Computation of the minimum data storage and applications in memory management for multimedia signal processing[1]

Ilie I. Luican[a], Hongwei Zhu[b] and Florin Balasa[c,*]
[a]*Department of Computer Science, University of Illinois at Chicago, Chicago, IL, USA*
[b]*ARM, Inc., Sunnyvale, CA, USA*
[c]*Department of Computer Science, Southern Utah University, Cedar City, UT, USA*

Abstract. The amount of the data storage in signal processing systems, whose behavior is described by loop-organized algorithmic specifications, has an important impact on the overall energy consumption, chip area, as well as system performance. This paper presents a methodology based on lattices [25] which can be used to address several memory management tasks for applications with high-level specifications, where the main data structures are multidimensional arrays. This methodology was used in the past for the exact computation of the minimum data storage in applications with procedural, affine specifications [2]. The paper discusses two applications of that technique in the memory management of data-dominated signal processing systems: (1) the evaluation of the impact of loop transformations on the data storage, and (2) the assessment and efficient implementation of models of mapping multidimensional signals into the physical memory.

Keywords: Memory management, memory size computation, signal-to-memory mapping, loop transformations, multimedia signal processing

1. Introduction

In many signal processing systems, particularly in the multimedia and telecommunication 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 product.

The behavior of these targeted VLSI systems, synthesized to execute mainly data-intensive 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). The data structures are multidimensional arrays whose indexes in the code are affine functions of the surrounding loop iterators. The class of specifications with these characteristics are often called *affine* specifications [4]. A piece of code in this class is shown in Fig. 1(a) and will be used along the paper as an illustrative example.[2]

For several decades, researchers have worked on different approaches for estimating or computing the *minimum* amount of memory locations necessary to store the data during the execution of a multidimensional signal processing application. The typical assumption

[*]Corresponding author. E-mail: balasa@suu.edu.
[1]This research was sponsored by the US National Science Foundation (DAP 0133318).

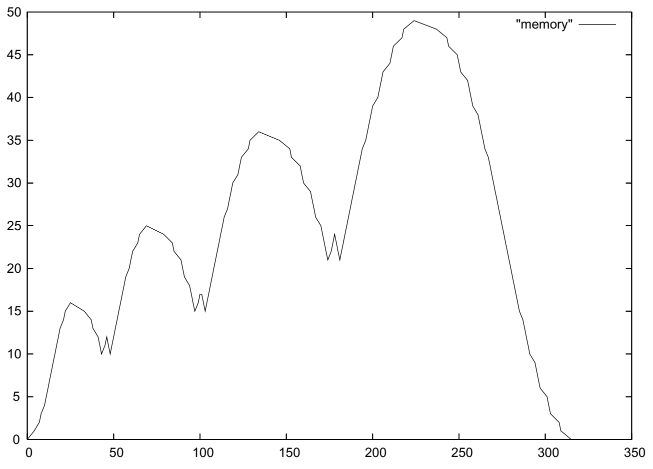
[2]Since the focus will be on the arrays A, B, C, and D, the left-hand side operand is irrelevant in the last assignment. It is assumed that no $A/B/C/D$- array element is used in the remainder of the code.

ISSN 1069-2509/08/$17.00 © 2008 – IOS Press and the author(s). All rights reserved

