System level memory optimization for hardware-software co-design

Koen Danckaert Francky Catthoor‡ Hugo De Man‡

IMEC, Kapeldreef 75, B-3001 Leuven, Belgium
‡Professor at the Katholieke Universiteit Leuven

Abstract

Application studies in the areas of image and video processing systems indicate that between 50 and 80% of the area cost in (application-specific) architectures for real-time multi-dimensional signal processing (RMSP) is due to data storage and transfer of array signals. This is true for both single- and multi-processor realizations, both customized and (embedded) programmable targets. This paper has two main contributions. First, to reduce this dominant cost, we propose to address the ***system-level storage organization*** for the *multi-dimensional* ***(M-D) signals*** as a first step in the overall methodology to map these applications, before the HW/SW-partitioning decision. Secondly, we will demonstrate the usefulness of this novel approach based on a realistic test-vehicle, namely a quad-tree based image coding application.

1 Introduction and related work

In multi-media applications and others that make use of large multi-dimensional array-type data structures, a considerable amount of memory is required which is dominant in the system cost. This is especially true for embedded systems [12].

A system designer crafting an image or video processing system faces a large design space at the specification level. Up to now, only few hardware synthesis systems (see refs in [12]) try to reduce the storage requirements for array-type data structures, always focussed on single-processor realizations and with (severe) model limitations. Also in our own previous work on ATOMIUM [12], we have focussed mainly single-processor storage, dealing with loop transformations, memory allocation and in-place storage reduction for complex M-D array signal processing. Only recently, we have started studying the effect in a parallel processor context [4], but then focussed on the parallelisation issue and not on the hardware-software (HW/SW) co-design aspects.

Many papers have been published on the HW/SW co-design issues, including modeling and simulation [10], generic integration and interfaces [3, 5, 7], custom HW/SW synthesis [8], and especially partitioning. The latter category includes manual approaches: coarse-grain [2] or fine-grain [1], and automatic approaches: starting from hardware allocation first [9] or from software allocation [6]. All of these approaches ignore however the heavy impact of the data storage cost if they would be applied on data-dominated applications as in image processing.

As a result, the number of transfers to large memories or the amount of cache misses in the software part is not minimized at all. Consequently, there is a large potential loss in power consumption and a significant overhead in cycles (due to cache misses or off-chip access). This is especially undesirable in an embedded application.

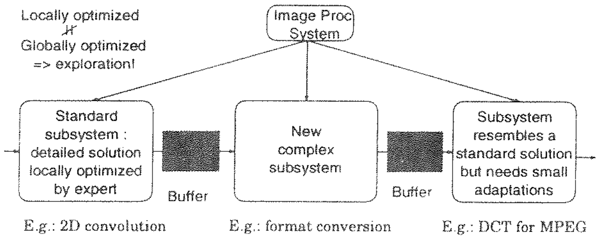

Figure 1: Data transfer and storage exploration for heterogeneous data dominated multi-process systems: possibilities for design support and optimization

Conventional HW/SW co-design approaches tackle the partitioning and load balancing issues as the only key point so they perform these first in the overall methodology. For the typical image processing system in figure 1, this means that all the submodules are first assigned to the best matched processors, and afterwards they are treated fully separately where they will be compiled in an optimized way onto the corresponding HW or SW processor. This strategy leads to a good load balancing solution but unfortunately, it will typically give rise to a significant buffer overhead for the mismatch between the data produced and consumed in the different submodules. To remedy this situation, the system-level memory management (SLMM) oriented methodology presented in this paper, first applies storage and transfer oriented optimizations *between* the different systems. Initially, all the submodules containing M-D processing are combined into one global specification model, and then optimized as a whole in terms of SLMM. It has to be stressed that it is indeed not required that the actual processor organization or even the processor partitioning is known initially in order to apply this SLMM methodology. We also apply more aggressive loop and data flow transformations than previously done. These transformations are able to significantly reduce the storage requirements for statically allocated memory in a multi-processing context.

Our SLMM techniques are complementary to the existing partitioning and load balancing techniques (as discussed earlier). In that way, they typically have little ef-

0-8186-7895-X/97 $10.00 © 1997 IEEE

fect on the finally obtained partitioning or load balancing properties, whereas a large storage size and/or data transfer gain can be obtained. They are also fully complementary to the traditional high-level synthesis step known as "register allocation/assignment" [11]. The latter should be applied after partitioning still because they are too much related to the detailed scheduling stage.

