Multimedia Rectangularly Addressable Memory

Georgi Kuzmanov, *Member, IEEE*, Georgi Gaydadjiev, *Member, IEEE*, and Stamatis Vassiliadis, *Fellow, IEEE*

***Abstract*—We propose a scalable data alignment scheme incorporating module assignment functions and a generic addressing function for parallel access of randomly aligned rectangular blocks of data. The addressing function implicitly embeds the module assignment functions and it is separable, which potentially enables short critical paths and saves hardware resources. We also discuss the interface between the proposed memory organization and a linearly addressable memory. An implementation, suitable for MPEG-4 is presented and mapped onto an FPGA technology as a case study. Synthesis results indicate reasonably small hardware costs in the order of up to a few thousand FPGA slices for an exemplary 512 × 1024 two-dimensional (2-D) addressable space and a range of access pattern dimensions. Experiments suggest that speedups close to 8× can be expected when compared to linear addressing schemes.**

***Index Terms*—Linear addressing, memory modules, module assignment functions, rectangular block addressing, separability.**

I. Introduction

THE problems of conflict-free parallel accesses of different data patterns have been extensively explored in several research areas. Vector processors designers have been interested in memory systems that are capable of delivering data at the demanding bandwidths of the increasing number of pipelines, see, e.g., [1]–[4]. Different approaches have been proposed for optimal alignment of data in multiple memory modules [1], [3]–[7]. Module assignment and addressing functions have been utilized in various interleaved memory organizations to improve the performance. In graphical display systems, researchers have been investigating efficient accesses of different data patterns: blocks (rectangles), horizontal and vertical lines, forward and backward diagonals [7], [8]. In this paper, we consider visual data representations. For such an application, the most computationally intensive algorithms, like motion estimation and the discrete cosine transform, operate on square pixel blocks, requiring a significant data throughput. Therefore, the emerging visual data compression standards have narrowed the problems toward high-performance implementations of rectangularly accessible data storages.

In this paper, we propose an addressing function for rectangularly addressable memory systems, with the following characteristics: Rectangular subarrays can be accessed in a two-dimensional (2-D) data storage with *high scalability*. The *addressing is separable*, which potentially saves hardware. We also introduce *implicit module assignment functions* and a *conflict free data routing* circuitry, which along with the *high flexibility* of the design parameters, allow *minimal number of memory modules* and *shortest critical paths*. Compared to related work, our proposal is the only one that combines the above characteristics altogether and utilizes the lowest number of memory modules. Therefore, our design is superior to related art in speed, scalability, flexibility, and low complexity.

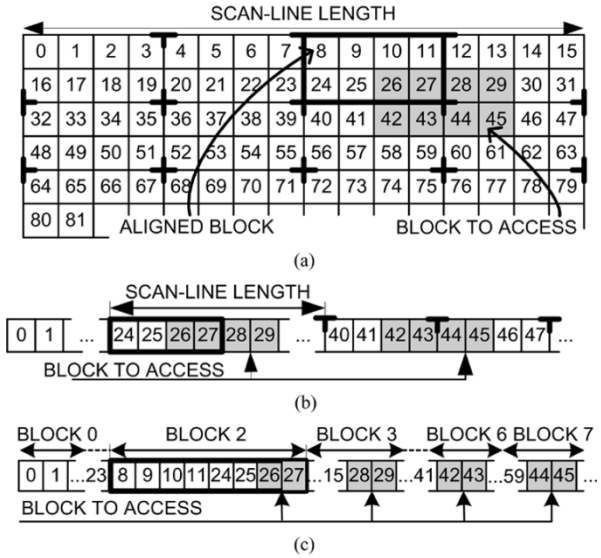


Fig. 1. Addressing problem in LAM. (a) Pixels in a video frame. (b) Scan-line alignment. (c) Block-based alignment.

The remainder of the paper is organized as follows. Section II introduces the particular addressing problem. In Section III, the addressing scheme is described and the corresponding memory organization with a possible implementation are discussed. Case study synthesis results for FPGA technology are reported and related work is compared to our design in Section IV. Finally, the paper is concluded with Section V.

II. Motivation

In this section, without loss of generality (our scheme applies equally to vector rectangular processing), we consider the memory addressing and accessing problem by considering the MPEG standards.

The Addressing Problem—A Motivating Example: Most of the MPEG data processing is performed over regions (blocks of pixels) from a frame. This generates memory problems with data alignment and access illustrated by the following *motivating example*. Assume a single port linearly addressable memory (LAM) and a plane divided into blocks with dimensions 4×2 byte pixels. Further, assume that the video information is stored as a scan-line [see Fig. 1(a)] and that the system is capable of accessing eight consecutive bytes per

Manuscript received March 5, 2004; revised April, 2005. This work was supported by the Dutch embedded systems research program PROGRESS (project AES.5021). The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Mihaela van der Schaar.

The authors are with the Computer Engineering Lab, EEMCS, Delft University of Technology, 2628 CD Delft, The Netherlands (e-mail: G.Kuzmanov@ewi.tudelft.nl; G.N.Gaydadjiev@ewi.tudelft.nl; S.Vassiliadis@sewi.tudelft.nl).

Digital Object Identifier 10.1109/TMM.2005.864345

1520-9210/$20.00 © 2006 IEEE

TABLE I
NUMBER OF LAM CYCLES IN DIFFERENT CASES

all aligned	mixed	none aligned
$\frac{n^2}{W} \cdot N$	$(\frac{n^2}{W} + n - 1) \cdot N$	$(\frac{n^2}{W} + n) \cdot N$

cycle. Because of non aligned blocks [see Fig. 1(b)], neither of the blocks containing pixels $\{8, 9, 10, 11, 24, 25, 26, 27\}$ and $\{26, 27, 28, 29, 42, 43, 44, 45\}$ is accessible by a single memory transfer. Even though the memory system could be accessing all data, because it can access linearly 8 bytes in a single memory cycle, in fact it can access, e.g., either bytes $\{26, 27, 28, 29\}$ or bytes $\{42, 43, 44, 45\}$, but not all 8 bytes $\{26, 27, 28, 29, 42, 43, 44, 45\}$.

