Mapping Control-Intensive Video Kernels onto a Coarse-Grain Reconfigurable Architecture: the H.264/AVC Deblocking Filter

C. Arbelo[1], A. Kanstein[2], S. López[1], J.F. López[1], M. Berekovic[3], R. Sarmiento[1] and J.-Y. Mignolet[3]
[1] Research Institute for Applied Microelectronics (IUMA), Department of Electronic Engineering and Control (DIEA), University of Las Palmas de Gran Canaria, E-35017, Spain.
[2] Freescale Inc., Toulouse, France
[3] IMEC, Leuven, Belgium

Abstract

Deblocking filtering represents one of the most compute intensive tasks in an H.264/AVC standard video decoder due to its demanding memory accesses and irregular data flow. For these reasons, an efficient implementation poses big challenges, especially for programmable platforms. In this sense, the mapping of this decoder's functionality onto a C-programmable coarse-grained reconfigurable architecture named ADRES (Architecture for Dynamically Reconfigurable Embedded Systems) is presented in this paper, including results from the evaluation of different topologies. The results obtained show a considerable reduction in the number of cycles and memory accesses needed to perform the filtering as well as an increase in the degree of instruction parallelism (ILP) when compared with an implementation on a Very Long Instruction Word (VLIW) dedicated processor. This demonstrates that high ILP is achievable on the ADRES even for irregular, data-dependent kernels.

1 Introduction

The architectural requirements imposed by nowadays multimedia applications are very strict, in the sense that high levels of performance must be achieved under severe area occupation and/or power dissipation constraints. In addition, the architecture must be flexible enough in order to support different versions of one particular application, while maintaining the design effort and time-to-market as low as possible.

This situation becomes even more stringent in the case of video coding applications based on the H.264/AVC [1] video coding standard. H.264/AVC represents the state of the art standard as it provides better coding efficiency when compared with its predecessors such as MPEG-2 [2], MPEG-4 [3] and H.263 [4].

This work was partially supported by the Spanish Ministry of Education and Science (MEC) under project TEC2005-08138-C02-01/MIC

However, these improved characteristics come at the expense of an increased computational cost. For the particular case of the decoder subsystem, the complexity increases by a factor of two.

Coarse-grained reconfigurable architectures [5] appear as potential candidates for the implementation of real time H.264/AVC video codecs, as they achieve high performance while maintaining a degree of flexibility close to general purpose DSP processors. The ADRES (Architecture for Dynamically Reconfigurable Embedded Systems) architecture developed at IMEC represents a suitable option for multimedia applications, as it outperforms other optimized DSP processors [6].

The results obtained by mapping the adaptive deblocking filter from a baseline profile H.264/AVC video decoder onto the ADRES architecture are presented in this paper. By using a proper set of optimization techniques, the deblocking filtering kernel has been considerably accelerated into the ADRES architecture, yielding as a result, an increase of the whole decoder performance.

The rest of this paper is organized as follows. Section 2 gives a functional overview of a generic H.264/AVC decoder with special emphasis on the deblocking filtering task, while in Section 3 the ADRES architecture together with its associated compiler named DRESC (Dynamically Reconfigurable Embedded Systems Compiler) are introduced. The details about the different versions and architectures developed for the adaptive deblocking filtering kernel, together with the implementation results obtained, are given in Section 4 and finally some concluding remarks as well as future research directions are outlined in Section 5.

2 The H.264/AVC decoder

The functional block diagram of a generic hybrid video decoder based on the H.264/AVC standard is shown in Fig. 1, where the deblocking filtering kernel has been highlighted.

978-3-9810801-2-4/DATE07 © 2007 EDAA

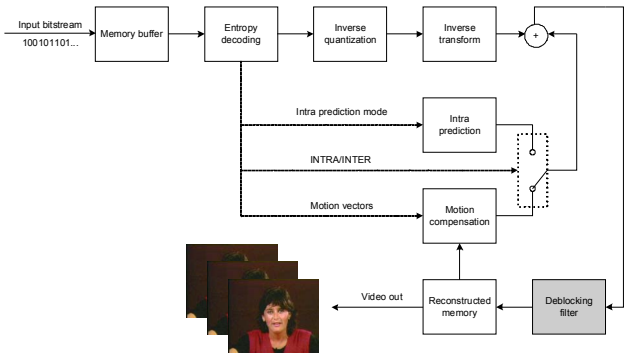

Fig. 1: Block diagram of a generic H.264/AVC decoder.

