Signal-to-Memory Mapping Analysis for Multimedia Signal Processing*

Ilie I. Luican Hongwei Zhu Florin Balasa

Dept. of Computer Science, University of Illinois at Chicago, Chicago, IL 60607, U.S.A.

Abstract – The storage requirements in data-dominant signal processing systems, whose behavior is described by array-based, loop-organized algorithmic specifications, have an important impact on the overall energy consumption, data access latency, and chip area. Finding the optimal storage of the usually large arrays from these behavioral specifications is an important step during memory allocation. This paper proposes more efficient algorithms for the intra-array storage mapping models of De Greef [5] and Tronçon [11], resulting in an implementation several time faster than the original ones.

1 Introduction

In many signal processing systems, particularly in the multimedia and telecom domains, data transfer and storage have a significant impact on both the system performance and the major cost parameters – power consumption and chip area. During the system development process, the designer must often focus first on the exploration of the memory subsystem in order to achieve a cost optimized end product [1, 2].

The behavior of these targeted VLSI systems, synthesized to execute mainly data-dominant applications, is described in a high-level programming language, where the code is typically organized in sequences of loop nests having as boundaries (usually affine) functions of loop iterators, conditional instructions where the arguments may be data-dependent and/or data-independent (relational and/or logic expressions of affine functions of loop iterators). In our target domain, the data structures are multi-dimensional arrays whose indices in the code are affine functions of loop iterators. The class of specifications with these characteristics are often called *affine* specifications [1].

The optimal mapping to memory of the multi-dimensional signals from these behavioral specifications is an important step during memory allocation.

1.1 Models of signal-to-memory mapping

De Greef *et al.* choose one of the canonical linearizations of the array (a permutation of its dimensions), followed by a modulo operation that wraps the set of "virtual" memory locations into a smaller set of actual physical locations [5]. Instead of a 1-dimensional window in the linearized space of addresses as in [5], Tronçon *et al.* proposed to compute an m-dimensional bounding box in the original m-dimensional index space of the array [11]. This is achieved by finding m modulo operands, computed separately as the maximal index differences in each dimension. See also Section 1.2 for more details on these two models.

Lefebvre and Feautrier, addressing parallelization of static control programs, developed in [6] an intra-array storage 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 studied the problem of memory reuse for systems of recurrence equations, a computation model used to represent algorithms to be compiled into circuits [8]. In their model, the loop iterators first undergo an affine mapping (into a linear space of *smallest* dimension – what they call a "projection") before modulo operations are applied to the array indices.

Darte *et al.* proposed a lattice-based mathematical framework for intra-array mapping, establishing a correspondence between valid linear storage allocations and integer lattices called *strictly admissible* relative to the set of differences of the conflicting indices [4]. They proposed two heuristic techniques for building strictly admissible integer lattices, hence building valid storage allocations.

1.2 Context and motivation of this research

Since De Greef's and Tronçon's models [5, 11] for signal-to-memory mapping[1] play an important part in this paper, they will be better explained and illustrated below.

To reduce the size of a multi-dimensional array mapped to memory, De Greef *et al.* consider all the possible linearizations of the array and, for any linearization, they compute the largest distance at any time between two live elements in the linearized array [5]. This distance plus 1 is then the storage required for the mapping

*This research was sponsored by the U.S. National Science Foundation (DAP 0133318).

[1]This paper focuses on *intra-array* mapping, whereas the *inter-array* in-place mapping (i.e., the optimization of the memory sharing between different arrays) [1] is outside the scope of the paper. Note that we addressed this latter problem in [13], since that algorithm computes the minimum storage requirement of an entire algorithmic specification containing an arbitrary number of multi-dimensional signals.

of the array into the data memory. Note that for an m-dimensional array there are $m!$ orderings of the indices. For instance, a 2-dimensional 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. (For $m \geq 6$, an heuristic is proposed [5] to limit the search.)

In the illustrative example from Fig. 1(a), the elements of the array A are produced column by column from left to right, inside each column being produced bottom-up. 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), 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 9×19=171. If the columns are concatenated decreasingly (($A[index_1][index_2]$, $index_1$=0,18), $index_2$=9,0), there are 9 pairs of elements simultaneously alive, mapped at a maximum distance. Two such elements are, for instance, $A[18][0]$ – produced in the iteration $(i\ ,\ j)$ = (18 , 0) and consumed in (28 , 0), and $A[8][9]$ – produced in the iteration (8 , 9) and consumed in (18 , 9), the distance between them in the linearization being 9×19+10=181.