Another approach to process block-organized data may be to reorder data into the LAM. If we position blocks into consecutive bytes [Fig. 1(c)], we will be able to access such blocks in a single memory cycle (e.g., pixels $\{8, 9, 10, 11, 24, 25, 26, 27\}$). In MPEG, however, some of the most demanding algorithms (e.g., motion estimation) require accessing block data at an arbitrary position in the frame, thus in memory. In the Fig. 1(c) example, accessing block $\{26, 27, 28, 29, 42, 43, 44, 45\}$ requires four cycles, even though the bandwidth is 8 bytes. This is because only two of its bytes can be accessed in one memory access cycle (i.e., either $\{26, 27\}$, or $\{28, 29\}$, or $\{42, 43\}$, or $\{44, 45\}$). Fig. 1(c) suggests that in such cases data fetching may become even less effective than the scan-line alignment scheme. In the rest of the presentation, for conciseness, we will refer to blocks like $\{8, 9, 10, 11, 24, 25, 26, 27\}$ in Fig. 1(a) as aligned, and to the remaining blocks (like $\{26, 27, 28, 29, 42, 43, 44, 45\}$) as nonaligned. The borders between aligned blocks in the figures are marked with thick line crosses.

General Problem Introduction and Proposed Solution: Consider a LAM with word length of W bytes (typically $W = 1, 2, 4, 8, 16$) and the time for linear memory access to be T_{LAM}. The time to access a single $a \times b$ subarray of one-byte pixels, depending on its alignment in the LAM (refer to the preceding motivating example) will be

1) aligned subarray: $(a \cdot b/W) \cdot T_{\mathrm{LAM}}$;
2) not aligned subarray: $((a/W) + 1) \cdot b \cdot T_{\mathrm{LAM}}$.

The time, required to access $Na \times b$ blocks with respect to their alignment will be

1) all N blocks aligned: $N \cdot (a \cdot b/W) \cdot T_{\mathrm{LAM}}$;
2) None of the blocks aligned: $N \cdot ((a/W) + 1) \cdot b \cdot T_{\mathrm{LAM}}$;
3) Mixed: $N \cdot [(1/a) \cdot (a/W) + (a - 1/a)((a/W) + 1)] \cdot b \cdot T_{\mathrm{LAM}} =$

$$= N \cdot \left(\frac{a}{W} + 1 - \frac{1}{a}\right) \cdot b \cdot T_{\mathrm{LAM}} \quad (1)$$

By *mixed* access scenario we mean accessing both aligned and nonaligned blocks. In (1), we assume that the probability to access an aligned block is $(1/a)$, while for a nonaligned block it is $(a - 1/a)$. For simplicity, but without losing generality, assume square blocks of $n \times n$, (i.e., $a = b = n$). Further assuming N blocks to access, we can estimate the number of LAM cycles as indicated in Table I. Obviously, the number of cycles to access an $n \times n$ block in a LAM is a square function of n, i.e., $O(n^2)$.

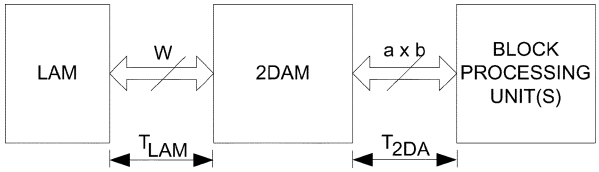


Fig. 2. Memory hierarchy with 2DAM.

TABLE II
ACCESS TIME PER $n \times n$ BLOCK IN LAM CYCLES, $t = (T_{2DA}/T_{\mathrm{LAM}})$

n	W	LAM			2DAM	
		WC	Mix.	BC	Mix./BC	WC
8	1 (8 bits)	72	71	64	8+t	64+t
	2 (16 bits)	40	39	32	4+t	32+t
	4 (32 bits)	24	23	16	2+t	16+t
16	1 (8 bits)	272	271	256	32+t	256+t
	2 (16 bits)	144	143	128	16+t	128+t
	4 (32 bits)	80	79	64	8+t	64+t

An appropriate memory organization may speed-up the data accesses. Consider the memory hierarchy in Fig. 2 with time to access an entire $n \times n$ block from the 2-D accessible memory (2DAM) to be T_{2DA}. In such a case, the time to access $Nn \times n$ subblocks in the mixed access scenario will be

$$\frac{N}{n} \cdot \frac{n^2}{W} \cdot T_{\mathrm{LAM}} + N \cdot T_{2DA}, \quad [\mathrm{sec}]$$
$$\Leftrightarrow \left(\frac{n}{W} + \frac{T_{2DA}}{T_{\mathrm{LAM}}}\right) \cdot N, [\mathrm{LAM\ cycles}].$$

That is the sum of the time to access the appropriate number of aligned blocks (i.e., (N/n)) from LAM plus the time to access all N blocks from the 2DAM. It is evident that in a mixed access scenario, the number of cycles to access an $n \times n$ block in the hierarchy from Fig. 2 is a linear function of N, i.e., $O(n)$ and depends on the implementation of the 2-D memory array. Table II presents access times per single $n \times n$ block. Time is reported in LAM cycles for some typical values of N and W. Three cases are assumed for LAM: 1) none of the N blocks is aligned—worst case (WC); 2) mixed block alignment (Mix.); and 3) all blocks are aligned—best case (BC). The last two columns contain cycle estimations for the organization from Fig. 2 where both mixed and best case scenarios assume that aligned blocks are loaded from the LAM to the 2DAM first and then nonaligned blocks are accessed from the 2DAM. The 2DAM worst case (contrary to LAM) assumes that all blocks to be accessed are aligned. Even in this worst case, the 2DAM-enabled hierarchy may perform better than LAM best case if the same aligned block should be accessed more than once (i.e., data are reusable). For example, assume accessing k times the same aligned block. In LAM (best case), this would take $k \cdot (n^2/W) = [(n^2/W) + (k - 1) \cdot (n^2/W)]$, while in 2DAM (worst case), it would cost $[(n^2/W) + (k - 1) \cdot (T_{2DA}T_{\mathrm{LAM}})]$ LAM cycles per block. Obviously, to have a 2DAM enabled memory hierarchy, faster than pure LAM, it would be enough if $(n^2/W) > (T_{2DA}/T_{\mathrm{LAM}})$. All estimations above strongly suggest that *a 2DAM with certain organization may dramatically reduce the number of accesses to the LAM (main memory), thus considerably speeding-up related applications*.