As it can be observed from this figure, the incoming video bitstream is stored on a memory buffer in order to be parsed and decoded by the entropy decoding stage. The syntax elements obtained after this process for each macroblock (16×16 luminance pixels and two blocks of 8×8 chrominance pixels) are demultiplexed and sent to the different functional kernels involved in the decoding process. In particular, the syntax elements related to the coding of the luminance and chrominance residual samples of the current macroblock (MB) are re-ordered by following a typical inverse scan procedure and passed to the inverse transform kernel. In parallel, a predictor is composed from previously decoded pixels in the same frame (intra coded macroblocks) or from pixels pointed by the received motion vectors belonging to frames previously decoded (inter coded macroblocks) depending on the information stored in the MB layer of the received bitstream. The decoded and inversely transformed residual samples are then added to the selected predictor. Finally, a deblocking filter reduces the presence of annoying blocking artifacts resulting from the block-based processing, so that the original macroblock is recovered with minimal quality losses.

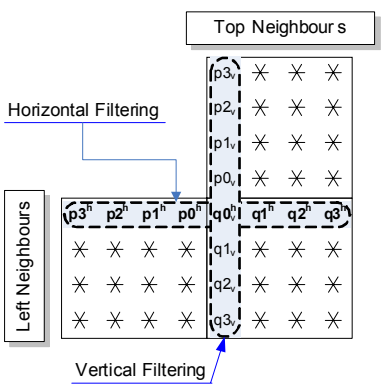

Fig. 2: Pixels distribution for horizontal and vertical filtering.

The deblocking consists of horizontally filtering the vertical edges and vertically filtering the horizontal ones, taking as the unfiltered inputs the pixels marked in Fig. 2. For each edge within a reconstructed MB, independently of its nature, filtering starts computing the boundary strength (bS), which is a number ranging from zero (no filtering) to 4, that varies adaptively per block according to the coding conditions and the block pixel values. For strength 4, up to three pixels on either side of the edge can be modified *(strong filter)*. On the other hand, for strengths between 1 and 3, only two pixels on either side of the edge may be affected (*weak filter*). Once the strength has been determined for an edge, a gradient-like analysis is performed to determine if the edge sharp should be preserved or filtered to attenuate blocking artifacts. In this sense, the filter is disabled, regardless of the filter strength, if $|p0-q0|\geq\alpha$ or $|p1-p0|\geq\beta$ or $|q1-q0|\geq\beta$, where α and β mainly depend on the quantization parameter.

3 The ADRES/DRESC framework

The ADRES coarse-grained array processor, as shown in Fig. 3, consists of an array of functional units (FUs), enhanced with register files (RFs) and connected through routing resources like wires, multiplexors and busses. ADRES is a templatized architecture that allows the construction of processors from an arbitrary number of function units, register files and interconnects.