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), there are 9 pairs of elements simultaneously alive, maximally distanced from each other. Two such elements are, for instance, $A[0][9]$ – produced in the iteration $(i\ ,\ j)$ = (0 , 9) and consumed in (10 , 9), and $A[10][8]$ – produced in the iteration (10 , 8) and consumed in (20 , 8). The distance between them in the linearization is 10×10-1=99. Finally, if the rows are concatenated decreasingly (($A[index_1][index_2]$, $index_2$=0,9), $index_1$=18,0), there are 9 pairs of elements simultaneously alive, as well (e.g., $A[18][0]$ and $A[8][9]$), placed at the maximum distance 10×10+9=109.

For the other 4 linearizations, the maximum distances obtained have the same values, since the array elements are stored in reverse order relative to one of the 4 linearizations analyzed above.

According to De Greef's model [5], the best linearization (among those considered) for the array A in this illustrative example is the concatenation row by row increasingly. A window of 99+1=100 successive memory locations (relative to a certain base address) is sufficient to store the array. Indeed, this linearization can be wrapped modulo 100, which is correct because two values simultaneously alive can never be mapped to the same location.

In order to avoid the inconvenience of analyzing different linearization schemes, Tronçon *et al.* proposed [11] to reduce the size of an m-dimensional array A mapped to the data memory, computing a window $W = (w_1, \ldots, w_m)$, whose elements can be used as operands in modulo operations that redirect all accesses to the array A. An access to the element $A[exp_1]\ldots[exp_m]$ is redirected to $A[exp_1 \bmod w_1]\ldots[exp_m \bmod w_m]$ (relative to a base address in the data memory). Since the mapping has to ensure a correct execution of the code, two distinct array elements simultaneously alive should not be mapped by the modulo operations to the same location. Each window element w_i is the maximum difference (in absolute value) between the i-th indices of any two A-elements simultaneously alive, plus 1. The window W determines the memory size required for storing the array.

In the illustrative example shown in Fig. 1(a), the window corresponding to the signal A is $W = (11\ ,\ 10)$. Indeed, the elements $A[0][9]$ – consumed in the iteration (10 , 9) – and $A[10][0]$ to $A[10][8]$ are simultaneously alive; the maximum difference between the first indices is 10, which yields a window element $w_1 = 10 + 1 = 11$. Similarly, $w_2 = 10$ since the elements $A[9][0], \ldots, A[9][9]$ are simultaneously alive. Note that there is no difference between the two models for 1-dimensional arrays.

This paper proposes more efficient algorithms for the intra-array storage mapping models of De Greef [5] and Tronçon [11], resulting in an implementation several time faster than the ones given by their authors. In addition, this paper better evaluates the two models using a software tool [13] able to compute the minimum linear window for any multi-dimensional array from an affine algorithmic specification.

The rest of the paper is organized as follows. Section 2 describes the global flow of the intra-array mapping, focusing on the more significant algorithmic aspects. Section 3 presents basic implementation aspects and discusses the experimental results. Section 4 summarizes the main conclusions of this work.

2 The array mapping methodology

2.1 Definitions and concepts

Definitions A *polyhedron* is a set of points $P \subset \Re^n$ satisfying a finite set of linear inequalities: $P = \{\ \mathbf{x} \in \Re^n \mid \mathbf{A}\cdot\mathbf{x} \geq \mathbf{b}\ \}$, where $\mathbf{A} \in \Re^{m\times n}$ and $\mathbf{b} \in \Re^m$. If P is a bounded set, then P is called a *polytope*. If $\mathbf{x} \in \mathbf{Z}^n$, then P is called a $\mathbf{Z}$-polyhedron/polytope. Each *array reference* $M[x_1(i_1,\ldots,i_n)]\ \cdots\ [x_m(i_1,\ldots,i_n)]$ of an m-dimensional signal M, in the scope of a nest of n loops having the iterators $i_1, \ldots, i_n$, is characterized by an *iterator space* and an *index* (or *array*) *space*. The iterator space signifies the set of all iterator vectors $\mathbf{i} = (i_1, \ldots, i_n) \in \mathbf{Z}^n$ in the scope of the array reference. The index space is the set of all index vectors $\mathbf{x} = (x_1, \ldots, x_m) \in \mathbf{Z}^m$ of the array reference. When the indices of an array reference are linear mappings with integer coefficients of the loop iterators, the index space consists of one or several *lattices* linearly bounded [10], that is, the image of an affine vector function over the iterator polytope $\{\ \mathbf{i} \in \mathbf{Z}^n \mid \mathbf{A}\cdot\mathbf{i} \geq \mathbf{b}\ \}$:

$$\{\ \mathbf{x} = \mathbf{T}\cdot\mathbf{i} + \mathbf{u} \in \mathbf{Z}^m \mid \mathbf{A}\cdot\mathbf{i} \geq \mathbf{b}\ ,\ \mathbf{i} \in \mathbf{Z}^n\} \quad (1)$$

where $\mathbf{x} \in \mathbf{Z}^m$ is the index vector of the m-dimensional signal and $\mathbf{i} \in \mathbf{Z}^n$ is an n-dimensional iterator vector. In our context, the elements of the matrices $\mathbf{T}$, $\mathbf{A}$ and of the vectors $\mathbf{u}$, $\mathbf{b}$ are integers.

