

# Signal Assignment Model for the Memory Management of Multidimensional Signal Processing Applications

Florin Balasa · Ilie I. Luican · Hongwei Zhu · Doru V. Nasui

Received: 13 October 2008 / Revised: 27 May 2009 / Accepted: 27 May 2009 / Published online: 20 June 2009  
© 2009 Springer Science + Business Media, LLC. Manufactured in The United States

**Abstract** Many signal processing systems, particularly in the multimedia and telecom domains, are synthesized to execute data-dominated applications. Their behavior is described in a high-level programming language, where the code is typically organized in sequences of loop nests and the main data structures are multidimensional arrays. Since data transfer and storage have a significant impact on both the system performance and the major cost parameters—power consumption and chip area, the designer must spend a significant effort during the system development process on the exploration of the memory subsystem in order to achieve a cost-optimized design. This paper presents a memory allocation methodology for multidimensional signal processing applications, focusing on the problem of efficiently mapping the multidimensional signals from the algorithmic specification into the physical memory. In a first phase, two previous mapping models are implemented within a common theoretical framework, which is advantageous from

both the point of view of computational efficiency and the amount of allocated data storage. Different from all the previous mapping models that aim to optimize the memory sharing between the elements of a same array (creating separate windows in the physical memory for distinct arrays), this proposed mapping model exploits—in a second phase—the possibility of memory sharing between the elements of different arrays. As a consequence, this signal assignment approach yields significant savings in the amount of data storage resulted after mapping.

**Keywords** Memory management · Memory allocation · Signal-to-memory assignment · Multidimensional signal processing · Storage requirement

## 1 Introduction

Many signal processing systems are synthesized to execute data-dominated applications in various domains including video and image processing, artificial vision and medical imaging, real-time 3D rendering, advanced audio and speech coding. The behavior of many of these systems is described in a high-level programming language, where the code is typically organized in sequences of loop nests and the main data structures are multidimensional arrays whose references have as indices linear functions of the loop iterators.

Since data transfer and storage have a significant impact on both the system performance and the major cost parameters—power consumption and chip area, the designer must spend a significant effort during the system development process on the exploration of the

---

F. Balasa (✉)  
Dept. of Computer Science, Southern Utah University,  
Cedar City, UT, USA  
e-mail: balasa@suu.edu

I. I. Luican  
Dept. of Computer Science, University of Illinois at Chicago,  
Chicago, IL, USA

H. Zhu  
ARM, Inc., Sunnyvale, CA, USA

D. V. Nasui  
American Int. Radio, Inc., Rolling Meadows, IL, USA

possible memory organizations in order to achieve a cost-optimized design [2, 3]. This is why the problem of memory allocation is central to any computer-aided design tool focusing on memory management.

Since these (typically large) arrays from the algorithmic specification must be stored during the execution of the application code, the memory allocation solution is significantly influenced on how these data structures from the specification are mapped into the physical memory. The goals of the *mapping* operation are the following:

- (a) to assign the arrays from the behavioral specification into an amount of data storage as small as possible; moreover, to be able to compute this amount of storage (after mapping) and be able to determine the memory location of any array element from the specification;
- (b) to use mapping functions simple enough in order to ensure an address generation hardware of a reasonable complexity;
- (c) to ascertain that any distinct scalar signals (array elements) *simultaneously alive* are mapped to distinct storage locations. The lifetime of a scalar signal is the time interval between the clock cycles when the scalar is *produced* or written, and when is read for the last time, i.e. *consumed*, during the code execution. Two scalars are simultaneously alive if their lifetimes do overlap. Obviously, in such a case, they must occupy different memory locations; otherwise, they can share the same location.

Several mapping models have been proposed in the past, trading off between the first two goals (that is, accepting a certain excess of storage to ensure a less complex address generation hardware) while ascertaining that the third goal be strictly satisfied.

To reduce the size of a multidimensional array mapped to memory, De Greef et al. consider all the possible *canonical* linearizations of the array; for any linearization, they compute the largest distance at any time between two live elements in the linearized array [4]. This distance plus 1 is then the storage “window” required for the mapping of the array into the data memory. More formally,  $|W_A| = \min \max \{ \text{dist}(A_i, A_j) \} + 1$ , 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.

Note that for an  $m$ -dimensional array there are  $m!$  orderings of the indices. For instance, a 2-dimensional (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. De Greef et al. consider in their model all these  $2^m \cdot m!$  possible linearizations, called *canonical*. (For  $m \geq 6$ , an heuristic is proposed to limit the search.)

In order to avoid the inconvenience of analyzing different linearization schemes, Tronçon et al. proposed [5] to reduce the size of an  $m$ -dimensional array  $A$  mapped to the data memory, computing a window  $W_A = (w_1, \dots, w_m)$ , whose elements can be used as operands in modulo operations that redirect all accesses to  $A$ . An access to the element  $A[\text{index}_1] \dots [\text{index}_m]$  is redirected to  $W_A[\text{index}_1 \bmod w_1] \dots [\text{index}_m \bmod w_m]$ ; in its turn, this bounding window is mapped (relative to a base address) into the physical memory by a typical canonical linearization, like row or column concatenation for 2-D arrays. Signal  $A$ 's window is denoted as  $W_A$  in any of the two models [4] and [5] (in spite of being 1-dimensional in the case of the former model and  $m$ -dimensional for the latter) since the meaning will be clear enough from the context. Each window element  $w_k$  is computed as the maximum difference (in absolute value) between the  $k$ -th indices 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 bounding window is identical to the storage window from model [4] for 1-D arrays. The amount of data memory required for storing (after mapping) the array  $A$  is the volume of the window  $W_A$ , that is,  $|W_A| = \prod_{k=1}^m w_k$ .

Lefebvre and Feautrier, addressing parallelization of static control programs, developed in [6] an approach based on modular mapping, as well. They first compute the lexicographically maximal “time delay” between the write and the last read operations, which is a super-approximation of the distance between conflicting index vectors (i.e., whose corresponding array elements are simultaneously alive). Then, the modulo operands are computed successively as follows: the modulo operand  $b_1$ , applied on the first array index, is set to 1 plus the maximal difference between the first indices over the conflicting index vectors; the modulo operand  $b_2$  of the second index is set to 1 plus the maximal difference between the second indices over the conflicting index vector, when the first indices are equal; and so on.

Quilleré and Rajopadhye perform first an affine mapping into a linear space of *smallest* dimension (what they call a “projection”) before modulo operations are applied to the array indices [7].

Darte et al. proposed a very refined mathematical framework, establishing a correspondence between valid linear storage allocations and integer lattices [8] called *strictly admissible* relative to the set of differences of the conflicting indices [9]. Heuristic techniques for building strictly admissible integer lattices, hence building valid storage allocations, are proposed.

All these signal-to-memory assignment approaches treat *separately* the arrays from the algorithmic specification, computing windows in the physical memory for each individual array. They exploit the possibility of memory sharing only between the elements of a same array. However, since the arrays are handled separately, the possibility of memory sharing between elements of different arrays is inherently ignored. This can lead to an excessive data storage, as Section 2.2 will illustrate. The implementation of some mapping techniques may detect simpler cases when, for instance, two entire arrays have disjoint lifetimes and, consequently, the two arrays may share the same (largest) window in the physical memory. Here, we are referring to the more general and typical situation when elements of different arrays are simultaneously alive.

Interestingly, the possibility of memory sharing between elements of different arrays with disjoint lifetimes was observed long time ago (e.g., [10]) and it has been taken into account by several approaches for memory size evaluation (e.g., [11, 12]). The memory sharing is sometimes called “inter-array in-place mapping” when the elements belong to different arrays, and “intra-array in-place mapping” when the elements belong to the same array [3, 11]. It must be emphasized that these terms can create confusion since they do not necessary refer to signal-to-memory mapping techniques, where an explicit correspondence between the array elements and their addresses in the physical memory is indicated; they rather refer to the possible reuse of data storage by array elements having disjoint lifetimes.