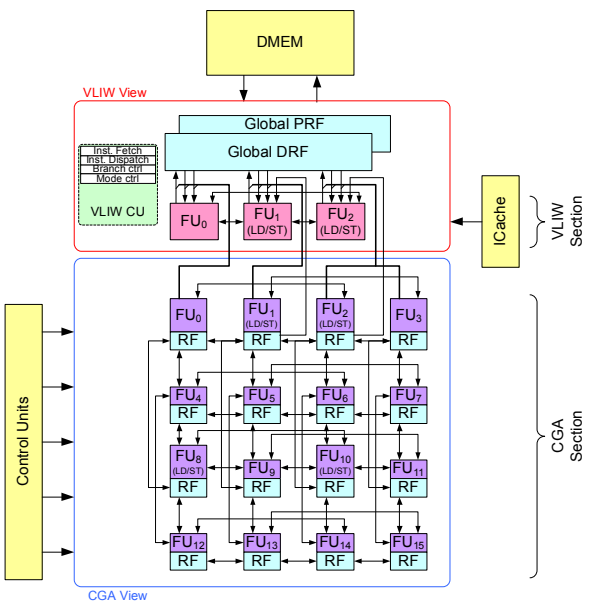

Fig. 3: Architecture of the ADRES coarse-grain reconfigurable array

The results of the ADRES FU, shown in Fig. 4, can be written to a local register file (RF), which is usually small and has fewer ports than the shared RF, or routed directly to the inputs of other FUs. The multiplexors shown in Fig.4 are used for routing data from different sources, while the configuration RAM acts as an instruction memory to control these components. It stores a number of configuration contexts locally, which are loaded on a cycle-by-cycle basis. In addition, ADRES provides predicates to let the compiler remove the control flow inside loops and efficiently implement prologues and

epilogues of SW pipelined loops with conditional execution.

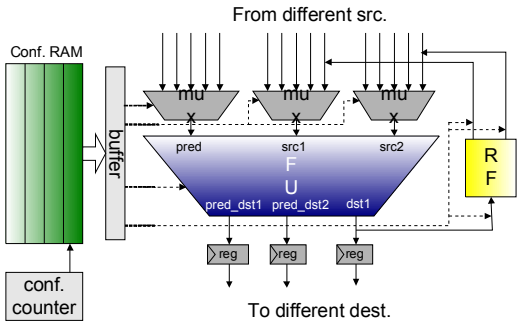

Fig. 4: ADRES coarse-grain array node

For a complex architecture like ADRES, an automatic design methodology and programming tools are essential. Therefore the ADRES architecture has been developed together with its own C compilation framework, called DRESC (Dynamically Reconfigurable Embedded System Compiler). It maps computation-intensive kernels, typically dataflow loops, onto the reconfigurable array, whereas the remaining code is mapped onto the VLIW processor. The data communication between the VLIW processor and the reconfigurable array is performed through the shared RF and shared memory.

The compiler framework is shown in Fig.5. A design starts from a C-language description of the application. On the basis of execution time and possible speedup, the profiling and partitioning step identifies the candidate computation-intensive loops (kernels) for mapping onto the reconfigurable array.

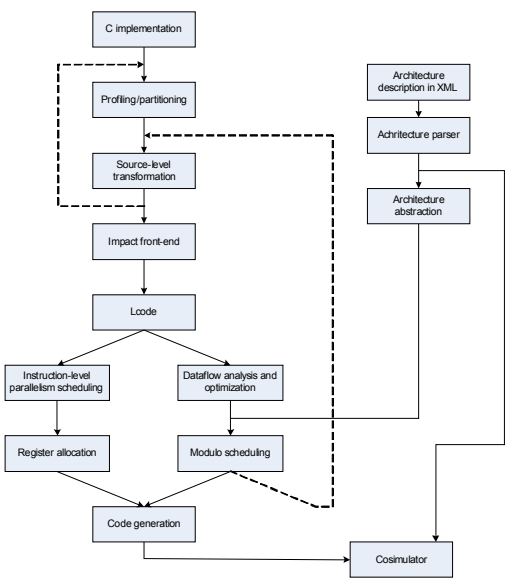

Fig. 5: Dynamically Reconfigurable Embedded System Compiler (DRESC) Framework

Source-level transformations allow the pipelining of the kernel software to maximize performance. In the next step, a VLIW compiler framework called IMPACT [9] is used as a front-end and for high-level optimizations. IMPACT provides the control flow analysis to optimally replace branching with predicated operations [10], emitting an intermediate representation, called Lcode, which is used as input for scheduling together with the XML-based architecture.

Applying a modulo scheduling algorithm to the Lcode achieves high parallelism for the kernels, whereas applying traditional instruction-level parallelism (ILP) scheduling techniques yields the available moderate parallelism for the non-loop code. The tools automatically identify and handle communications between these two parts and generate scheduled code for both the reconfigurable array and the VLIW processor.

Predicates play an important role in the transformation of the code for modulo scheduling, because loop code must be free of branches and must feature a single exit point. This part of optimizations is provided by IMPACT. Also, predicates are needed to efficiently implement software pipelining, by guarding the execution during the prologue and epilogue stages implemented within the loop body. In array mode predicates are not used to guard operations but they provide the write enables to the register file and the memory, and special paths are used to route predicates to the loop stop signal.