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

for ( i=0; i<=6; i++ )                                          //   1st loop nest
  for ( j=0; j<=3; j++)
    if ( 3<=i+j  &&  i+j<=6 )   A[i][j] = 1;
for ( i=0; i<=8; i++ )                                          //   2nd loop nest
  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++ )                                         //   3rd loop nest
  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++ )                                         //   4th loop nest
  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=7; i<=19; i++ )                                         //   5th loop nest
  for ( j=0; j<=6; j++)
    if ( 13<=i+j  &&  i+j<=19 )   ... = D[i-7][j];
```

(a)



(b)

Fig. 1. (a) Illustrative example of affine specification. (b) Trace of the data storage requirement during the code execution. The abscissae are numbers of elementary loop iterations and the ordinates are memory locations necessary to store the data.

is that any scalar (array element) must be stored only during its lifetime – from the moment when it is produced until it is used for the last time as an operand. For instance, the storage requirement for the illustrative example in Fig. 1(a) is 49 memory locations, although the total number of array elements in the code is significantly larger. This is due to the fact that scalars having disjoint lifetimes can share the same memory location. The variation of the occupied data storage during the code execution is displayed in Fig. 1(b).

Most of the initial work was done at scalar level due to the register-transfer behavioral specifications of the earlier digital systems. After being modeled as a clique partitioning problem [29], the register allocation and assignment have been optimally solved for nonrepetitive schedules, when the life-time of all the scalars is fully determined [9]. The similarity with the problem of channel routing without vertical constraints [12] has been exploited in order to determine the minimum register requirements (similar to the number of tracks in a channel), and to optimally assign the scalars to registers (similar to the assignment of one-segment wires to tracks) in polynomial time by using the *left-edge* algorithm [15]. A suboptimal extension for repetitive and conditional schedules has been proposed in [10]. A lower bound on the register cost can be found at any stage of the scheduling process using force-directed scheduling [21]. Integer Linear Programming (ILP) techniques are used in [8] to find the optimal number of memory locations during a simultaneous scheduling and allocation of functional units, registers, and buses. Employing circular graphs [19,26] proposed optimal register allocation/assignment solutions for repetitive schedules. A lower bound for the register count is found in [18] without fixing the schedule, through the use of ASAP and ALAP constraints on the operations. A good overview of these techniques can be found in [7].

Common to all the scalar-based techniques is that they are computationally expensive or even fail altogether when used by flattening large multidimensional arrays, each array element being considered a separate scalar. As already mentioned, the nowadays data-intensive signal processing applications are described by high-level, loop-organized, algorithmic specifications whose main data structures are typically multidimensional arrays. Flattening the arrays from the specification of a video or image processing application would typically result in many thousands or even millions of scalars.

To overcome the shortcomings of the scalar-based techniques, several research teams have tried to split the arrays into suitable units before or as a part of the estimation. This reduces the number of elements the estimator must handle compared to the scalar-based methodology. We shall now review different published contributions using this strategy, starting with techniques that assume a procedural execution of the application code, that is, where the loop structure and sequence of instructions induce the (fixed) execution ordering.

In [30], a production time axis is created for each array. This models the relative production and consumption time, or date, of the individual array accesses. The difference between these two dates equals the number of array elements produced between them. The maximum difference found for any two depending instances gives the storage requirement for this array. The total storage requirement is the sum of the requirements for each array. An ILP approach is used to find the date differences. Since each array is treated separately, only the internal in-place mapping of an array (intra-array in-place) is considered; the possibility of mapping arrays in-place of each other (inter-array in-place or memory sharing between different arrays) is not exploited.

