Hardwired MPEG-4 Repetitive Padding

Georgi Kuzmanov, *Member, IEEE*, Stamatis Vassiliadis, *Fellow, IEEE*, and Jos T. J. van Eijndhoven

***Abstract*—We consider two hardwired solutions for repetitive padding, a performance restricting algorithm for real time MPEG-4 execution. The first solution regards application specific implementations, the second regards general purpose processing. For the application specific implementations we propose a systolic array structure. To determine the chip area and speed, we have synthesized its VHDL models for two field-programmable gate array families—Xilinx and Altera. Depending on the implemented configuration, the unit can process between 77 K and 950 K macroblocks per second (MB/s) when mapped on FPGA chips containing less than 10 K logical gates and frequency capabilities below 100 MHz. The second approach regards an augmentation of a general-purpose arithmetic logical units with an extra functionality added to perform repetitive padding. At trivial hardware costs of a few hundred 2×2 AND-OR logic gates, we achieve an order of magnitude speed-up compared to nonaugmented general purpose processor padding. The proposed hardware solutions meet the requirements of all MPEG-4 visual profile levels. Both approaches have been proven to be scalable and fit into different architectural concepts and operand widths.**

***Index Terms*—Arithmetic-logical-unit (ALU) augmentation, field-programmable gate array (FPGA), hardwired repetitive padding, MPEG-4, systolic structure.**

I. Introduction

ASSUMING audio-visual data compression standards, MPEG-4 [1] is the first to address content-based coding. To allow the efficient implementation of the specific standard requirements, several application profiles are defined. Within each profile, a number of levels constrain the computational complexity and the required data bandwidth of the application.

Complexity analysis [2] indicates that real-time software implementations of the intermediate CoreProfile@Level1 require more than 5 billion reduced instruction set computer (RISC)-like instructions per second. Consequently, we can safely conclude, that real time implementations of the highest profiles and levels of MPEG-4 would cost substantially more instructions per second (up to the order of 100 billion). These processing requirements will significantly exceed the capabilities of the general purpose processors, despite near future technology improvements. The work presented in this paper addresses one important feature in MPEG-4, the *repetitive padding* technique, defined at all levels in the core and main profiles of the standard. Software profiling results, reported in [2]–[4], indicate that padding is a computationally demanding and time consuming process, which restricts the real time operation of the MPEG-4 codecs. We present two general hardware approaches to implement the repetitive padding algorithm in real time. The first approach assumes MPEG-4 application-specific processing (ASIP) designs. It can be used as a hardware accelerator for an ASIP MPEG-4 processor or reconfigurable processing [5]. The second approach aims at hardware augmentations of general-purpose arithmetic-logical-units (ALUs) with application specific functional extensions. We show that both of the approaches are beneficial for improving the execution of the repetitive padding at little cost. More specifically, the following is shown regarding performance and cost:

- **Performance**—real time processing for all MPEG-4 profiles and levels. Assuming available technologies, data processing rates from 77 K up to 280 K macroblocks per second (MB/s) are achieved employing 4–16 simple processing elements (PEs) mapped on similar Xilinx and Altera field-programmable gate arrays (FPGAs). We show that higher processing speeds are achievable when more PEs (e.g., 32, 64) are implemented. It is established that the required operating frequency is low. The 16-pixel line processing FPGA implementations produce results at frequencies between 11 and 25 MHz. For a 64-bit augmented ALU example, running at 1 GHz, 7.8 million MB/s can be achieved.
- **Hardware costs**—we establish that scalable implementations, tunable to all Profiles@Levels requirements are feasible. To achieve the performance mentioned previously, a low number of FPGA cells (419 Xilinx configurable logic blocks (CLBs) and 1024 Altera LCs) is required for a 16-pixel processing unit. Only 344 AND-OR gates hardware penalty costs are required for a 64-bit padding-augmented ALU. We also show that the 32- and 128-bit implementations cost 172 and 688 extra AND-OR gates, respectively.

The remainder of the discussion in this paper is organized as follows. In Section II, we give some background knowledge and the motivation for our research. Section III describes in details the ASIP padding structure. The general purpose ALU padding augmentation is presented in Section IV. Section V gives quantitative evaluations of both approaches and presents analytical and simulation results in numbers. Finally, Section VI concludes the discussion.

II. Background and Motivation

For content-based coding, MPEG-4 uses the concept of a video object plane (VOP). A VOP is an arbitrarily shaped region of a frame, which usually corresponds to a semantic object

Manuscript received February 22, 2002; revised December 18, 2003. This work was supported by PROGRESS, the embedded systems research program of the Dutch organization for Scientific Research NWO, the Dutch Ministry of Economic Affairs, the Technology Foundation STW, and PHILIPS Research Labs, Eindhoven, The Netherlands. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Harrick M. Vin.

G. Kuzmanov and S. Vassiliadis are with the Computer Engineering Laboratory, EEMCS, TU Delft, 2600 GA Delft, The Netherlands (e-mail: G.Kuzmanov@EWI.TUDelft.NL; S.Vassiliadis@EWI.TUDelft.NL.)

J. T. J. van Eijndhoven is with the Department of Information and Software Technology, PHILIPS Research, Eindhoven, The Netherlands (e-mail:jos.van.eijndhoven@philips.com).

Digital Object Identifier 10.1109/TMM.2005.843365

in the visual scene. A sequence of VOPs in the time domain is referred to as a video object (VO). Each VOP is described by its *shape* and *texture*. *Shape* is mainly represented in binary format. This format represents the shape as a bitmap, referred to as *binary alpha plane*. Each pixel in this plane takes one of two possible values, which indicate whether the pixel belongs to the object or not. The binary alpha plane is divided into 16×16-pixel blocks called *binary alpha blocks* (BAB). *The texture* of a VOP represents its color by *macroblocks* (one 16×16 array of luminance and two 8×8 arrays of chrominance pixels). As its preceding visual data compression (MPEG) standards, MPEG-4 adopts *motion compensation* techniques, to exploit temporal redundancies in the encoded video sequences. The difference is that in MPEG-4 motion compensation is defined over VOPs instead of frames.