The modulo scheduling algorithm utilizes a graph-based architecture representation, called MRRG (Modulo Routing Resource Graph), to model resources in a unified way, expose routing possibilities and enforce modulo constraints. The algorithm is based on congestion negotiation and simulated annealing methods. Starting from an invalid schedule that overuses resources, it tries to reduce overuse over time until a valid schedule is found [11]. One main advantage of the DRESC framework is its flexibility: the tools are designed to be retargetable within the ADRES template.

Some code transformations are required to allow DRESC to map code onto the reconfigurable array, and to achieve high performance. These transformations are described in the following section in an exemplary way, using the deblocking filtering kernel from H.264/AVC.

4 Mapping of the H.264/AVC deblocking filter onto the ADRES architecture

This section describes the code versions developed in order to optimize the deblocking filter performance as much as possible together with the most significant mapping results obtained for each version. It is important to highlight that the results shown in this section have been obtained by using the H.264/AVC decoder public C code from libavcodec (available in the FFmpeg library on sourceforge.net/projects/ffmpeg), with some modifications to make the mapping on ADRES feasible.

4.1 Preliminary profiling results

Before mapping the code corresponding to the H.264/AVC decoder onto the ADRES architecture, it is

necessary to select which loops should be mapped onto the coarse grain array and which ones should be executed on the VLIW. For this purpose, in order to identify which functions involved in the deblocking filter are the most time consuming, profiling tools such as *Gprof* and *Quantify* have been used.

The profiling results have been obtained by decoding the foreman_cif_bl.264 bitstream. This input bitstream results from encoding 300 frames of the FOREMAN raw video sequence in CIF format (352×288 pixels) with the H.264 baseline profile public C code [12] using no rate control. From these results, it is inferred that the functions where most time is spent, and therefore, the candidates for parallelization on the reconfigurable matrix, are the following:

- filter_mb
- filter_mb_luma_W
- filter_mb_lumaV_S
- filter_mb_lumaH_S
- filter_mb_chroma_W
- filter_mb_chroma_S

The function filter_mb is the main function and is responsible for two tasks: For calculating the values of the boundary strength (bS) for the horizontal and vertical edges, and for invoking the appropriate functions according to the type of filtering; vertical or horizontal, luminance or chrominance. The function filter_mb_luma_W performs the vertical or horizontal luminance weak filter, whereas the next two compute the vertical and horizontal luminance strong filter, respectively. Similarly, the functions filter_mb_chro-ma_W and filter_mb_chroma_S perform the vertical and horizontal weak and strong filter for the chrominance samples.

4.2 Versions developed

Several versions have been developed in order to achieve a better deblocking filter performance in terms of cycle counts, the most representative ones being summarized in Fig. 6.

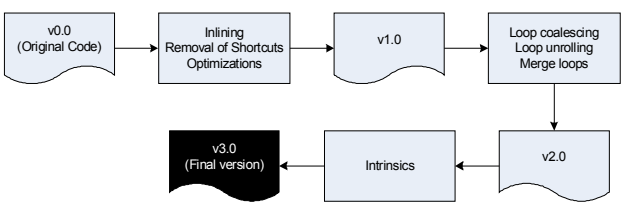

Fig. 6: Diagram of the developed versions

The first modifications performed are the constraint-removing transformations, necessary to map the loops on the array. These loop requirements basically consist on the removal of the exit points and the function calls inside the loop body. In this sense, function inlining is a widely used optimization technique to reduce the overhead associated with function calls at the expense of increase code size if the inline function is called in multiple places. Once it is possible to map the loops onto ADRES, several optimizations can be done to speed up the code, since a loop may not produce good performance in its original form though it is pipelineable. This purpose can be achieved by introducing some techniques such as *loop coalescing*, *loop unrolling* and *loop merging*; followed by the inclusion of special functions called *intrinsics*.

Loop coalescing consists on the combination of a loop nest into a single loop, thus increasing the number of iterations of the loop. Currently the DRESC compiler can only pipeline the innermost loop of a nested loop; if the outer loops contribute to a significant portion of the total execution time, or the total number of iterations of the innermost loops is too small so that the overhead of prologue and epilogue is dominant, only pipelining the innermost loops won't produce good performance.