Grun et al. use the data-dependency relations between the array references in the code to find the number of array elements produced or consumed by each assignment [11]. The storage requirement at the end of a loop equals the storage requirement at the beginning of the loop, plus the number of elements produced within the loop, minus the number of elements consumed within the loop. The upper bound for the occupied memory size within a loop is computed by producing as many array elements as possible before any elements are consumed. The lower bound is found with the opposite reasoning. From this, a memory trace of bounding rectangles as a function of time is found. The total storage requirement equals the peak bounding rectangle. If the difference between the upper and lower bounds for this critical rectangle is too large, better estimates can be achieved by splitting the corresponding loop into two loops and rerunning the estimation. In the worst-case situation, a full loop-unrolling is necessary to achieve a satisfactory estimate.

Zhao et al. introduced a methodology for so-called "exact" memory size estimation for array computation [32]. It is based on live variable analysis and integer point counting for intersection/union of mappings of parameterized polytopes. In this context, a polytope is the intersection of a finite set of half-spaces and may be specified as the set of solutions to a system of linear inequalities. It is shown that it is only necessary to find the number of live variables for one statement in each innermost loop nest to get the minimum memory size estimate. The live variable analysis is performed for each iteration of the loops however, which makes it computationally hard for large multidimensional loop nests.

In [24], the specifications are limited only to perfectly nested loops. A reference window is used for each array. At any moment during execution, the window contains array elements that have already been referenced and will also be referenced in the future. These elements are hence stored in the local memory. The maximal window size gives the memory requirement for the array. If multiple arrays exist, the maximum reference window size equals the sum of the windows for individual arrays. Inter-array in-place is consequently not considered.

All the techniques above estimate the memory size assuming a single memory. Hu et al. perform hierarchical memory size estimation, taking data reuse and memory hierarchy allocation into account [13]. In-place mapping is not incorporated in the current version, but is indicated as part of future work.

In contrast to the array-based methods described so far in this section, the storage requirement estimation technique presented in [1] assumes a non-procedural execution of the application code, that is, the execution ordering is still not (completely) fixed. The approach is based on traversing a dependence graph (based on an extended data dependence analysis) resulting in a number of non-overlapping array sections (so called basic sets) and the dependences between them. The basic set sizes and the numbers of the dependences are found using an efficient lattice point counting technique. The maximal combined size of simultaneously alive basic sets found through a greedy graph traversal gives an estimation of the storage requirement.

The estimation technique described in [14] assumes a *partially fixed* execution ordering. The authors employ a data dependence analysis similar to [1], but with a significant improvement: they add the capability of taking into account available execution ordering information (based mainly on loop interchanges), thus avoiding the possible overestimates due to the total ordering freedom (less the data dependence constraints).

This paper discusses the basic ideas of a polyhedral framework, operating mainly with polytopes and lattices [25], which can be used to address several memory management tasks in multimedia and multidimensional signal processing applications having high-level specifications, where the main data structures are multidimensional arrays. In particular, this methodology has been recently used as the core of a non-scalar technique for *computing exactly* the minimum size of the data memory in multidimensional signal processing algorithms, when the specifications are *procedural*,[3] i.e.,

[3]This assumption, adopted by several previous works [24,30,32] which perform only *approximate* evaluations, is based on the fact

the execution flow is induced by the loop structure and the instruction order [2].

Moreover, the paper presents two applications of this technique in the memory management of the multidimensional signal processing systems. Note that the previous works on memory evaluation do not discuss this issue (that is, how the evaluation of the amount of data memory is used in the design flow), mentioning rather vaguely that this step is necessary in the early design stage of the memory subsystem. The first application is typical to the system-level exploration phase, assisting the designer in the evaluation of different code (and, especially, loop) transformations concerning their impact on the data storage.

The second application is the mapping of the array elements from the application code to physical addresses in the data memory. (An overview on the previous signal-to-memory mapping techniques will be given in Section 4.) The paper will show that the aforementioned polyhedral framework can be used to obtain efficient implementations of different mapping models. Even more important, the computation of the minimum data storage can be used to better assess the global quality of the mapping, which is not attempted by any of the previous works.