1) The Repetitive Padding Algorithm: The purpose of padding in MPEG-4 is to ensure more accurate block matching in motion compensation algorithms for arbitrary shaped visual objects. The padding process defines the full-color values for pixels outside the shape of a VOP. Macroblocks, which lie on the boundary of the VOP are referred to as boundary blocks and are processed with *repetitive padding*. Exterior macroblocks (completely outside the VOP) are padded using the *extended padding method*, which has low processing complexity, and will not be discussed further in this paper. Repetitive padding, described in [6], is equivalent to the following steps.

> ***Step 1.*** *Initialization*. Define any pixel outside the object boundary to be zero. Make a duplicate binary alpha map.
> ***Step 2.*** *Horizontal Repetitive Padding*. Scan each horizontal line of a block. Each scan line is composed of *zero* and *nonzero* line segments (according to the shape bits in the binary alpha map).
>
> - In zero segments, between an end point of the scan line and the end point of a nonzero segment, all zero pixels are replaced by the pixel value of the end pixel of nonzero segment.
> - In zero segments, between the end points of two different nonzero segments, all zero pixels take the average value of these two end points.
>
> Nonzero segments are not processed. All shape bits, corresponding to padded pixels are set in the duplicate binary alpha map.
> ***Step 3.*** *Vertical Repetitive Padding*. Scan each vertical line of the block and perform the identical procedure as described for the horizontal line. The updated shape information from the duplicate binary alpha map is used.

Fig. 1 illustrates the repetitive padding algorithm with a simplified example of a 4×4 pixel BAB and a 4×4 pixel luminance block. The original data structures are in the left part of the figure, where the definition of the zero and nonzero pixels is depicted according to Step 1. The luminance block contains color values (indicated by $\{A, B, C, D, E\}$) for the pixels, belonging to the shape of the VOP (nonzero pixels) and $\{x\}$ value (don't care) for the zero pixels. The central two squares of Fig. 1 illustrate the resulting data after the horizontal repetitive padding (Step 2). The duplicate BAB is indicated by S′ and the 4×4 luminance block is padded accordingly. In this part, the example

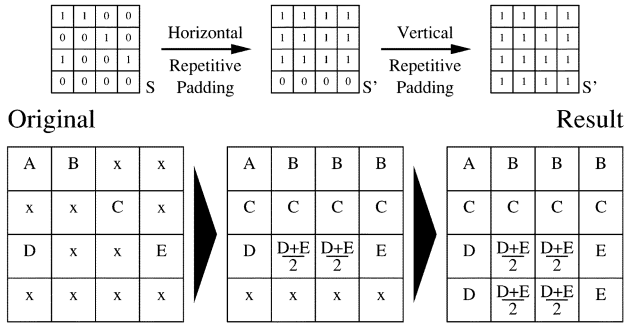

Fig. 1. Repetitive padding algorithm.

TABLE I
VISUAL PROFILES@LEVELS DEFINITIONS AND PROCESSING SPEED

Profile	Level	Session Size	Numb. VO	Max. MB/s	Bound. MB/s
Main	L4	1920x1088	32	489600	244800
	L3	CCIR 601	32	97200	48600
	L2	CIF	16	23760	11880
	L1	N.A.	N.A.	N.A.	N.A.
Core	L2	CIF	16	23760	11880
	L1	QCIF	4	5940	2970
Simple Scalable	L2	CIF	4	23760	N.A.
	L1	CIF	4	7425	N.A.
Simple	L3	CIF	4	11880	N.A.
	L2	CIF	4	5940	N.A.
	L1	QCIF	4	1485	N.A.

illustrates both cases, mentioned in Step 2, i.e., replicating a boundary pixel and estimating the average of two boundary pixels. Finally (Step 3), the vertical repetitive padding is performed, identically to the horizontal, and the resulting BAB and luminance blocks are shown in the right-most area of Fig. 1. The same procedure is executed for each of the two chrominance blocks from the padded macroblock, as well.

2) Motivation: In this paper we advocate hardwired solutions for repetitive padding. The rational behind such a proposal is as follows: Unlike its predecessors, MPEG-4 is much more demanding in terms of computational complexity with even more data intensive algorithms. This is illustrated in Table I, which represents the required data processing speed according to the MPEG-4 Visual Profiles@Levels definitions [7]. The *Core* Profile is the first to deal with arbitrary-shaped and temporally scalable objects, useful where a relatively simple content interactivity is required (e.g., Internet multimedia). The most demanding visual profile appears to be the *Main* Profile. It augments the functionality of the Core profile by coding of interlaced, semi-transparent, and sprite objects. It can be used for interactive and entertainment-quality broadcast and DVD applications [1]. At the highest level of the Main profile (L4 in Table I) a session with a frame size of 1920×1088 is processed, containing up to 32 VOs at a maximum of 489 600 macroblocks per second (MB/s). The last column of the table represents the required *boundary macroblocks per second*, which is an important criterion for evaluating the devices we are presenting further in this paper. Considering the above explanations, we can conclude that the performance demands of the Simple MPEG-4 Profile are approximately the same as of MPEG-2, since in this profile only rectangular video

TABLE II
COMPUTATIONAL DEMANDS OF CORE@L1 AND MAIN@L4

Profile	MPEG-4 Algorithm	# VO	Bound. MB/s	SW-MIPS Required
Core@L1	All MPEG-4	1	742	4 500
	Algorithms	4	2 970	18 000
	Repetitive	1	742	175
	Padding	4	2 970	700
Main@L4	Repetitive	1	7 650	1 794
	Padding	32	244 800	57 400

objects are defined. Therefore, the challenge is to meet the requirements of the most-demanding Core and Main Visual Profile Levels of MPEG-4, where arbitrary-shaped visual objects are processed.