A novel signal-to-memory assignment approach exploiting the possibility of memory sharing between elements of different arrays was briefly presented in [1]. This paper explains the advantages of the novel assignment model, describes the computation methodology, and illustrates the algorithm flow. To the best of our knowledge, it is the only mapping approach with the capability of inter-array memory sharing even when the arrays do not have disjoint lifetimes, thus producing better savings of data storage than all the previous techniques.

The rest of the paper is organized as follows. Section 2 thoroughly discusses a few illustrative examples, analyzing the results of several mapping

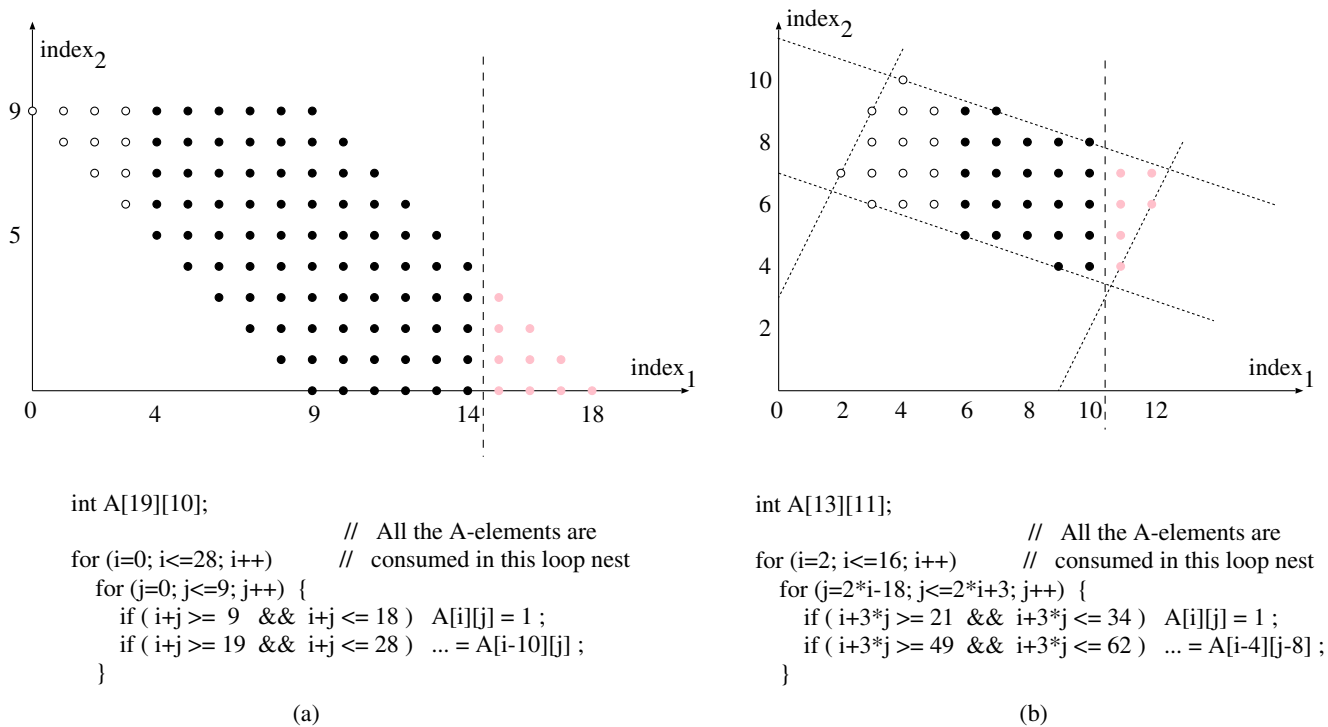
techniques and explaining the motivation of this research. Section 3 presents the basic ideas of this novel signal assignment algorithm used in a memory allocation framework. Section 4 addresses implementation aspects and discusses the experimental results. Section 5 summarizes the main conclusions of this work.

## 2 Discussion on Illustrative Examples

The first part of this discussion will focus on the previous mapping models [4] and [5], since they are relevant for the rest of the paper. The second part will illustrate the advantage of mapping with inter-array memory sharing.

### 2.1 Previous Mapping Models Exemplified

The mapping model [4] will be first illustrated for the loop nest from Fig. 1a. The graph above the code represents the array (index) space of signal  $A$ . The points represent the  $A$ -elements  $A[\text{index}_1][\text{index}_2]$  which are produced (and also consumed) in the loop nest. In the figure, the  $A$ -elements are produced column by column from left to right. (Note that a column of points in the figure is actually a row of the 2-D signal.) Inside each column, the  $A$ -elements are produced bottom-up; they are consumed in the same order, but with a “delay” of 10 columns. 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 ( $(A[\text{index}_1][\text{index}_2], \text{index}_1 = 0, 18), \text{index}_2 = 0, 9$ ), the two elements simultaneously alive and placed the farthest apart from each other are  $A[9][0]$  and  $A[9][9]$ . The distance between their positions in the linearization is  $9 \times 19 = 171$ . If the columns are concatenated decreasingly ( $(A[\text{index}_1][\text{index}_2], \text{index}_1 = 0, 18), \text{index}_2 = 9, 0$ ), there are 9 pairs of elements simultaneously alive, mapped at a maximum distance. Two such elements are, for instance,  $A[14][0]$ —produced in the iteration ( $i = 14, j = 0$ ) and consumed in ( $i = 24, j = 0$ ), and  $A[4][9]$ —produced in the iteration ( $i = 4, j = 9$ ) and consumed in ( $i = 14, j = 9$ ), the distance between them in the linearization being  $9 \times 19 + 10 = 181$ .



**Figure 1** **a** Illustrative example where the memory allocation is better in storage point of view when the mapping model [4] is used. The graph shows the array (index) space of the 2-D signal  $A$  at the end of the iteration ( $i = 14$ ,  $j = 4$ ). **b** Illustrative example

having a similar code structure, whose allocation solution is better when the mapping model [5] is used. The graph shows the index space of the 2-D signal  $A$  at the end of the iteration ( $i = 10$ ,  $j = 8$ ).

Now, if we consider the array linearization by row concatenation in the increasing order of the rows ( $(A[\text{index}_1][\text{index}_2]$ ,  $\text{index}_2 = 0, 9)$ ,  $\text{index}_1 = 0, 18)$ , there are 9 pairs of elements simultaneously alive, maximally distanced from each other. Two such elements are, for instance,  $A[4][5]$ —produced in the iteration ( $i = 4$ ,  $j = 5$ ) and consumed in ( $i = 14$ ,  $j = 5$ ), and  $A[14][4]$ —produced in the iteration ( $i = 14$ ,  $j = 4$ ) and consumed in ( $i = 24$ ,  $j = 4$ ). The distance between them in the linearization is  $10 \times 10 - 1 = 99$ . Finally, if the rows are concatenated decreasingly ( $(A[\text{index}_1][\text{index}_2]$ ,  $\text{index}_2 = 0, 9)$ ,  $\text{index}_1 = 18, 0)$ , there are 9 pairs of elements simultaneously alive, as well (e.g.,  $A[14][0]$  and  $A[4][9]$ ), placed at the maximum distance  $10 \times 10 + 9 = 109$ .

For the other four linearizations, the maximum distances obtained have the same values (i.e., 171, 181, 99, 109), since the array elements are stored in reverse order relative to one of the four linearizations analyzed above.

According to the mapping model [4], the best linearization (among those considered above) for the array  $A$  is the concatenation row by row, increasingly. A memory window  $W_A$  of  $99 + 1 = 100$  successive locations (relative to a certain base address) is sufficient to

store the array without mapping conflicts: it is sufficient that any access to  $A[\text{index}_1][\text{index}_2]$  be redirected to  $W_A[(10 * \text{index}_1 + \text{index}_2) \bmod 100]$ .