The rest of the paper is organized as follows. Section 2 introduces the basic mathematical concepts of our formal methodology for addressing memory management tasks. Section 3 discusses the first memory management application: the exploration of functionally equivalent specifications, having different storage characteristics. Section 4 addresses the second memory management application – the assessment of different signal-to-memory mapping models. Section 5 presents implementation aspects and experimental results. Section 6 summarizes the conclusions of this research.

2. Polyhedral framework for memory management

An array reference can be typically represented as the image of an affine vector function $\mathbf{i} \longmapsto \mathbf{T}\cdot\mathbf{i}+\mathbf{u}$ over a **Z**-polytope (its iterator space) $\{\,\mathbf{i}\in\mathbf{Z}^n \mid \mathbf{A}\cdot\mathbf{i}\geqslant\mathbf{b}\,\}$, therefore, a *lattice* [25] which is *linearly bounded* [27].

that in present industrial design, the specification usually includes a full fixation of the execution ordering. Even if this is not the case, the designer can still explore different equivalent specifications, as it will be exemplified in Section 3.

For instance, the array reference $A[i+2*j+3][j+2*k]$ from the loop nest

Example 1.
```
  for (i=0; i<=2; i++)
      for (j=0; j<=3; j++)
         for (k=0; k<=4; k++)
            if ( 6*i+4*j+3*k ⩽12 )
                ··· A[i+2*j+3][j+2*k] ···
```

has the iterator space

$$P=\left\{\begin{bmatrix}i\\j\\k\end{bmatrix}\in\mathbf{Z}^3\left|\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\\-6&-4&-3\end{bmatrix}\begin{bmatrix}i\\j\\k\end{bmatrix}\geqslant\begin{bmatrix}0\\0\\0\\-12\end{bmatrix}\right.\right\}.$$

(The inequalities $i\leqslant 2$, $j\leqslant 3$, and $k\leqslant 4$ are redundant.)

The A-elements of the array reference have the indices x, y:

$$\left\{\begin{bmatrix}x\\y\end{bmatrix}=\mathbf{T}\cdot\mathbf{i}+\mathbf{u}=\begin{bmatrix}1&2&0\\0&1&2\end{bmatrix}\begin{bmatrix}i\\j\\k\end{bmatrix}+\begin{bmatrix}3\\0\end{bmatrix}\left|\begin{bmatrix}i\\j\\k\end{bmatrix}\in P\right.\right\}.$$

The points of the index space lie inside the **Z**-polytope $\{x\geqslant 3\,,\ y\geqslant 0\,,\ 3x-4y\leqslant 15\,,\ 5x+6y\leqslant 63\,,\ x,y\in\mathbf{Z}\}$, whose boundary is the image of the boundary of the iterator space P (see Fig. 2). However, it can be shown that only those points (x,y) satisfying also the inequalities $-6x+8y\geqslant 19k-30$, $x-2y\geqslant -4k+3$, and $y\geqslant 2k\geqslant 0$, for some positive integer k, belong to the index space; these are the black points in the right quadrilateral from Fig. 2. In this illustrative example, each point in the iterator space is mapped to a distinct point of the index space, but this is not always the case.

Two operations are relevant in our framework: the *intersection* and the *difference* of lattices. While the intersection of lattices was addressed by other works as well (in different contexts, though) like, for instance [27], the *difference* operation is far more difficult. These operations are formally explained in [2]. Due to the fact that the lattice-based framework is the core of our memory management methodology and knowing its capabilities is critical for understanding the rest of the paper, the *intersection* and *difference* operations will be informally explained below.

2.1. The intersection of two linearly bounded lattices

If two lattices are derived from the same multidimensional signal, their matrices $\mathbf{T}$ and vectors $\mathbf{u}$ from their

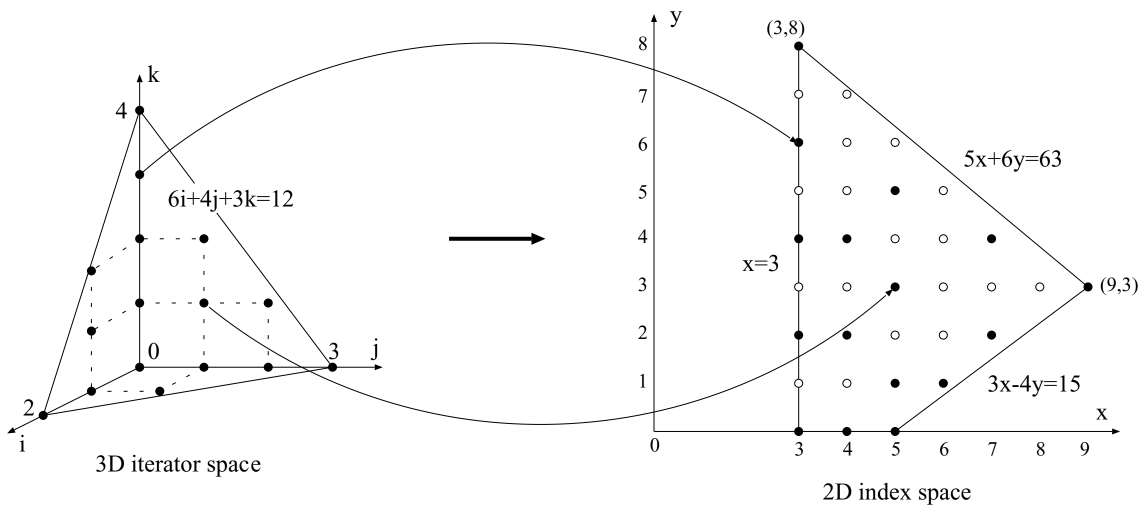


Fig. 2. The mapping of the iterator space into the index space of the array reference $A[i+2*j+3][j+2*k]$.

mapping have obviously the same number of rows – the dimension of the indexed signal. Intersecting the two linearly bounded lattices means, first of all, solving a linear Diophantine system[4] having the elements of the iterator vector as unknowns. If the system has no solution, the intersection is empty. Otherwise, the solution of the Diophantine system is the image of an affine vector function like any other lattice [25]. If the set of coalesced constraints of the two lattices has integer solutions, then the intersection is a new lattice linearly bounded. Otherwise, the intersection is empty.

2.2. The difference of two lattices

Since the difference of two lattices is not always a lattice (as an illustrative example will show below), the goal is to determine a *cover* of the difference set, therefore a set of lattices whose union be equal to the difference. The *difference* operation (see [2] for a theoretical description), will be explained using an example:

Example 2.
```
 for (k=0; k<=6; k++)
     for (l=0; l<=18; l++)
        ··· A[k][l] ···
for (i=0; i<=2; i++)
    for (j=0; j<=3; j++)
       ··· A[3*i][5*i+2*j] ···
```

[4]Finding the integer solutions of the system. Solving a linear Diophantine system was proven to be of polynomial complexity, all the known methods being based on bringing the system matrix to the Hermite Normal Form [25].

The index space of the array reference `A[k][l]` can be represented as

$$\begin{aligned} Lbl_1 &= \{x=k,\ y=l \mid 6 \geqslant k \geqslant 0\ ,\ 18 \geqslant l \geqslant 0\} \\ &= \{6 \geqslant x \geqslant 0\ ,\ 18 \geqslant y \geqslant 0,\ x,y \in \mathbf{Z}\}, \end{aligned}$$

and it is shown in Fig. 3(a).

The index space of the array reference `A[3*i][5*i+2*j]` can be represented as