It should be emphasized too that decisions made at the SLMM level do translate into constraints on the M-D signal access, which directly influence the search space of the subsequent partitioning and processor mapping tasks. This is for instance true due to the restrictions on loop ordering and index expressions. Still, *only* the relative ordering of blocks of statements is decided and not yet the fully sequential execution or the "scheduling". Typically, these restrictions do not negatively affect the final outcome itself (see section 3)!

The overall approach will be illustrated with a real application in section 3. The application is described first in section 2.

2 The QSDPCM algorithm

2.1 Algorithm description

QSDPCM (Quadtree Structured Difference Pulse Code Modulation) is a compression technique for video [13]. It involves a motion estimation step, and a quadtree encoding of the motion compensated frame-to-frame difference signal. The algorithm optimizes both the displacement vector and the quadtree mean decomposition jointly such that the total frame-to-frame update information can be coded with a minimum number of bits. In this paper, we will assume that the images are in CIF format (528 x 288 for the luminance/chrominance signals together), and that the frame rate is 25 Hz.

A global view of the algorithm is given in figure 2. In a first step, the actual image is 4×4 mean subsampled, and matched with a 4×4 subsampled version of the reconstructed previous image. The initial guess of the displacement is computed using a 4×4 block matching (BM) algorithm with a search interval of +/- 4 pixels (full search) in the subsampled images. The resulting displacement vector is used as an inital displacement in the second stage, where an 8×8 BM algorithm with a search interval of +/- 2 pixels is applied on 2×2 mean subsampled versions. The displacement vector obtained in this way provides the initial guess for the QSDPCM algorithm.

The optimum displacement vector and the best quadtree decomposition are finally determined in a joint optimization procedure. For each displacement in a +/- 1 interval around the initial guess, the 16×16 difference signal is computed and 2×2 mean subsampled. The resulting 8×8 difference signals are quadtree encoded (with the local block means being Huffman coded). The displacement which requires the minimum number of bits for the quadtree decomposition is selected.

In a bottom-up quadtree decomposition of a two-dimensional signal, four adjacent subblocks are tested if they can accurately be represented by their mean value.

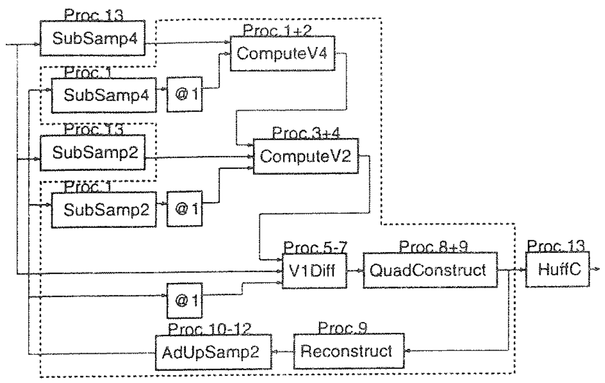

Figure 2: The QSDPCM video encoding algorithm and one processor partitioning option

If so, they are merged to a 4× larger subblock. The procedure can then be repeated recursively until the largest possible block size (8×8) is reached.

2.2 Algorithm code and cycles

From figure 2, it is clear that many submodules of the algorithm operate in a cycle with an iteration period bound (IPB) of 1 (i.e. one frame). Indeed, to compute the motion vectors in the 4×4 subsampled images, the previous frame must already have been reconstructed (and 4×4 subsampled too). The following submodules are not in this critical cycle: the computation of the 4×4 and 2×2 subsampled versions of the initial image, and the Huffman coding of the displacement vectors and quadtrees.

Figure 3 illustrates the algorithm structure as Silage code. In the main function, we see that the incoming and the reconstructed image are first 4×4 and 2×2 subsampled. Then, the estimated motion vectors are computed, first in the 4×4 and then in the 2×2 subsampled image. Based on these, the final motion vectors are computed with pixel accuracy. First, the nine possible 2×2 subsampled difference images are computed, and then they are quadtree encoded to determine which one requires the fewest bits. Finally everything is Huffman coded. The image is then reconstructed, and 2×2 adaptive upsampled. It is this upsampled image which will be the reference image for the next frame to be encoded. Also a 2×2 and 4×4 subsampled version of this reference are needed. Only when these reference images are available, we can start computing the motion vectors of the next frame. The following table gives the number of arithmetic operations in each function, for one block (first line) and for a whole image (second line).