A summary of the computational complexity of the QCIF, Core Profile Level 1 of MPEG-4, is reported in [2]. Since this is the lowest profile level, which utilizes the padding algorithm, we shall consider its real-time requirements as the minimum for a hardware implementation. At this level, the computational power, reported for the software encoding of a single object is in the order of 4500 million (RISC-like) instructions per second (MIPS). Assuming a software performance optimization by a factor of up to ten (accepted to be feasible in [2]), the total computational complexity is within the computational capabilities of the contemporary general purpose processors (500–1000 MIPS). In the case of four video objects (see Table I); however, the real-time software feasibility becomes problematic with its requirements of approximately four times higher computational workload. Given the above considerations, the need of a hardware acceleration of MPEG-4 is evident, even at this low profile level. Further analysis of the requirements for the software implementation indicates that the padding algorithm occupies some 175 MIPS for a single video object, or around 700 MIPS for the maximum four video objects, stated at Level 1 of the Core profile (Table I). Considering Table I, we can estimate that the required speed of 5940 MB/s for the Core Profile Level 1 is approximately 82 times lower than the speed requirements of the highest—Main@Level4 Profile (489 600 MB/s). A simple arithmetic estimation indicates that for the highest MPEG-4 profile level, the nonoptimized software padding would require approximately 57 000 MIPS and when extremely optimized (ten times speed-up)—on the order of 6000 MIPS. Even for the significantly less complex decoder part of MPEG-4, the padding algorithm will require some 24 000 MIPS for nonoptimized software implementation down to 2500 MIPS in dramatically optimized programming. All these approximated estimations of the MPEG-4 requirements are systematized in Table II.

III. APPLICATION-SPECIFIC PROCESSOR APPROACH

Since padding is performed over horizontal and vertical pixel lines identically, we propose a scalable systolic structure to process pixel blocks per line basis. Consequently, we define a PE and a topology to connect functional groups of PEs.

1) PE: A single PE, which is dedicated to process each pixel of a block, is depicted in Fig. 2. The same PE is used for luminance and chrominance padding. The following equations describe the functionality of the PE:

$$\mathrm{OS} = (S \vee \overline{\mathrm{LI}_N \vee \mathrm{RI}_N}) \tag{1}$$

$$O = \mathrm{OS} \wedge I \vee \overline{\mathrm{OS}} \wedge [(\mathrm{LI}_{\overline{N}} + \mathrm{RI}_{\overline{N}}) \gg i] \tag{2}$$

$$i = \mathrm{LI}_N \wedge \mathrm{RI}_N$$

$$\mathrm{LO} = S \wedge |\overline{S,I}| \vee \overline{S} \wedge \mathrm{RI}, \quad \mathrm{RO} = S \wedge |\overline{S,I}| \vee \overline{S} \wedge \mathrm{LI} \tag{3}$$

$$S' = S \vee \mathrm{LI}_N \vee \mathrm{RI}_N \tag{4}$$

where $\vee$ and $\wedge$ represent logical OR and AND operations, respectively, overline stands for logical negation, $A \gg i$ denotes "shift A with i positions right", and $+$ is an arithmetic summation of binary vectors. OS stands for output select signal; N represents the width of the processed data (we assume $\mathrm{N} = 8$); LI, RI are left and right input vectors with width $\mathrm{N}+1$; LO, RO are left and right output vectors with width $\mathrm{N}+1$; I, O are data input and output vectors with width N; S is the shape (input) bit before processing; S' is a mask output bit after processing. $\mathrm{LI}_{\overline{N}}$ denotes the first N (least-significant) bits of LI (bits 0 to $\mathrm{N}-1$); $\mathrm{LI}_N$is the Nth (the most-significant) bit of LI, used for shape, and $\overline{|S,I|}$ denotes the concatenation of bit S and vector I.

The operation of the PE is explained by the following.

- If the input shape bit S is set (the pixel belongs to the object and should not be padded), then:
  - The output O takes the value of the input I, i.e., the pixel keeps its color.
  - The value of the input (pixel) I is propagated to the left and to the right (via outputs $\mathrm{LO}_{\overline{N}}$ and $\mathrm{RO}_{\overline{N}}$) for further processing. The shape input bit S is propagated by the same multiplexers and occupies the most-significant bits of LO and RO.
  - The output bit S' is set, the pixel has been processed.
- If the input bit $S = 0$ (the pixel has to be padded), then:
  - the output O takes the average of $\mathrm{LI}_{\overline{N}}$ and $\mathrm{RI}_{\overline{N}}, \mathrm{RI}_{\overline{N}}, \mathrm{LI}_{\overline{N}}$ or I, depending on LI_N and RI_N. Note: If RI_N or LI_N is zero, the corresponding $\mathrm{RI}_{\overline{N}}$ or $\mathrm{LI}_{\overline{N}}$ should also be initialized to zero—see (2).
  - The LI value is propagated via RO and the RI—via LO including color and shape information.
  - The output bit S' is set, the pixel has been processed.

2) The Systolic Structure: To process a pixel line, padding elements are concatenated (Fig. 3) with the left-most and the right-most inputs initialized to zero (including $\mathrm{LI}_N = 0, \mathrm{RI}_N = 0$). We can easily evaluate the processing speed of the structure, given its operating frequency[1]. Let us assume a chain of n $\mathrm{PE}(\mathrm{n} = 4, 8, 16)$, operating at frequency F_n Hz. Further assume $\mathbf{N_n^{P8}}$ *and* $\mathbf{N_n^{P16}}$ *denoting the numbers of cycles, necessary to process an 8-pixel (chrominance) and a 16-pixel (luminance) line respectively*. Some potential values of these parameters are shown in Table III. The processing of 16 pixels by any $n \cdot \mathrm{PE}$ configuration will take $(N_n^{P16})/(F_n)$ s and for a 256-pixel luminance block-$(16 \cdot N_n^{P16}/F_n)$ [s]. Identically, the processing of two 8×8-pixel chrominance blocks will take $(16 \cdot N_n^{P8})/(F_n)$ [s] to the same unit configuration. Since a macroblock consists

[1]We distinguish (data) processing speed, measured in macroblocks per second (MB/s) from the device operating speed (frequency), measured in Hertz (Hz).