$$Lbl_2 = \left\{ \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3 & 0 \\ 5 & 2 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \middle| \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \geqslant \begin{bmatrix} 0 \\ -2 \\ 0 \\ -3 \end{bmatrix} \right\}$$

As $\mathbf{T}=\begin{bmatrix} 3 & 0 \\ 5 & 2 \end{bmatrix}$, $\mathbf{T}^{-1}=\frac{1}{6}\begin{bmatrix} 2 & 0 \\ -5 & 3 \end{bmatrix}$, $\mathbf{u}=\begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $\mathbf{x}=\begin{bmatrix} x \\ y \end{bmatrix}$, the index space of the array reference can be represented using the indexes x and y as coordinates: from $\mathbf{A}\cdot\mathbf{i} \geqslant \mathbf{b}$, it follows $\mathbf{A}\cdot\mathbf{T}^{-1}\cdot(\mathbf{x}-\mathbf{u}) \geqslant \mathbf{b}$, that is, the inequalities $6 \geqslant x \geqslant 0, 18 \geqslant -5x+3y \geqslant 0$, representing the quadrilateral in Fig. 3(b). Not all the points (x,y)$\in$ $\mathbf{Z}^2$ in the quadrilateral can be index values of the array reference. Only those points satisfying the divisibility conditions: $6|2x$ (or $3|x$) and $6|(-5x+3y)$ (that is, x is a multiple of 3, and 6 divides exactly $-5x+3y$) belong to the index space. These divisibility conditions result from the necessity that the elements of the iterator vector $\mathbf{i}=\mathbf{T}^{-1}\cdot(\mathbf{x}-\mathbf{u})$ be integer. In conclusion,

$$Lbl_2 = \{x = 3i,\ y = 5i + 2j | 2 \geqslant i \geqslant 0, 3 \geqslant j \geqslant 0\}$$
$$= \{6 \geqslant x \geqslant 0\ ,\ 18 \geqslant -5x + 3y \geqslant 0, 3|x,$$
$$6| -5x + 3y, x, y \in \mathbf{Z}\},$$

and it is shown in Fig. 3(b).

Taking one of the 4 inequalities of Lbl_2 and adding its negated inequality to the minuend Lbl_1 will create a lattice which (if not empty) is disjoint from Lbl_2 and is included in Lbl_1. For instance, negating the inequality $18 \geqslant -5x+3y$ from Lbl_2, we obtain $19 \leqslant -5x+3y$, or $19 \leqslant -5k + 3l$ with the iterators of Lbl_1. Adding it to the iterator space of Lbl_1, we obtain

$$L1 = \{x = k, y = l | 6 \geqslant k \geqslant 0, 18 \geqslant l,$$
$$3l \geqslant 5k + 19\}$$

shown in Fig. 3(c).

Negating the inequality $-5x + 3y \geqslant 0$, we obtain

$$L2 = \{x = k, y = l | 6 \geqslant k \geqslant 0, l \geqslant 0,$$
$$5k - 1 \geqslant 3l\ \}.$$

Empty lattices result by negating the other inequalities ($6 \geqslant x \geqslant 0$).

The other lattices in the difference must violate at least one of the divisibility conditions $3|x$ and $6|(-5x + 3y)$. To obtain them, we simply replace the vector **u** in Lbl_2, keeping the same periodicity of the index space from Lbl_2 (which here is 3 and 2 along the two axes), but doing a translation along the axes. Therefore, taking

$$\mathbf{u} = \begin{bmatrix} k_1 \\ k_2 \end{bmatrix} \neq \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

where $k_1 = 0, 1, 2$ and $k_2 = 0, 1$, five new lattices are obtained. For instance, choosing $(k_1, k_2) = (1,0)$, replacing $x = 3i + k_1$ and $y = 5i + 2j + k_2$ in the inequalities $6 \geqslant x \geqslant 0, 18 \geqslant -5x + 3y \geqslant 0$ of Lbl_2, we get $1 \geqslant i \geqslant 0, 3 \geqslant j \geqslant 1$. Therefore,

$$L3 = \{x = 3i + 1,\ y = 5i + 2j\ |1 \geqslant i \geqslant 0,$$
$$3 \geqslant j \geqslant 1\}$$

is included in $Lbl_1 - Lbl_2$, and it is shown in Fig. 3(c) with 6 gray points. The other four lattices are:

$$L4 = \{x = 3i + 2, y = 5i + 2j | 1 \geqslant i \geqslant 0,$$
$$4 \geqslant j \geqslant 2\}$$
$$L5 = \{x = 3i, y = 5i + 2j + 1 | 2 \geqslant i \geqslant 0,$$
$$2 \geqslant j \geqslant 0\}$$
$$L6 = \{x = 3i + 1, y = 5i + 2j + 1 | 1 \geqslant i \geqslant 0,$$
$$3 \geqslant j \geqslant 1\}$$
$$L7 = \{x = 3i + 2,\ y = 5i + 2j + 1 | 1 \geqslant i \geqslant 0,$$
$$4 \geqslant j \geqslant 2\}$$

Note that the decomposition is minimal (although not unique): it is not possible to obtain a decomposition of $Lbl_1 - Lbl_2$ with less than 7 lattices for this example – which is a "difficult" one! In most of the practical cases encountered, the difference can be represented as only one single lattice and, if this is the case, the algorithm (informally explained above) will find it. Otherwise, in the general case, the minimality cannot be guaranteed, unless all the combinations of inequalities can be negated, instead of selecting only one at a time. This would increase the computation time, without practical benefits. □

These basic operations can be used to decompose all the array references from the application code into *disjoint* bounded lattices. This latter operation has a very significant consequence, reducing the level of difficulty of the memory management problems: it is equivalent to dealing with algorithmic specifications where all the array references are disjoint from each other. This idea has been recently used for the exact computation of the minimum data storage in affine, procedural specifications [2].

The next two sections will focus on applications of this technique, as well as of the polyhedral framework, in the memory management of signal processing systems.

3. The exploration of functionally equivalent specifications