Example 1:
```
$for\ (i = 0;\ i \leq 3;\ i++)$
    $for\ (j = 0;\ j \leq 2;\ j++)$
        $if(\ 3i \geq 2j\ )\ \cdots A[2i+3j][5i+j]\cdots$
```

```
int A[19][10];
                                    //   All the A-elements are
for (i=0; i<29; i++)           //   consumed in this loop nest
    for (j=0; j<10; j++)  {
        if ( i+j >=  9   &&  i+j <= 18 )   A[i][j] = ...
        if ( i+j >= 19  &&  i+j <= 28 )   ... = A[i-10][j] ;
    }
```

(a)

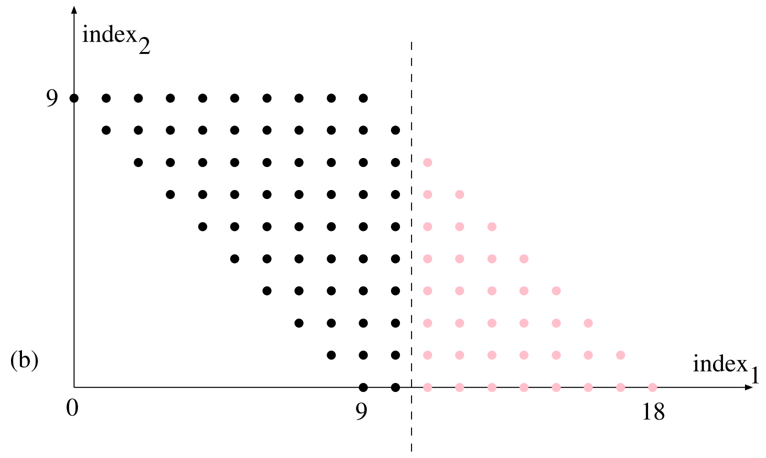

Figure 1: (a) Illustrative example. (b) The array space of signal A. The points represent the A-elements $A[index_1][index_2]$ which are produced in the loop nest. The black points to the left of the dashed line are all alive, representing the A-elements produced till the beginning of the iteration $(i\ ,\ j)$ = (10 , 9), when the first array element is going to be consumed (which is $A[0][9]$).

The iterator space (see Fig. 2) of the array reference $A[2i+3j][5i+j]$ is the **Z**-polytope $P = \{\ \mathbf{i} \in \mathbf{Z}^2 \mid \mathbf{A}\cdot\mathbf{i} \geq \mathbf{b}\ \} =$

$$\left\{ \left[\begin{array}{c} i \\ j \end{array}\right] \in \mathbf{Z}^2 \;\middle|\; \left[\begin{array}{rr} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ 3 & -2 \end{array}\right] \left[\begin{array}{c} i \\ j \end{array}\right] \geq \left[\begin{array}{r} 0 \\ -3 \\ 0 \\ -2 \\ 0 \end{array}\right] \right\}$$

or, in non-matrix format: $P = \{0 \leq i \leq 3\ ,\ 0 \leq j \leq 2\ ,\ 2j \leq 3i,\ \ i,j \in \mathbf{Z}\}$. The index space of the array reference can be expressed in this case as the bounded lattice:

$$\left\{ \left[\begin{array}{c} x \\ y \end{array}\right] = \left[\begin{array}{cc} 2 & 3 \\ 5 & 1 \end{array}\right] \left[\begin{array}{c} i \\ j \end{array}\right] + \left[\begin{array}{c} 0 \\ 0 \end{array}\right] \;\middle|\; \left[\begin{array}{c} i \\ j \end{array}\right] \in P \right\}$$

where x and y are the indices of the array reference. The (black) points of the index space (see Fig. 2) lie inside the **Z**-polytope $\{\ 26 \geq 5x - 2y \geq 0\ ,\ 39 \geq -x + 3y\ ,\ y \geq x\ ,\ x, y \in \mathbf{Z}\}$, the image of the boundary of the iterator space. In this example, each point in the iterator space is mapped to a distinct point of the index space, which is not always the case. $\square$

2.2 Computing a 1D window for a lattice of signals

The computation method employed by De Greef *et al.* consists of a sequence of integer linear programming (ILP) optimizations for each array linearization [5]. Tronçon *et al.* use, basically, sequences of emptiness checks for **Z**-polytopes derived from the code [11].