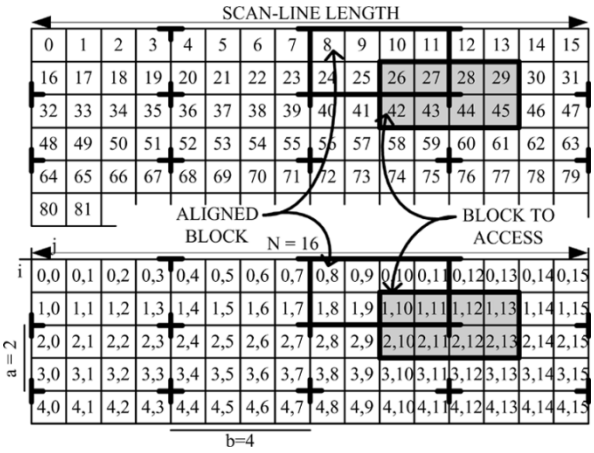


Fig. 3. Mapping scan-lines into 2-D addresses (considered example).

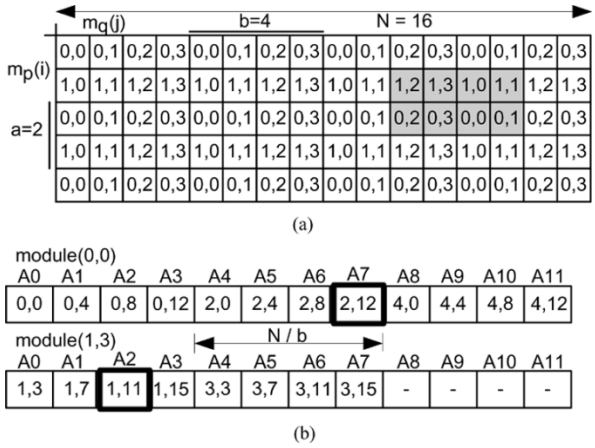


Fig. 4. Examples for $a = 2, b = 4, N = 16$. (a) Module assignments of the 2-D pixel area and (b) 2-D addresses and linear addressing within modules.

III. Block Addressable Memory

In this section, we propose the addressing scheme, the memory organization, and a potential implementation.

Addressing Scheme: Assume $M \times N$ image data stored in $k = a \times b$ memory modules ($1 \le a \le M; 1 \le b \le N$). Furthermore, assume that each module is linearly addressable. We are interested in parallel, conflict-free access of $a \times b$ blocks at any (i,j) location, defined as

$$B(i,j) = \{I(i+p, j+q) | 0 \le p < a, 0 \le q < b\},$$
$$0 \le i \le M - a, \quad 0 \le j \le N - b.$$

To align data in k modules without data replication, we organize these modules in a 2-D $a \times b$ matrix. A module assignment function, which maps a piece of data with 2-D coordinates (i,j) in memory module $(p,q) : 0 \le p < a, 0 \le q < b$, is required. We separate the function denoted as $m_{p,q}(i,j)$, into two mutually orthogonal assignment functions $m_p(i)$ and $m_q(j)$. We define the following module assignment functions for each module at position (p,q):

$$m_p(i) = (i - p) \bmod a \tag{2}$$
$$m_q(j) = (j - q) \bmod b \tag{3}$$

The addressing function for module (p,q) with respect to coordinates (i,j) is defined as

$$A_{p,q}(i,j) = (i \operatorname{div} a + c_i) \cdot \frac{N}{b} + j \operatorname{div} b + c_j \tag{4}$$
$$c_i = \begin{cases} 1, & i \bmod a > p \\ 0, & \text{otherwise.} \end{cases} \qquad c_j = \begin{cases} 1, & j \bmod b > q \\ 0, & \text{otherwise.} \end{cases}$$

Obviously, if $p = a - 1 \Rightarrow c_i = 0$ for $\forall i$; if $q = b - 1 \Rightarrow c_j = 0$ for $\forall j$, respectively. In essence, c_i, and c_j are the module assignment functions, implicitly embedded into the linear address $A_{p,q}(i,j)$.

Example: Consider the motivating example of Section II and the pixel area from Fig. 1(a). The same pixel area is mapped into a 2-D addressing space with N = 16 as depicted in Fig. 3. In this new mapping, we address data by columns and rows, as 2-D addressing is the actual addressing performed at algorithmic level. That is, byte 27 is referred to as $(1,11)$. Consequently, we have to perform the physical memory partitioning and assignment of data. Assume that data will be stored into linearly byte addressable memory modules, organized in a 2×4 matrix. Because in our example we have $5 \times 16 = 80$-byte memory, we subdivide the physical memory into eight modules in total, 10 bytes each. Each pixel has to be allocated in a specific module by the assignment function. The memory module assignments of all pixels from the considered pixel area for a = 2, b = 4 are depicted in Fig. 4(a). In the Figure, the pixel with 2-D address $(1,11)$ from Fig. 3 is allocated by the module assignment function in module $(1,3)$. At the second addressing level, the linear address of each individual pixel within the module *(intramodule address)*, has to be determined. The addressing function (4) generates a unique intramodule address within an uniquely assigned memory module, for each and every byte from the 2-D addressing space. The intramodule address of pixel $(1,11)$, determined by (4) is 2, denoted as A2 in module $(1,3)$ [see Fig. 4(b)]. Consequently, the addressing scheme is in fact performed at two levels- module assignment and intramodule addressing.

We access blocks rather than bytes (for the example—2×4 bytes). Blocks are addressed by the 2-D coordinates of their upper-left pixels. Consider the shaded nonaligned block 26–45 addressed as $B(1,10)$ (see Fig. 3). Note that the pixels of a block are accessed from all eight modules simultaneously, in parallel. Using (2)–(4), we can calculate the linear address of the pixels from the considered block for each module (p,q) with respect to 2-D address $i,j = (1,10)$:

- **module** $(p,q) = (0,0)$:

$$\left.\begin{array}{l} i \bmod a = 1 > p \Rightarrow c_i = 1 \\ j \bmod b = 2 > q \Rightarrow c_j = 1 \end{array}\right\} \Rightarrow A_{0,0}(1,10) = 7.$$

- **module** $(p,q) = (1,3)$

$$\left.\begin{array}{l} i \bmod a = 1 = p \Rightarrow c_i = 0 \\ j \bmod b = 2 = q \Rightarrow c_j = 0 \end{array}\right\} \Rightarrow A_{1,3}(1,10) = 2.$$

That is, the pixels of block $i,j = (1,10)$ will be allocated at address 7 in module $(p,q) = (0,0)$ and at address 2 in module $(p,q) = (1,3)$. Identically, the intramodule addresses of the remaining six pixels of the considered block can be calculated for each of the remaining six modules to be $A_{0,1}(1,10) = 7, A_{0,2}(1,10) = 6, A_{0,3}(1,10) =$

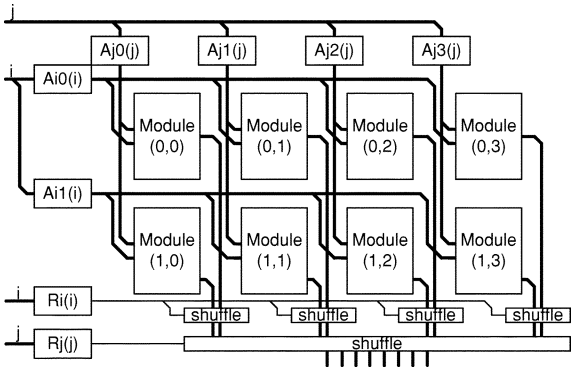


Fig. 5. The 2DAM for $a = 2, b = 4$, and $N = 2^n \geq 16$.