The tool implemented based on the algorithm computing the minimum data storage – described in Section 2 – could be easily adapted to generate the memory trace during the execution of the application code. The memory traces generated in Fig. 4(a) display the data storage variation for Durbin's algorithm, an algebraic kernel used in many signal processing applications for solving Toeplitz systems of equations. The abscissae are the numbers of datapath instructions in the code; the ordinates are data memory locations in use. Figure 4(b) displays the variation of the storage requirement for a dynamic programming application. The memory trace generated in Fig. 4(c) shows a detail of the data storage variation during the execution of a 2D Gaussian blur filter algorithm from a medical image processing

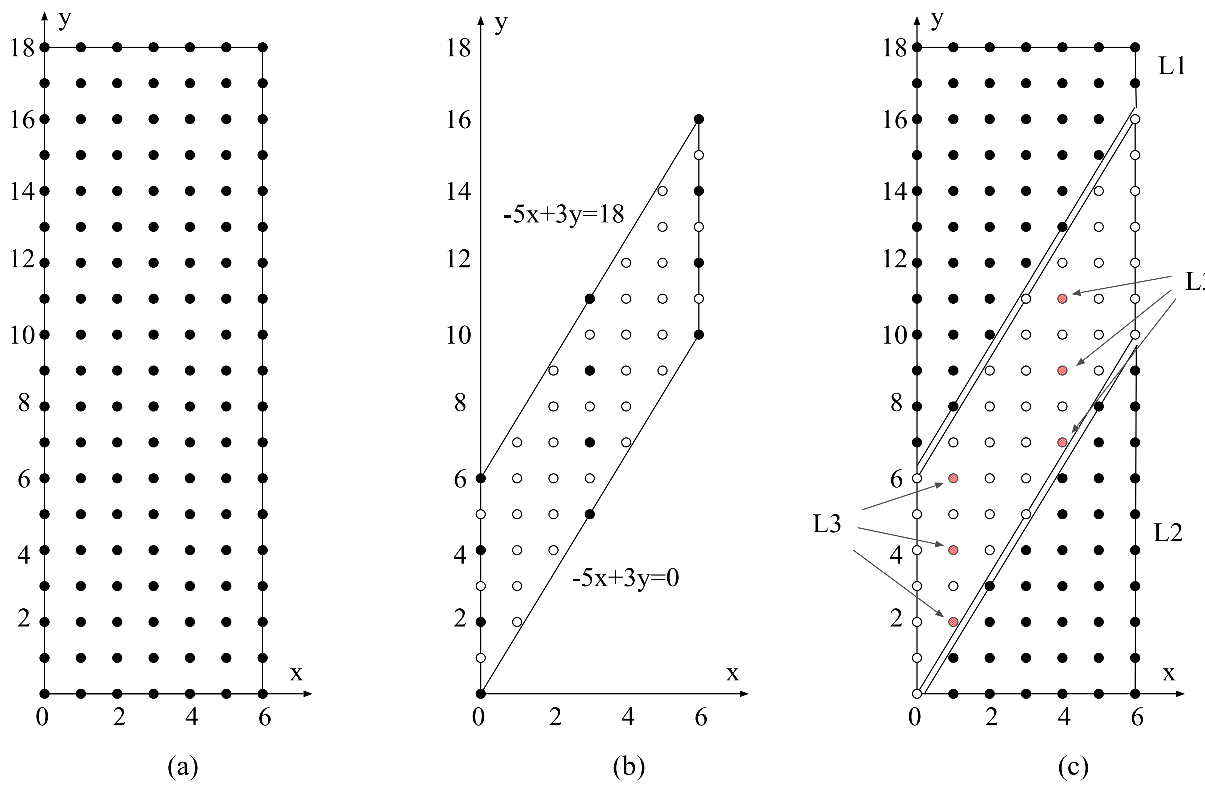


Fig. 3. (a) The index space (the lattice Lbl_1) of the array reference `A[k][l]` in *Example 2*. (b) The 12 black points are the index space (the lattice Lbl_2) of the array reference `A[3*i][5*i+2*j]`. (c) Computation of the difference $Lbl_1 - Lbl_2$. The difference has (at least) 7 disjoint lattices as components. Two components are $L1$ (all the lattice points in the upper quadrilateral) and $L2$ (all the lattice points in the lower triangle). Another component is the lattice $L3$ that covers the 6 gray points in the middle area. The other lattices $L4$-$L7$ cover the middle area as well, except the points in Lbl_2.

application which extracts contours from tomography images in order to detect brain tumors.

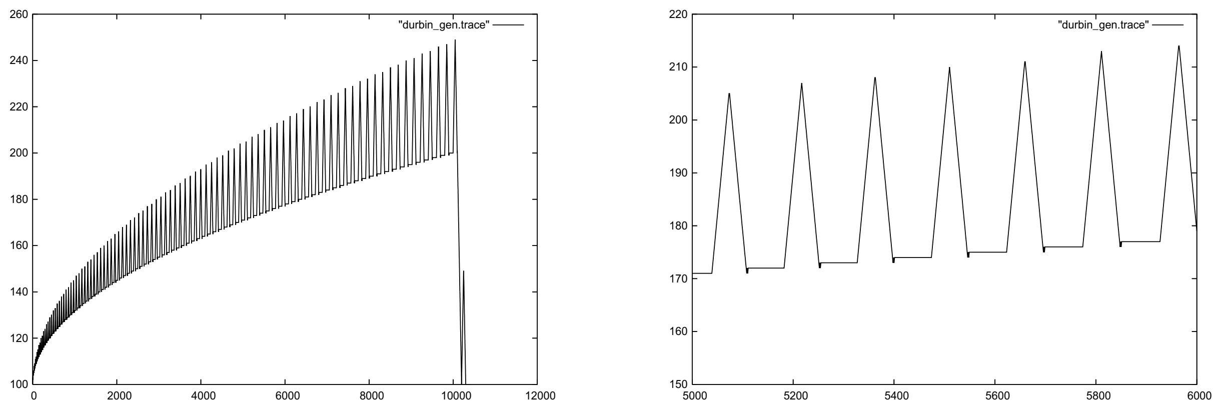
The computation of the minimum data storage is also useful in evaluating the impact of different code (and, in particular, loop) transformations on the data storage. For instance, the minimum memory size needed by the array A in the exemplifying code with two loop nests from Fig. 5 is 4,096 locations. The variation of the storage requirement has a simple pattern: in the first loop nest, it increases uniformly due to the production of the A-elements; in the second loop nest, it decreases uniformly due to the consumption of the same elements, as shown in the first graph. After the fusion of the nested loops, the storage requirement decreases to 3,104 locations, the new trace of the memory variation being displayed in the second graph of the figure. Interchanging the loops drastically decreases the storage requirements (with 98%) to only 64 locations, the final trace being the third graph shown in Fig. 5.

Different variants of code of a same application can be compared one against another in storage point of view, without the need of performing a proper memory allocation for each variant – a significantly more expensive solution.

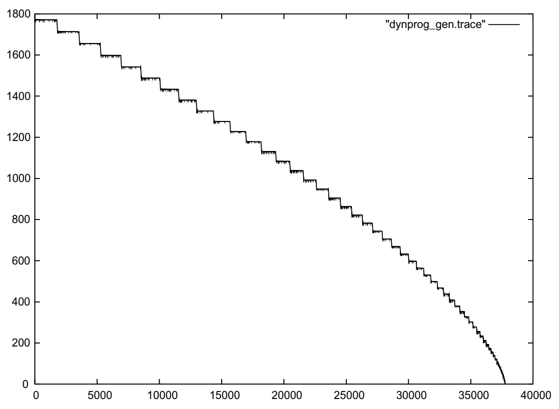
4. Mapping multidimensional arrays to the physical memory

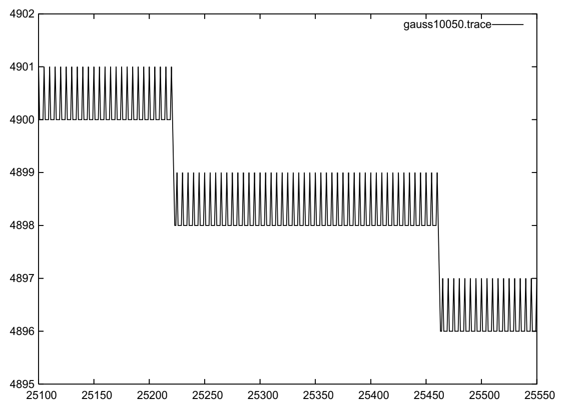
The minimum data storage represents the *tight lower bound* for which the execution of the code is still possible. However, in practice, this amount of storage is difficult to reach (although still possible!) since it would require a complex hardware for address generation. Instead, signal-to-memory mapping techniques are used to compute the physical addresses in the data memory for the array elements in the application code. These mapping models actually trade-off an excess of storage for a less complex address generation hardware.

In many digital signal processing (DSP) applications, the array patterns are predictable, regular, and periodic. A sequencer-based architecture (see Fig. 6) can be



(a)





(b) (c)

Fig. 4. (a) Memory traces for the execution of Durbin's algorithm ($N = 100$). The first graph is the entire trace, the second is a detail. (b) The variation of storage requirement for a dynamic programming application. (c) Memory trace for the execution of a 2D Gaussian blur filter algorithm ($N = 100$, $M = 50$). The graph is a detailed trace, covering the 4-th inner-loop iteration in the part of the code performing the "vertical blur."

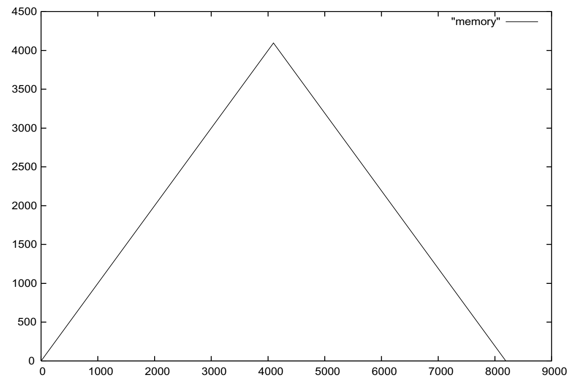
used for pipelined memory accesses in streamed data applications [20]. Moreover, a sequencer-based architecture can be used for dynamic address computations, as well. One possibility is that the addresses be computed inside the datapath unit and, then, transferred – using common data buses – to the memory sequencer, composed of a memory access scheduler, a dynamic address controller, an address generator, and an address translation table (transforming the logical addresses into physical ones) [17]. A second possibility, more expensive but more energetically efficient and providing an increased performance, is to have a specialized datapath internal to the sequencer such that the transfers between the datapath and the sequencer be reduced. The computation resources inside the sequencer can typically perform operations like addition, multiplication, increment, modulus.

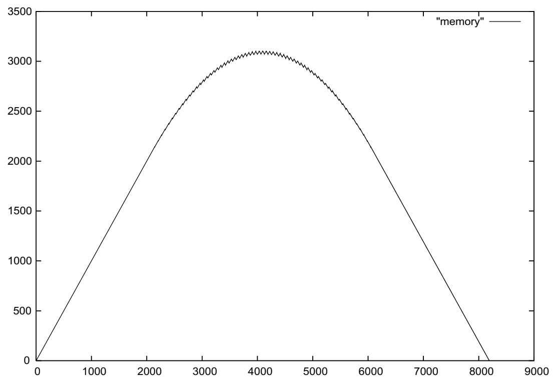
A brief overview of signal-to-memory mapping techniques is given below. De Greef et al. choose one of the canonical linearizations of the array[5] (a permutation of its dimensions), followed by a modulo operation that wraps the set of "virtual" memory locations into a smaller set of actual physical locations [6].

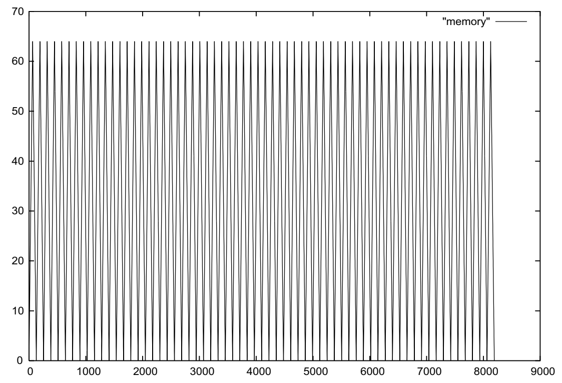
Tronçon et al. proposed to compute an m-dimensional bounding box in the original m-dimensional index space of the array [28]. This is achieved by find-

[5]The row and, respectively, the column concatenations of a 2D array are canonical linearizations.