We are using a different methodology based on the decomposition of the array references in disjoint bounded lattices (1) (see the next subsection). This framework is common to both models. The specific difference is the computation of the 1-dimensional windows for the lattices of signals, which is going to be explained in this section.

Given an array reference or, more general, a bounded lattice (1) whose elements are all alive:

(a) In Tronçon's model [11], the problem is to compute a 1-dimensional window for every index of the array reference (or lattice), that is, the extreme points of the projection of the lattice on every axis. The idea of the algorithm is to find a transformation **S** such that the extreme values of some iterator correspond to the extreme values of some index. In this way, the problem reduces to computing the projection of a **Z**-polytope, which is well-studied [7, 12].

Algorithm 1

Suppose we study the projection on the k-th axis. The k-th index has the expression: $x_k = \mathbf{t}_k \cdot \mathbf{i} + u_k$, where $\mathbf{t}_k$ is the k-th row of the matrix **T** in (1).

Step 1 Let **S** be a unimodular matrix[2] bringing $\mathbf{t}_k$ to the Hermite Normal Form [9]: $[h_1\ 0 \cdots 0] = \mathbf{t}_k \cdot \mathbf{S}$. (If the row $\mathbf{t}_k$ is null, then the window reduces to one point: $x_k^{min} = x_k^{max} = u_k$.)

Step 2 After applying the unimodular transformation **S**, the new iterator polytope becomes: $\bar{P} = \{\ \bar{\mathbf{i}} \in \mathbf{Z}^n \mid \mathbf{A}\cdot\mathbf{S}\cdot\bar{\mathbf{i}} \geq \mathbf{b}\ \}$.

Step 3 Compute the extreme values of $\bar{i}_1$ (denoted $\bar{i}_1^{min}$ and $\bar{i}_1^{max}$) by projecting the polytope $\bar{P}$ on the first axis [7]. Then, $x_k^{min} = h_1\bar{i}_1^{min} + u_k$ and $x_k^{max} = h_1\bar{i}_1^{max} + u_k$. $\square$

The algorithm will be illustrated projecting the array reference from *Example 1* on the first axis and finding the extreme values of the first index x.

From *Example 1*, $[x] = \left[\begin{array}{cc} 2 & 3 \end{array}\right] \left[\begin{array}{c} i \\ j \end{array}\right] + [0]$. The unimodular matrix $\mathbf{S} = \left[\begin{array}{rr} -1 & 3 \\ 1 & -2 \end{array}\right]$ (see, e.g., [9] for building **S**) brings $\mathbf{t}_1 = \left[\begin{array}{cc} 2 & 3 \end{array}\right]$ to the Hermite Normal Form: $\mathbf{t}_1 \cdot \mathbf{S} = \left[\begin{array}{cc} 1 & 0 \end{array}\right]$. Since $\left[\begin{array}{c} i \\ j \end{array}\right] = \left[\begin{array}{rr} -1 & 3 \\ 1 & -2 \end{array}\right] \left[\begin{array}{c} \bar{i} \\ \bar{j} \end{array}\right]$, the initial iterator space P (see *Example 1*) becomes $\bar{P} = \{0 \leq -\bar{i} + 3\bar{j} \leq 3\ ,\ 0 \leq \bar{i} - 2\bar{j} \leq 2\ ,\ 5\bar{i} \leq 13\bar{j},\ \ \bar{i}, \bar{j} \in \mathbf{Z}\}$. Eliminating $\bar{j}$ from these inequalities with a Fourier-Motzkin technique [3], the extreme values of the *exact shadow* [7] of $\bar{P}$ on the first axis are $\bar{i}^{min} = 0$ and $\bar{i}^{max} = 12$,

[2] A square matrix whose determinant is $\pm$ 1.

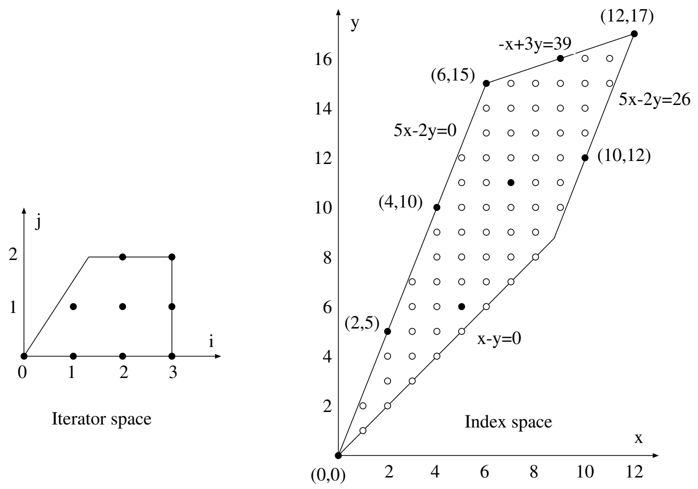

Figure 2: The mapping of the iterator space of the array reference $A[2i+3j][5i+j]$ to its index space (a bounded lattice).