Sub4	Sub2	V4	V2	V1	Quad	Rec	AdUp2
272 161K	320 190K	2673 1587K	3225 1915K	5184 3079K	2646 1571K	384 228K	4608 2737K

3 SLMM approach illustration

3.1 Design based on initial description

When a design is made based on this initial description and when flexibility is desired in the final implementation,

```
func main(image: P[N][M]) : P =
begin
 sub2[][]=SubSamp2(image[][]); sub4[][]=SubSamp4(image[][]);
 (v4x[][],v4y[][])=ComputeV4(sub4[][],rec4[][]@1);
 (v2x[][], v2y[][])=ComputeV2(sub2[][],rec2[][]@1,v4x[][],v4y[][]);
 (diffs[][][][][][])=V1Diff(image[][],rec[][]@1,v2x[][],v2y[][]);
 (quadmeans[][][][], quadcode[][][], v1x[][], v1y[][]) =
       QuadConstruct(diffs[][][][][][]);
  huffcode[] = HuffCoding(quadmeans[][][][], quadcode[][][],
                          v1x[][], v1y[][]);
  recsub[][] = Reconstruct(quadmeans[][][][], quadcode[][][],
                           rec2[][]@1, v1x[][], v1y[][]);
  rec[][] = AdUpSamp2(rec2[][]);
  rec2[][] = SubSamp2(rec[][]); rec4[][] = SubSamp4(rec[][]);
end;

func SubSamp2(image: P[N][M]) sub: P[][] =
begin
  (i: 0 .. N div 2 -1) ::
  (j: 0 .. M div 2 -1) ::
    sub[i][j] = (image[i][j] + image[i+1][j] + ...)/4;
end;

func ComputeV2(act2, prev2: P[N div 2][M div 2];
        v4x,v4y: A[N div Nb][B div Nb]) v2x, v2y: A[][] =
begin
  (xb : 0 .. N div Nb -1) ::  /* for each block */
  (yb : 0 .. M div Nb -1) ::
    (vx[xb][yb], vy[xb][yb]) = V2_block(act2[][],
             prev2[][], xb, yb, v4x[xb], v4y[yb]);
end;

func V2_block(act2, prev2: P[N div 2][M div 2]; xb, yb: A;
             v4x, v4y: A) v2x, v2y : A =
begin
  (vx : -2 .. 2) ::  /* Scan through search region */
  (vy : -2 .. 2) ::  /* Keep 8x8 current block + 8x12 region */
    begin            /* of prev frame in foreground          */
      (i : 0 .. 7) :: /* Accumulate differences over */
      (j : 0 .. 7) :: /* 8x8 block                   */
        SumAbsDiff[vx][vy][i*8+j+1] =
          SumAbsDiff[vx][vy][i*8+j] + abs(act2[xb*8+i][yb*8+j]
            -prev2[xb*8+v4x*2+vx+i][yb*8+v4y*2+vy+j]);
      /* Select vx,vy with minimum SumAbsDiff */
    end;
end;
```

Figure 3: Original Silage code

typically some submodules will be implemented in hardware and some in software. In this case, the submodules in the cycle can be assigned to hardware due to the high computation complexity and the high throughput requirement in this critical cycle. An exception can be made for SubSamp2, SubSamp4 and Reconstruct because these do not contain many operations.

When we take into account that programmable digital signal processors (SW) run at about 50 to 100 MHz, that the frame rate is 25 fr/s and that the arithmetic operation efficiency of these DSP processors is usually about 25-50% (the rest is lost in overhead for condition and address handling and the like), we arrive at a maximal load of 1 million operations per frame per SW processor. Given the operations in table 2.2, we need about 12 SW processors for the operations in the critical cycle. On the other hand, the functions which are not in the cycle can be combined into a single SW software processor.

Several approaches are feasible now for the critical cycle. We will illustrate that the data storage cost has a major effect on the cost related to HW/SW partitioning. Indeed, in the initial description of this algorithm, each submodule operates on a whole frame of the incoming video stream. This means that between two submodules, buffers are needed to store the results for a whole frame. For example, between SubSamp4 and ComputeV4, the two 4×4 subsampled images (actual and reconstructed previous frame) must be stored in a buffer. This has to be a background buffer as it is too big to be stored in foreground. If we have assigned SubSamp4 to software and ComputeV4 to hardware, then this buffer cannot be optimized away anymore. In the global initial QSDPCM description this leads to an overhead of buffers for 742K words, and an overhead of 2245K transfers operating on these frame size buffers. This number already assumes that a memory hierarchy is present, and that it is used in an optimal way. Without memory hierarchy (caching), there would be even 9599K background memory transfers.

