

# Energy-Aware Memory Allocation Framework for Embedded Data-Intensive Signal Processing Applications\*

Florin BALASA<sup>†a)</sup>, Nonmember, Ilie I. LUICAN<sup>††</sup>, Student Member, Hongwei ZHU<sup>†††</sup>, and Doru V. NASUI<sup>††††</sup>, Nonmembers

**SUMMARY** Many signal processing systems, particularly in the multimedia and telecommunication domains, are synthesized to execute data-intensive applications: their cost related aspects—namely power consumption and chip area—are heavily influenced, if not dominated, by the data access and storage aspects. This paper presents an energy-aware memory allocation methodology. Starting from the high-level behavioral specification of a given application, this framework performs the assignment of the multidimensional signals to the memory layers—the on-chip scratch-pad memory and the off-chip main memory—the goal being the reduction of the dynamic energy consumption in the memory subsystem. Based on the assignment results, the framework subsequently performs the mapping of signals into both memory layers such that the overall amount of data storage be reduced. This software system yields a complete allocation solution: the exact storage amount on each memory layer, the mapping functions that determine the exact locations for any array element (scalar signal) in the specification, and an estimation of the dynamic energy consumption in the memory subsystem.

**key words:** multidimensional signal processing, memory management, memory allocation, dynamic energy consumption, signal-to-memory assignment

## 1. Introduction

Many multidimensional signal processing systems, particularly in the areas of multimedia and telecommunications, are synthesized to execute data-intensive applications, the data transfer and storage having a significant impact on both the system performance and the major cost parameters—power and area.

In particular, the memory subsystem is, typically, a major contributor to the overall energy budget of the entire system. The *dynamic* energy consumption is caused by memory accesses, whereas the *static* energy consumption is due to leakage currents. Savings of dynamic energy can be potentially obtained by accessing frequently used data from smaller on-chip memories rather than from the large off-chip main memory, the problem being how to optimally as-

sign the data to the memory layers. As on-chip storage, the scratch-pad memories (SPMs)—compiler-controlled static random-access memories, more energy-efficient than the hardware-managed caches—are widely used in embedded systems, where caches incur a significant penalty in aspects like area cost, energy consumption, hit latency, and real-time guarantees\*\*. Different from caches, the SPM occupies a distinct part of the virtual address space, with the rest of the address space occupied by the main memory. The consequence is that there is no need to check for the availability of the data in the SPM. Hence, the SPM does not possess a comparator and the miss/hit acknowledging circuitry [4]. This contributes to a significant energy (as well as area) reduction. Another consequence is that in cache memory systems, the mapping of data to the cache is done during the code execution, whereas in SPM-based systems this can be done at compilation time, using a suitable algorithm—as this paper will show.

The energy-efficient assignment of signals to the on- and off-chip memories has been studied since the late nineties. These previous works focused on partitioning the signals from the application code into so-called *copy candidates* (since the on-chip memories were usually caches), and on the optimal selection and assignment of these to different layers into the memory hierarchy. For instance, Kandemir and Choudhary analyze and exploit the temporal locality by inserting local copies [12]. Their layer assignment builds a separate hierarchy per loop nest and then combines them into a single hierarchy. However, the approach lacks a global view on the lifetimes of array elements in applications having imperfect nested loops. Brockmeyer et al. use the steering heuristic of assigning the arrays having the lowest access number over size ratio to the lowest memory layer first, followed by incremental reassignments [5]. Hu et al. can use *parts* of arrays as copies, but they typically use cuts along the array dimensions [10] (like rows and columns of

Manuscript received March 19, 2009.

Manuscript revised June 22, 2009.

<sup>†</sup>The author is with the Dept. of Computer Science and Information Systems, Southern Utah University, U.S.A.

<sup>††</sup>The author is with the Dept. of Computer Science, University of Illinois at Chicago, U.S.A.

<sup>†††</sup>The author is with the Physical IP Division, ARM Inc., San Jose, California, U.S.A.

<sup>††††</sup>The author is with American International Radio, Inc., Rolling Meadows, Illinois, U.S.A.

\*The content is based on the conference paper published in the proceedings of ASP-DAC 2009 [3].

a) E-mail: balasa@suu.edu

DOI: 10.1587/transfun.E92.A.3160

\*\*A detailed study [4] comparing the tradeoffs of caches as compared to SPMs found in their experiments that the latter exhibit 34% smaller area and 40% lower power consumption than a cache of the same capacity. Even more surprisingly, the runtime measured in cycles was 18% better with an SPM using a simple static knapsack-based allocation algorithm. As a general conclusion, the authors of the study found absolutely no advantage in using caches, even in high-end embedded systems in which performance is important. (Caches have been a big success for desktops though, where the usual approach to adding SRAM is to configure it as a cache.)

matrices). Udayakumaran and Barua propose a dynamic allocation model for SPM-based embedded systems [17], but the focus is on global and stack data, rather than on multidimensional signals. Issenin et al. perform a data reuse analysis in a multi-layer memory organization [11], but the mapping of the signals into the hierarchical data storage is not considered. The energy-aware partitioning of an on-chip memory in multiple banks has been studied by several research groups, as well. Techniques of an exploratory nature analyze possible partitions, matching them against the access patterns of the application [7]. Other approaches exploit the properties of the dynamic energy cost and the resulting structure of the partitioning space to come up with algorithms able to derive the optimal partition for a given access pattern [1].

Despite many advances in memory design techniques over the past two decades, existing computer-aided design (CAD) methodologies are still ineffective in many aspects. In the previous works, the reduction of the dynamic energy consumption in hierarchical memory subsystems is mainly based on heuristic explorations of the solution space rather than on a formal methodology. Also, several models of mapping the multidimensional signals into the physical memory were proposed in the past (see [13] for an overview). However, they all failed (a) to provide efficient implementations, (b) to prove their effectiveness in hierarchical memory organizations, and (c) to provide quantitative measures of quality for the mapping solutions. Moreover, the reduction of power consumption and the mapping of signals in hierarchical memory subsystems were treated in the past as separate problems.