The assignment model proposed by Tronçon et al. [5] circumvents the need of analyzing different linearization schemes by computing a maximal bounding window, having the same dimension as the signal, large enough to cover in any moment of the computation all the array elements simultaneously alive. In the illustrative example shown in Fig. 1a, the window corresponding to the signal  $A$  is  $W_A = (11, 10)$ . Indeed, the maximum horizontal and vertical distances between black points is  $d_1 = 10$  and  $d_2 = 9$ , respectively. A 2-D window whose elements are  $w_1 = d_1 + 1$  and  $w_2 = d_2 + 1$  is large enough to store the  $A$ -elements without any mapping conflict if any access to  $A[\text{index}_1][\text{index}_2]$  is redirected to  $W_A[\text{index}_1 \bmod 11][\text{index}_2 \bmod 10]$ .

From the above discussion, it follows that the storage allocation for signal  $A$  is 100 locations if the mapping model [4] is used, whereas it is  $w_1 \times w_2 = 110$  locations if the model [5] is applied. The better allocation result of the mapping model [4] for the illustrative code in Fig. 1a should not be construed as proof that the strategy based on canonical linearizations is generally advantageous in storage point of view. In the

illustrative example shown in Fig. 1b, where the code has a similar structure, the mapping model [5] yields a better allocation result.

Indeed, the 2-D window corresponding to the array  $A$  is  $W_A = (5, 6)$ : it can be easily verified that the maximum distance between the first (respectively, second) indices of two alive elements cannot exceed 4 (respectively, 5). For instance, the simultaneously alive elements  $A[6][9]$  and  $A[10][4]$  (see the black points in Fig. 1b) have both indices the farthest apart from each other. Therefore, a memory access to the element  $A[\text{index}_1][\text{index}_2]$  can be safely redirected to  $W_A[\text{index}_1 \bmod 5][\text{index}_2 \bmod 6]$ , the data storage requirement after mapping being 30 locations.

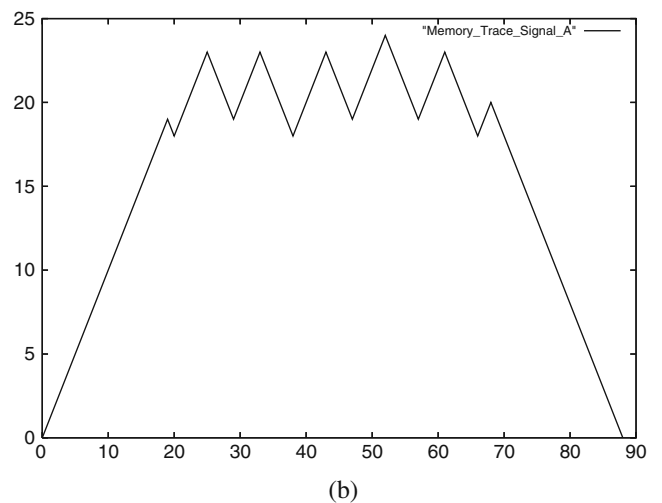
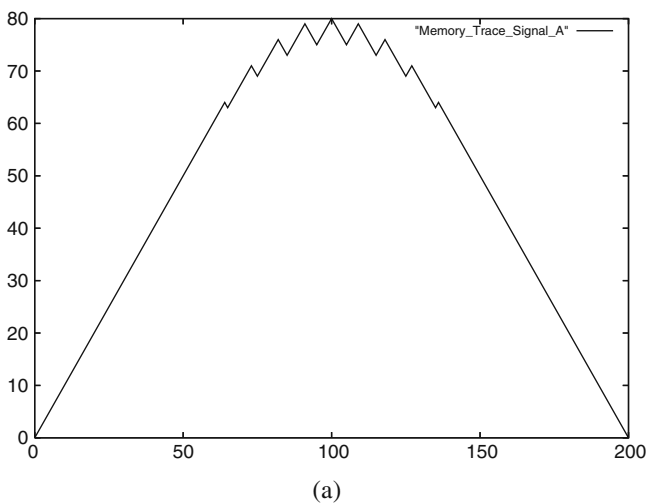
Taking into account that the ranges of the two indexes are  $12 - 2 + 1 = 11$  and, respectively,  $10 - 4 + 1 = 7$ , the maximum distance in the canonical linearization by column concatenation, in the increasing order of the columns, is 53 (e.g., between the elements  $A[9][4]$  and  $A[7][9]$ ), whereas in the decreasing order of the columns, it is 59. In the canonical linearization by row concatenation, the maximum distances are 31 (e.g., between the elements  $A[6][5]$  and  $A[10][8]$ ) and 33, respectively. It follows that the memory requirement using the mapping model [4] is 32 locations, slightly worse than the allocation result using the  $m$ -dimensional window model [5]. For many benchmarks, the latter model finds smaller window sizes: this happens especially when the array space contains *holes* and, also, when the size of the array can be reduced in every dimension, since, in such cases, any linearization will contain a number of unused array elements. Anyway,

the two illustrative examples in Fig. 1 show that, given an algorithmic specification, one cannot decide in advance which of the assignment techniques [4] and [5] yields a better solution.

It should be noticed that, actually, only 80 locations are really needed for the illustrative code in Fig. 1a as no more than 80  $A$ -elements can be simultaneously alive: see again Fig. 1a, where the 80 black dots represent the live elements at the end of the iteration ( $i = 14, j = 4$ ). Similarly, the minimum data storage required for the execution of the code in Fig. 1b is 24 locations: see the graph in Fig. 1b, where the 24 black dots represent the live elements at the end of the iteration ( $i = 10, j = 8$ ). Figure 2 shows the variation of the storage requirement for the signal  $A$  during the execution of both codes. These memory traces are generated using the tool described in [12], allowing to evaluate the minimum data storage. A *minimum* array window is not only difficult to compute but, typically, difficult to use in practical allocation problems as, in most of the cases, it requires a significantly more complex addressing hardware. A signal-to-memory mapping model like the ones described above trades-off an excess of storage for a less complex address generation hardware.

### 2.2 Exploiting Inter-Array Memory Sharing

Let us analyze now the illustrative example in Fig. 3a, where the  $A$ -elements produced in the first loop nest are consumed in the second loop nest, and the



**Figure 2 a** Memory trace (variation of the storage requirement) for the signal  $A$  from the code example in Fig. 1a. The abscissae are the number of assignment operations and the ordinates are

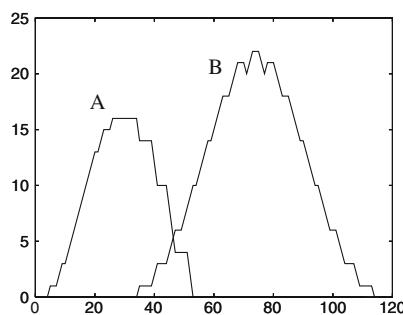
the number of occupied memory locations. **b** Memory trace for the signal  $A$  from the code example in Fig. 1b.

```
int A[7][4], B[11][6];
```

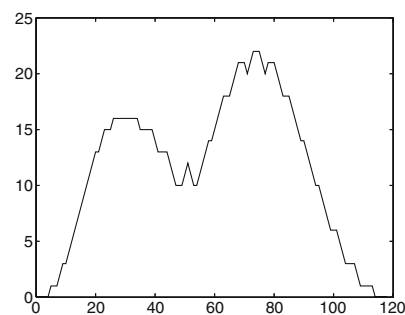
```
for ( i=0; i<=6; i++)
  for ( j=0; j<=3; j++)
    if ( 3<=i+j && i+j<=6 ) A[i][j] = ... ;
```

```
for ( i=0; i<=14; i++)
  for ( j=0; j<=5; j++)
  { if ( 5<=i+j )
    if ( i<=3 ) B[i][j] = A[i][j-2] + A[6-i][5-j] ;
    else if ( i+j<=10 ) B[i][j] = ... ;
    if ( 9<=i+j && i+j<=14 ) ... = B[i-4][j];
  }
```

(a)



(b)



**Figure 3** **a** Illustrative example with two arrays. **b** The variation of the storage requirements during the execution of the illustrative code: for the signals *A* and *B* (the first graph), and for the whole code (the second graph).