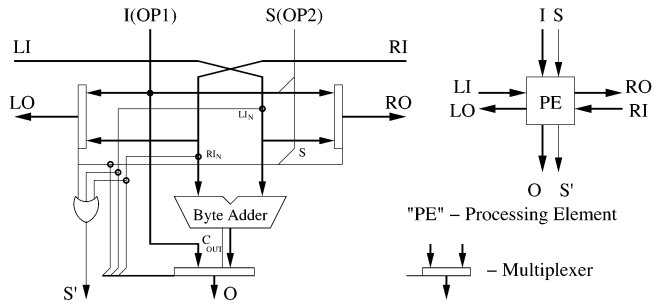

Fig. 2. A padding PE.

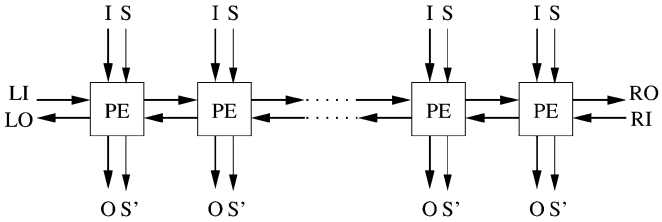

Fig. 3. Single scan line/column padding structure.

TABLE III
VALUES OF N_n^{P8} AND N_n^{P16}

n	Average		Worst Case	
	N_n^{P8}	N_n^{P16}	N_n^{P8}	N_n^{P16}
4	2.5	5.5	3	7
8	1	2.5	1	3
16	0.5	1	0.5	1
16·k	0.5	1	0.5	1

of 256 luminance and $128(2 \times 64)$ chrominance pixels, padded vertically and horizontally, a whole macroblock will be padded for $(32/F_n) \cdot (N_n^{P8} + N_n^{P16})$ [s]. If we implement a configuration, which processes several (say k) 16-pixel lines in parallel, we can formulate the *processing speed* as follows:

$$\text{processing\_speed} = \frac{F_n \cdot k}{32 \cdot (N_n^{P8} + N_n^{P16})}. \tag{5}$$

Formulation (5) is still valid for $n < 16$, assuming that $k = 1$.

In Table III, we separate the values for each of the parameters into two groups, namely: average values and worst-case values. The numbers represent the count of processing cycles at operating frequency F_n for different numbers of PEs (column "n"). The cycle count N_n^{P8} is disproportionately greater for chrominance line padding when $n < 8$ compared to the case when $n \geq 8$. Identical is the case with the cycle count N_n^{P16} when $n < 16$ compared to the case when $n \geq 16$. This is because if $n < 16$, we cut the data propagation chain within the line to be padded. In such cases extra processing cycles are required to complete the computations, because padding is highly data dependent regarding the data within a line/column [more details are discussed in the section to follow and Fig. 6(b)].

3) Possible Configurations: The proposed structure is scalable and can contain an arbitrary number of PEs. Moreover, it is possible to implement several structures, to process multiple lines in parallel. We report three cases of configurations with practical significance.

- *16 PE unit*—processes one luminance or two chrominance lines/columns per operating cycle.
- *8 PE or 4 PE unit*—processes a half/quarter of luminance or entire/half chrominance line/column. An additional control circuit is required, to maintain intermediate computational results.
- $32, 64, \ldots, 256$ *PE unit*—processing two or more luminance and four or more chrominance lines/columns per operating cycle. The extreme configuration would process the whole macroblock.

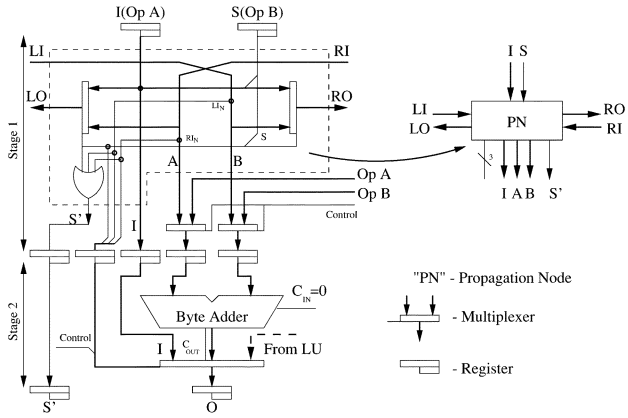

Fig. 4. ALU augmentation for a single-pixel padding.

IV. THE AUGMENTED ALU

Here, we consider a general-purpose ALU, augmented to support repetitive padding via subword data parallelism. We *accommodate padding without creating critical* ALU *paths and preserve the ALU functionality*. Since 8-bit integer data are frequently used, in this paper we assume the same data formats (our scheme with proper considerations will accommodate "N-bit" quantities as stated in MPEG-4, N being 10, 12, etc.).

1) Pixel Processing: A single byte processing structure, which is dedicated to process each pixel of a block, is depicted in Fig. 4. Its organization is similar to the one illustrated in Fig. 2, but it is extended with additional pipelining to fit into the general-purpose ALU cycle. We pipeline the processing flow by dividing it into two stages. The first stage contains a propagation node (PN) and two multiplexers. The multiplexers are required to preserve the original functionality of the ALU. A byte-controlled adder and an output multiplexer build the second pipeline stage. The byte controlled adder is a part of the original multi-byte ALU adder, with controllable carries between the bytes. The padding output multiplexer can be merged with the existing ALU output multiplexer, depicted in Fig. 4 by the dash-lined arrow, leading from the logical unit (LU). The function of the PN is to propagate the appropriate values to its adjacent PNs and to supply data and control signals to the byte controlled adder and the output multiplexer. Its functionality can be described by (3) and (4).