$6, A_{1,0}(1,10) = 3, A_{1,1}(1,10) = 3, A_{1,2}(1,10) = 2$. Fig. 4(b) illustrates the internal linear addressing and data alignment within the considered two memory modules. Note that having the intramodule addresses of all pixels in the considered block, we only need to know which module contains the upper-left pixel $(i,j) = (1,10)$ to reorder the data properly. The upper-left pixel of block $B(1,10)$ is calculated (from the zeroes of (2) and (3) to be located in module $(p,q) = (1,2)$. *Thus, having each and every of the 8 block pixels localized in each and every of the 8 modules, we can access the entire block in one cycle by accessing all the modules in parallel.* Yet identically, it can be shown that any 2×4 block, regardless its position, can be accessed in a single cycle. Recall that $B(1,10)$ is the 2-D notation of block $\{26-45\}$ from the motivating example. That block was accessible in two or four cycles from a conventional 8 byte LAM, thus two to four times slower than the proposed scheme at the same bandwidth of 8 bytes per cycle.

Memory Organization and Implementation: Equations (2)–(4) are generally valid for any natural value of parameters a, b, and N. To implement the proposed addressing and module assignment functions, however, we will consider practical values of these parameters. Since pixel blocks processed in MPEG algorithms have dimensions up to 16×16, values of practical significance for parameters a and b are the powers of two up to 16 (i.e., 1, 2, 4, 8, 16). For the particular implementation example we will consider $a \times b = 2 \times 4$.

Module Addressing: The module addressing function is *separable* thus, the function can be represented as a sum of two functions of a *single* and *unique* variable each (i.e., variables i and j). That is, $A_{p,q}(i,j) = Ai_p(i) + Aj_q(j)$ allowing the address generators to be implemented per column and per row (see Fig. 5) instead of implemented as individual addressing circuits for each of the memory modules. Taking into account the separability of $A_{p,q}(i,j)$ and considering an arbitrary range of picture dimensions to be stored, we can define $C_h = N = 2^n, n \geq 4$ as "horizontal capacity" of the 2DAM (to be discussed later). The requirements for the frame sizes of all MPEG standards and for video object planes (VOPs)[9] in MPEG-4 are constituted to be multiples of 16, thus, N is a multiple of 2^4 by definition. Assuming the discussed practical values of N and b, further analysis of (4) suggests that $j \operatorname{div} b + c_j < (N/b)$ and $(j \operatorname{div} b + c_j)_{\max} = (N/b) - 1$, i.e., no carry can be ever generated between $Ai_p(i)$ and $Aj_q(j)$. Therefore, we can implement $A_{p,q}(i,j)$ for every module (p,q) by simply routing signals to the corresponding address generation blocks without actually summating $Ai_p(i) + Aj_q(j)$. Fig. 6(a) illustrates address generation circuitry of q-addresses $(Aj_q(j))$ for all modules except the first $(1 \leq q < b)$. With respect to (4), if c_j is 1 the quotient $j \operatorname{div} b$ should be incremented by one, otherwise it should not be changed. To determine the value of c_j, a Look-Up-Table (LUT) with $j \bmod b$ inputs can be used. For the assumed practical values of a and $b(\leq 16)$, such a LUT would have at most 4 inputs, i.e., *c_j is a binary function of at most 4 binary digits*. Row p-addresses are generated identically. For $p = 1$ or $q = 3, c_i = 0, c_j = 0$ respectively. Therefore, address generation in these cases does not require a LUT and an incrementor. Instead, it is just routing $i \operatorname{div} a$ and $j \operatorname{div} b$ to the corresponding memory ports, i.e., blocks $Ai_1(i)$ and $Aj_3(j)$ in Fig. 5 are empty. Fig. 6(b) depicts all 4 LUTs for the case $a \times b = 2 \times 4$. The usage of LUTs to determine c_i and c_j is not mandatory, fast pure logic can be utilized instead.

Data Routing Circuitry: In Fig. 5, the shuffle blocks, together with blocks $R_p(i)$ and $R_q(j)$, illustrate the data routing circuitry. The shuffle blocks are in essence circular barrel shifters, i.e., having the complexity of a network of multiplexors. An $n \times n$ shuffle is actually an $n \rightarrow 1$ n-way multiplexor. In the example from Fig. 5, the i-level shuffle blocks are four $(2 \rightarrow 1)$ 16-bit multiplexors and the j-level one is $(4 \rightarrow 1)$ 64-bit. To control the shuffle blocks, we can use the module assignment functions for $p = q = 0$, i.e., $R_i(i) = i \bmod a$ and $R_j(j) = j \bmod b$. These functions calculate the (p,q)-coordinates of the "upper-left" pixel of the desired block, i.e., pixel (i,j). For the assumed practical values of a and b being powers of two, the implementation of $R_i(i)$ and $R_j(j)$ is simple routing of the least-significant $\log_2(a)$-bits [resp., $\log_2(b)$] to the corresponding shuffle level.