If we do not need the flexibility, one or more dedicated hardware processor(s) can be designed to perform the functions which are in the cycle. How many will depend on the used hardware synthesis methodology. The IPB of 1 precludes simple pipelining. However, by performing a pipeline interleaving combined with loop folding operation such that each of the processors works on 1/12 of the frame successively, it is still possible to break this cycle which allows more freedom in processor partitioning.

When the flexibility is needed in all the functions, we again have several options. We can use data level parallelism by partitioning the frame into 12 equal parts and distributing the processing of these pixels over 12 processors. The advantage of this approach is that it is simple to program but the memory overhead is high, namely still 742K (540K with in-place mapping) words in total i.e. 61K (45K) per processor. The amount of transfers is 2245K. Moreover, each processor has to run the full code for all the functions. Alternatively, we can use task level parallelism by the more complex pipeline interleaving manipulation and by assigning the different functions (or parts of them) in chunks of 1 million operations over the different SW processors. This leads to the processor partitioning indicated in figure 2 by the processor numbers. The advantages are that the code size per processor is relatively low and especially that we need only 325K words with our memory management approach. The reason for this is that most buffers need only be present between two stages (although double-buffered). E.g. the buffer which contains the array diffs, is $342/12 = 28K$,and is only present (twice) between V1Diffs and QuadConstruct. In the data parallel case, *each* processor needs this 28K (although not double-buffered), which equals 342K total. The disadvantage is that the design time will be much higher due to the complex processor partitioning and memory management.

In summary, if we do not take into account the optimized storage related costs during the evaluation of the processor partitioning decisions, the data parallel option could have been selected. If later on the individual processor designers are faced with the given partitioning they would not be able to return to the much less costly task parallel case.

However, this is not the end of the story. This way of partitioning a system based on the initial description

```
func main(image: P[N][M]) : P =
begin
  (xb : 0 .. N div Nb -1) ::  /* for each block */
  (yb : 0 .. M div Nb -1) ::
  begin
    sub2[xb][yb][][] = SubSamp2_block(image[][],xb,yb);
    sub4[xb][yb][][] = SubSamp4_block(image[][],xb,yb);
    (v4x[xb][yb], vec4y[xb][yb]) = V4_block(sub4[xb][yb][][],
         rec4[xb][yb][][]);
    (v2x[xb][yb], vec2y[xb][yb]) = V2_block(sub2[xb][yb][][],
         rec2[xb][yb][][],v4x[xb][yb], v4y[xb][yb]);
    ....
end;
```

Figure 4: Globally transformed code

and only afterwards evaluating the data storage cost will lead to suboptimal designs. Typically it still gives rise to large buffers between the different submodules, even if it is already optimized towards memory as described above. Unfortunately, if the HW/SW partitioning is performed first in the design trajectory, these remaining buffers afterwards cannot be optimized away anymore.

For the QSDPCM application, we can do much better still by *applying aggressive storage oriented transformations before the HW/SW partitioning.*

3.2 Global optimizations

To apply our system-level memory optimizations, all functions (submodules) are first taken together in one big function. In this way, global transformations can be done which have a very big impact on the memory cost (and thus on the power and area cost).

We can apply a loop merging operation to this description, so that we have two outer loops that iterate over the block indices (see figure 4). Indeed, there are no dependences between two blocks of the same frame at all, so it is easy to see that this is a valid transformation[1].

Now we have an algorithm that operates block per block. All computations are done on the first block before we begin processing the second one. In this way, buffer memory for only one block (instead of one frame) will be required between the submodules. This reduces both the area and power requirements of the application because these small buffers can be stored in foreground memories.

The pipeline interleaving transformation is still possible. This will now allow pipelining at the level of blocks. While ComputeV4 is being executed on block x, ComputeV2 is executed on block $x-1$, and so on.

3.3 Optimizations between modules

Between V1Diff and QuadConstruct, nine difference blocks have to be kept in memory. It is obviously much better to merge the loops which iterate over the nine possible displacements in these functions. Then after computing one difference block, it is immediately quadtree encoded. Note however, that these quadtrees (which occupy less memory) have to be written to background anyway (only one of them will have to be read back). A possible choice would be not to write them to background, but only remember how many bits they took, and to recompute the best one afterwards. Then however, the 16×16 and 18×18 blocks from which the difference image was computed, would have to be read back, so here this is not good.