2) Line/Column Padding: To process a line or a column from a block by an n-byte ALU, we have to implement a chain

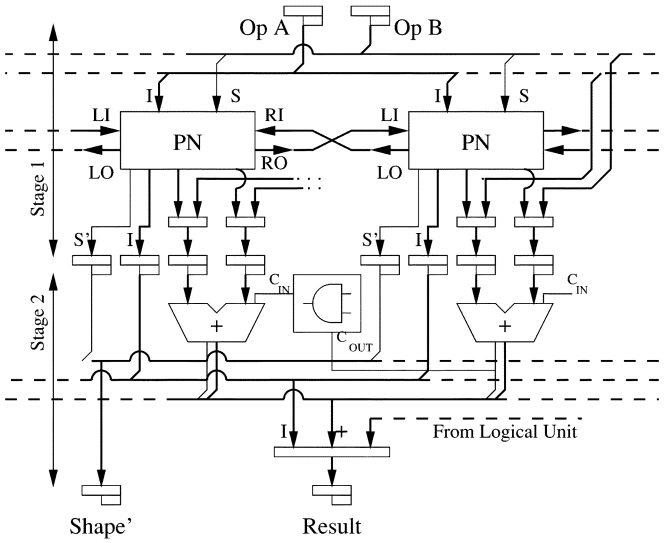

Fig. 5. Scan line/column padding augmentation of an ALU.

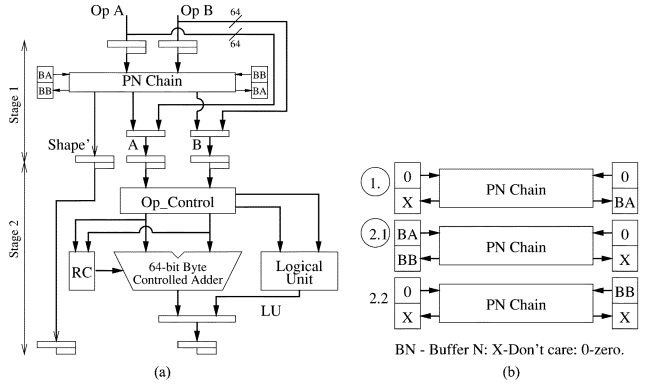

Fig. 6. Generalized 64-bit padding enabled ALU. (a) Pipeline stages—general view. (b) Luminance processing—data initialization and buffering by cycles.

of n PN (i.e., n-pixel parallel padding) similar to the structure in Fig. 3. A section of such processing circuitry for two adjacent pixels is depicted in Fig. 5. The added ALU logic has to perform the functions described in (1)–(4). Each node from the propagation chain propagates the pixel values from left to right and from right to left when its corresponding shape bit S is zero. When a shape bit is 1, the corresponding PN transmits the value of the pixel (driven via input I) to its adjacent nodes. The proper multiplexer input is selected to drive the appropriate value out of the ALU. This structure is scalable and can contain an arbitrary number of PNs, depending on the ALU width (i.e., n elements for an n-byte ALU).

3) Putting Everything Together: We will describe the padding process flow, performed by a 64-bit ALU, a general view of which is depicted in Fig. 6(a). The operand control circuit (Op_Control) is a part of the operands critical path. It is responsible for setting the adder operands and performs operations like sign extension, operand masking etc. The result control (RC) circuit deals with flags handling like overflows, carries, equal zero, etc.

Data buffering and initialization are identical for the application specific implementation, described earlier. Fig. 6(b) depicts a general view of the 64-bit initialization and cycle partitioning for luminance line/column padding. Since a luminance line (128-bit) can not be processed in one pass by a 64-bit ALU, we assume that the left-most half (the left-most subline) of the line is processed first. Depending on the right-most shape bit of the subline, padded in the first cycle, the full-line padding would require two or three cycles. If the right-most shape bit of the first half of the luminance line is "0", three cycles in total will be required to pad the whole luminance line, otherwise (when shape bit is "1"), two cycles would be sufficient. Since the discussed right-most shape bit is available before the next operands (describing the other half of the line) are issued, we have branch determination (a perfect branch prediction) [8] for the pipeline. We can avoid this branching, assuming a worst-case scenario, i.e., three cycles to process any luminance line/column. An implementation according such an assumption would lead to simpler control while still yielding very high processing speeds. The BA and BB buffers in Fig. 6(b) contain initialization and intermediate 8-bit data along with one bit for shape handling. Equation (5) is also valid for a padding augmented ALU. We can consider n as the number of bytes the ALU processes in a cycle and N_n^{P8}, N_n^{P16}—the average number of cycles spent to process one chrominance and one luminance line *in a long data sequence*. We can conclude that *in a long data sequence, a complete luminance line processing by a 64-bit padding augmented and pipelined ALU would take on average 2.5 cycles to perform* $(N_n^{P16} = 2.5)$. *The padding of a chrominance line by the same ALU would take approximately one cycle* $(N_n^{P8} = 1)$.

V. Quantitative Evaluation

Here, we evaluate the proposed implementations using the measurements *processing speed*(MB/s) and *chip area* (logical blocks). We have taken into account: *parallelism, pipelining, and system inherent delay*. We also compare our proposal to other existing schemes.

A. Systolic Array Implementation

We have experimented with different numbers of PEs. Without loss of generality and for embedding our results into the MOLEN [5] experimental platform, we assume reconfigurable technology. We have written synthesizeable VHDL models of a single PE and a generic multi-element structure of PEs. To get realistic values for the parameters of the unit, we have synthesized the VHDL models for two different FPGA technologies, namely the Xilinx xc4085xlpg559-09 and the Altera epf10k20rc240-4 chips, which can be run at comparable frequencies (around 100 MHz). For both chip families we have evaluated structures of 4, 8, and 16 PEs and speed has been reported in megahertz. Two extra evaluations for 32 and 64 PEs Xilinx mappings illustrate how the data organization and the number of PEs influence the performance of the unit.