Loop unrolling expands a loop as each new iteration contains several copies of a single iteration. In this way, the number of iterations of the loop is reduced as many times as copies are made. Applied to the ADRES architecture, it helps to increase the size of loop bodies so that pipelining is more efficient since more instructions can be scheduled for parallel execution.

The last loop optimization, loop merging, combines different loops into a single one, increasing the loop body and therefore reducing the overhead of prologue and epilogue.

Another technique to be considered to improve the performance of the code is the addition of *intrinsics*. An intrinsic is a function known by the compiler directly mapped to a sequence of one or more assembly language instructions. Intrinsic functions perform simple and useful operations that are difficult to express concisely in C or C++, with the additional advantage that no calling linkage is required. Two different types of intrinsics were added; common intrinsics and special ones known as SIMD (*Single Instruction Multiple Data*).

Common intrinsics perform special arithmetic operations in a more efficient way, whereas SIMD are instructions designed to accelerate the application by exploiting the parallelism since they let one instruction perform the same operation on multiple data elements. After exhaustively studying the code for potential opportunities to use intrinsics functions, the inclusion of intrinsics to perform average, clipping and shift, and round operations, appear as potential alternatives to speed up the filtering functions.

The SIMD instructions implemented are the dot product operation and the inner sum operation. The first one consists on the sum of 4 8-bit multiplications and the other one performs the sum of 4 8-bit values. These instructions enhance the performance of the luminance and chrominance filters.

4.3 Mapping results

To verify the complete application, a co-simulator is used in order to simulate the compiled application using two different bitstreams of 300 frames each. The

bitstreams used are NEWS and FOREMAN, both of them in CIF format (352x288 pixels). While the FOREMAN sequence has been encoded with the H.264 reference encoder as previously mentioned, the NEWS sequence has been encoded with a proprietary encoder and rate control set to 256kbps. The selected testbench sequences have very different spatial and temporal characteristics as NEWS represents the typical "head and shoulders" sequence with low motion, large static background areas and no context changes, whereas FOREMAN is a highly textured sequence with a chaotic motion field produced by the combination of local and global movements. The first frame of each sequence is shown in Fig. 7.

Fig. 7: First frame of NEWS (left) and FOREMAN (right) sequences

Relating to the architecture, a 4x4 instance from ADRES template is used consisting on 12 distributed RFs (DRFs) and a register file shared by the VLIW processor and 16 FUs. These include 16 ALUs, 8 multipliers and 4 load/store units being each of them connected not only to connected to the 4 nearest neighbours, but also with FUs within one hop (4x4_dresc_arch_meshplus_12DRF).

The results of the mapped loops for NEWS sequence are listed in Table 1, where the total cycle count and IPC (Instructions Per Cycle) for the three versions developed are presented. This last parameter reflects the parallelism achieved, being the maximum value equal to the number of FUs (16 in this case).

	VLIW		ADRES					
	V0.0		v1.0		v2.0		v3.0	
	cycles	IPC	cycles	IPC	cycles	IPC	cycles	IPC
filter_mb	224.109.963	0,98	167.827.505	3,63	122.195.422	6,64	109.293.556	7,31
filter_mb_luma_W	107.052.758	2,66	50.553.210	9,19	38.087.280	13,49	29.799.770	11,47
filter_mb_lumaV_S	3.079.483	3,26	812.258	14,24	895.061	13,78	622.994	13,01
filter_mb_lumaH_S	3.316.625	3,17	914.525	13,39	902.750	13,47	690.800	13,23
filter_mb_chroma_W	47.999.672	1,87	34.946.144	7,34	14.478.264	12,15	11.797.104	9,26
filter_mb_chroma_S	3.792.376	1,89	1.369.032	8,90	889.084	11,38	810.404	8,83

Tab. 1: Number of cycles and IPC for the deblocking filter functions mapped onto ADRES

The results prove that mapping the loops on ADRES substantially decreases the cycle counts besides increasing the parallelism from approximately 2 to 12 instructions per cycle. It has to be pointed out that especially with the deblocking filter, the IPC increases proportionally more than the cycle count decreases, because more instructions have been predicated when mapping the conditional code to the array, and therefore more code is executed speculatively