and those extreme points are valid projections (i.e., there exists $\bar{j}$'s such that $(\bar{i}^{min}, \bar{j})$ and $(\bar{i}^{max}, \bar{j})$ satisfy the inequalities defining $\bar{P}$). Since $h_1 = 1$ and $u_1 = 0$, it follows immediately that $x^{min} = 0$ and $x^{max} = 12$, which can be easily observed from Fig. 2. Projecting the lattice on the second axis, the second row of the affine mapping is $\mathbf{t}_2 = \begin{bmatrix} 5 & 1 \end{bmatrix}$ and the unimodular matrix is $\mathbf{S} = \begin{bmatrix} 0 & -1 \\ 1 & 5 \end{bmatrix}$. With a similar computation, it follows that $y^{min} = 0$ and $y^{max} = 17$. Therefore, based only on the array reference $A[2i+3j][5i+j]$ from *Example 1*, the window of the array A is W=(13,18).

(b) In De Greef's model [5], the problem is to compute a 1-dimensional window for every canonical linearization of the array.

Take, for instance, the linearization by row concatenation (in the increasing order of the rows) in the illustrative example from Fig. 1. Then, it can be easily observed that in the bounded lattice of live elements, the ones at the maximum distance from each other are the elements with (lexicographically[3]) minimum and, respectively, maximum indices. For instance, in the lattice of live array elements in Fig. 1 (only the black points, to the left of the dashed line), the elements $A[0][9]$ and $A[10][8]$ are mapped at the maximum distance (which is 99), as explained in Section 1.2. The index vectors $[index_1\ index_2]$=[0 9] and [10 8] are the minimum and, resp., the maximum in lexicographic order from the entire lattice.

Similarly, in the linearization by column concatenation (in the increasing order of the columns), the 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. In the illustrative example from Fig. 1, the elements $A[9][0]$ and $A[9][9]$ are the farthest away from each other (at 171 unit distance) among the live elements.

It follows that finding the points in a lattice having the (lexicographically) minimum and maximum index vectors is crucial for De Greef's model. For any canonical linearization, it is sufficient to apply an index permutation first, followed at the end by the inverse permutation of the resulting index vectors. If in the linearization some dimension is traversed backwards, then a simple transformation reversing the index variation must be also applied.

The idea of the algorithm resembles *Algorithm 1*'s idea: we find a transformation $\mathbf{S}$ such that the minimum (maximum) iterator vector in the iterator polytope is mapped to the minimum (maximum) index vector in the lattice. But those iterator vectors are easier to compute by successive projections of $\mathbf{Z}$-polytopes.

Algorithm 2

The index vectors are given by the affine vector mapping $\mathbf{x} = \mathbf{T} \cdot \mathbf{i} + \mathbf{u}$, the iterator vectors $\mathbf{i}$ satisfying the constraints $\mathbf{A} \cdot \mathbf{i} \geq \mathbf{b}$.

Step 1 Let $\mathbf{S}$ be a unimodular matrix bringing $\mathbf{T}$ to the Hermite Normal Form [9]: $\mathbf{H} = \mathbf{T} \cdot \mathbf{S}$.

Step 2 After applying the unimodular transformation $\mathbf{S}$, the new iterator polytope becomes: $\bar{P} = \{ \bar{\mathbf{i}} \in \mathbf{Z}^n \mid \mathbf{A} \cdot \mathbf{S} \cdot \bar{\mathbf{i}} \geq \mathbf{b} \}$.

Step 3 Compute the maximum (minimum) value of $\bar{i}_1$ (the first element of $\bar{\mathbf{i}}$) by projecting the polytope $\bar{P}$ on the first axis [7]. Then, replacing this value in $\bar{P}$, compute the maximum (minimum) value of $\bar{i}_2$ by projection on the second axis, and so on. The iterator vector whose elements are determined as explained above is the maximum (minimum) iterator vector in lexicographic order, denoted $\bar{\mathbf{i}}^{max}$ (resp., $\bar{\mathbf{i}}^{min}$). Then, $\mathbf{x}^{min} = \mathbf{H} \cdot \bar{\mathbf{i}}^{min} + \mathbf{u}$ and $\mathbf{x}^{max} = \mathbf{H} \cdot \bar{\mathbf{i}}^{max} + \mathbf{u}$. $\Box$