*B*-elements produced in the second loop nest are consumed in the second one as well. The variation of the storage requirements for each of the signals *A* and *B*, as well as for the entire code, are shown in Fig. 3b. If we assume that the *A*-elements and *B*-elements are stored in separate windows of the physical memory, the minimum data memory (maximum storage requirement) for *A* is 16 locations, while for *B* it is 23 locations (therefore, a total of 39 locations). Otherwise, since part of the *B*-elements can be stored in memory locations previously occupied by consumed *A*-elements, the minimum data storage ensuring the code execution is only 23 locations, as shown by the second graph in Fig. 3b.

An analysis—similar to the one in Section 2.1—of the canonical linearizations of the signals *A* and *B* yields mapping windows of sizes  $|W_A| = 22$  and  $|W_B| = 25$  locations, with assignment functions  $A[\text{index}_1][\text{index}_2] \mapsto W_A[(4 * \text{index}_1 + \text{index}_2) \bmod 22]$  and  $B[\text{index}_1][\text{index}_2] \mapsto W_B[(6 * \text{index}_1 + \text{index}_2) \bmod 25]$ , relative to some base address. Hence, the model [4] yields a data memory of  $|W_A| + |W_B| = 22 + 25 = 47$  locations.

Applied to the illustrative example in Fig. 3a, the mapping model [5] computes 2-D windows for the signals *A* and *B*, large enough along each dimension to bound their elements simultaneously alive. Without further details, these bounding windows result to be  $W_A = (7, 4)$  and  $W_B = (5, 6)$ . The assignment functions mapping the array spaces into these windows are  $A[\text{index}_1][\text{index}_2] \mapsto W_A[\text{index}_1 \bmod 7][\text{index}_2 \bmod 4]$  and, respectively,  $B[\text{index}_1][\text{index}_2] \mapsto W_B[\text{index}_1 \bmod 5][\text{index}_2 \bmod 6]$ . Hence, the model [5] yields a data memory of  $|W_A| + |W_B| = 28 + 30 = 58$  locations.

Both these previous signal assignment techniques yield data memory allocation solutions quite poor for this illustrative example, at least twice the size of the

minimum data memory (of 23 locations). The main cause of this relatively poor behavior is the separate handling of the arrays. Since the mapping windows are all disjoint, no assignment technique adopting such a strategy could yield a better result than the sum of the minimum storage requirement per signal, that is,  $16 + 23 = 39$  locations in this case.

Now, suppose that the *A*-elements and *B*-elements are mapped first as in De Greef's model [4]:  $A[\text{index}_1][\text{index}_2] \mapsto W_A[(4 * \text{index}_1 + \text{index}_2) \bmod 22]$  and  $B[\text{index}_1][\text{index}_2] \mapsto W_B[(6 * \text{index}_1 + \text{index}_2) \bmod 25]$ . If the two memory windows  $W_A$  and  $W_B$  are contiguous, the maximum distance between locations simultaneously occupied by live elements is 32. This is the distance between the locations occupied by  $A[3][0]$  and  $B[3][4]$ :  $\text{dist}(W_A[12], W_B[22]) = 32$ , if  $W_B$  follows  $W_A$  in the physical memory. This means that a common window  $W_{AB}$  of size 33 is actually sufficient for both signals. Indeed, it can be verified that mapping *A* and *B* in the same window  $W_{AB}$  such that  $A[\text{index}_1][\text{index}_2] \mapsto W_{AB}[(4 * \text{index}_1 + \text{index}_2) \bmod 22]$  and  $B[\text{index}_1][\text{index}_2] \mapsto W_{AB}[(22 + (6 * \text{index}_1 + \text{index}_2) \bmod 25) \bmod 33]$  does not create any conflict between simultaneously alive elements. Note that an amount of 33 storage locations after mapping is better than the lower bound of 39 locations when the arrays are handled separately. Actually, there is an even better result of only 25 locations, very close to the absolute minimum of 23 locations when the assignment function for the *B*-elements is  $B[\text{index}_1][\text{index}_2] \mapsto W_{AB}[(6 * \text{index}_1 + \text{index}_2 + 14) \bmod 25]$ , the mapping of *A* being the same.

There is a price to be paid, though: the cost of the address generation hardware may increase (here, the mapping function for *B* is more complex), which is not unexpected. However, in the system-level exploration phase, the designer must be offered as many possible meaningful options.

This illustrative example was used to convincingly prove that developing signal-to-memory assignment techniques that allow memory sharing between different arrays can be very beneficial in terms of data storage.

The next section will present a signal-to-memory assignment algorithm for multidimensional signal processing applications. In a first phase, the two mapping models [4] and [5] are implemented within a common theoretical framework, which is advantageous both from the point of view of computational efficiency and the amount of allocated data storage – since the allocation solution for each signal is the best of the two models. In a second phase, the assignment algorithm contains a mechanism of investigating and exploiting the possibility of memory sharing between elements of different arrays.

Different from all the previous works which do not provide realistic metrics of quality for their memory allocation solutions, this memory management software tool also computes the minimum storage requirement of each multidimensional signal in the specification (therefore, the optimal memory sharing between the elements of the same array), as well as the minimum data storage for the entire (procedural) algorithmic specification (therefore, the optimal memory sharing between all the array elements and scalars in the code) [12].

### 3 The Signal Assignment Model

The idea of the signal assignment algorithm is to start from the mapping solution of either the model based on canonical linearizations [4], or the model based a multidimensional bounding window [5] – which one is better for the given algorithmic specification (see the discussion in Section 2.1). Afterwards, to search for a pairwise grouping of the array windows that will yield the maximum benefit in terms of data storage reduction by mutual memory sharing.

The flow of the algorithm is described below. Afterwards, an example will illustrate its main steps.

#### 3.1 The Flow of the Signal Assignment Algorithm

**Step 1** For every array  $A$  in the algorithmic specification, compute the size of the memory window  $|W_A|$  as the minimum window of the two assignment models [4] and [5].

It must be emphasized that, although we use the same definitions of array windows as the previous

assignment models [4] and [5], our computation methodology is entirely different.

The computation method employed by De Greef et al. consists of a sequence of integer linear programming (ILP) optimizations for each array linearization [4]. Tronçon et al. perform first a “liveness analysis”. During this phase, a set of program points is firstly decided. For instance, if the code contains  $n$  nested loops,  $2n + 2$  program points are considered: one at the beginning of each loop body and one at the exit from each loop body, together with points at the start and at the end of the code. Every program point is annotated with a set of  $\mathbf{Z}$ -polyhedra, whose integer solutions specify sets of live array elements. In this way, all the live array elements in the chosen program points can be determined. Afterwards, an evaluation of the lower-bounds of window sides  $w_k$  is performed. Starting with  $w_k = 1$ , all the integer solutions differing in the  $k$ -th coordinate ( $x'_k \neq x''_k$ ) of the computed  $\mathbf{Z}$ -polyhedra are tested whether  $x'_k \bmod w_k \neq x''_k \bmod w_k$  or not. The equality implies that a potential mapping conflict was found and, consequently, the window side  $w_k$  should be increased. Subsequent incremental upward adjustments of  $w_k$  are performed for all the chosen program points till no mapping conflict occurs.

In contrast, our methodology is developed within a polyhedral framework operating with ( $\mathbf{Z}$ -) polytopes and linearly bounded lattices [13]. A ( $\mathbf{Z}$ -) polytope is a bounded and closed  $n$ -dimensional polyhedron, restricted to the points having integer coordinates (since the loop iterators are integers). A linearly bounded lattice is the image of an affine vector function  $\mathbf{i} \mapsto \mathbf{x} = \mathbf{T} \cdot \mathbf{i} + \mathbf{u}$  over a ( $\mathbf{Z}$ -) polytope. The reason of using such a framework is that the set of index vectors (named *index/array space*)  $\mathbf{x} = [x_1, \dots, x_m]^T$  of an array reference  $M[x_1(i_1, \dots, i_n)] \dots [x_m(i_1, \dots, i_n)]$  of an  $m$ -dimensional signal  $M$ , in the scope of a nest of  $n$  loops having the iterators  $i_1, \dots, i_n$ , can be typically represented as a linearly bounded lattice in affine specifications, whereas the set of iterator vectors (named *iterator space*)  $\mathbf{i} = [i_1, \dots, i_n]^T$  can be typically represented as ( $\mathbf{Z}$ -) polytopes. (The iterator space of an array reference is not always a convex polytope; it can be a non-convex polyhedron, or even a union of convex and non-convex polyhedra. But, nevertheless, it can be *decomposed* into a finite set of disjoint ( $\mathbf{Z}$ -) polytopes.)