This paper presents a memory allocation methodology for embedded data-intensive signal processing applications. Starting from the high-level behavioral specification of a given application, where the code is organized in sequences of loop nests and the main data structures are multidimensional arrays, this framework performs the assignment of the multidimensional signals to the memory layers—the on-chip scratch-pad memory and the off-chip main memory—the goal being the reduction of the dynamic energy consumption in the memory subsystem. The previous works do not take into account the non-uniform access patterns within same arrays; they make distinction only between the access intensity of *entire* arrays [5] (e.g., array A is more accessed than array B), or they try to heuristically identify the more accessed parts by imposing constraints on their shape and/or size [10] (e.g., the rows 1-2 of array A are more accessed than its rows 3-4). In contrast to the previous works, our analysis is more refined, allowing to formally identify those intensely-accessed areas of the array space—independent of their shape, size, or array dimensions. Assigning the most heavily-accessed parts of the arrays into the scratch-pad layer (requiring less energy consumed per access) entails important savings of the dynamic energy consumption in the memory subsystem. This is why our framework is *energy-aware*.

Based on the assignment results, the framework subse-

quently performs the mapping of signals into both memory layers such that the overall amount of data storage be significantly reduced. Different from the previous works, this mapping technique is designed to work in hierarchical memory organizations, operating with parts of the arrays that can be assigned to different physical memories. The polyhedral framework, common to both design phases (the signal assignment to the memory layers and the signal mapping into the data memories), entails a high computation efficiency since both phases rely on similar polyhedral operations.

This software system yields a complete allocation solution: the exact storage amount on each memory layer, the mapping functions that determine the exact locations for any array element (scalar signal) in the specification, metrics of quality for the allocation solution, and an estimation of the dynamic energy consumption in the memory subsystem using the CACTI power model [19].

The rest of the paper is organized as follows. Section 2 presents the algorithm that assigns the signals to the memory layers, aiming to minimize the dynamic energy consumption in the hierarchical memory subsystem subject to SPM size constraints. Section 3 describes the global flow of the memory allocation approach, focusing on the mapping aspects. Section 4 discusses on implementation and presents experimental results. Finally, Sect. 5 summarizes the main conclusions of this research.

## 2. Energy-Aware Signal Assignment to the Memory Layers

The algorithms describing the functionality of real-time multimedia and telecom applications are typically specified in a high-level programming language, where the code is organized in sequences of loop nests having as boundaries linear functions of the outer loop iterators. Conditional instructions are very common as well, and the multidimensional array references have (possibly complex) linear indexes (the variables being the loop iterators).

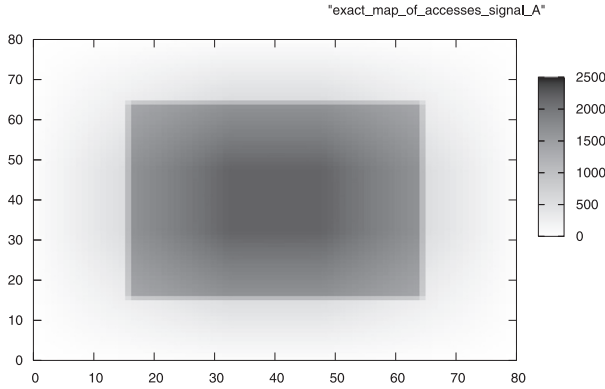
Figure 1 shows an illustrative example whose structure is similar to the kernel of a motion detection algorithm [6]. The problem is to automatically identify those parts of arrays from the given application code that are more intensely accessed, in order to steer their assignment to the energy-efficient data storage layer (the on-chip scratch-pad mem-

```

optDelta[0] = 0 ; // int A[81][81]: input;
for ( i=16; i<=64; i++ )
  for ( j=16; j<=64; j++ )
  { Delta[i][j][0] = 0 ;
    for ( k=i-16; k<=i+16; k++ )
      for ( l=j-16; l<=j+16; l++ )
        Delta[i][j][33*k-33*i+l-j+545] = A[i][j] - A[k][l]
          + Delta[i][j][33*k-33*i+l-j+544] ;
    optDelta[49*i+j-799] = Delta[i][j][1089] + optDelta[49*i+j-800];
  }
opt[0] = optDelta[2401];

```

**Fig. 1** Illustrative example whose structure is similar to a motion detection kernel ( $m = n = 16$ ,  $M = N = 64$ ) [6].



**Fig. 2** Exact map of memory read accesses (obtained by simulation) for the 2-D signal  $A$  from the illustrative code in Fig. 1.

ory) such that the dynamic energy consumption in the hierarchical memory subsystem be reduced.

The number of storage accesses for each array element can certainly be computed by the simulated execution of the code. The result of such a simulation is displayed in Fig. 2, where the area represents the so-called *index space* of the 2-D signal  $A$  from the illustrative code in Fig. 1. For each pair of possible indexes (between 0 and 80), the number of accesses was counted and the level of grey depends on the intensity with which the array elements are accessed (the darker the color, the higher the number of accesses). The array elements near the center of the index space are accessed with high intensity (for instance,  $A[40][40]$  is accessed 2,178 times;  $A[16][40]$  is accessed 1,650 times), whereas the array elements at the periphery are accessed with a significantly lower intensity (for instance,  $A[0][40]$  is accessed 33 times and  $A[0][0]$  only once).

The drawbacks of such an approach are twofold. First, the simulated execution may be computationally ineffective when the number of array elements is very significant, or when the application code contains deep loop nests. Second, even if the simulated execution were feasible, such a scalar-oriented technique would not be helpful since the addressing hardware of the data memories would result very complex. An address generation unit (AGU) is typically implemented to compute arithmetic expressions in order to generate sequences of addresses [14]; a set of array elements is not a good input for the design of an efficient AGU.

Our proposed computation methodology for energy-aware signal assignment to the memory layers is described below, after defining a few basic concepts.