2.3 The global flow of the mapping algorithm

The global flow of the algorithm finding the sizes of the mapping-to-memory windows (relative to some base address which can be decided during the memory allocation phase) for every multi-dimensional signal in an affine behavioral specification will be explained below. Similar to [4, 5, 6, 11], the code is assumed to be in *single-assignment* form, that is, each array element is written at most once (but it can be read an arbitrary number of times). We shall implicitly apply Tronçon's model [11], using *Algorithm 1* (see Section 2.2) in the flow. But this global algorithm can be easily switched to De Greef's model [5], using instead *Algorithm 2* and, also, taking into account the discussion on the index permutation (Section 2.2) in order to test all the canonical linearizations. To illustrate the algorithm, we shall use:

```
Example 2: for $(k = 0;\ k \leq 10;\ k++)$  // Loop nest 1
               for $(l = 0;\ l \leq 5;\ l++)$  $A[k][l] = \cdots;$
           for $(j = 0;\ j \leq 5;\ j++)$     // Loop nest 2
               for $(i = 0;\ i \leq 2j;\ i++)$  $\cdots = A[i][j];$
           for $(i = 1;\ i \leq 5;\ i++)$     // Loop nest 3
               for $(j = 0;\ j \leq i-1;\ j{+}{+})$  $\cdots = A[2i][j+1];$
```

This example is not restrictive containing only one array since the mapping algorithm is applied *independently* for each multi-dimensional array (signal) in the specification.

[3]Let $\mathbf{i} = [i_1, \ldots, i_n]^T$ and $\mathbf{j} = [j_1, \ldots, j_n]^T$ be two vectors. The vector $\mathbf{j}$ is larger lexicographically than $\mathbf{i}$ (written $\mathbf{j} \succ \mathbf{i}$) if $(j_1 > i_1)$, or $(j_1 = i_1$ and $j_2 > i_2)$, $\ldots$, or $(j_1 = i_1, \ldots, j_{n-1} = i_{n-1}$, and $j_n > i_n)$. The $minimum/maximum$ vector from a set of vectors is the smallest/largest vector in the set relative to the lexicographic order.

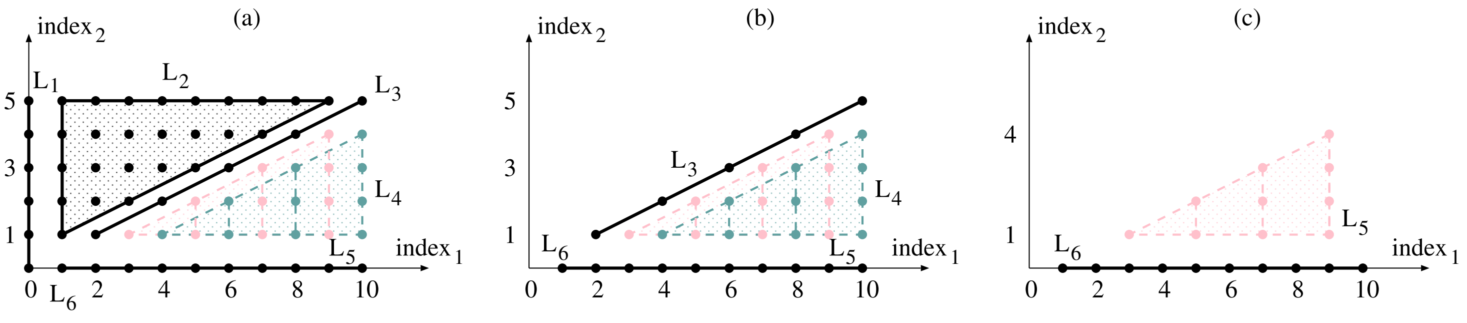

Figure 3: (a) The decomposition of the array space of signal A (*Example 2*) in 6 disjoint lattices. All these lattices are alive after the first loop nest. (b) The live lattices of signal A after the second loop nest. (c) The live lattices after the third loop nest.

Algorithm 3

Step 1 *Decompose signal's array references into disjoint lattices.*

Figure 3(a) shows the result of this decomposition for the three array references of signal A from *Example 2*. This decomposition is obtained analytically. using intersections and differences of lattices – operations quite complex which cannot be explained here due to lack of space. The resulting lattices have the following expressions (in non-matrix format, in order to save space):

$L_1 = \{x = 0,\ y = t \mid 5 \geq t \geq 0\}$
$L_2 = \{x = t_1,\ y = t_2 \mid 5 \geq t_2 \geq 1\ ,\ 2t_2 - 1 \geq t_1 \geq 1\}$
$L_3 = \{x = 2t,\ y = t \mid 5 \geq t \geq 1\}$
$L_4 = \{x = 2t_1 + 2,\ y = t_2 \mid 4 \geq t_1 \geq t_2 \geq 1\}$
$L_5 = \{x = 2t_1 + 1,\ y = t_2 \mid 4 \geq t_1 \geq t_2 \geq 1\}$
$L_6 = \{x = t,\ y = 0 \mid 10 \geq t \geq 1\}$

While the first array reference in *Example 2* is the union of all the six lattices, the second array reference is $L_1 \cup L_2 \cup L_3$, and the third is $L_3 \cup L_4$. (Note that the 2nd and 3rd array references have in common the array elements covered by the lattice L_3.) Assuming that the elements covered by L_1 and L_2 are consumed (accessed as operands for the last time) in the second loop nest, and the elements covered by L_3 and L_4 are consumed in the third loop nest, we can find the elements alive at the borderline between these blocks of code. For instance, the array elements covered by the lattices L_3, L_4, L_5, and L_6 are still alive after the second loop nest (see Fig. 3(b)); the elements covered by L_5 and L_6 are alive after the third loop nest, as well (see Fig. 3(c)).