The key idea of our methodology is the reduction of the computation of memory windows for entire *arrays* to the computation of windows for *lattices*. Note that the way an array window is defined depends on the mapping model employed, as exemplified in Section 2.1 for the two previous models [4] and [5]. Consequently, the computation of windows for lattices will be also

dependent on the mapping model employed. Leaving this difference aside, the rest of the computation is practically independent of the mapping model.

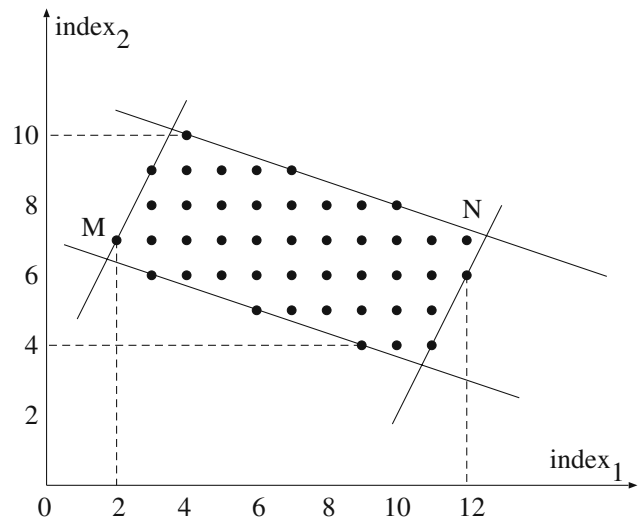
Figure 4 displays an illustrative code and the index (array) space of the array reference  $A[i][j]$ , all the points ( $A$ -elements) being black (alive) after the execution of the loop nest. According to the mapping model [5], the problem is to compute the integer projections [14] of this lattice on the  $m = 2$  coordinate axes, the elements of the  $m$ -dimensional memory window being the sizes of these  $m$  projections. For this example,  $W = (12 - 2 + 1, 10 - 4 + 1) = (11, 7)$ . Technical details of this computation can be found in [15].

According to the model [4], the window of the lattice depends on the canonical linearization. Taking, for instance, the linearization by row concatenation, with the rows sorted increasingly, it can be observed that the maximum distance between live elements (black points) is reached when the index vectors of the  $A$ -elements are the minimum and, respectively, the maximum relative to the lexicographic order.<sup>1</sup> These array  $A$ -elements are represented by the points  $M = A[2][7]$  and  $N = A[12][7]$  in Fig. 4, and  $\text{dist}(M, N) = (x_N - x_M)\Delta y + (y_N - y_M) = (12 - 2) \times 7 + (0 - 0) = 70$ . In the linearization by column concatenation in the increasing order of the columns, 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 indices is applied first. More general, for any canonical linearization, it is sufficient to apply an index permutation first, followed by the computation of (lexicographically) minimum and maximum index vectors, whose indexes are inversely permuted in the end. The distance between these points (whose coordinates are the two vectors) plus 1 is the window size. The minimum window size over all the canonical linearization is the window size of the lattice. Technical details of this computation can be found in [15], as well.

Keeping in mind the computation methodology for the memory windows of lattices, the computation flow of *Step 1* is the following:

**Step 1a** *Extract the array references from the given algorithmic specification and decompose the array*

<sup>1</sup>Let  $\mathbf{x} = [x_1, \dots, x_m]^T$  and  $\mathbf{y} = [y_1, \dots, y_m]^T$  be two  $m$ -dimensional vectors. The vector  $\mathbf{y}$  is larger lexicographically than  $\mathbf{x}$  (written  $\mathbf{y} > \mathbf{x}$ ) if  $(y_1 > x_1)$ , or  $(y_1 = x_1 \text{ and } y_2 > x_2)$ ,  $\dots$ , or  $(y_1 = x_1, \dots, y_{m-1} = x_{m-1}, \text{ and } y_m > x_m)$ .



```
int A[13][11];
```

```
for (i=2; i<=16; i++)
  for (j=2*i-18; j<=2*i+3; j++) {
    if ( i+3*j >= 21 && i+3*j <= 34 ) A[i][j] = ... ;
  }
```

**Figure 4** The computation of the memory window of an array reference.

*references for every multidimensional signal into disjoint (linearly bounded) lattices.*

This partitioning into disjoint lattices—used also in [12]—can be performed analytically, by recursively intersecting the array references of every multidimensional signal in the code. This is a complex operation which is thoroughly discussed and exemplified in [12]. In this way, the next substeps will have to deal only with windows of lattices in order to eventually obtain the windows of entire arrays.

**Step 1b** *Compute underestimations of the window sizes for each indexed signal taking into account the live elements at the boundaries between the loop nests.*

Let  $A$  be an  $m$ -dimensional signal in the algorithmic specification, and let  $\mathcal{P}_A$  be the set of disjoint lattices partitioning the index space of  $A$ . A high-level pseudo-code of the computation of  $A$ 's preliminary windows is given below. Preliminary window sizes for each canonical linearization according to DeGreef's model [4] are computed first, followed by the computation of the window size underestimate according to Tronçon's model [5] in the same framework operating with lattices. The meaning of the variables are explained as comments.

```

for (each canonical linearization  $\mathcal{C}$ ) {
  for (each disjoint lattice  $L \in \mathcal{P}_A$ ) // compute the
    (lexicographically) minimum and maximum ...
    compute  $x^{\min}(L)$  and  $x^{\max}(L)$ ; // ... index
    vectors of  $L$  relative to  $\mathcal{C}$ 
  for (each boundary  $n$  between the loop nests  $n$  and
     $n+1$ ) { // the start of the code is boundary 0
    let  $\mathcal{P}_A(n)$  be the collection of disjoint lattices
      of  $A$ , which are alive at the boundary  $n$ ;
      // these are disjoint lattices produced
      before the boundary and consumed
      after it
    let  $X_n^{\min} = \min_{L \in \mathcal{P}_A(n)} \{x^{\min}(L)\}$  and
       $X_n^{\max} = \max_{L \in \mathcal{P}_A(n)} \{x^{\max}(L)\}$ ;
     $|W_{\mathcal{C}}(n)| = \text{dist}(X_n^{\min}, X_n^{\max}) + 1$ ; // The distance
      is computed in the canonical linearization  $\mathcal{C}$ 
    }
     $|W_{\mathcal{C}}| = \max_n \{|W_{\mathcal{C}}(n)|\}$ ; // the window size
    according to [4] for the canonical linearization  $\mathcal{C}$ 
  } // (possibly, an underestimate) 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  $n+1$ )
  { // the start of the code is boundary 0
  let  $\mathcal{P}_A(n)$  be the collection of disjoint lattices of  $A$ ,
    which are alive at the boundary  $n$ ;
  for (each dimension  $k$  of signal  $A$ ) {
    let  $X_k^{\min} = \min_{L \in \mathcal{P}_A(n)} \{x_k^{\min}(L)\}$  and
       $X_k^{\max} = \max_{L \in \mathcal{P}_A(n)} \{x_k^{\max}(L)\}$ ;
     $w_k(n) = X_k^{\max} - X_k^{\min} + 1$ ; // The  $k$ -th side
      of  $A$ 's bounding window at boundary  $n$ 
    }
  }
  for (each dimension  $k$  of signal  $A$ )  $w_k = \max_n \{w_k(n)\}$ ;
  //  $k$ -th side of  $A$ 's window over all boundaries
   $|W| = \prod_{k=1}^m w_k$ ; // the window size according to [5]
  (possibly, an underestimate)
}

```

Steps 1a and 1b find the exact values of the window sizes for both models when every loop nest either produces or consumes (but not both!) the signal's elements. When elements of the signal are both produced and consumed in the same loop nest (like in the illustrative examples from Fig. 1), then the window sizes obtained after Step 1b may be only underestimates since an increase of the storage requirement can happen inside such a loop nest. In order to determine the exact values of the window sizes, an additional step is required.