Let  $M[x_1(i_1, \dots, i_n)] \cdots [x_m(i_1, \dots, i_n)]$  be an *array reference* of an  $m$ -dimensional signal  $M$ , in the scope of a nest of  $n$  loops having the iterators  $i_1, \dots, i_n$ . The array reference is characterized by an *iterator space* and an *index (or array) space*. The iterator space signifies the set of all iterator vectors  $\mathbf{i} = (i_1, \dots, i_n) \in \mathbf{Z}^n$  in the scope of the array reference, and it can be typically represented by a so-called  $\mathbf{Z}$ -polytope (a polyhedron bounded and closed, restricted to the set  $\mathbf{Z}^n$ ):  $\{\mathbf{i} \in \mathbf{Z}^n \mid \mathbf{A} \cdot \mathbf{i} \geq \mathbf{b}\}$ . E.g., for the array reference  $A[k][l]$  from the code in Fig. 1, the iter-

ator vector is the column vector  $\mathbf{i} = [i \ j \ k \ l]^T$ , the matrix  $\mathbf{A}$  has 8 rows (and 4 columns) derived from the lower and upper boundaries of the 4 nested loops, and  $\mathbf{b}$  is a column vector of 8 elements. The index space is the set of all index vectors  $\mathbf{x} = (x_1, \dots, x_m) \in \mathbf{Z}^m$  of the array reference. When the indexes of an array reference are linear mappings with integer coefficients of the loop iterators, the index space consists of one or several *linearly bounded lattices* [15]:  $\{\mathbf{x} = \mathbf{T} \cdot \mathbf{i} + \mathbf{u} \in \mathbf{Z}^m \mid \mathbf{A} \cdot \mathbf{i} \geq \mathbf{b}, \mathbf{i} \in \mathbf{Z}^n\}$ . E.g., for the same array reference  $A[k][l]$ ,  $\mathbf{T} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$  and

$$\mathbf{u} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

**Step 1** Let  $M$  be an indexed signal from the algorithmic specification. Decompose the array references  $M[x_1(i_1, \dots, i_n)] \cdots [x_m(i_1, \dots, i_n)]$  into disjoint lattices [2].

The motivation of the decomposition of the array references relies on the following intuitive idea: the disjoint lattices belonging to many array references are actually those parts of the array space of  $M$  more heavily accessed during the code execution. This decomposition into disjoint lattices — used also in [2], where it is explained in detail — can be performed analytically, by recursively intersecting the array references of signal  $M$ .

**Step 2** Compute the average number of memory accesses for each disjoint lattice of signal  $M$ .

The total number of memory accesses to a given linearly bounded lattice of  $M$  is computed as follows:

**Step 2.1** Select an array reference of  $M$  and intersect the given lattice with it. If the intersection result is not an empty set, it follows that the selected array reference and the given lattice have  $M$ -elements in common. The intersection is done in order to determine the expressions of the loop iterators for these common  $M$ -elements. (An example will be given below.)

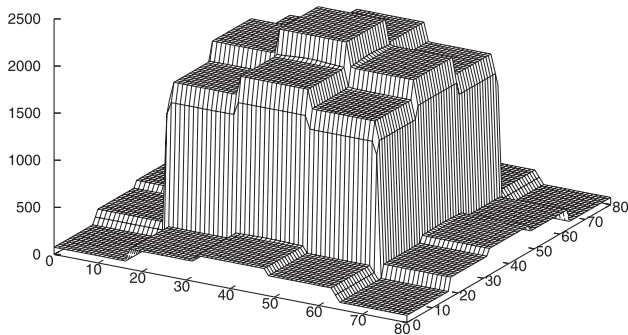
**Step 2.2** Compute the number of points in the (non-empty) intersection — a linearly bounded lattice as well [15]: this yields the number of memory accesses to the given lattice, as part of the selected array reference.

**Step 2.3** Repeat steps 2.1 and 2.2 for all the signal's array references in the code, cumulating the numbers of accesses to the given lattice.

*Step 2* is executed for each disjoint lattice obtained at *Step 1*. The overall result is a map of the memory accesses to the array space of signal  $M$ .

For example, let us consider one of signal  $A$ 's lattices<sup>†</sup>  $\{64 \geq x, y \geq 16\}$ . Intersecting it with the array reference  $A[k][l]$  (see the code in Fig. 1), we obtain the lattice  $\{i = t_1, j = t_2, k = t_3, l = t_4 \mid 64 \geq t_1, t_2, t_3, t_4 \geq 16, t_1 + 16 \geq t_3 \geq t_1 - 16, t_2 + 16 \geq t_4 \geq t_2 - 16\}$ . The size of this set is 1,809,025, which is the number of memory accesses to the given lattice as part of the array reference  $A[k][l]$ . Since the given lattice is also included

<sup>†</sup>When the lattice has  $\mathbf{T}=\mathbf{I}$  — the identity matrix — and  $\mathbf{u}=\mathbf{0}$ , the lattice is actually a  $\mathbf{Z}$ -polytope, like in this example.



**Fig. 3** Computed 3D map of memory *read* accesses for the signal *A* from the illustrative code in Fig. 1.

in the array reference  $A[i][j]$ , a similar computation yields 2,614,689 accesses to the same lattice as part of  $A[i][j]$ . Hence, the total amount of memory accesses to the given lattice is  $2,614,689 + 1,809,025 = 4,423,714$ . Since the number of *A*-elements covered by this lattice is 2,401, the average number of accesses for this lattice is 1,842.45.

Figure 3 displays a computed map of memory accesses for the signal *A*, where *A*'s index space is in the horizontal plane  $xOy$  and the numbers of memory accesses are on the vertical axis  $Oz$ . This computed map is an *approximation* of the exact map in Fig. 2, since the access distribution within each lattice is considered uniform (equal to the average value of accesses). Computing such approximate maps of accesses has an important advantage: the (usually very time-expensive) simulation is not needed any more, being replaced by algebraic computations. Note that a finer granularity in the decomposition of the index space of a signal into disjoint lattices entails a computed map of accesses closer to the exact map.

**Step 3** *Select the lattices having the highest access numbers, whose total size does not exceed the maximum SPM size (assumed to be a design constraint), and assign them to the SPM layer. The other lattices will be assigned to the main memory.*

Storing on-chip all the signals is, obviously, the most desirable scenario in point of view of dynamic energy consumption. This is usually not possible since the SPM size is, typically, limited to smaller values than the overall data storage requirement of the algorithmic specification. In our tests (Sect. 4), we compute the ratio between the expected dynamic energy reduction and the SPM size after mapping (see Sect. 3); the value of the SPM size maximizing this ratio is selected, the goal being to obtain the maximum benefit in energy point of view for the smallest SPM size.

### 3. Mapping Signals within Memory Layers

This design phase has the following goals: (a) to map the signals (already assigned to the memory layers) into amounts of data storage as small as possible, both for the SPM and the main memory; (b) to compute these amounts of storage after mapping on both memory layers (allocation

solution) and be able to determine the memory location of each array element from the specification (assignment solution); (c) to use mapping functions simple enough in order to ensure an address generation hardware of a reasonable complexity; (d) to ascertain that any scalar signals (array elements) *simultaneously alive* are mapped to distinct storage locations.