Step 2 *Using* Algorithm 1*, compute the extreme values of each signal's index for the live elements at the boundaries between blocks of code.*

For instance, after the third loop nest, applying *Algorithm 1* on the live lattices L_5 and L_6, the extreme values of A's first index are $x^{min} = 1$ and $x^{max} = 10$; the extreme values of A's second index are $y^{min} = 0$ and $y^{max} = 4$, as it can be easily noticed from Fig. 3(c). The global extreme values are, obviously, $X^{min} = 0$, $X^{max} = 10$, and $Y^{min} = 0$, $Y^{max} = 5$ (see Fig. 3(a)), resulting a window W=(11,6).

Steps 1 and *2* find the index windows when every block of code either produces or consumes (but not both!) the signal's elements.

Step 3 *Update the extreme values of the signal's indices for the live elements within each block of code where array elements are both produced and consumed.*

In the illustrative example from Fig. 1, A-elements are both produced and consumed in the loop nest. In such a situation, the basic idea is to compute the iterator vectors when array elements are accessed for the last time [13] and, subsequently, apply *Algorithm 1* to the live lattices corresponding to those iterations. For instance, the element $A[0][9]$ is consumed in the iteration $(i\ ,\ j)$ = (10 , 9). The corresponding lattice produced *before* that iteration is $\{x = i,\ y = j \mid 10 \geq i \geq 0,\ 9 \geq j \geq 0,\ 18 \geq i+j \geq 9\}$ (covering the black points in Fig. 1); *Algorithm 1* yields $x^{min} = 0$, $x^{max} = 10$, and $y^{min} = 0$, $y^{max} = 9$.

3 Experimental results

A software tool performing the mapping to the data memory of the (multi-dimensional) arrays from a given algorithmic specification has been implemented in C++, incorporating the algorithms described in this paper.

Table 1 summarizes the results of our experiments, carried out on a PC with a 1.85 GHz Athlon XP processor and 512 MB memory. The benchmark tests are multimedia applications and typical algebraic kernels used in signal processing. Columns 2 and 3 display the numbers of array references and scalars in the specification code. Columns 4 and 5 show the data memory size (i.e., the total window sizes of the arrays) when the mapping is done according to Tronçon's model [11] and according to De Greef's model [5], respectively. The CPU times were obtained using the algorithms described in this paper. Column 6 displays the sum of the *minimum* array windows[4] (optimized intra-array mapping). Column 7 shows the minimum storage requirements of the applications, when the memory sharing between different arrays is optimized as well. The data in these last two columns were obtained with another tool of ours presented in [13].

The main conclusions of these experiments are the following:

[4]The minimum array window of signal A from the illustrative example in Fig. 1 is 80, that is, the minimum number of memory locations enabling the code execution. Indeed, there are at most 80 A-elements simultaneously alive and this happens when the iteration (i,j)=(14,10) is completed. Note that Tronçon's and De Greef's models give window sizes of 110 and 100, respectively (see Section 1.2).

Application	# Array references	# Scalars	Mem. size / CPU using model [11]	Mem. size / CPU using model [5]	Memory size (optimal intra-array mapping)	Min. memory size (optimal inter+intra array mapping)
Motion detection	11	318,367	9,525 / 12 sec	9,636 / 20 sec	9,525	9,524
Regularity detection	19	4,752	4,353 / 3 sec	3,879 / 9 sec	2,449	2,304
2D Gaussian blur filter	20	177,167	48,646 / 34 sec	50,448 / 76 sec	48,646	16,515
SVD updating	87	386,472	17,554 / 18 sec	16,754 / 48 sec	10,204	8,725
Voice coder	232	33,619	13,104 / 14 sec	13,224 / 25 sec	12,963	11,890

Table 1: Experimental results. The computation times were obtained with our implementation. Columns 6, 7 were computed with [13].