**Step 1c** Update the mapping windows for each indexed signal in every loop nest producing and consuming elements of the signal.

Let  $L \in \mathcal{P}_A$  be a disjoint lattice of an indexed signal  $A$  which is consumed in a certain loop nest where  $A$ -elements are produced as well. Then,

```

for (each  $A$ -element covered by the lattice  $L$ ) {
  compute the iteration vector  $\mathbf{i}$  when the  $A$ -element
  is consumed (accessed for the last time);
  // this requires the computation of the maximum
  iterator vector relative to the lexicographic
  order [12]
  // if  $L$  is included in several array references,
  the overall maximum iterator vector is
  considered
  determine the disjoint lattices  $\mathcal{L}_p$  of  $A$  partially
  produced until the  $A$ -element is consumed in
  iteration  $\mathbf{i}$ ;
  determine the disjoint lattices  $\mathcal{L}_c$  of  $A$  partially
  consumed until the  $A$ -element is consumed in
  iteration  $\mathbf{i}$ ;
  for (each canonical linearization  $\mathcal{C}$ )
    for (each lattice  $L'$  in  $\mathcal{L}_p \cup \mathcal{L}_c$ ) {
      compute  $x^{\min}(L')$  and  $x^{\max}(L')$ ;
       $|W_{\mathcal{C}}| = \max\{|W_{\mathcal{C}}|, \text{dist}(x^{\min}(L'),
        x^{\max}(L')) + 1\}$ ;
      //  $|W_{\mathcal{C}}|$  may increase
    }
  for (each dimension  $k$  of signal  $A$ )
    for (each lattice  $L'$  in  $\mathcal{L}_p \cup \mathcal{L}_c$ ) {
      compute  $x_k^{\min}(L')$  and  $x_k^{\max}(L')$ ;
       $w_k = \max\{w_k, x_k^{\max}(L') - x_k^{\min}(L') + 1\}$ ;
      //  $w_k$  may increase
      update  $|W|$  if  $w_k$  increased its value;
      //  $|W|$  may increase
    }
  }
}

```

The guiding idea is that local or global maxima of  $w_k$  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  $|W_{\mathcal{C}}|$  are reached immediately before the consumption of an  $A$ -element, which may entail a decrease of the maximum distance between live elements.

Finally, the memory window for signal  $A$  has the minimum size over the two models:  $|W_A| = \min\{|W|, \min_{\mathcal{C}}\{|W_{\mathcal{C}}|\}\}$ . The mapping functions are also available (as exemplified in Section 2.1).

The next steps investigate the possibility of memory sharing between the elements of different arrays.

**Step 2** Build a complete graph  $G$ , where each vertex represents an array in the application code. Compute weights for every edge  $(A, B)$  in the following way:

- if the two arrays  $A$  and  $B$  have disjoint lifetimes, the weight is  $\min\{|W_A|, |W_B|\}$ ;
- when the lifetimes of the two arrays overlap, compute the maximum distance between the locations occupied by simultaneously alive  $A$ - and  $B$ -elements, taking into account the mapping functions found at *Step 1* and assuming the two memory windows are contiguous; the size of the common window  $W_{AB}$  is this maximum distance plus 1; the weight of the edge  $(A, B)$  is  $|W_A| + |W_B| - |W_{AB}|$ . This weight represents the data storage saved when the two arrays  $A$  and  $B$  share the same memory space versus the situation when the two arrays would be stored separately (in disjoint memory windows).

**Step 3** Find the maximum weighted matching in the graph  $G$ . A matching in graph is a set of edges, no two of which meet at a common vertex. The weight of the matching is the sum of the weights of its edges. A maximum weighted matching represents a matching of maximum weight, as shown in Fig. 5 for a graph with 18 vertices. In this case, the matching will produce the most beneficial pairwise grouping of the arrays in terms of storage reduction. The matching solution will maximize the overall savings of data storage when the

arrays are sharing pairwise the memory space. Note that even larger savings could be achieved, in principle, if more complex array groups (larger than two) shared the same memory space, but the computation effort would become prohibitive.

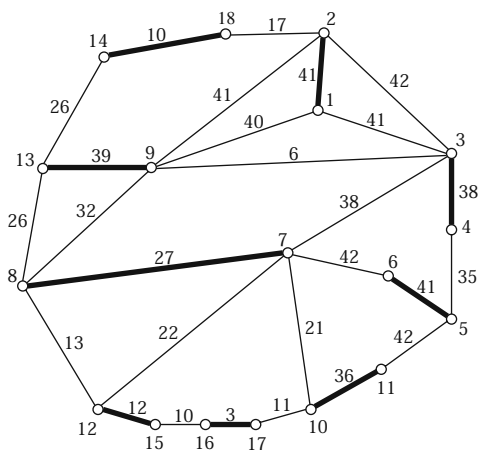
Maximum matching has been a subject of interest in graph theory for many years. Matching algorithms were first developed for bipartite graphs. For non-bipartite graphs, most of the best matching algorithms are based on the work of Claude Berge [16], who proposed searching for augmenting paths as a general strategy for maximum matching. Based on Berge's theorems, Edmonds proposed an efficient algorithm whose computation time is proportional to  $V^4$ , where  $V$  is the number of vertices [17]. The algorithm works in  $O(V)$  stages (of cubic complexity each) by finding augmenting paths by a tree search combined with a process of shrinking certain subgraphs called *blossoms* into single nodes of a reduced graph (most often Edmond's algorithm is called the "blossom shrinking algorithm"). Edmond's algorithm has been refined by Gabow [18], who obtained an overall complexity of  $O(V^3)$ . The fastest known algorithm (under the assumption of integral weights that are not particularly high) was developed by Gabow and Tarjan [19]. Further theoretical and practical improvements for large-scale matching problems were proposed by Applegate and Cook [20]. We are using an implementation of Gabow's algorithm [18] due to Ed Rothberg, available online [21]. The running times of this implementation is very suitable for our needs since the graphs built by the assignment algorithm have rather small numbers of vertices (each representing an array in the application code).

**Step 4** Compute the overall data storage corresponding to the maximum weighted matching in the graph. In principle, the overall amount of data memory after mapping is the sum of the window sizes resulted after matching. This step takes into account the possibility of disjoint lifetimes between the (pairs of) arrays in this matching, in which case the (common) windows can overlap and share the largest of the two windows. Finally, determine the mapping functions for each array.

### 3.2 Example Illustrating the Flow of the Algorithm

The algorithm will be illustrated on the code from Fig. 6a.

*Step 1* computes the sizes of the memory windows for each array in the code, according to the assignment models [4] and [5], implemented as described in



**Figure 5** The maximum weighted matching in a weighted graph with 18 vertices (bold edges).

**Figure 6** (a) Example illustrating the flow of the algorithm. (b) Table with the sizes of the mapping memory windows for the signal assignment models [4] and [5], as computed at Step 1. The last row of the table displays the minimum window sizes, computed as in [12]. (c) The complete graph built at Step 2 and the maximum weighted matching computed at Step 3.