Since the mapping models [8] and [16] play an important part in this section, they will be explained and illustrated below.

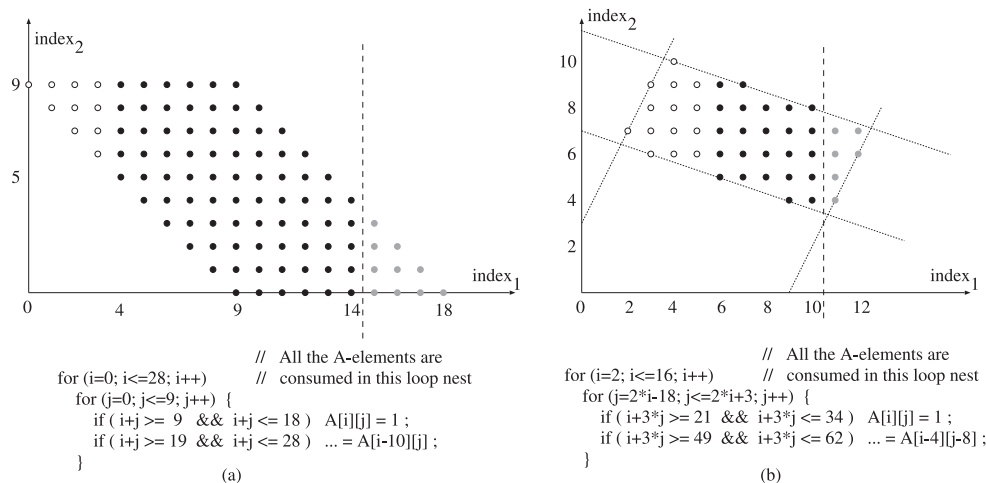
To reduce the size of a multidimensional array mapped to memory, the model [8] considers all the possible *canonical*<sup>†</sup> linearizations of the array; for any linearization, the largest distance at any time between two live elements is computed. This distance plus 1 is then the storage “window” required for the mapping of the array into the data memory. More formally,  $|W_A| = \min \max \{ dist(A_i, A_j) \} + 1$ , where  $|W_A|$  is the size of the storage window of a signal *A*, the minimum is taken over all the canonical linearizations, while the maximum is taken over all the pairs of *A*-elements  $(A_i, A_j)$  simultaneously alive.

This mapping model will be illustrated for the loop nest from Fig. 4(a). The graph above the code represents the array (index) space of signal *A*. The points represent the *A*-elements  $A[index_1][index_2]$  which are produced (and consumed as well) in the loop nest. The points to the left of the dashed line represent the elements produced till the end of the iteration ( $i = 14, j = 4$ ), the black points being the elements still alive (i.e., produced and still used as operands in the next iterations), while the circles representing *A*-elements already ‘dead’ (i.e., not needed as operands any more). The light grey points to the right of the dashed line are *A*-elements still unborn (to be produced in the next iterations).

If we consider the array linearization by column concatenation in the increasing order of the columns ( $(A[index_1][index_2], index_1=0,18), index_2=0,9$ ), two elements simultaneously alive, placed the farthest apart from each other, are  $A[9][0]$  and  $A[9][9]$ . The distance between them is  $9 \times 19 = 171$ . Now, if we consider the array linearization by row concatenation in the increasing order of the rows ( $(A[index_1][index_2], index_2=0,9), index_1=0,18$ ), the maximum distance between live elements is 99 (e.g., between  $A[4][5]$  and  $A[14][4]$ ). For all the canonical linearizations, the maximum distances have the values {99, 109, 171, 181}. The best canonical linearization for the array *A* is the row concatenation, a memory window  $W_A$  of  $99 + 1 = 100$  successive locations being large enough to store the array without mapping conflicts: it is sufficient that any access to  $A[index_1][index_2]$  be redirected to  $W_A[(10 * index_1 + index_2) \bmod 100]$ , since the distance between the points  $(index_1, index_2)$  and  $(0,0)$  is  $10 * index_1 + index_2$ .

In order to avoid the inconvenience of analyzing dif-

<sup>†</sup>For instance, a 2-D array can be typically linearized concatenating the rows or concatenating the columns. In addition, the elements in a given dimension can be mapped in the increasing or decreasing order of the respective index.



**Fig. 4** (a-b) Illustrative examples having a similar code structure. The mapping model by array linearization yields a better allocation solution for the former example, whereas the bounding window model behaves better for the latter one.

ferent linearization schemes, another possibility is to compute a maximal  $m$ -dimensional bounding window  $W_A = (w_1, \dots, w_m)$  large enough to encompass at any time the simultaneously alive ( $m$ -dimensional)  $A$ -elements. An access to the element  $A[\text{index}_1] \dots [\text{index}_m]$  can then be redirected without any conflict to the window location  $W_A[\text{index}_1 \bmod w_1] \dots [\text{index}_m \bmod w_m]$ . Each window element  $w_k$  is computed as the maximum difference in absolute value between the  $k$ -th indexes of any two  $A$ -elements  $(A_i, A_j)$  simultaneously alive, plus 1. More formally,  $w_k = \max \{ |x_k(A_i) - x_k(A_j)| \} + 1$ , for  $k = 1, \dots, m$ . This ensures that any two array elements simultaneously alive are mapped to distinct memory locations. The amount of data memory required for storing (after mapping) the array  $A$  is the volume of the bounding window  $W_A$ , that is,  $|W_A| = \prod_{k=1}^m w_k$ .

In the illustrative example shown in Fig. 4(a), the bounding window of the signal  $A$  is  $W_A = (11, 10)$ . It follows that the storage allocation for signal  $A$  is 100 locations if the linearization model is used, and  $w_1 \times w_2 = 110$  locations when the bounding window model is applied. However, in the example shown in Fig. 4(b), where the code has a similar structure, the bounding window model yields a better allocation result—30 storage locations, since the 2-D window of  $A$  is  $W_A = (5, 6)$ , whereas the linearization model yields 32 locations (the best canonical linearization being the row concatenation in the increasing order of rows).