```
                                   //   All the A-elements are
for (i=0; i<191; i++)              //   produced in this loop nest
   for (j=0; j<64; j++)  {
      if ( i+j >= 63   &&  i+j <= 126 )   A[i][j] = ... ;
   }
                                   //   All the A-elements are
for (i=0; i<191; i++)              //   consumed in this loop nest
   for (j=0; j<64; j++)  {
      if ( i+j >=127  &&  i+j <= 190 )   ... = A[i-64][j];
   }
```







```
                                   //   All the A-elements are
for (i=0; i<191; i++)              //   consumed in this loop nest
   for (j=0; j<64; j++)  {
      if ( i+j >= 63   &&  i+j <= 126 )   A[i][j] = ... ;
      if ( i+j >=127  &&  i+j <= 190 )   ... = A[i-64][j];
   }
```

```
                                   //   All the A-elements are
for (j=0; j<64; j++)               //   consumed in this loop nest
   for (i=0; i<191; i++)  {
      if ( i+j >= 63   &&  i+j <= 126 )   A[i][j] = ... ;
      if ( i+j >=127  &&  i+j <= 190 )   ... = A[i-64][j];
   }
```

Fig. 5. Memory traces showing the effect of loop fusion and loop interchange on the storage requirement. The storage requirement of the initial code is 4,096 memory locations. After loop fusion, the needed data storage decreases to 3,104 locations; additionally, after loop interchange, it becomes only 64 locations (about 1.5% of the initial value).

ing m modulo operands, computed separately as the maximal index differences in each dimension.

Lefebvre and Feautrier, addressing parallelization of static control programs, developed in [16] 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 vectors, 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 [23]. 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 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 [5]. They proposed two heuristic techniques for building strictly admissi-

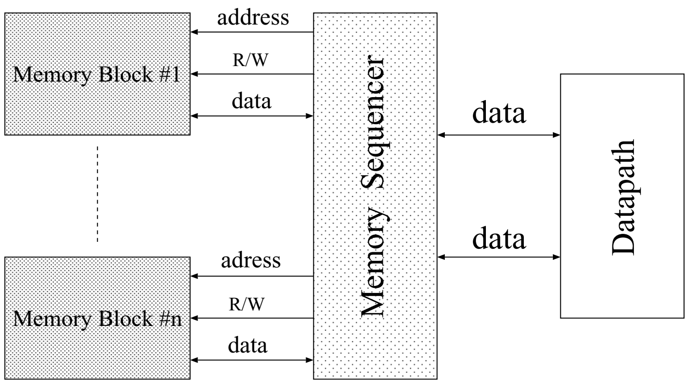


Fig. 6. Datapath and memory units using a sequencer-based architecture [17].

ble integer lattices, hence building valid storage allocations.

4.1. *The evaluation of the mapping models*

None of these past research works dealing with the mapping techniques was able to provide consistent information on *how good their models are*, that is, how large is the oversize of their resulting storage amount after mapping in comparison to the minimum data storage. The effectiveness of a certain mapping model was assessed only *relatively*, that is, the authors compared the storage resulted after applying their mapping model either to the total number of array elements, or to the storage results when applying another mapping model [5,6,28]. This *relative* evaluation is not sufficient, though, since it does not give a precise picture on the *absolute* quality of the model.

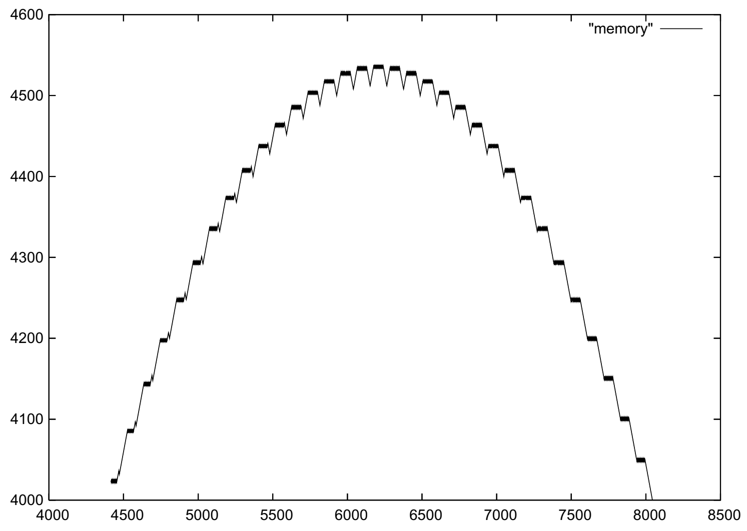
The polyhedral framework described in Section 2 can be used to determine the absolute minimum data storage [2] and, in addition, it can compute the minimum windows for each array in the application code (that is, the maximum number of each array's elements simultaneously alive). For instance, the signal A from the illustrative example in Fig. 7 needs a minimum window of 1,752 memory locations since there are at most 1,752 A-elements simultaneously alive (as computed by the tool based on the algorithm presented in Section 2). Similarly, the minimum window (or the optimal intra-array in-place mapping [4]) of signal B is 3,104 storage locations. However, since the elements of the arrays A and B can share the same locations if their lifetimes are disjoint, the minimum storage requirement is, actually, 4,536 locations, less than the sum of the minimum windows of A and B ($1,752+3,104 = 4,856$). (This is also called the optimal inter-array in-place mapping [4].)

A detail of the memory trace generated by our tool is displayed in Fig. 7.

Now, the mapping model proposed in [6] computes storage windows for each array in the code, considering all the canonical linearizations of the array; for each linearization, the largest distance between two live elements is computed. This distance plus 1 is the data storage allocated for the array – according to [6]. For instance, the linearizations considered for a 2D array are the ones obtained by concatenating the rows, or concatenating the columns, in the increasing or decreasing order of the indexes. When applied to the code in Fig. 7, the mapping model [6] yields a storage window of 2,304 locations[6] for signal A, therefore, 31.51% more storage than necessary (i.e., 1,752); similarly, the model yields a window of 4,096 for signal B, thus 31.96% extra storage (in comparison to 3,104). Moreover, this model [6] would allocate 2,304 + 4,096 = 6,400 locations for the entire code, therefore, 41.09% more storage than the minimum requirement of 4,536 locations.

Tabel 1 shows the sizes of the memory windows for each array in the illustrative code from Fig. 1(a) according to the signal-to-memory mapping models [6,28]. As a comparison, the minimum storage requirements for every individual array are displayed as well. It can be seen that doing the mapping according to the model [6] would require almost 3 times (296%) more data storage than really necessary (the minimum data storage being 49 locations), while the mapping model [28] behaves even worse for this example, requiring 369% additional storage. On the other hand, Tronçon's mod-

[6]Since, e.g., the elements $A[0][47]$ and $A[48][46]$ are simultaneously alive and their distance in the row-by-row concatenation is 48 × 48−1 = 2,303.