Finally, the overall vs. the kernel cycle counts for the sequences commented before are outlined in Table 2. Comparing the results for the last version against the ones obtained from the original code, it is noticeable a speed-up of approximately 3 times for the kernel. The speed-up is constrained by the high complexity of the deblocking filter which requires conditional processing on the block edges and sample level, resulting in many conditional branches that become a challenge for parallel processing. The overall speed-up is lower since only one of the most computationally intensive kernels, the deblocking filter, has been mapped and optimized. In order to achieve better results some other significant kernels, such as the motion compensation and the inverse transform, should be mapped.

	v0.0		v3.0		speed-up	
	overall	kernel	overall	kernel	overall	kernel
NEWS	2.207.500.396	389.350.877	1.984.299.764	153.014.628	1,11	2,54
FOREMAN	3.075.140.626	745.777.008	2.576.058.975	230.920.308	1,19	3,23

Tab. 2: Mapping speed-up obtained

4.4 Architecture exploration

Since ADRES is a template for a CGRA, architecture variations can be easily derived using the XML-based architecture description language. Architectural aspects such as number of resources, interconnection topologies and number of distributed register files can be easily specified by the description language. This flexibility together with the automatic support from the DRESC framework, allow the exploration of new architectures to find an optimal instance for a given application.

From the architecture instance used for the kernel optimization (mesh-plus connectivity with 12 DRFs), two experiments have been done varying only one architecture parameter and fixing the others. The first experiment consists on modifying the interconnections between FUs. The three instances used are shown in Fig. 8 while the results for each of them, in terms of cycle counts and IPC, are listed in Table 3.

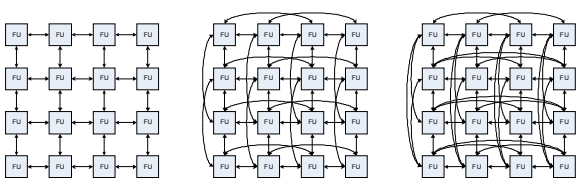

Fig. 8: Architecture instances (a) Mesh, (b) Mesh-Plus and (c) Mesh-Full.

As expected, the more interconnections between FUs, the less the cycles are needed, due to the fact that there is a better routability between components and therefore, the operations can be easily scheduled. On the other hand this

may lead to more wires and wider multiplexors, which indicates more area, longer delays and higher power consumption.

	Mesh		Mesh-Plus		Mesh-Full	
	cycles	IPC	cycles	IPC	Cycles	IPC
filter_mb	109.375.152	6,93	109.293.556	7,31	108.701.536	7,18
filter_mb_luma_W	35.442.330	9,97	29.799.770	11,47	28.918.120	11,45
filter_mb_lumaV_S	686.082	10,93	622.994	13,01	626.937	13,27
filter_mb_lumaH_S	753.600	11,86	690.800	13,23	690.800	13,34
filter_mb_chroma_W	13.227.056	7,62	11.797.104	9,26	10.099.036	11,62
filter_mb_chroma_S	959.896	7,16	810.404	8,83	806.140	8,87

Tab 3. : Cycles and IPC for different interconnection topologies

The second experiment consists on varying the number of distributed register files. Figure 9 shows the instances used and the results are presented in Table 4.

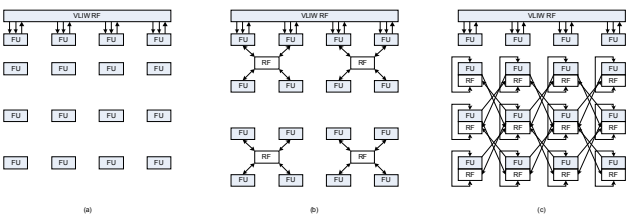

Fig. 9.: Architecture instances (a) No local DRFs, (b) 4 shared local DRFs and (c) 12 local DRFs.