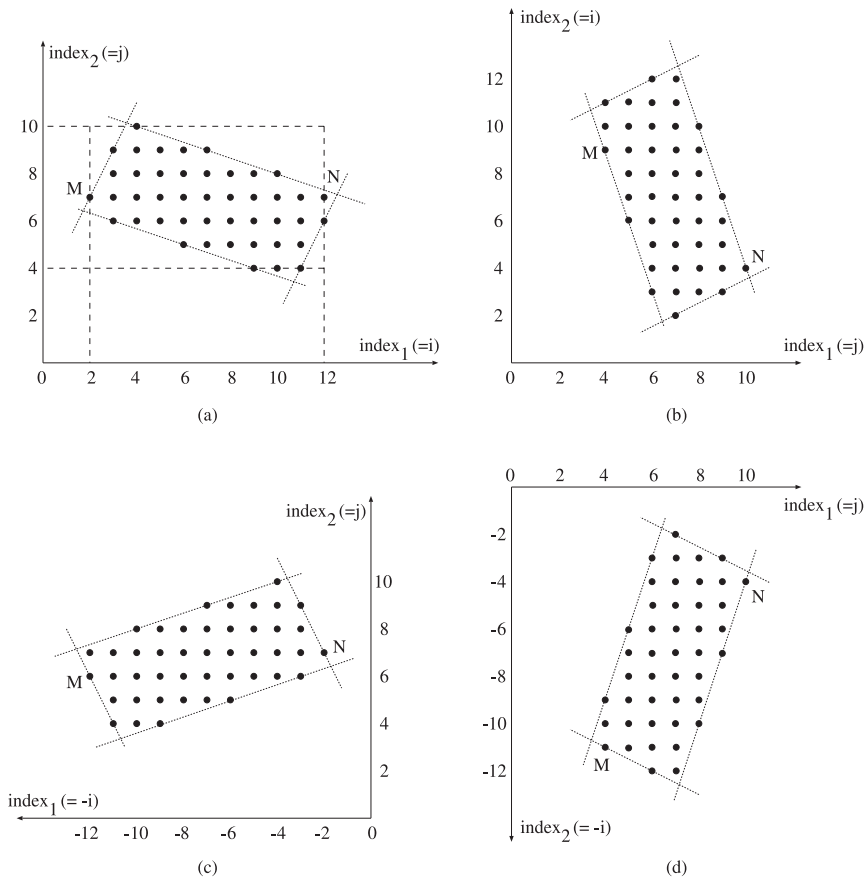
Our software system incorporates both mapping models, their implementation being based on the same polyhedral framework operating with lattices, used also in Sect. 2. This is advantageous both from the point of view of computational efficiency and relative to the amount of allocated data storage—since the mapping window for each signal is the smallest one of the two models. Moreover, this methodology can be applied independently to the memory layers, providing a complete storage allocation/assignment solution for distributed memory organizations.

Before explaining the global flow of the algorithm, let us examine the simple case of a code with only one array reference in it: take, for instance, the two nested loops from Fig. 4(b), but without the second conditional statement that consumes the  $A$ -elements. In the bounding window model,  $W_A = (11, 7)$  can be determined by computing the integer projections on the two axes of the lattice of  $A[i][j]$ :  $\{x = i, y = j \mid i \geq 2, -i \geq -12, -2i + j \geq -18, 2i - j \geq -3, i + 3j \geq 21, -i - 3j \geq -34\}$ , represented graphically by all the points inside the quadrilateral from Fig. 5(a). It can be observed from the figure that this can be reduced to the computation of the integer projections of a polytope (even when there are holes in the index space), which is a well-studied problem [18].

In the linearization model, the distance between two  $A$ -elements  $A_1[x_1][y_1]$  and  $A_2[x_2][y_2]$ , assuming row concatenation in the increasing order of the rows, is:  $\text{dist}(A_1, A_2) = (x_2 - x_1)\Delta y + (y_2 - y_1)$ , where  $\Delta y$  is the range of the second index (here, equal to 7) in the array space<sup>†</sup>. Then, the  $A$ -elements at a maximum distance have the minimum and, respectively, the maximum index vectors relative to the lexicographic order. These array  $A$ -elements are represented by the points  $M = A[2][7]$  and  $N = A[12][7]$  in Fig. 5(a), and  $\text{dist}(M, N) = (12 - 2) \times 7 + (0 - 0) = 70$ . Similarly, in the linearization by column concatenation, the array elements at the maximum distance from each other are still the elements with (lexicographically) minimum and maximum index vectors, *provided an interchange of the indexes is applied first*. These are the points  $M = A[4][9]$  and  $N = A[10][4]$  in Fig. 5(b), and the distance between them is  $(10 - 4) \times 11 + (4 - 9) = 61$ .

More general, the maximum distance between the

<sup>†</sup>To ensure that the distance is a nonnegative number, we shall assume that  $[x_2 \ y_2]^T > [x_1 \ y_1]^T$  relative to the lexicographic order. The vector  $\mathbf{y} = [y_1, \dots, y_m]^T$  is larger lexicographically than  $\mathbf{x} = [x_1, \dots, x_m]^T$  (written  $\mathbf{y} > \mathbf{x}$ ) if  $(y_1 > x_1)$ , or  $(y_1 = x_1 \text{ and } y_2 > x_2)$ , ..., or  $(y_1 = x_1, \dots, y_{m-1} = x_{m-1}, \text{ and } y_m > x_m)$ .



**Fig. 5** (a) The index space of the array reference  $A[i][j]$  from the code in Fig. 4(b) without the second conditional assignment. (b) The index space of the array reference  $A[j][i]$  (with indexes interchanged) assuming the same iterator space. (c-d) The index spaces of the array references  $A[-i][j]$  and  $A[j][-i]$ .

points of a live lattice in a canonical linearization is the distance between the (lexicographically) minimum and maximum index vectors, providing an index permutation is applied first. The distance between the array elements  $A_i[x_1^i][x_2^i] \dots [x_m^i]$  and  $A_j[x_1^j][x_2^j] \dots [x_m^j]$  is:  $dist(A_i, A_j) = (x_1^j - x_1^i)\Delta x_2 \dots \Delta x_m + (x_2^j - x_2^i)\Delta x_3 \dots \Delta x_m + \dots + (x_{m-1}^j - x_{m-1}^i)\Delta x_m + (x_m^j - x_m^i)$  where the index vector of  $A_j$  is lexicographically larger than of  $A_i$  ( $\Delta x_i$  the range of  $x_i$ ). Algorithms for the computation of both the integer projections of a lattice and the (lexicographically) minimum/maximum index vectors in a lattice were proposed in [13].