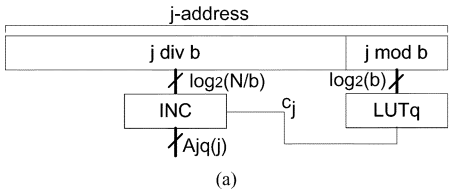
2DAM Capacity: Earlier, we have defined the "horizontal capacity" of 2DAM as $C_h = N = 2^n, n \geq 4$. C_h is the *maximal scanline length in bytes (pixels), the 2DAM can store without addressing conflicts*. The "vertical capacity" of 2DAM is denoted as C_v and defined as the *maximal number of C_h–byte (C_h-pixel) scanlines the 2DAM can store*. Finally, *the capacity C_{2DAM} of a 2DM is defined as the couple $(C_h \times C_v)$-bytes (pixels).*

LAM Interface: Fig. 7 depicts the organization of the interface between LAM and 2DAM (recall Fig. 2) for the modules considered in Fig. 5. Data bus width of the LAM is denoted by W (in number of bytes). In the particular example, W is assumed to be 2, therefore modules have coupled data busses. For each (i,j) address, the AGEN block sequentially generates addresses to the LAM and distributes write enable (WE) signals to a corresponding module couple. Two module WE signals $(\mathrm{WE}_i, \mathrm{WE}_j)$ are assumed for easier row and column selection. In the general case, the AGEN block should sequentially generate $(a \cdot b/W)$ LAM addresses for each (i,j) address. Provided that pixel data is stored into LAM in scan-line manner, the LAM addresses to be generated are defined as

$$A_{\mathrm{LAM}}(i,j) = \{a \cdot (i \operatorname{div} a) + k\} \cdot N + b \cdot (j \operatorname{div} b) + l \cdot W.$$

Which, assuming that only aligned blocks will be accessed from the LAM (i.e., (i,j) are aligned), can be simplified:

$$A_{\mathrm{LAM}}(i,j) = (i+k) \cdot N + j + l \cdot W$$
$$k = 0, 1, \ldots, a-1; l = 0, 1, \ldots, \tfrac{b}{W} - 1. \quad (5)$$



(a)

j mod b		c_j q=0	c_j q=1	c_j q=2	i mod a	c_i p=0
0	0	0	0	0	0	0
0	1	1	0	0	1	1
1	0	1	1	0	-	-
1	1	1	1	1	-	-

(b)

Fig. 6. Module address generation. (a) Generation Circuit of q-addresses for $1 \leq q < b$. (b) LUTs contents for $a = 2, b = 4$.

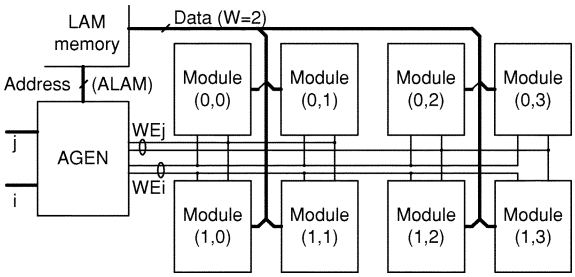


Fig. 7. LAM interface for $W = 2, a = 2, b = 4$.

In the 2DAM, the data are simultaneously written in modules:

$$(p,q) = (k, l \cdot W), (k, l \cdot W + 1), \ldots, (k, l \cdot W + W - 1) \quad (6)$$

as each byte of the word is stored at local module address:

$$A^{\mathrm{LAM}}_{p,q}(i,j) = (i \operatorname{div} a) \cdot \frac{N}{b} + j \operatorname{div} b. \quad (7)$$

Note, that accessing only aligned blocks from the LAM enables thorough bandwidth utilization. When only aligned blocks are addressed, all address generators issue the same address, due to (4). Therefore, during write operations into 2DAM, the same addressing circuitry can be used as for reading. If the modules are true dual port, the write port addressing can be simplified to just proper wiring of both i and j address lines because the incrementor and the LUTs from Fig. 6(a) are not required. Therefore, module addressing circuitry is not depicted in Fig. 7.

Addressing Consistency: In the following, we will prove that the described scheme provides a consistent LAM and 2DAM addressing. It means that each and every byte is allocated in the same memory module and at the same intramodule address by both LAM and 2DAM addressing schemes.

Lemma 1: $x \bmod z = x - n \cdot z$ iff $0 \leq x - n \cdot z < z; \forall x, n, z \in \mathbb{N}$.

Proof: 1. If $x \bmod z = x - n \cdot z \Rightarrow 0 \leq x - n \cdot z < z; \forall x, n, z \in \mathbb{N}$ is true by the definition of mod operation. 2. If $0 \leq x - n \cdot z < z \Rightarrow xmodz = x - n \cdot z; \forall x, n, z \in \mathbb{N}$. Let $x \bmod z = x - p \cdot z$. Then, by definition $0 \leq x - p \cdot z < z$. Assume $p \neq n \Rightarrow |p - n| \geq 1$. We derive the system:

$$\left\| \begin{array}{l} 0 \leq x - n \cdot z < z \\ 0 \leq x - p \cdot z < z \end{array} \right.$$

Its only solution $p = n$ contradicts to the assumption. ■

Lemma 2: $(x - y) \bmod z = (x \bmod z - y) \bmod z; \forall y < z; \forall x, y, z \in \mathbb{N}$.

Proof: By definition $x \bmod z = x - n1 \cdot z$ and $(x \bmod z - y) \bmod z = (x \bmod z - y) - n2 \cdot z$. $\Rightarrow$ By substitution and based on Lemma 1, we derive: $(x \bmod z - y) \bmod z = (x - n1 \cdot z - y) - n2 \cdot z = (x - y) - (n1 + n2) \cdot z = (x - y) \bmod z$ ■

Lemma 3: $(x \operatorname{div} y) \cdot y = x - x \bmod y; \forall x, y \in \mathbb{N}$.

Proof:

$$\left\| \begin{array}{l} x \bmod y = p \\ x \operatorname{div} y = k \\ k \cdot y + p = x \end{array} \right. \Rightarrow \begin{array}{c} (x \operatorname{div} y) \cdot y = \\ = k \cdot y = x - p = \\ = x - x \bmod y \end{array}$$

■

Theorem 1: **(Consistency between the 2DAM and the LAM addressing schemes).** Assume the 2DAM and LAM addressing interface schemes defined by (2)–(4) and (5)–(7), respectively. Any byte (i', j') is allocated in the same memory module at the same intramodule address by both addressing schemes.

Proof: (Consistency of module assignments.) Consider byte (i', j'). In consistence with (5), we define $k = i' \bmod a$ and $l = (j' \bmod b) \operatorname{div} W$. Considering the LAM interface and Lemma 3, the module, where byte (i', j') should be stored is calculated as follows:

$$\begin{aligned} &(p,q) \\ &= (k, l \cdot W + (j' \bmod b) \bmod W) \\ &= (k, \{(j' \bmod b) \operatorname{div} W\} \cdot W + (j' \bmod b) \bmod W) \\ &= (k, (j' \bmod b) - (j' \bmod b) \bmod W \\ &\quad + (j' \bmod b) \bmod W) \\ &\Rightarrow (p,q) = (k, j' \bmod b) \end{aligned} \quad (8)$$

Considering (2)–(3) for the 2DAM module allocation and Lemma 2, we derive:

$$\begin{array}{c|c} m_p(i') = & m_q(j') = \\ = (i' - p) \bmod a = 0 & = (j' - q) \bmod b = 0 \\ (i' \bmod a - p) \bmod a = 0 & (j' \bmod b - q) \bmod b = 0 \\ (k - p) \bmod a = 0; k < a & j' \bmod b < b \end{array}$$
$$\Rightarrow p = k; q = j' \bmod b. \quad (9)$$

Equations (8) and (9) indicate that any byte (i', j') will be allocated in the same memory module both by the LAM interface and by the 2DAM read circuitry.