	No DRFs		4-Shared DRFs		12-DRFs	
	cycles	IPC	cycles	IPC	Cycles	IPC
filter_mb	129.067.024	7,81	112.845.676	7,23	109.293.556	7,31
filter_mb_luma_W	38.616.270	10,96	31.739.400	10,03	29.799.770	11,47
filter_mb_lumaV_S	950.263	12,12	686.082	13,56	622.994	13,01
filter_mb_lumaH_S	1.008.725	10,65	804.625	10,73	690.800	13,23
filter_mb_chroma_W	13.137.684	10,48	11.886.476	10,26	11.797.104	9,26
filter_mb_chroma_S	952.028	8,83	818.272	10,19	810.404	8,83

Tab. 4: Cycles and IPC for instances with different DRFs

Results show an important decrease in the cycle counts when the number of DRFs is increased. As each FU has a local RF, the number of required cycles is reduced, since it is not necessary to spend cycles loading and storing the intermediate data to the global RF. In contrast, the addition of DRFs involves an augment of the area needed and in the number of configuration bits to provide an address for each port.

5 Conclusions and further research

This paper presents the procedure and the most significant results obtained by mapping a H.264/AVC standard-compliant deblocking filter onto the ADRES architecture, In particular, 6 functions that belong to this kernel have been optimized and mapped onto the array in order to accelerate the code. Big efforts have been made to overcome the inherent complexity of the deblocking filter. In addition, architectural tradeoffs have been inspected in order to accelerate the filtering kernel at the cost of an increase in the hardware cost.

The results show a speed-up of approximately 3 times for the kernel and 1.15 times for overall performance over a 4-issue VLIW, as well as an important increase in the degree of parallelism.

Acknowledgment

The authors wish to thank IMEC for supporting the student internships which made this work possible, and for providing access to the tools.

References

[1] T. Wiegand (Ed.), "Draft ITU-T Recommendation H.264 and Draft ISO/IEC 14496-10 AVC," *Joint Video Team of ISO/IEC/JTC1/SC29/WG11 & ITU-T SG16/Q.6 Doc. JVT-G050*, Pattaya, Thailand, March 2003.

[2] ISO/IEC JTC1/SC29 CD 11544, "Coded Representation of Picture and Audio Information - Progressive Bi-Level Image Compression", Recommendation H.262 Nov., 1993.

[3] MPEG-4 Part 2: Visual (IS 14496-2), Doc. N2502, Atlantic City, N.J., USA, October 1998.

[4] "Video coding for Low Bit Rate Communication", ITU-T Recommendation H.263, July 1995.

[5] B. Mei, *A Coarse-Grained Reconfigurable Architecture Template and its Compilation Techniques*, PhD Thesis IMEC, January 2005.

[6] M. Berekovic, A. Kanstein, D. Desmet, A. Bartic, B. Mei and J.Y. Mignolet, "Mapping of Video Compression Algorithms on the ADRES Coarse-Grain Reconfigurable Array", *Workshop on Multimedia and Stream Processors'05*, Barcelona, November 12, 2005.

[7] H. Huebert, B. Stabernack and H. Richter, "Tool-Aided Performance Analysis and Optimization of an H.264 Decoder for Embedded System", *Consumer Electronics, 2004 IEEE International Symposium on Consumer Electronics*, pp. 400 - 405, Sept. 1-3, 2004.

[8] X. Quan, L. Jilin, W. Shijie, Z. Jiandong. "H.264/AVC Baseline Profile Decoder Optimizations on Independent Platform", *International Conference on Wireless Communications, Networking and Mobile Computing, 2005*, Volume 2, pp. 1253 - 1256, 23-26 Sept. 2005

[9] P. P. Chang and S. A. Mahlke and W. Y. Chen and N. J. Warter and W. W. Hwu, "IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors", *Proc. of the 18th International Symposium on Computer Architecture (ISCA)*, pp. 266-275, 1991.

[10] Scott A. Mahlke, "Exploiting Instruction-Level Parallelism in the Presence of Conditional Branches", *PhD thesis*, University of Illinois, Urbana IL, September 1996

[11] B. Mei, S. Vernalde, D. Verkest, H. De Man, R. Lauwereins, "Exploiting Loop-Level Parallelism on Coarse-Grained Reconfigurable Architectures Using Modulo Scheduling", *Proc of DATE 2003*, Munich, March, 2003

[12] Available at: http://iphome.hhi.de/suehring/tml/index.htm