If in the linearization some dimension is traversed backwards, then a simple transformation reversing the index variation must be also applied as shown in Fig. 5(c) — row concatenation in the decreasing order of the columns, where  $dist(M, N) = 71$  — and in Fig. 5(d) — column concatenation in the decreasing order of the columns, where  $dist(M, N) = 73$ .

*Allocation algorithm:* For each memory layer (SPM and main memory) compute the mapping windows for every indexed signal having lattices assigned to that layer.

**Step 1** Compute underestimations of the window sizes on the current memory layer for each indexed signal, taking

into account only the live signals at the boundaries between the loop nests.

Let  $A$  be an  $m$ -dimensional signal and  $\mathcal{P}_A$  be the set of disjoint lattices of  $A$  assigned to the current memory layer by the algorithm described in Sect. 2. The notations are explained in the pseudo-code.

```

for ( each canonical linearization  $C$  of signal  $A$  ) {
  for ( each disjoint lattice  $L \in \mathcal{P}_A$  )
    compute  $x^{min}(L)$  and  $x^{max}(L)$ , the lexicographically
      minimum and maximum index vectors of  $L$  relative to  $C$ ;
  for ( each boundary  $n$  between the loop nests  $n$  and  $n + 1$  ) {
    let  $\mathcal{P}_A(n) \subseteq \mathcal{P}_A$  be the set of lattices alive at the boundary  $n$ ;
    // (lattices produced/consumed before/after the boundary)
     $X_n^{min} = \min_{L \in \mathcal{P}_A(n)} \{x^{min}(L)\}$ ;
     $X_n^{max} = \max_{L \in \mathcal{P}_A(n)} \{x^{max}(L)\}$ ;
     $|W_C(n)| = dist_C(X_n^{min}, X_n^{max}) + 1$ ;
    // Distance computed relative the canonical linearization  $C$ 
  }
   $|W_C| = \max_n \{ |W_C(n)| \}$ ;
  // window size over all boundaries for the linearization  $C$ 
}
 $|W_1| = \min_C \{ |W_C| \}$ ; // window size in the linearization model
for ( each disjoint lattice  $L \in \mathcal{P}_A$  )
  for ( each dimension  $k$  of signal  $A$  )
    compute  $x_k^{min}(L)$  and  $x_k^{max}(L)$ , the extremes of the
      integer projection of  $L$  on the  $k$ -th axis;

```

```

for ( each boundary  $n$  between the loop nests  $n$  and  $n + 1$  ) {
  let  $\mathcal{P}_A(n) \subseteq \mathcal{P}_A$  be the set of lattices alive at the boundary  $n$ ;
  for ( each dimension  $k$  of signal  $A$  ) {
     $X_k^{min} = \min_{L \in \mathcal{P}_A(n)} \{x_k^{min}(L)\};$ 
     $X_k^{max} = \max_{L \in \mathcal{P}_A(n)} \{x_k^{max}(L)\};$ 
     $w_k(n) = X_k^{max} - X_k^{min} + 1;$ 
    // The  $k$ -th side of  $A$ 's bounding window at boundary  $n$  }
  }
for ( each dimension  $k$  of signal  $A$  )
   $w_k = \max_n \{w_k(n)\};$  // the  $k$ -th side of  $A$ 's bounding window
 $|W_2| = \prod_{k=1}^n w_k;$  // the bounding window size

```

*Step 1* finds the exact values of the window sizes for both mapping models only when every loop nest either produces or consumes (but not both!) the signal's elements. Otherwise, when in a certain loop nest elements of the signal are both produced *and* consumed (see the illustrative example from Fig. 4(a)), then the window sizes obtained at the end of *Step 1* may be only underestimates since an increase of the storage requirement can happen *inside* the loop nest. Then, an additional step is required to find the exact values of the window sizes in both mapping models.

**Step 2** Update the mapping windows for each indexed signal in every loop nest producing and consuming elements of the signal.

The guiding idea is that local or global maxima of the bounding window size  $|W_2|$  are reached immediately before the consumption of an  $A$ -element, which may entail a shrinkage of some side of the bounding window encompassing the live elements. Similarly, the local or global maxima of  $|W_C|$  are reached immediately before the consumption of an  $A$ -element, which may entail a decrease of the maximum distance in the linearization  $C$  between live elements. Consequently, for each  $A$ -element consumed in a loop nest which also produces  $A$ -elements, we construct the disjoint lattices partially produced and those partially consumed until the iteration when the  $A$ -element is consumed. Afterwards, we do a similar computation as in *Step 1* which may result in increased values for  $|W_1|$  and/or  $|W_2|$ .

Finally, the amount of data memory allocated for signal  $A$  on the current memory layer is the smallest data storage provided by the bounding window and the linearization models:  $|W_A| = \min \{ |W_1|, |W_2| \}$ . In principle, the overall amount of data memory after mapping is  $\sum_A |W_A|$ —the sum of the mapping window sizes of all the signals having lattices assigned to the current memory layer. In addition, a post-processing step attempts to further enhance the allocation solution: our polyhedral framework allows to efficiently check whether two multidimensional signals have disjoint lifetimes, in which case the signals can share the largest of the two windows. More general, an incompatibility graph [9] is used to optimize the memory sharing among all the signals at the level of whole code.

#### 4. Experimental Results

The polyhedral framework for the memory management of multidimensional signal processing applications has been

implemented in C++, incorporating the algorithms described in this paper. The behavioral specifications of the applications are expressed in a subset of the C language, illustrated in the examples used in the paper.

Tables 1 and 2 summarize the results of our experiments, carried out on a PC with an Intel Core 2 Duo 1.8 GHz processor and 512 MB RAM running Ubuntu 6.06. The benchmarks used (columns 1) are algebraic kernels (like Durbin's algorithm for solving Toeplitz systems) and algorithms used in multimedia applications (like, for instance, an MPEG4 motion estimation algorithm).

Table 1 focuses on our signal mapping methodology (Sect. 3). Columns 2 and 3 display the numbers of array references and array elements (scalar signals) in the specification code. The next column shows the memory allocation results—the overall data storage after mapping. The 5th column displays two metrics of quality for the memory allocation solutions: (a) the sum of the *minimum* array windows (that is, the optimum memory sharing between elements of same arrays), and (b) the minimum storage requirement for the execution of the application code (that is, the optimum memory sharing between all the scalar signals or array elements in the code). The results from this column are obtained with the algorithm [2], also part of this framework.