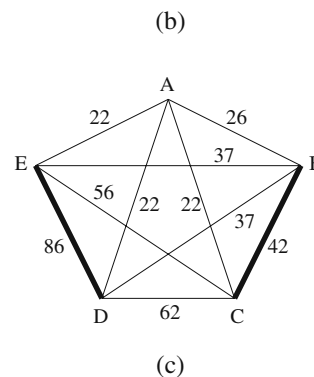
```

int A[7][4], B[9][5], C[11][6], D[13][7], E[15][8];

for (i=0; i<=6; i++)
  for (j=0; j<=3; j++)
    if (3<=i+j && i+j<=6) A[i][j] = 1;
for (i=0; i<=8; i++)
  for (j=0; j<=4; j++)
    if (4<=i+j)
      if (i<=3) B[i][j] = A[i][j-1] + A[6-i][4-j];
      else if (i+j<=8) B[i][j] = 2;
for (i=0; i<=10; i++)
  for (j=0; j<=5; j++)
    if (5<=i+j)
      if (i<=4) C[i][j] = B[i][j-1] + B[8-i][5-j];
      else if (i+j<=10) C[i][j] = 3;
for (i=0; i<=12; i++)
  for (j=0; j<=6; j++)
    if (6<=i+j)
      if (i<=5) D[i][j] = C[i][j-1] + C[10-i][6-j];
      else if (i+j<=12) D[i][j] = 4;
for (i=0; i<=14; i++)
  for (j=0; j<=7; j++)
    if (7<=i+j)
      if (i<=6) E[i][j] = D[i][j-1] + D[12-i][7-j];
      else if (i+j<=14) E[i][j] = 5;
for (i=8; i<=22; i++)
  for (j=0; j<=7; j++)
    if (15<=i+j && i+j<=22) ... = E[i-8][j];
    
```

(a)

Memory windows	Array A	Array B	Array C	Array D	Array E	Array total
Model [4]	22	37	56	79	106	300
Model [5]	28	45	66	91	120	350
Min  W	16	25	36	49	64	190



Section 3.1. These window sizes are displayed in the first and the second rows of the table in Fig. 6b. For comparison, the table in Fig. 6b displays also the minimum storage requirement separately for each array [12]. The best result for each array is selected.

Step 2 builds a complete weighted graph with 5 vertices, one for each array in the code. Afterwards, it computes the weights of its edges. This graph is shown in Fig. 6c. For instance, the arrays A and E have disjoint lifetimes and, consequently, the weight of the edge (A, E) is  $\min\{|W_A|, |W_E|\} = \min\{22, 106\} = 22$ . Since A can share the memory space of E, the storage saving is 22 locations—the window size of A (see the first row of the table in Fig. 6). On the other hand, since D and E have overlapping lifetimes, the size of the common memory window of the two signals results to be  $|W_{DE}| = 99$  locations. The weight of the edge (D, E) is  $|W_D| + |W_E| - |W_{DE}| = 79 + 106 - 99 = 86$  locations, which is the memory saving when D and E share the same storage space.

The size of the common memory window for signals whose lifetimes are not disjoint, like the signals D and E, is determined as follows. Considering the window  $W_D$  followed contiguously in the memory by the window  $W_E$ , we compute the minimum distance between locations occupied by live D- and E-elements. This computation involves the mapped disjoint lattices of D and E, whose lifetime information is known. If the minimum distance results larger than zero, than the two windows can be overlapped by this amount. For the

signals D, E the minimum distance is 86, hence yielding a common window of  $|W_{DE}| = 79 + 106 - 86 = 99$  locations. Note that the weight of the edge (D, E) is actually this minimum distance. The computation is repeated assuming that the window  $W_E$  is followed contiguously by  $W_D$  and the better of the two results is chosen.

Step 3 computes the maximum weighted matching in the graph. The edges in this matching are (D, E) and (B, C). It follows that the arrays D and E will share a common memory space ( $W_{DE}$ ) of 99 locations. Similarly, B and C will share a common window ( $W_{BC}$ ) of size 51, while A will be stored by itself in a memory window of size 22. The amount of data memory after mapping is, for the time being,  $|W_{DE}| + |W_{BC}| + |W_A| = 99 + 51 + 22 = 172$  locations.

Step 4 finds that, actually, A has a disjoint lifetime from both arrays in the pair (D, E), so it can share their window. Therefore, the final result is  $|W_{DE}| + |W_{BC}| = 99 + 51 = 150$  locations, whereas the previous models [4] and [5] yield 300 and 350 locations, respectively.

The mapping functions for each array are:

$$\begin{aligned}
 A[i][j] &\mapsto W_{DE}[(4i + j) \bmod 22] \\
 B[i][j] &\mapsto W_{BC}[(5i + j) \bmod 37] \\
 C[i][j] &\mapsto W_{BC}[(37 + (6i + j) \bmod 56) \bmod 51] \\
 D[i][j] &\mapsto W_{DE}[(7i + j) \bmod 79] \\
 E[i][j] &\mapsto W_{DE}[(79 + (8i + j) \bmod 106) \bmod 99]
 \end{aligned}$$

Note that the assignment algorithm can be stopped after *Step 1*, if desired: after the first phase, this algorithm will already offer an allocation solution at least as good as the best of the two models [4] and [5], since the algorithm picks the smaller window for each of the arrays. *Step 1* can even work with only one of the previous models, rather than with both of them. But, since any of the models could yield the best result (see the examples in Section 2.1) and since the computation overhead is not significant (the implementation of both models in a common polyhedral framework being advantageous both from the point of view of computational efficiency and the amount of data storage), the execution of *Step 1* with two models seems to be the right choice.

Optionally, *Steps 2–4* can be run in a second phase, to investigate possibilities of memory sharing between distinct arrays. More reduction of the data storage can thus be obtained, the price being some additional computation time and, possibly, a more complex address generation unit (ADU), as suggested by the mapping functions for the signals *C* and *E*. This unit will need to compute additions, multiplications, and modulo operations, like many typical ADUs [22]. Having more than one allocation solution should be a benefit for the designer since it offers the possibility of trading-off the amount of data storage against the complexity of the ADU.

## 4 Experimental Results

The polyhedral framework for the memory management of multidimensional signal processing applications has been implemented in C++, incorporating the algorithm described in this paper. The algorithmic specifications of the applications are expressed in a subset of the C language, illustrated in the examples used in the paper.

Table 1 summarizes 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 benchmark tests (column 1) are signal processing applications and typical algebraic kernels used in this domain: (1) a motion detection algorithm used in the transmission of real-time video signals on data networks [3]; (2) the kernel of a motion estimation algorithm for moving objects (MPEG-4); (3) Durbin's algorithm for solving Toeplitz systems with  $N$  unknowns; (4) a singular value decomposition (SVD) updating algorithm [23] used in spatial division multiplex access (SDMA) modulation in mobile communication receivers, in beamforming, and Kalman filtering; (5) the kernel of a voice coding

application—essential component of a mobile radio terminal.

Columns 2 and 3 display the numbers of array references and array elements in the specification code. The next group of three columns summarizes the allocation results: the sum of the memory windows computed according to the [4] model, the sum of the memory windows computed according to the [5] model, and the final allocation result—the overall data storage after mapping. (For the illustrative example in Fig. 3, the values in these columns would be 47, 58, and 25—as explained in Section 2.2.) The 7-th 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 [12]. (Again, for the illustrative example in Fig. 3, these two values are 39 and 23.) The last column displays the total running times for the results shown in columns 4–7.

The data memory sizes after mapping yielded according to the models [4] and [5] (columns 4 and 5) are always lower-bounded by the first values in the 7-th column, since the memory window of an array cannot be smaller than the minimum storage requirement of the array. However, our allocation results (column 6) can be better than the first values in the 7-th column, since the array windows may overlap significantly (the effect of memory sharing between arrays).

The two metrics of performance from column 7 help the designer to assess the quality of the allocation solution. E.g., for the motion detection, the data memory after mapping results equal to the absolute lower bound—the minimum storage requirement for the code execution. It follows that the allocation solution is optimal and cannot be improved. For the motion estimation the individual array windows are optimal as well, but the memory allocation solution could still be improved enhancing the memory sharing between elements of different arrays.

For these benchmark tests, the memory windows computed according to the model [4] resulted equal or smaller than the windows computed according to the model [5], although this is not a general conclusion—as exemplified in Section 2.1. (The largest difference is observed for the SVD updating.) However, the largest part of the overall running times (column 8) is spent computing the windows in column 4. This is due in part to the number of canonical linearizations that have to be analyzed.