(Consistency of intramodule addresses.) Assume *(i,j)* is the aligned block, containing byte (i', j'), i.e., $i \operatorname{div} a = i' \operatorname{div} a, j \operatorname{div} b = j' \operatorname{div} b$. Consider (4):

$A_{p,q}(i', j') = (i' \operatorname{div} a + c_i) \cdot (N/b) + j' \operatorname{div} b + c_j$, from (9): $p = i' \bmod a$ and $q = j' \bmod b \Rightarrow c_i = c_j = 0, \Rightarrow A_{p,q}(i', j') = (i' \operatorname{div} a) \cdot (N/b) + j' \operatorname{div} b \Rightarrow$(Rec. assumption) $A_{p,q}(i', j') = (i \operatorname{div} a) \cdot (N/b) + j \operatorname{div} b$, identical to (7). ■

Example: We consider a single (arbitrary chosen) byte and show that it is allocated in the same memory module and at the same intramodule address both by the LAM and by the 2DAM addressing schemes. Assume that visual data is scan-line aligned in LAM with word length of 2 bytes and big-endian convention. Consider the byte with 2-D address $(1, 11)$ (see Fig. 3). The memory hierarchy of Fig. 2 indicates that byte

$(1, 11)$ has to be loaded from the LAM into the 2DAM by means of the proposed LAM interface. Assuming that the 2DAM is first loaded in its entirety, all aligned blocks of the considered 5×16-byte area are to be loaded from the LAM into the 2DAM. Byte $(1, 11)$ is assigned in the LAM as part of aligned block $(0, 8)$. The LAM addresses of the four 2-byte words containing the pixels of the block are $A_{\mathrm{LAM}} = 8, 10, 24, 26$, see Fig. 3. The LAM address of the 2-byte word, containing the considered pixel $(1, 11)$ is calculated from (5) to be: $A_{\mathrm{LAM}}(0, 8)_{k=1,l=1} = (0 + 1) \cdot 16 + 8 + 1 \cdot 2 = 26$. Recall Fig. 3, where byte $(1, 11)$ had LAM address 27. Thus, in the assumed big-endian LAM convention, the considered byte 27 is the MSB of the 2-byte memory word aligned at address 26. Considering (6), this 2-byte word should be stored into modules $(1, 2)$ and $(1, 3)$, see Fig. 7. The MSB, i.e., byte 27, should be stored into module $(p, q)_{k=1,l=1} = (k, l \cdot W + W - 1) = (1, 3)$. Its intramodule address with respect to the LAM interface is calculated from (7) to be: $A_{1,3}^{\mathrm{LAM}}(0, 8) = (0 \text{ div } 2) \cdot (16/4) + 8 \text{ div } 4 = 2$ That is, *byte* $(1, 11)$ *with LAM address 27, will be stored by the LAM-to-2DAM interface into module (1,3) at intramodule address 2*. Consider the 2DAM addressing scheme, the shaded nonaligned block $(1, 10)$ in Figs. 3 and 4, and (2)–(4). Indeed, *considering the 2DAM addressing scheme, byte* $(1, 11)$ *can be read from address location 2 of module* $(1, 3)$, as it was shown in the previous example.

Critical Paths: Assuming generic synchronous memories we separate the critical paths into two: address generation and data routing. For the proposed circuit implementation, the address generation critical path (CP_A) is determined by $\mathrm{CP}_A = \max(\mathrm{CP}_{\mathrm{add}(M/a)}, \mathrm{CP}_{\mathrm{add}(N/b)}) + \mathrm{CP}_{\mathrm{LUT}}$. That is the critical path of either a $\log_2((M/a))$-bit, or a $\log_2((N/b))$-bit adder, whichever is longer, and the critical path of one (maximum 4-input) LUT. The data routing critical path (CP_D) is: $\mathrm{CP}_D = \mathrm{CP}_{\mathrm{mux}_a} + \mathrm{CP}_{\mathrm{mux}_b}$. That is, the sum of the critical paths of one $a \rightarrow 1$ multiplexor and one $b \rightarrow 1$ multiplexor.

IV. Experimental Results and Related Work

We note that our proposal is general, therefore we do not consider implementations bound to any particular computer architecture or specific multimedia software. Thus, we are allowed to analyze the proposed memory organization regardless the system implementation platform and the particular multimedia application. By doing so, we isolate the performance benefits due to our proposal only. Binding the memory to any particular processor system would introduce results dependent on the considered architectural context. Moreover, intermingling architectural features with the proposed memory organization would not give a clear indication of the benefits due to the memory organization only. It would rather introduce architectural discussions outside the scope of this paper.

In this section, we present an experimental case study for a number of FPGA-based designs and compare to related works.

Case Study: A generic VHDL model of the memory organization has been developed and synthesized for the recent Virtex II Pro FPGA technology of Xilinx. We consider reconfigurable implementations as we also envision that the proposed organization can be embedded in an FPGA augmented processor (e.g., [10]) being part of its reconfigurable memory subsystem. Table III contains synthesis results for the 2vp50ff1152 FPGA device (the last column displays some of the resources available on the chip). The on-chip memory volume allows frames or VOPs sized up-to 512×1024 pixels to be stored. It should be noted that more than one frame can be stored in the memory and accessed, depending on the particular frame format. For example, up-to fourteen CIF frames (144×176) can be stored into the implemented 512×1024 storage. This issue is much more beneficial in MPEG-4, where the arbitrary shaped VOPs to be stored vary both in size and number for each particular codec session. Synthesis data for practical MPEG pattern sizes of 2×4, 4×8, 8×8, and 16×16-pixels indicate that respective structures can be efficiently implemented with a fraction of the available FPGA resources. Only the 16×16 pattern creates a resource conflict with regard to the available IO pins of the chip. This conflict, however, should not be considered as a problem, since structures with bandwidth of that magnitude are usually intended for on-chip implementations. In the "Adders" rows of Table III, the notation "bits/#" denotes the number of bits in an adder and the corresponding number of such adders, respectively. Results indicate that in the most common case of 8×8 block patterns, 3287 Virtex II Pro slices are required, which is 13% of the 2vp50ff1152 FPGA device resources.