In ComputeV4, it is possible to interchange the loop which iterates over all possible displacements and the loop which scans over the 4×4 block itself. If the displacement loop is the outer one, the 4×4 block of the actual image and a 12×4 region of the previous image must be kept in foreground (if we work row-wise or column-wise) to avoid duplicate transfers from background memory to the datapath. If the block-scanning loop is the outer one, we can compute for each individual pixel the contribution to the mean absolute difference, and this for all $9 \times 9 = 81$ positions (see figure 5). This will obviously not be the best solution in this case, as it requires 81 values to be kept in foreground instead of 16.

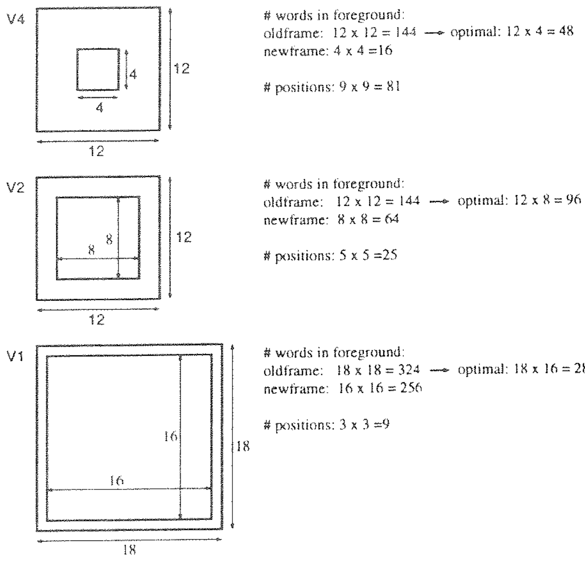

Figure 5: Memory requirements for ComputeV4,2,1

In ComputeV2 however, this loop interchange transformation is beneficial w.r.t. memory. If the displacement loop is the outer one here (figure 3), an 8×8 block of the actual image and a 12×8 region of the previous image must be kept in foreground. If the block-scanning loop is the outer one (figure 6), we need only storage for $5 \times 5 = 25$ positions of the displacement vector. Moreover, only a 12×5 region of the previous image must be in foreground (because each new pixel must be compared with all old pixels in a 5×5 square).

Another advantage of this modification is that, by merging the block-scanning loop (which is now the outer loop) with the block-scanning loop of SubSamp2, the 2×2 subsampled values do not have to be stored in background memory between SubSamp2 and ComputeV2. Note that this would also not be possible if we had assigned these functions to different chips in the initial partitioning. On

[1]Note that also a loop tiling transformation of the loops in SubSamp2 and SubSamp4 has been applied to make the global loop merging possible. Hence, the subsampled images are now 4-dimensional signals.

```
func V2_block(act2:P[8][8];prev2: P[N][M];v4x,v4y:A) v2x,v2y: A=
begin
  (i: 0 .. 7) :: /* For each pixel in current block, compute  */
  (j: 0 .. 7) :: /* contrib to 5x5 possible SumAbsDiff values */
  begin
    (vx : -2 .. 2) ::  /* Scan through search region */
    (vy : -2 .. 2) ::  /* Keep 5x5 SumAbsDiff values + 5x12  */
    begin              /* region of prev frame in foreground */
      SumAbsDiff[vx][vy][i*8+j+1] = SumAbsDiff[vx][vy][i*8+j]
       +abs(act2[i][j]-prev2[xb*8+v4x*2+vx+i][yb*8+v4y*2+vy+j]);
    end;
  end;
  /* Select vx,vy with minimum SumAbsDiff */
end;
```

Figure 6: Transformed code for V2_block

the other hand, it influences the way we will partition the new transformed algorithm: SubSamp2 and ComputeV2 must now be kept together.

In V1diff and Quadconstruct, this analysis is a bit more difficult to make, as for every possible displacement, not the mean absolute difference is computed (which would require only $3 \times 3 = 9$ values), but the quadtree decomposition. So if we want to reduce the required memory (18×18 pixels of the previous and 16×16 of the actual image), we have to merge the loops which compute the difference signal and the quadtree decomposition.

If we work line per line, when 2 lines of the difference signal have been computed, the first level of the bottom-up construction of the quadtree can already be applied, and the results written to background memory. Only four values are needed for the further construction of the quadtree (and must be kept in foreground). So we need to keep 2 rows of 8 values, 2 rows of four values, and 2 rows of 2 values in foreground, and this for all nine possible quadtree decompositions. Instead of a 16×16 block, we then only have to keep two rows (2×16) of the actual image block in foreground (note that one line is not enough because the quadtree is constructed on the 2×2 subsampled difference signal). Likewise, we need only four rows (4×18) of the previous image block.