**Table 1** Experimental results.

Application parameters	Num. array references	Num. array elements	Mem. size after mapping			$\sum \min W  / \min(\text{Mem}_{\text{code}})$	CPU [s]
			$\sum  W $ [4]	$\sum  W $ [5]			
Motion detection	11						
M = N = 32, m = n = 4		72,543	2,741	2,741	2,740	2,741 / 2,740	3
M = N = 64, m = n = 4		318,367	9,525	9,525	9,524	9,525 / 9,524	13
M = N = 120, m = n = 8		3,749,063	33,285	33,285	33,284	33,285 / 33,284	196
Motion estimation	17						
M = 32, N = 16		265,633	4,513	4,513	3,624	4,513 / 2,465	22
M = 48, N = 32		2,368,865	11,873	11,873	10,054	11,873 / 7,265	182
M = 64, N = 32		4,208,449	18,241	18,241	14,670	18,241 / 10,049	464
Durbin algorithm	27						
N = 50		2,749	201	203	164	176 / 124	2
N = 100		10,499	401	403	327	351 / 249	4
SVD updating	86						
n = 20		25,908	2,464	3,243	2,120	1,664 / 1,430	55
n = 30		85,213	5,494	7,263	4,751	3,694 / 3,195	162
n = 40		199,418	9,724	12,883	8,356	6,524 / 5,660	285
Voice coder	251	33,751	14,498	14,525	12,690	12,965 / 11,899	130

Our polyhedral framework entails significantly faster computations of the initial mapping windows. The running times reported in [4] and [5] are typically of the order of minutes or even tens of minutes, whereas our implementation runs for the same examples, or of similar complexity, in tens of seconds at most (only for columns 4 and 5). For instance, the *voice coding* application was processed by [5] in over 25 min using a 300 MHz Pentium II; in contrast, our running time is significantly shorter—only 12 s for *Step 1*—in spite of the different computation platforms. (For evaluating the difference of performance between the two computation platforms [24], a scaling factor of at most 20 between CPU times is fairly accurate.) In another common benchmark—the SVD updating algorithm [23]—their test has only 6,038 array elements, which corresponds to matrices of order  $n = 12$  in the application code. Their reported run time is 87 s [5], whereas ours was about 1.5 s (only for column 5). In spite of the difference between the computation platforms, we can safely state that our implementation is several times faster. The investigation of inter-array memory sharing increases our running times, but the additional savings of data memory are quite significant.

## 5 Conclusions

This paper has addressed the problem of mapping the multidimensional arrays from the high-level algorithmic specifications of signal processing applications into the physical memory. Different from all the previous techniques that handle the arrays independently from

one another, this novel mapping approach exploits the possibility of inter-array memory sharing, yielding significant savings of data storage. Moreover, the computation methodology—based on operations with polytopes and lattices—allows the extension of the mapping approach to distributed memory organizations.

This assignment model will be part of a power-aware memory management framework for embedded data-intensive signal processing applications. In a first phase, this framework will provide solutions for the optimization of the dynamic energy consumption—which expands when memory accesses occur—in hierarchically-organized memory subsystems. Subsequently, the framework development will focus on the reduction of the static energy—spent as long as memory is powered on.

## References

- Luican, I. I., Zhu, H., & Balasa, F. (2008). Efficient assignment algorithm for mapping multidimensional signals into the physical memory. In *Proc. IEEE int. conf. acoustics, speech, and signal processing* (pp. 1409–1412). Las Vegas NV.
- Panda, P. R., Catthoor, F., Dutt, N., Dankaert, K., Brockmeyer, E., Kulkarni, C., et al. (2001). Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 6(2), 149–206.
- Catthoor, F., Danckaert, K., Kulkarni, C., Brockmeyer, E., Kjeldsberg, P. G., Van Achteren, T., et al. (2002). *Data access and storage management for embedded programmable processors*. Boston: Kluwer.
- De Greef, E., Catthoor, F., & De Man, H. (1997). Memory size reduction through storage order optimization for embedded parallel multimedia applications. In A. Krikelis (Ed.),

- Parallel computing* (Special issue on “Parallel Processing and Multimedia”) (Vol. 23, no. 12, pp. 1811–1837). Amsterdam: Elsevier.
5. Tronçon, R., Bruynooghe, M., Janssens, G., & Catthoor, F. (2002). Storage size reduction by in-place mapping of arrays. In A. Coresi (Ed.), *Verification, model checking and abstract interpretation* (pp. 167–181).
  6. Lefebvre, V., & Feautrier, P. (1998). Automatic storage management for parallel programs. *Parallel Computing*, 24, 649–671.
  7. Quilleré, F., & Rajopadhye, S. (2000). Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems*, 22(5), 773–815.
  8. Schrijver, A. (1986). *Theory of linear and integer programming*. New York: Wiley.
  9. Darte, A., Schreiber, R., & Villard, G. (2005). Lattice-based memory allocation. *IEEE Transactions on Computers*, 54, 1242–1257.
  10. Verbauwhede, I., Catthoor, F., Vandewalle, J., & De Man, H. (1991). In-place memory management of algebraic algorithms on application specific processors. In E. Deprettere, et al. (Eds.), *Algorithms and parallel VLSI architectures*. Amsterdam: Elsevier.
  11. Kjeldsberg, P. G., Catthoor, F., & Aas, E. J. (2003). Data dependency size estimation for use in memory optimization. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 22(7), 908–921.
  12. Balasa, F., Zhu, H., & Luican, I. I. (2007). Computation of storage requirements for multi-dimensional signal processing applications. *IEEE Transactions on VLSI Systems*, 15(4), 447–460.
  13. Thiele, L. (1992). Compiler techniques for massive parallel architectures. In P. Dewilde (Ed.), *State-of-the-art in computer science*. Boston: Kluwer.
  14. Verdoolaege, S., Beyls, K., Bruynooghe, M., & Catthoor, F. (2005). Experiences with enumeration of integer projections of parametric polytopes. In R. Bodik (Ed.), *Compiler construction: 14th int. conf.* (Vol. 3443, pp. 91–105). New York: Springer.
  15. Luican, I. I., Zhu, H., & Balasa, F. (2007). Signal-to-memory mapping analysis for multimedia signal processing. In *Proc. asia & south-pacific design automation conf.* (pp. 486–491). Yokohama, Japan.
  16. Berge, C. (1957). Two theorems in graph theory. *Proceedings of the National Academy of Sciences of the United States of America*, 43, 842–844.
  17. Edmonds, J. (1965). Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17, 449–467.
  18. Gabow, H. N. (1973). *Implementation of algorithms for maximum matching on non-bipartite graphs*. Ph.D. Thesis, Stanford University.
  19. Gabow, H. N., & Tarjan, R. E. (1991). Faster scaling algorithms for general graph-matching problems. *Journal of the ACM*, 38(4), 815–853.
  20. Applegate, D., & Cook, W. (1993) Solving large-scale matching problems. In D. Johnson & C. C. McGeoch (Eds.), *Network flows and matchings. DIMACS series in discrete mathematics and theoretical computer science* (Vol. 12, pp. 557–576). Providence: American Mathematical Society.
  21. Rothberg, E. (2009). FTP Directory. <ftp://dimacs.rutgers.edu/pub/netflow/>.
  22. Talavera, G., Jayapala, M., Carrabina, J., & Catthoor, F. (2008) Address generation optimization for embedded high-performance processors: A survey. *Journal of Signal Processing Systems*, 53(3), 271–284.
  23. Moonen, M., Dooren, P. V., & Vandewalle, J. (1992). An SVD updating algorithm for subspace tracking. *SIAM Journal on Matrix Analysis and Applications*, 13(4), 1015–1038.
  24. Tom’s Hardware (2009). *Benchmark marathon: 65 CPUs from 100 MHz to 3066 MHz*. [http://www.tomshardware.com/2003/02/17/benchmark\\_marathon/index.html](http://www.tomshardware.com/2003/02/17/benchmark_marathon/index.html).



**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.