4, for the traditional approach it increases by a factor of 9 in the best case and a factor of 20 in the worst case. This explosion in the number of lines of

code at the gate level for the traditional approach have a strong impact on design-time, timing and area. Actually, the more lines of the gate level description has, the more time will be spent to get the final design meeting the timing constraints (longer iteration loops) and the more gates the final implementation will have (see Table 4).

Design time is illustrated in Table 2 in terms of CPU time invested for the scheduling and logic synthesis phases. Using our approach by a factor of 3-4 reduction in logic synthesis is obtained and a factor of 2-4 reduction in gate count is observed. Moreover, the critical path also benefits of our linear evolution of design complexity through abstraction levels (see Table 3). This proves that traditional high-level synthesis adds too many redundant architecture units (and gates) to the final implementation, in this way also increasing the size of the critical path. By better exploring the time-multiplexing alternatives, at a higher abstraction level than the traditional primitive operation level, the time-multiplexing overhead gets significantly reduced.

Active Area (no. gates)	Input loop				Calc_out loop			
	Direct use of BC		ADOPT approach		Direct use of BC		ADOPT approach	
Mapping	hierch	flat	hierch	flat	hierch	flat	hierch	flat
Comb-logic	3070	2447	767	588	1678	1476	1187	980
Muxes	131	103	21	15	75	49	24	28
Flip-flops	1388	960	265	248	652	631	419	478
Total	4589	3510	1053	851	2405	2156	1630	1486

Table 4: Active area (no. gates) after re-timing for the MP loops: using traditional high-level synthesis and complemented by Adopt

Finally, through all four tables we can also compare the results obtained using both the flattened and hierarchical logic synthesis scripts. Flattening the design before logic synthesis leads to better optimisation at the logic level but at the expense of more synthesis time. This increase of synthesis time has an exponential dependency with respect to the size of the behavioural description. Therefore, even when this effect is not fully exposed in our tables (due to the relative small size of our driver), in general this difference can be very important. Still, we can observe that when selecting hierarchical logic synthesis for the ADOPT approach, the overall quality of results (i.e., synthesis time, gate count and timing) outperforms the traditional high-level synthesis strategy even when using the time consuming flattened logic synthesis approach. This leads to the important conclusion that our approach bounds the potential complexity explosion when going from the behaviour to the gate level while still maintaining (and even improving) a high quality of the results. This smooth dependency with design complexity allows to absorb efficiently newer generations of complex applications while (re)using existing methodologies and tools in a very design-time and in a cost efficient way.

CONCLUSIONS

In this paper, we have shown how to explore at the high-level the address generation related issues before actual synthesis. This approach is complemented with an efficient use of a behavioral synthesis as back-end, which is based on the use

of application specific resources instead of the traditional primitive level arithmetic operators. Based on these principles, we have presented a guided tour of our design flow using a real-life application driver representative for the quite broad application domain of data-intensive applications. The applied design flow has successfully shortened design time with a factor from man-months to man-weeks of five. Moreover, the implementation obtained have resulted in a factor of 3-4 better throughput at a similar cost when compared to a more conventional RT based implementation approach.

References

[1] P.Lippens, J.Van Meerbergen, A.Van der Werf, W.Verhaegh, B.McSweeney, J.Huisken, O.McArdle, *Phideo: A silicon compiler for high speed algorithms*, Proc. 2nd ACM/IEEE European Conf. for Design Automation, pp. 436-441, 1991.

[2] D.Grant, P.Denyer, *Address generation for array access based on modulus m counters*, Proc. 2nd ACM/IEEE European Conf. for Design Automation, pp. 118-122, 1991.

[3] D.Grant, J.Van Meerbergen, P.Lippens, *Optimization of address generator hardware*, Proc. 5th ACM/IEEE European Design $ Test Conf., pp. 325-329, 1994.

[4] M.Miranda, F.Catthoor, H.De Man, *Address equation multiplexing for real time signal processing applications*, VLSI Signal Processing, VII, pp. 188-197, New York, 1994. J. Rabaey, P. Chau and J. Eldon (eds.). IEEE Press.

[5] M.Miranda, F.Catthoor, M.Janssen, H.De Man, *ADOPT: Efficient hardware address generation in distributed memory architectures*, Proc. 9th ACM/IEEE Intln. Symp. on System Synthesis, pp. 20-25, 1996.

[6] M.Miranda, M.Kaspar, F.Catthoor, H.De Man, *Architectural exploration and optimisation for counter based hardware address generation*, Proc. 8th ACM/IEEE European Design & Test Conf., pp. 293-298, 1997.

[7] S.Note, W.Geurts, F.Catthoor, H.De Man, *Cathedral-III: Architecture driven high-level synthesis for high throughput DSP applications*, Proc. 28th ACM/IEEE Design Automation Conf., pp. 597-602, 1991.

[8] W.Geurts, F.Catthoor, H.De Man, *Time constrained allocation and assignment techniques for high throughput signal processing*, Proc. 29th ACM/IEEE Design Automation Conf., pp. 124-127, 1993.

[9] M.McFarland, A.Parker, R.Camposano, *The high-level synthesis of digital systems*, Proc. of the IEEE, special issue on "The future of computer-aided design", Vol. 78, No. 2, pp. 301-318, Feb. 1990.

[10] D.Knapp, T.Ly, D.MacMillen, R.Miller, *Behavioral Synthesis Methodology for HDL-Based Specification and Validation*, Proc. of 32nd ACM/IEEE Design Automation Conf., 1995

[11] M.Miranda, F.Catthoor, M. Janssen, H.De Man, *High-Level Address Optimization and Synthesis Techniques for Data-Transfer Intensive Applications*, IEEE Trans. on VLSI Systems, no.4, vol.6, pp. 677-686. December, 1998.

[12] E.R Berlekamp, *Algebraic Coding Theory*, McGraw-Hill: New-York, 1968.

[13] F.Catthoor, S.Wuytack, E.De Greef, F.Balasa, L.Nachtergaele, A.Vandecappelle, *Custom Memory Management Methodology – Exploration of Memory Organisation for Embedded Multimedia System Design*, ISBN 0-7923-8288-9, Kluwer Acad. Publ., Boston, 1998.

[14] S.Gupta, M.Miranda, F.Catthoor, R.Gupta, *Analysis of High-level Address Code Transformations for Programmable Processors*, In Proc. IEEE Design and Test in Europe Conf., March, Paris, 2000.

[15] J.M.Janssen, F.Catthoor, H.De Man, *A specification invariant technique for operation cost minimisation in flow-graphs*, Proc. 7th ACM/IEEE Intln. Symp. on High-level Synthesis, pp. 146-157, 1994.

[16] J.M.Janssen, F.Catthoor, H.De Man, *A specification invariant technique for regularity improvement between flow-graph clusters*, Proc. 7th ACM/IEEE European Design & Test Conf., pp. 138-143, 1996.

[17] S.Carlson, *Introduction to HDL-based design using VHDL*, Synopsys Inc, California, 1990.