1) Area and Speed Evaluation: Table IV reports the area estimates for the Xilinx chip in the absolute units the vendor defines—CLBs and in percentage of the available gate array area. For the Altera chip, results are reported in Table V in similar manner but the units for the absolute area are defined by Altera as logical cells (LCs). The speed estimations for both FPGA families suggest similar results. Besides the operating frequency, measured in megahertz, we also evaluated the actual

TABLE IV
RESULTS FOR THE XILINX xc4085xlpg559-09 CHIP

# PE	# CLBs		Speed		
	total	%	MHz	MB/s	
				Average	Worst C.
4	45 of 3136	4	24.5	95 700	76 600
8	206 of 3136	7	18.2	162 500	142 200
16	419 of 3136	14	11.4	237 500	237 500
32	838 of 3136	27	**11.4**	**475 000**	**475 000**
64	1676 of 3136	53	**11.4**	**950 000**	**950 000**

TABLE V
RESULTS FOR THE ALTERA epf10k20rc240-4 CHIP

# PE	# LCs		Speed		
	total	%	MHz	MB/s	
				Average	Worst C.
4	254 of 1152	22	24.8	96 900	77 500
8	511 of 1152	44	19.8	176 800	154 700
16	1024 of 1152	88	13.4	279 200	279 200

data processing speed of the different configurations. Since VOPs may vary in size and resolution, the MPEG-4 requirements group has defined the binding criteria for implementation complexity in terms of *transferred macroblocks per second* (Table I). For consistency with this definition, in the last two columns of Tables IV and V, we have estimated the processing speed in MB/s according to (5)—more specifically, the average and worst-case values. The reported numbers indicate that the padding structure can meet the real-time requirements for a broad range of visual resolutions. Note, that for structures with PE number, which is a multiple of 16, the average and worst-case speeds are equal. This is due to the data dependency within a line and the 16-pixel wide data structure to be processed. In such large structures, the required number of cycles to pad a 16-pixel line is fixed ($N_n^{P16} = \text{const}$).

The *Core* and *Main* profile levels of MPEG-4 require processing speeds in the range 2970–244 800 Boundary MB/s to maintain from 4 up to 32 VOPs. It is obvious that the operating speeds, achieved by the proposed padding unit, completely match the required values.

B. ALU Augmentation

As indicated in the previous section, our assumption is: *accommodate padding without creating critical ALU paths and preserve the ALU functionality*. Furthermore, it is of interest to establish if the expenses in terms of hardware are significant.

1) Critical Paths/Speed Estimation: Before addressing the critical path of the stages, we discuss generally what is considered to be an ALU critical path. Assuming that an ALU operation is performed in a single cycle, the ALU critical path in a general purpose design can be approximated to be twice the delay of an adder plus a small constant of two to four logic stages. Given that a 64-bit adder using 2×2 AND-OR (or equivalent) gates requires seven logic stages [9], the ALU can be approximated by at least 14 2×2 AND-OR logic stages. Regarding the critical path penalty issue, it has been noted that byte/nibble controlled adders (used in the past to perform, for example, decimal operations) will not increase the cycle time. The reason for such a possibility is that the masking is embedded implicitly in the stages. For a precise description and discussion for a controlled adder, more complex than the one proposed here, the interested reader is referred to [10].

The computation of padding requires two pipeline stages [Fig. 6(a)]: one computing the PN operation and one performing the masked ALU operation. We note that a pipeline stage is a machine cycle, while a logic stage is the delay of a gate.

a) Pipeline Stage 1: The following operations are performed in this stage.

- Operands are routed through the propagation chain.
- Data, to be processed in Stage 2, is loaded in the pipeline latches.
- Control signals for the output multiplexer are generated.

The first-stage critical path is clearly linear to the length of input data and it is a serial operation. This critical path is equal to the number of bytes in the ALU operands plus one multiplexers. Given that the worst case is the largest input ALU, implemented in practice (64 bit), the critical path is equal to the delay of $1 + \frac{64}{8} = 9$ multiplexers which fits into a single ALU cycle. For usual 32-bit units, the delay is equal to five multiplexer delays. It should be noted, that operands are passed through the PNs only when a padding operation is performed. They are bypassing the first pipeline stage for a conventional ALU operation, adding one extra input in the already existing bypassing multiplexer. Consequently, the first cycle of padding computations will not imply a critical path problem. The bottomline is that for evaluating the performance of the scheme, proposed here, it is safe to assume no augmentation to the processor cycle times.

b) Pipeline Stage 2: In this pipeline stage, the following operations that could comprimise the critical path are performed.

- The byte-controlled adder performs masked additions over the data stored in the pipeline latches.
- The output multiplexer issues the appropriate results according to the generated control signals (see Fig. 4).

The ALU critical path penalty could have been augmented by a single two-way AND element (added possibly to the critical path of the byte controlled adder). Such a penalty has been shown to be avoidable [10] with implicit computations. Thus it should not extend the critical paths of a general purpose ALU implementation. The critical path penalty for reading from the general purpose register or bypassing the operands of the ALU is a 2–1 multiplexer and it should be noted that such a multiplexer already exists. It is used, for example, to perform bypassing of operands from other units, direct data passing from caches etc. The only foreseeable penalty is adding a single input to the already existing multiplexer, which is not anticipated to create critical path problems.

In estimating the expected performance we note that *an ALU instruction takes 1 cycle, padding takes two cycles latencies*. To estimate the possible speed achievable from the proposed solution we consider the following: Let us assume an $n \cdot 8$-bit padding augmented ALU operating at frequency F_n [Hz]. Let us assume values of n that have practical significance—4, 8, 16, (i.e., 32, 64, 128-bit ALU). To evaluate the speed of the ALU we can use (5), which gives results for long data sequences. Assuming a value of $F_n = 1$ GHz (which is currently easily achievable for general purpose processors) and using the data

TABLE VI
HARDWARE ESTIMATION AND PROCESSING SPEED AT $F_n = 1$ GHz

ALU bits	n	Number of extra gates	Speed [MB/s]	
			Average	Worst Case
32-bit	4	172	3 906 250	3 125 000
64-bit	8	344	8 928 600	7 812 500
128-bit	16	688	20 833 300	20 833 300