3.4 Memory optimized design

The only background memories we still need are the actual and previous frame buffer (2 x 152K, which can be reduced to 166K in total by appropriate in-place mapping between these two frames). The number of transfers to background has been reduced from 2245K to 1238K.

If we choose for a data parallel solution in this case, each of the 12 processors will be working on another block. So each processor has much code to execute. The frame memory has to be organised such that it allows simultaneous accesses from all processors. One solution is to use a 12-port memory. A much better solution is to use 12 memories, and store 1/12 of the number of blocks in each memory, in an interleaved way.

In our optimizations, we have imposed the constraint that following pairs of functions must be on the same chip (so without pipelining between them): Sub4 & V4, Sub2 & V2, V1 & Quad. This means that a purely algorithmic parallel solution is not possible here. A mixed data/algorithmic parallel solution is possible though, where e.g. two processors execute Sub2 & V2, each on another block. Between the pipelined functions, double buffers are still needed, but our optimizations have made those very small.

4 Conclusion

In this paper, we have demonstrated that the HW/SW partitioning approach which is typically followed in conventional HW/SW co-design papers does not lead to good results for data-dominated applications as image and video processing. Instead of performing the processor partitioning prior to the hardware synthesis or software compiling steps per processor, first a system-level memory management approach should be applied to the global algorithm, leading to significant reshuffling and modification of the initial processes or submodules. Only then the partitioning step should be applied. The feasibility and effect of this new approach has been substantiated on a realistic image processing application, leading to a reduction of the storage size from 742K 12-bit words to 166K words and a decrease of the number of memory accesses from 2245K to 1238K.

References

[1] E.Barros, W.Rosenstiel, "A method for hardware/software partitioning", *Proc. CompEuro Conf.*, Den Haag, The Netherlands, May 1992.

[2] K.Buchenrieder, A.Sedlmeier, C.Veith, "HW/SW codesign with PRAMs using Codes", *Proc. IFIP Conf. Hardware Description Languages*, Elsevier, Amsterdam, pp.55-68, 1993.

[3] P.Chou, R.Ortega, G.Borriello, "The Chinook hardware/software co-synthesis system", *Proc. 8th ACM/IEEE Intnl. Symp. on System-Level Synthesis*, Cannes, France, Sep. 1995.

[4] K.Danckaert, F.Catthoor, H.De Man, "System-level memory management for weakly parallel image processing", *Proc. EuroPar Conference*, Lyon, France, August 1996. "Lecture notes in computer science" series, Springer Verlag, pp.217-225, 1996.

[5] H.De Man, I.Bolsens, B.Lin, K.Van Rompaey, S.Vercauteren, D.Verkest, "Co-design of DSP systems", NATO Advanced Study Institute on "Hardware/Software Co-design", Tremezzo, Italy, June 1995.

[6] R.Ernst, J.Henkel, T.Benner, "Hardware-software cosynthesis for microcontrollers", *IEEE Design and Test of Computers*, Vol.10, No.4, pp.64-75, Dec. 1993.

[7] D.Gajski, F.Vahid, S.Narayan, J.Gong, "Specification and design of embedded systems", Prentice Hall, 1994.

[8] G.Goossens, F.Catthoor, H.De Man, "Integration of signal processing systems on IC architectures with mixed hardware/software", *IFIP Intnl. Workshop on Hardware/Software Co-design*, Grassau, Germany, May 1992.

[9] R.Gupta, G.De Micheli, "Hardware-software cosynthesis for digital systems", *IEEE Design and Test of Computers*, Vol.10, No.3, pp.29-41, Sep.1993.

[10] A.Kalavade, E.Lee, "A hardware-software codesign methodology for DSP applications", *IEEE Design and Test of Computers*, Vol.10, No.3, pp.16-28, Sep.1993.

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

[12] L.Nachtergaele, F.Catthoor, F.Balasa, F.Franssen, E.De Greef, H.Samsom, H.De Man, "Optimisation of memory organisation and hierarchy for decreased size and power in video and image processing systems", *Proc. Intnl. Workshop on Memory Technology, Design and Testing*, San Jose CA, pp.82-87, Aug. 1995.

[13] P. Strobach, "QSDPCM – A New Technique in Scene Adaptive Coding," *Proc. 4th Eur. Signal Processing Conf.*, EUSIPCO-88, Grenoble, France, Elsevier Publ., Amsterdam, pp.1141–1144, Sep 1988.