The data memory size after mapping is typically lower-bounded by the first value in the 5th column, since the memory window of an array cannot be smaller than the minimum storage requirement of the array. The exception which occurs for the motion detection kernel is due to the existence of a scalar signal whose lifetime is disjoint from the lifetime of another array in the code. As explained at the end of Sect. 3, signals having disjoint lifetimes can share the same window. The importance of the second value in the 5th column is mainly theoretical, serving for the evaluation of the allocation results (column 4). A minimum data storage could be obtained in principle, but the price to be paid is a very complex address generation hardware. By means of the signal-to-memory mapping model, an excess of storage in memory allocation is traded-off against a less complex address generation hardware (practically all the mapping models using modulo computations [14]). This framework has the unique feature of yielding a *measure* of this compromise—the amount of additional storage over the absolute lower bound.

The last group of columns displays the CPU times when running the mapping algorithms [8], [16], and the current one in Sect. 3, respectively. Note that our algorithm takes into consideration *both* models, selecting for each array the smaller memory window. When the nested loops either produce or consume (but not both) elements of same arrays (as in the case of the Durbin benchmark), our algorithm is significantly faster since only *Step 1* need to be executed. Even if this is not the case, the current algorithm is regularly faster due to our polyhedral framework, efficiently operating with polytopes and lattices.

Table 2 displays in columns 2 and 3 the total memory accesses and the dynamic energy consumption in the case of

**Table 1** Experimental results on signal mapping. Column 4 displays the data storage requirements obtained after executing the mapping algorithm explained in Sect. 3; the last column shows the corresponding running times.

Application parameters	Num. array references	Num. array elements	Memory size after mapping	$\sum_A \min\{ W_A \} / \min\{Mem\_Size\}$	CPU [sec]		
					[8]	[16]	
Motion detection	11						
M=N= 32, m=n=4		72,543	2,740	2,741 / 2,740	10	7	3
M=N= 64, m=n=4		318,367	9,524	9,525 / 9,524	31	22	13
M=N=120, m=n=8		3,749,063	33,284	33,285 / 33,284	150	80	56
Motion estimation	17						
M=32, N=16		265,633	4,513	4,513 / 2,465	50	30	14
M=48, N=32		2,368,865	11,873	11,873 / 7,265	162	75	57
M=64, N=32		4,208,449	18,241	18,241 / 10,049	218	101	77
Durbin algorithm	27						
N = 50		2,749	201	176 / 124	8	3	1
N = 100		10,499	401	351 / 249	14	5	2
SVD updating	86						
n = 20		25,908	2,464	1,664 / 1,430	50	31	20
n = 30		85,213	5,494	3,694 / 3,195	76	45	29
n = 40		199,418	9,724	6,524 / 5,660	105	62	42
Voice coder	251	33,751	14,498	12,965 / 11,890	74	55	41

**Table 2** Experimental results on signal assignment to memory layers.

Application parameters	#Memory accesses	Dyn. energy 1-layer [ $\mu J$ ]	SPM size	Dyn. energy saved [5]	Dyn. energy saved [10]	Dyn. energy saved	CPU [sec]
Motion detection	136,242	486	841	30.2%	44.5%	49.2%	4
M=N=32, m=n=4							
Motion estimation	864,900	3,088	1,416	38.7%	40.7%	50.7%	23
M=32, N=16							
Durbin algorithm	1,004,993	3,588	764	55.2%	58.5%	73.2%	28
N = 500							
SVD updating	6,227,124	22,231	12,672	35.9%	38.4%	46.0%	37
n = 100							
Vocoder	200,000	714	3,879	30.8%	32.5%	39.5%	8

a single (off-chip) memory layer. Using the CACTI power model, we chose as parameters a technology of 65 nm, 4 memory banks, and a line size of 16 bytes. Assuming a two-layer memory organization, columns 4-7 display the SPM size and the savings of dynamic energy<sup>†</sup> applying, respectively, a previous model steered by the total number of accesses for whole arrays [5], another previous model steered by the most accessed array rows/columns [10], and the current assignment model (Sect. 2), versus the single-layer memory scenario (column 3). Finally, column 8 shows the CPU times.

The SPM sizes (column 4) are computed as follows: the lattices of all the arrays in the application are ordered decreasingly based on the average number of accesses per array element; in this order (such that the most accessed lattices come first), the lattices are gradually assigned to the SPM, increasing the SPM size with discrete amounts; for each new SPM size, the CACTI model computes the energy per access; afterwards, we determine the reduction of dynamic energy versus the scenario when all the signals are stored off chip. We choose the SPM size maximizing the ratio between the dynamic energy reduction and the size.

Our results are regularly better than the other models since, building the map of memory accesses for each array (see Fig. 3), our framework identifies with accuracy those

parts of arrays intensely accessed, whose assignment to the SPM layer yields the highest benefit in terms of dynamic energy consumption. For instance, the energy consumptions for the motion estimation benchmark were, respectively, 1894, 1832, and 1522  $\mu J$ ; the saved energies relative to the energy in column 3 are displayed as percentages in columns 5-7.

Besides the memory allocation solution, the signal mapping algorithm computes the mapping functions for all the arrays, so we can determine the exact locations for any array element in the specification. This provides the necessary information for the automated design of the address generation unit, which is one of our future development directions.

## 5. Conclusions

This paper has presented an integrated CAD methodology for power-aware memory allocation, targeting embedded data-intensive signal processing applications. The memory management tasks — the signal assignment to the memory layers and their mapping to the physical memories — are efficiently addressed within a common polyhedral framework.

<sup>†</sup>The use of CACTI to estimate the dynamic energy consumption of SPMs is explained in an appendix of [4].