1. *Our memory exploration tools* (the one described in this paper together with the one presented in [13]) *can provide an accurate evaluation of the signal-to-memory mapping models.* While [5] and [11] compute only the size of their array windows, they are unable to provide any consistent information on *how good their models are*, that is, how large is the oversize of their array windows in comparison to the minimal windows (col. 6) and storage requirements (col. 7). While a minimum array window may be difficult to achieve in practice (in many cases, requiring a significantly more complex hardware), a signal-to-memory mapping model actually trades-off an excess of storage for a less complex address generation hardware. Our methodology allows to compute this amount of extra storage, providing useful data for the system-level exploration design phase. Table 1 shows that there are applications (like the motion detection, the Gaussian blur filter) where both mapping models give very good solutions, close (or, even, equal) to the optimal array windows (col. 6). On the other hand, there are also applications where the window sizes computed according to the models are significantly larger than the optimal ones (e.g., 72% and 64% larger for the SVD updating); in such cases, the designer must take the decision whether to go along with any of the models or imagine a different solution.

2. *Trončon's model gives, in general, smaller window sizes than De Greef's model.* This happens especially when the array space contains *holes* (see, for instance, Fig. 2) and, also, when the size of the array can be reduced in any dimension since, in such cases, any linearization will contain a number of unused array elements. On the other hand, there are cases when De Greef's model finds a linearization yielding a smaller size window, as it happens in the illustrative example (Fig. 1) or in the SVD updating application. Also, De Greef's model runs regularly slower in our implementation since there are several linearizations to be analyzed, the rest of the computations being similar to the other model.

3. *The implementation of the mapping models based on the algorithms presented in this paper is several times faster than the original implementations.* The computation times reported in [5] and [11] are typically of the order of minutes, whereas our implementation runs for the same examples or of similar complexity in tens of seconds at most. For instance, the *voice coding* application was processed by [11] in over 25 minutes (using a 300 MHz Pentium II) and by [5] in over 27 minutes (unspecified platform); in contrast, we did the computations in only 14 seconds on a 1.85 GHz Athlon XP and in 38 seconds on a Sun Ultra 10 workstation. We can safely state that this implementation is several times faster.

4 Conclusions

This paper has addressed the problem of intra-array mapping of the multi-dimensional signals to the data memory in multimedia behavioral specifications. This paper has thoroughly analyzed two mapping models, assessing their effectiveness and proposing more efficient algorithms which led to an implementation several times faster than the original ones.

References

[1] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, A. Vandecappelle, *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design*, Kluwer Academic Publishers, Boston, 1998.

[2] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P. G. Kjeldsberg, T. Van Achteren, and T. Omnes, *Data Access and Storage Management for Embedded Programmable Processors*, Boston, USA: Kluwer Acad. Publ., 2002.

[3] G.B. Dantzig, B.C. Eaves, "Fourier-Motzkin elimination and its dual," *J. Combinatorial Theory (A)*, vol. 14, pp. 288-297, 1973.

[4] A. Darte, R. Schreiber, G. Villard, "Lattice-based memory allocation," *IEEE Trans. Computers*, vol. 54, pp. 1242-1257, Oct. 2005.

[5] E. De Greef, F. Catthoor, H. De Man, "Memory size reduction through storage order optimization for embedded parallel multimedia applications", special issue on "Parallel Processing and Multimedia" (ed. A. Krikelis), in *Parallel Computing*, Elsevier, vol. 23, no. 12, pp. 1811-1837, Dec. 1997.

[6] V. Lefebvre, P. Feautrier, "Automatic storage management for parallel programs," *Parallel Computing*, vol. 24, pp. 649-671, 1998.

[7] W. Pugh, "A practical algorithm for exact array dependence analysis," *Comm. of the ACM*, vol. 35, no. 8, pp. 102-114, Aug. 1992.

[8] F. Quilleré, S. Rajopadhye, "Optimizing memory usage in the polyhedral model," *ACM Trans. Programming Languages and Syst.*, vol. 22, no. 5, pp. 773-815, 2000.

[9] A. Schrijver, *Theory of Linear and Integer Programming*, John Wiley, New York, 1986.

[10] L. Thiele, "Compiler techniques for massive parallel architectures," in *State-of-the-art in Computer Science*, Kluwer Acad. Publ., 1992.

[11] R. Trončon, M. Bruynooghe, G. Janssens, F. Catthoor, "Storage size reduction by in-place mapping of arrays," *Verification, Model Checking and Abstract Interpretation*, pp. 167-181, 2002.

[12] S. Verdoolaege, K. Beyls, M. Bruynooghe, F. Catthoor, "Experiences with enumeration of integer projections of parametric polytopes," in *Compiler Construction: 14th Int. Conf.*, pp. 91-105, 2005.

[13] H. Zhu, I.I. Luican, F. Balasa, "Memory size computation for multimedia processing applications," *Proc. Asia & South-Pacific Design Automation Conf.*, pp. 802-807, Yokohama, Japan, Jan. 2006.