In Table IV, transfer speedup estimations are presented, assuming $T_{\mathrm{LAM}} = 10$ ns. Calculations are made according to the figures and notations presented in Table II. In BC, all blocks are assumed to be nonaligned, while in WC the very unlikely scenario that all blocks are aligned and accessed only once is considered. T_{2DA} values are derived from the synthesis reports for the designs considered in Table III. Figures in Table IV indicate that even in the unfavorable case when 2DAM is slower than the LAM, considerable transfer speedups of up to 8x can be achieved, due to the proposed memory organization.

TABLE III
Synthesis for Frames up-to 512×1024 (Device 2vp50ff1152)

$a \times b$	2 x 4	4 x 8	8 x 8	16 x 16	Avail.
2-1mux	192	1280	3072	16384	N.A.
Adders:	4	10	14	30	N.A.
bits/#	8/1	7/3	6/7	5/15	N.A.
bits/#	8/3	7/7	7/7	6/15	N.A.
# Slices	534	1512	3287	15408	24640
%	1	6	13	63	100
# LUT4	928	2630	5723	26805	49280
%	1	5	11	54	100
IOs	100	292	548	2084	756
BRAM	8x 64K	32x 16K	64x 8K	256x 2K	522K

TABLE IV
Estimated Transfer Speedups for $T_{\mathrm{LAM}} = 10$ ns.

$a \times b$	T_{2DA}	t= $\frac{T_{2DA}}{T_{LAM}}$	W	Transfer Speedup		
				BC	Mix.	WC
8x8	16,7ns	1,67	1 (8 bits)	7,45	7,34	0,97
			2 (16 bits)	7,05	6,88	0,95
			4 (32 bits)	6,54	6,27	0,91
16x16	18,8ns	1,88	1 (8 bits)	8,03	8,00	0,99
			2 (16 bits)	8,05	8,00	0,99
			4 (32 bits)	8,10	8,00	0,97

TABLE V
COMPARISON TO OTHER PROPOSED SCHEMES

Related Work	scalability	# modules	implementation drawbacks or limitations
Budnik, Kuck [1]	$\sqrt{N} \times \sqrt{N}$ from $N \times N$	prime $m > N = 2^n$	*mod(m)*, crossbar, no addressing
Lawrie [3]	$\sqrt{N} \times \sqrt{N}$	$m = 2.N; N = 2^{2n+1}$	*mod(m)*, no addressing
Voorhis, Morin [4]	$p \times q$ from $M \times N$	$m \geq p \times q$	not separable,*mod(pq),mod(pq+1),*
Kim, Prasanna [5]	$\sqrt{N} \times \sqrt{N}$ from $N \times N$	$m = N$	certain blocks are inaccessible
De-lei Lee [6]	$\sqrt{N} \times \sqrt{N}$ from $N \times N$	$m = N$	many modules for higher N
Sproull et al. [8]	8×8	8×8	time-space multiplexing, not general
Park [7]	$p \times q$ from $M \times N$	prime $m > p \times q$	not separable, many adders, big LUTs
HiPAR-DSP [11], [12]	$N \times N$	$m = (1 + N)^2$	$2 \times N + 1$ additional modules, *mod(m)*
HiPAR-DSP16 [14]	$p \times q$ from $M \times N$	$m >> p \times q$	big number of modules, *mod(m)*
This proposal	$p \times q$ from $M \times N$	$m = p \times q$	none of the above, rectangular patterns only

Related Work: Accessing blocks of memory has been a main concern for vector (array) processors researchers and developers for long time. Two major groups of memory organizations for parallel data access have been reported in literature—organizations with and without data replication (redundancy). We are interested only in those without data replication. Another division is made with respect to the number of memory modules—equal to the number of accessed data points and exceeding this number. Organizations with a prime number of memory modules can be considered as a subset of the latter. Their essential drawback is that the addressing functions are nonseparable and are more complex, thus slower and costly to implement. We have organized our comparison with respect to block accesses, discarding other data patterns, due to the specific requirements of visual data compression. It should be noted, however, that our design can easily support horizontal and vertical lines of length $a \times b$.

To compare designs, two basic criteria have been established: scalability and implementation drawbacks in terms of speed and/or complexity. Comparison results are summarized in Table V. Budnik and Kuck [1] described a scheme for conflict free access of $\sqrt{N} \times \sqrt{N}$ square blocks out of $N \times N$ arrays, utilizing $m > N = 2^n$ memory modules, where m is a prime number. Their scheme allows the complicated full crossbar switch as the only possibility for data alignment circuitry and many costly *modulo(m)* operations with m not a power of two. In a publication, related to the development of the Burroughs Scientific Processor, Lawrie [3] proposes an alignment scheme with data switching, simpler than a crossbar switch, but still capable to handle only $\sqrt{N} \times \sqrt{N}$ square blocks out of $m = 2N$ modules, where $N = 2^{2n+1}$. Both schemes in [1] and [3] require larger number of modules than the number of simultaneously accessed elements (N). Furthermore, in both papers authors do not describe the addressing circuitries for their schemes. Voorhis and Morin [4] suggest various addressing functions considering $p \times q$ subarray accesses and different number of memory modules m: both $m = p \times q$ and $m > p \times q$. Neither of the functions proposed in [4] is separable, which leads to an extensive number of address generation and module assignment logic. In [5], the authors propose a scheme based on Latin squares and capable of accessing $\sqrt{N} \times \sqrt{N}$ square blocks out of $N \times N$ arrays but not from random positions. Similar drawbacks has the scheme proposed in [6]. One early graphical display system, described in [8], can be considered a partial case of our scheme, since authors describe square 8×8 submatrix accesses and memory alignment similar to the proposed in our scheme. The authors in [8] did not consider rectangular subarray accesses, which are not directly deducible from the proposed reading. No formalization of the addressing functions was presented either. A more recent display system memory, capable of simultaneous access of $p \times q$ rectangular subarrays is described in [7]. It utilizes a prime number of memory modules, which enables accesses to numerous data patterns, but disallows separable addressing. Large LUTs (in size and number) and long critical paths containing consecutive additions are the other drawbacks of [7]. Therefore, it is slower and requires more memory modules than our proposal. A memory organization, capable of accessing $N \times N$ square blocks, aligned into $(1 + N)^2$ memory modules was described in [11]. The same scheme was used for the implementation of the matrix memory of the first version of HiPAR-DSP [12].Besides the restriction to square accesses only, that memory system uses a redundant number of modules, due to additional DSP-specific access patterns considered. A definition of rectangular $p \times q$ block random addressing scheme from the architectural point of view dedicated for multimedia systems was introduced in [13], but no particular organization was presented there. In the latest version of HiPAR16 [14], the matrix memory was improved so that a restricted number of rectangular patterns could also be accessed. This design, however, still uses excessive number of memory modules as p and M respectively q and N should not have common divisors; e.g., to access the example 2×4 pattern, the HiPAR16 memory requires $3 \times 5 = 15$ memory modules, instead of eight for our proposal. The memory of [14] would require more-complicated circuitry. Similarly to [8], [12], [14] assume separability, however, the number of utilized modules is even higher than the closest prime number to $p \times q$. An alternative solution, proposed in [15], is the utilization of hardwired register buffers. Such an approach is limited by the implementable registers size and high routing complexity - in contrast to the current proposal, which allows arbitrary larger data to be accessed. Compared to [1], [3], [5]–[8], [11], [12], [14], our scheme enables higher scalability and lower number of memory modules. This directly affects the design complexity, which has been proven to be very low in our case. Address function separability reduces the number of address generation logic and critical path penalties, thus it enables faster implementations. Regarding address separability, we differentiate from [1], [3]–[7], where address separability is not supported. As a result, *our design is envisioned to have the shortest critical path penalties among all referenced works*.