## References

- [1] F. Angiolini, L. Benini, and A. Caprara, "An efficient profile-based algorithm for scratchpad memory partitioning," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol.24, no.11, pp.1660–1676, Nov. 2005.
- [2] F. Balasa, H. Zhu, and I.I. Luican, "Computation of storage requirements for multi-dimensional signal processing applications," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol.15, no.4, pp.447–460, April 2007.
- [3] F. Balasa, I.I. Luican, H. Zhu, and D.V. Nasui, "System-level exploration tool for energy-aware memory management in the design of multidimensional signal processing systems," *Proc. Asia & South-Pacific Design Automation Conf.*, pp.443–448, Yokohama, Jan. 2009.
- [4] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Comparison of cache- and scratch-pad based memory systems with respect to performance, area and energy consumption," *Technical Report #762*, University of Dortmund, Sept. 2001.
- [5] E. Brockmeyer, M. Miranda, H. Corporaal, and F. Catthoor, "Layer assignment techniques for low energy in multi-layered memory organisations," *Proc. ACM/IEEE Design Aut. & Test in Europe*, pp.1070–1075, Munich, Germany, March 2003.
- [6] E. Chan and S. Panchanathan, "Motion estimation architecture for video compression," *IEEE Trans. Consum. Electron.*, vol.39, pp.292–297, Aug. 1993.
- [7] S. Coumeri and D.E. Thomas, "Memory modeling for system synthesis," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol.8, no.3, pp.327–334, June 2000.
- [8] E. De Greef, F. Catthoor, and H. De Man, "Memory size reduction through storage order optimization for embedded parallel multimedia applications," *Parallel Comput.*, vol.23, no.12, pp.1811–1837, Elsevier, 1997.
- [9] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.
- [10] Q. Hu, A. Vandecapelle, M. Palkovic, P.G. Kjeldsberg, E. Brockmeyer, and F. Catthoor, "Hierarchical memory size estimation for loop fusion and loop shifting in data-dominated applications," *Proc. Asia & South-Pacific Design Automation Conf.*, pp.606–611, Yokohama, Jan. 2006.
- [11] I. Issenin, E. Brockmeyer, M. Miranda, and N. Dutt, "Data reuse analysis technique for software-controlled memory hierarchies," *Proc. Design Automation and Test in Europe*, 2004.
- [12] M. Kandemir and A. Choudhary, "Compiler-directed scratch-pad memory hierarchy design and management," *Proc. 39th ACM/IEEE Design Automation Conf.*, pp.690–695, Las Vegas, NV, June 2002.
- [13] I.I. Luican, H. Zhu, and F. Balasa, "Signal-to-memory mapping analysis for multimedia signal processing," *Proc. Asia & South-Pacific Design Automation Conf.*, pp.486–491, Yokohama, Jan. 2007.
- [14] G. Talavera, M. Jayapala, J. Carrabina, and F. Catthoor, "Address generation optimization for embedded high-performance processors: A survey," *J. Signal Processing Systems*, vol.53, no.3, pp.271–284, Dec. 2008.
- [15] L. Thiele, "Compiler techniques for massive parallel architectures," in *State-of-the-art in Computer Science*, ed. P. Dewilde, Kluwer Acad. Publ., 1992.
- [16] R. Tronçon, M. Bruynooghe, G. Janssens, and F. Catthoor, "Storage size reduction by in-place mapping of arrays," in *Verification, Model Checking and Abstract Interpretation, Lecture Notes Computer Sc.*, ed. A. Coresi, pp.167–181, Springer, New York, 2002.
- [17] S. Udayakumaran and R. Barua, "Compiler-decided dynamic memory allocation for scratch-pad based embedded systems," *Proc. Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pp.276–286, New York, NY, Oct. 2003.
- [18] S. Verdoolaege, K. Beyls, M. Bruynooghe, and F. Catthoor, "Experiences with enumeration of integer projections of parametric polytopes," *Compiler Construction: 14th Int. Conf.*, ed. R. Bodik, vol.3443, pp.91–105, Springer, New York, 2005.
- [19] S. Wilton and N. Jouppi, "CACTI: An enhanced access and cycle time model," *IEEE J. Solid-State Circuits*, vol.31, no.5, pp.677–688, May 1996.

**Florin Balasa** received the M.S. and Ph.D. degrees in computer science from the Polytechnical University of Bucharest, Bucharest, Romania, in 1981 and 1994, respectively, the M.S. degree in mathematics from the University of Bucharest, Bucharest, Romania, in 1990, and the Ph.D. degree in electrical engineering from the Katholieke Universiteit Leuven, Leuven, Belgium, in 1995. He is currently an associate professor of computer science at Southern Utah University, Cedar City, Utah, U.S.A. From 1990 to 1995, he was with the Interuniversity Microelectronics Center (IMEC), Leuven, Belgium. From 1995 to 1999, he was a senior design automation engineer at the Advanced Technology Division of Conexant Systems (formerly Rockwell Semiconductor Systems), Newport Beach, California, U.S.A. From 2000 to 2007, he was with the Dept. of Computer Science at the University of Illinois at Chicago, Chicago, U.S.A. His research interests span different areas of VLSI CAD. Dr. Balasa was a recipient of the U.S. National Science Foundation CAREER Award.

**Ilie I. Luican** received the B.S. and M.S. degrees in computer science from the Polytechnical University of Bucharest (PUB), Bucharest, Romania, in 2002 and 2003, respectively. From 2003 to 2004, he was with the Automatic Control Department of PUB. He is currently pursuing the Ph.D. degree in the Department of Computer Science, University of Illinois at Chicago, Chicago. His research interests are mainly focused on high-level synthesis and memory management for real-time multi-dimensional signal processing.

**Hongwei Zhu** received the B.S. degree in electrical engineering from Xi'an Jiaotong University, P.R. China, in 1996, and his M.S. degree in electrical and electronics engineering from Nanyang Technological University, Singapore, in 2001. He received his Ph.D. degree in computer science from University of Illinois at Chicago (UIC), U.S.A., in 2007. Before joining UIC, he has worked as a senior R&D engineer at JVC Asia Pte. Ltd., Singapore. Currently, he is a software engineer in the Design Automation Group of the Physical IP Dept. at ARM, Sunnyvale, California, U.S.A. His research interests are mainly focused on memory management for real-time multidimensional signal processing and combinatorial optimization in CAD VLSI.

**Doru V. Nasui** received the M.S. degree in computer science from the Polytechnical University of Bucharest, Bucharest, Romania, in 1981. He is the co-founder of American International Radio, Inc. — one of the Motorola's largest worldwide distributors — where he is currently President/CEO since 1991. From 1987 to 1990, he was a senior software engineer at Motorola, Schaumburg, Illinois, U.S.A. His research interests are focused on the design and manufacturing of mobile communication systems and CAD techniques for telecom VLSI systems.