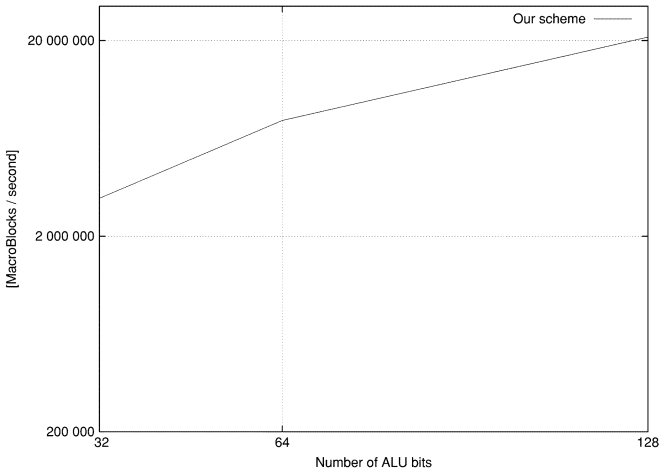

Fig. 7. Processing speed for different ALU operand sizes and $F_n = 1$ GHz (in logarithmic scale).

from Table III into (5), we calculated the implementation speeds, given in Table VI.

The most demanding profile level, *level 4* of the *Main* MPEG-4 profile, requires 244 800 Boundary MB/s (maximum 489 600 MB/s) for a high resolution session type (1920×1088) and 32 objects (Table I). This rates are an order of magnitude lower than what the augmented ALU implementations can achieve (see Fig. 7). The potentials of the structure indicate capabilities to meet even more-demanding future profiles of the visual data compression standards [11].

2) Hardware Estimations: We choose the 2×2 AND-OR logic block as a basis for the hardware estimations. The reason for such a choice lies on the fact that such a block is commonly available to most technology libraries [9]. A 1-bit 2 to 1 multiplexor is a 2×2 AND-OR gate. The hardware penalty for a single-byte padding structure is: 2×9-bit multiplexers, 2×8-bit multiplexers and 1 OR gate. That accounts for 35 2×2 AND-OR gates. An n-byte implementation will cost $n \cdot 35$ AND-OR gates plus additional cost for the ALU multiplexer of $n \cdot 8$ gates, i.e., $n \cdot 43$ 2×2 AND-OR gates. Table VI contains the exact values of the hardware penalties for different ALU sizes. It is noted that our estimations, as indicated in Table VI, strongly suggest that the hardware cost is negligible.

C. Related Work

The repetitive padding algorithm is described in [6] and [12], but some modifications have also been reported. In [13] and [14], new algorithms are proposed to modify the original repetitive padding. All of them suggest software improvements but do not focus on the hardware execution, nor on performance. The VLSI hardware padding accelerator, reported in [15], has a complex organization and control—it is dedicated for 64-bit data, contains three subunits, operating in four internal states with low flexibility and scalability. The achieved processing speed is 245 000 MB/s at clock frequency 100 MHz. Compared to it, our proposals have the following advantages.

- *Faster processing*—if we assume the same operating speed (100 MHz) for both ASIC and ALU schemes, our 64-bit implementations are three times faster (781 250 versus 245 000 MB/s).
- *Flexibility and scalability*—our approaches allow high levels of scalability and flexibility.
- *Simple hardware*—with their trivial control scheme, our implementations are simpler than the design in [15]. Our hardware overhead is just a few multiplexers versus the three subunits and complex interconnect.

A hardware acceleration of the padding, which appears to be faster, is discussed in [16]. In such a proposal, the padding algorithm is modified to support specific instruction set extensions as the horizontal and vertical padding processes are divided into two phases each. These two phases consequently scan the lines/columns into two opposite directions and perform the padding operations. In the proposed solution, there is a hardwired dedicated padding unit with high control overhead, supporting eight new instructions. Its estimated processing speed at 100-MHz clock frequency is 250 000 MB/s for 32-bit data width. Our proposal differentiates with the schemes described there in the following.

- *Higher processing speeds.* In [16], the processing speed is reported only for a 32-bit unit. Although the design is claimed to be scalable and implementable for larger data types, for 64- and 128-bit units data is not reported. For 32-bit data our units are at worst over 20% faster (312 500 versus 250 000 MB/s). According to the scheme proposed in [16], the processing speed of the unit increases (at most) linearly with the operand width. Our approaches allow an exponential speed up when the data width increases due to the better data processing scheme. For 64-bit data and the same clock frequency our units can be estimated to be over 50% faster, while for 128-bit data, the estimated speed up is over a factor of two.
- *Simpler control.* To perform the padding algorithm, our units require only one additional instruction, while in [16] eight new instructions are introduced. As a rule in computer engineering, a higher number of additional instructions imposes more severe architectural modifications and more complicated data paths and control circuitry in the implementation. It is always preferable to limit the opcodes added into an architecture. Our proposal is clearly better with only one (the minimum) instruction in addition.

Both [15] and [16] present hardware estimations for 0.35 μm CMOS technology and do not report any technology independent hardware estimations (e.g., number of logical gates). Consequently, we can not make an exact and independent comparison between the hardware size complexities of these units. Finally, in the present paper we use the standard repetitive padding

algorithm (differentiates from [13] and [14]) as we scan each line and column of a macroblock bidirectionally in parallel (differentiates from [16] where the padding algorithm is modified), thus saving a number of processing cycles. Our two approaches for the hardware acceleration of the algorithm are scalable (differentiates from [15]) and differentiate from all of the above mentioned references with the reconfigurable implementation and the general-purpose ALU modification.

VI. Conclusion

Two hardware approaches to realize the MPEG-4 repetitive padding algorithm in real-time were discussed in this paper. First, a design of a simple dedicated systolic structure was proposed. Its reconfigurable hardware costs and performance had been evaluated for two FPGA technologies—Altera and Xilinx. The simulation results indicate that the proposed padding unit can easily meet the real-time requirements of the Core and Main MPEG-4 profiles at trivial hardware costs. The second approach, proposed in this paper, described a scheme for general purpose ALU augmentation, which accelerates the MPEG-4 padding algorithm by orders of magnitude. We proposed a pipelined implementation preserving the original functionality of the target ALU. At a trivial hardware cost of only a few hundred elementary 2×2 AND-OR logic gates, we could easily achieve a real-time performance at the most-demanding MPEG-4 profile levels.