V. Conclusion

We presented a scalable memory organization capable of addressing randomly aligned rectangular data patterns out of a 2-D data storage. High performance is achieved by reduced number of data transfers between memory hierarchy levels, efficient bandwidth utilization, and short hardware critical paths. In the proposed design, data are located in an array of byte addressable memory modules by an addressing function, implicitly containing module assignment functions. An interface to a linearly addressable memory has been provided to load the array of modules. Theoretical analysis and experimental evidence suggest that the proposed 2-D addressing scheme has advantages over existing related art.

Acknowledgment

The authors would also like to thank J. P. Wittenburg for his valuable opinion, which helped to improve the quality of this material.

References

[1] P. Budnik and D. J. Kuck, "The organization and use of parallel memories," *IEEE Trans. Comput.*, vol. 20, no. 12, pp. 1566–1569, 1971.
[2] P. M. Kogge, *The Architecture of Pipelined Computers*. New York: McGraw-Hill, 1981.
[3] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Comput.*, vol. C-24, no. 12, pp. 1145–1155, 1975.
[4] D. C. van Voorhis and T. H. Morrin, "Memory systems for image processing," *IEEE Trans. Comput.*, vol. C-27, no. 2, pp. 113–125, 1978.
[5] K. Kim and V. K. Prasanna, "Latin squares for parallel array access," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 4, pp. 361–370, 1993.
[6] D. lei Lee, "Scrambled Storage for Parallel Memory Systems," in *Proc. IEEE Int. Symp. Computer Architecture*, 1988, pp. 232–239.
[7] J. W. Park, "An efficient buffer memory system for subarray access," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 3, pp. 316–335, 2001.
[8] R. F. Sproull, I. Sutherland, A. Thomson, S. Gupta, and C. Minter, "The 8 by 8 display," *ACM Trans. Graph.*, vol. 2, no. 1, pp. 32–56, 1983.
[9] *MPEG-4 Video Verification Model Version 16.0*. ISO/IEC JTC11/SC29/WG11, N3312.
[10] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The molen polymorphic processor," *IEEE Trans. Comput.*, vol. 53, no. 11, pp. 1363–1375, 2004.
[11] J. Kneip, K. Ronner, and P. Pirsch, "A data path array with shared memory as core of a high performance DSP," in *Proc. Int. Conf. on Application Specific Array Processors*, Aug. 1994, pp. 271–282.
[12] J. P. Wittenburg, M. Ohmacht, J. Kneip, W. Hinrichs, and P. Pirsh, "HiPAR-DSP: A parallel VLIW RISC processor for real time image processing applications," in *Proc. 3rd Int. Conf. on Algorithms and Architectures for Parallel Processing (ICAPP 97)*, Dec. 1997, pp. 155–162.
[13] G. Kuzmanov, S. Vassiliadis, and J. van Eijndhoven, "A 2D addressing mode for multimedia applications," in *Workshop on System Architecture, Modeling, and Simulation (SAMOS 2001)*, vol. 2268, July 2001, pp. 291–306.
[14] H. Kloos, J. Wittenburg, W. Hinrichs, H. Lieske, L. Friebe, C. Klar, and P. Pirsch, "HiPAR-DSP 16, a scalable highly parallel DSP core for system on a chip: Video and image processing applications," in *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, vol. 3, May 2002, pp. 3112–3115.
[15] M. B. Haverkamp, G. Kuzmanov, and S. Vassiliadis, "Implementing 2D memory buffers for MPEG," *PRORISC 2003*, pp. 90–94, Nov. 2003.

Georgi Kuzmanov (S'95–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 Laboratory at TU Delft. His research interests include reconfigurable computing, media processing, computer arithmetic, computer architecture and organization, vector processors, and embedded systems.

Georgi Gaydadjiev (M'01) was born in Plovdiv, Bulgaria, in 1964.

He is currently an Assistant Professor with the Computer Engineering Laboratory, Delft University of Technology (TU Delft), The Netherlands. His research and development experience includes 15 years in hardware and software design at System Engineering Ltd., Pravetz, Bulgaria, and Pijnenburg Microelectronics and Software B.V., Vught, The Netherlands. His research interests include embedded systems design, advanced computer architectures, hardware/software co-design, VLSI design, cryptographic systems, and computer systems testing.

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, Mathematics, and Computer Science (EEMCS) Department, Delft University of Technology (TU Delft), The Netherlands. He previously served in the Electrical Engineering faculties of Cornell University, Ithaca, NY, and the State University of New York (SUNY), Binghamton. For a decade, he worked with IBM, where he was involved in a number of advanced research and development projects.

Dr. Vassiliadis has received numerous awards for his work, including 24 publication awards, 15 invention awards, and an outstanding innovation award for engeneering/scientific hardware design. His 72 U.S. patents rank him as the top all-time IBM inventor. He received an honorable mention Best Paper award at the ACM/IEEE MICRO25 in 1992 and Best Paper awards in the IEEE CAS (1998), IEEE ICCD (2001), and PDCS (2002).