References

[1] *MPEG-4 Overview*, ISO/IEC JTC11/SC29/WG11 N4030, Mar. 2001.

[2] J. Kneip, S. Bauer, J. Vollmer, B. Schmale, P. Kuhn, and M. Reissmann, "The MPEG-4 video coding standard–A VLSI point of view," in *Proc. IEEE Workshop on Signal Processing Systems (SIPS98)*, Oct. 1998, pp. 43–52.

[3] H.-C. Chang, L.-G. Chen, M.-Y. Hsu, and Y.-C. Chang, "Performance analysis and architecture evaluation of MPEG-4 video codec system," in *Proc. IEEE Int. Symp. Circuits and Systems*, vol. II, Geneva, Switzerland, May 2000, pp. 449–452.

[4] S. Vassiliadis, G. Kuzmanov, and S. Wong, "MPEG-4 and the new multimedia architectural challenges," in *Proc. 15th Int. Conf. SAER'2001*, St. Konstantin, Bulgaria, Sep. 2001.

[5] S. Vassiliadis, S. Wong, and S. Cotofana, "The MOLEN $\rho\mu$-coded processor," in *Proc. 11th Int. Conf. Field Programmable Logic and Applications*, Belfast, N. Ireland, U.K., Aug. 2001.

[6] *MPEG-4 Video Verification Model Version 16.0*, ISO/IEC JTC11/SC29/WG11, N3312.

[7] *ISO/IEC 14496-2. Generic Coding of Audio-Visual Objects-Part 2: Visual. Final Proposed Draft*, ISO/IEC JTC11/SC29/WG11 N2802, Jul. 1999.

[8] M. Putrino and S. Vassiliadis, "Resolution of branching with prediction," *Int. J. Electron.*, vol. 66, no. 2, pp. 163–172, Feb. 1989.

[9] C. Chang, S. Vassiliadis, and J. Delgado-Frias, "An investigation of binary CLA and ripple CMOS adder designs," *Microprocess. Microprogramm. J.*, vol. 40, no. 1, pp. 1–21, Jan. 1994.

[10] M. Putrino, S. Vassiliadis, and E. Schwarz, "Parallel binary byte adder/subtracter," *Int. J. Electron.*, vol. 65, no. 2, pp. 139–153, Feb. 1988.

[11] *New MPEG-4 Profiles Under Consideration*, ISO/IEC JTC11/SC29/WG11, Jul. 2001.

[12] Y. Q. Shi and H. Sun, *Image and Video Compression for Multimedia Engineering*. Boca Raton, FL: CRC, 2000.

[13] E. A. Edirisinghe, J. Jiang, and C. Grecos, "Shape adaptive padding for MPEG-4," *IEEE Trans. Consumer Electron.*, vol. 46, no. 3, pp. 514–520, Aug. 2000.

[14] J.-H. Moon, J.-H. Kweon, and H.-K. Kim, "Boundary block-merging (BBM) technique for efficient texture coding of arbitrarily shaped object," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 9, no. 1, pp. 35–43, Feb. 1999.

[15] C. Heer and K. Migge, "VLSI hardware accelerator for the MPEG-4 padding algorithm," in *Proc. IS&T:SPIE Conf. Media Processors*, vol. 3655, 1999, pp. 113–119.

[16] M. Berekovic, H.-J. Stolberg, M. B. Kulaczewski, P. Pirsh, H. Moler, H. Runge, J. Kneip, and B. Stabernack, "Instruction set extensions for mpeg-4 video," *J. VLSI Signal Process.*, vol. 23, no. 1, pp. 27–49, Oct. 1999.

Georgi Kuzmanov (S'04–M'05) was born in Sofia, Bulgaria, in 1974. He received the M.Sc. degree in computer systems from the Technical University of Sofia in 1998 and the Ph.D. degree in computer engineering from Delft University of Technology (TU Delft), Delft, The Netherlands, in 2004.

Between 1998 and 2000, he was with "Info MicroSystems" Ltd., Sofia, where he was involved in several reconfigurable computing and ASIC projects as a Research and Development Engineer. He is currently with the Computer Engineering Group at TU Delft. His research interests include reconfigurable computing, media processing, computer arithmetic, computer architecture and organization, vector processors, and embedded systems.

Stamatis Vassiliadis (M'86–SM'92–F'97) was born in Manolates, Samos, Greece, in 1951.

He is currently a Chair Professor in the Electrical Engineering Department, Delft University of Technology, Delft, The Netherlands. He previously served in the Electrical Engineering faculties of Cornell University, Ithaca, NY, and the State University of New York, Binghamton. For a decade, he worked with IBM, where he was involved in a number of advanced research and development projects. He received numerous awards for his work, including 24 publication awards, 15 invention awards, and an outstanding innovation award for engeneering/scientific hardware design. His 70 U.S. patents rank him as the top all-time IBM inventor.

Dr. Vassiliadis received an honorable mention Best Paper award at the ACM/IEEE MICRO25 in 1992 and Best Paper awards at the IEEE CAS (1998), IEEE ICCD (2001), and PDCS (2002).

Jos T. J. van Eijndhoven was born in Roosendaal, The Netherlands, in 1957. He received the Master's degree in 1981 and the Ph.D. degree in 1984 for work on piecewise linear circuit simulation, both in electrical engineering, from Eindhoven University of Technology, The Netherlands.

He became a Senior Research Member in the Design Automation Group of Eindhoven University of Technology. In 1986, he spent a sabbatical period at the IBM T. J. Watson Research Laboratory, Yorktown Heights, NY, where he conducted research on high-level synthesis. In 1998, he moved to Philips Research Laboratories, Eindhoven, to work on the architectural design of programmable multimedia hardware and the associated mapping of media processing applications.