```
int A [95][48] ;
int B[127][64];
                                    //   All the A- and B-elements
for (i=0; i<191; i++)               //   are consumed in this loop nest
   for (j=0; j<64; j++)  {
      if ( i+j >= 47   &&  i+j <=  94   &&  j <= 47 )   A[i][j] = ... ;
      if ( i+j >= 95   &&  i+j <= 142  &&  j <= 47 )   ... = A[i-48][j];
      if ( i+j >= 63   &&  i+j <= 126 )                B[i][j] = ... ;
      if ( i+j >=127   &&  i+j <= 190 )                ... = B[i-64][j];
   }
```

Fig. 7. Illustrative example for mapping evaluation and a detail of its memory trace.

Table 1

The sizes of the mapping memory windows of the arrays in the signal-to-memory mapping models [6,28] for the example in Fig. 1(a). The last row displays the minimum window sizes determined with the algorithm computing the minimum data storage in [2]

Mapping windows	Array A	Array B	Array C	Array D	Array Total
Mapping model [6]	22	37	56	79	194
Mapping model [28]	28	45	66	91	230
Min. window size	16	25	36	49	126

el [28] can yield smaller window sizes than De Greef's model [6] when the array (index) space contains *holes* or 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). For a better system-level exploration, it is thus desirable to have implemented several mapping approaches.

In conclusion, the computation of the minimum data storage can be used to evaluate the performance of *any* signal-to-memory mapping model, being a useful tool in the system-level exploration.

4.2. Efficient implementation of mapping models

Even more relevant, the polyhedral framework which steers the computation of the minimum data storage can be used to efficiently implement different mapping models. The computation method employed by De Greef et al. consists of a sequence of integer linear programming (ILP) optimizations for each array linearization [6]. Tronçon et al. use, basically, sequences of emptiness checks for **Z**-polytopes derived from the code [28]. The software tools implemented by the authors of these models exhibit significant running times (often of the order of tens of minutes). Our methodology based on the decomposition of the array references in disjoint bounded lattices can be used to achieve more efficient implementations. We shall exemplify using the mapping model [6].

De Greef et al. analyze all the canonical linearizations of each array [6] (that is, the indexes vary like the iterators of a perfect loop nest and the distinct linearizations are permutations of the loops). For any linearization, the largest distance at any time between two live array elements is computed. This distance plus 1 is then the storage required for the mapping of the array into the data memory, relative to the chosen linearization. The linearization yielding the minimum largest distance is finally selected. The values of the mapping function are the positions of the array elements in the selected linearization, followed by a *modulo* operation (whose operand is the corresponding distance plus 1) that wraps the set of "virtual" memory locations into a smaller set of actual physical locations.

Since the computation of the bounding window of an array can be reduced to the computation of the bounding windows of its lattices, we shall address this latter problem using for illustration the example below:

Example 3.

```
for (i=2; i<=7; i++)
    for (j=1; j<=-2*i+15; j++)
       if (j<=i+1)  ···
          A[2*i-j+5][3*i+2*j-7] ···
```

Let us assumed that all the A-elements of the array reference `A[2*i-j+5][3*i+2*j-7]` are alive. Take, for instance, the linearization of A by row concatenation (in the increasing order of the rows). Then, it can be easily observed that, in the bounded lattice representing the array reference, the A-elements at the maximum distance from each other are the elements with (lexicographically) minimum and, respectively, maximum indices, that is $A[6][5]$ and $A[18][16]$ (see Fig. 8). 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 our illustrative example, the elements $A[8][1]$ and $A[10][18]$ are the farthest away from each other.

If 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 $\mathbf{A}\cdot\mathbf{i} \geqslant \mathbf{b}$, the algorithm computing the array elements situated at the maximum distance in the canonical linearization is described below:

Algorithm 1.

Step 1. Let $\mathbf{S}$ be a unimodular matrix (a square matrix whose determinant is $\pm$ 1) bringing $\mathbf{T}$ to the Hermite Normal Form [25]:

$$\mathbf{H} \;=\; \mathbf{T}\cdot\mathbf{S}.$$

Step 2. After applying the unimodular transformation $\mathbf{S}$, the new iterator polytope becomes:

$$\bar{P} = \{\, \bar{\mathbf{i}} \in \mathbf{Z}^n \mid \ \mathbf{A}\cdot\mathbf{S}\cdot\bar{\mathbf{i}} \geqslant \mathbf{b}\,\}.$$

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 [22]. 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 $\bar{\mathbf{i}}^{\max}$ (respectively, $\bar{\mathbf{i}}^{\min}$).

Then, $\mathbf{x}^{\min} = \mathbf{H}\cdot\bar{\mathbf{i}}^{\min} + \mathbf{u}$ and $\mathbf{x}^{\max} = \mathbf{H}\cdot\bar{\mathbf{i}}^{\max} + \mathbf{u}$. $\square$

The algorithm will be illustrated for the array reference `A[2*i-j+5][3*i+2*j-7]`, assuming first the linearization by row concatenation. Since the unimodular matrix

$$\mathbf{S} = \begin{bmatrix} 0 & 1 \\ -1 & 2 \end{bmatrix},$$

brings matrix

$$\mathbf{T} = \begin{bmatrix} 2 & -1 \\ 3 & 2 \end{bmatrix}$$

to the Hermite Normal Form:

$$\mathbf{H} = \mathbf{T}\cdot\mathbf{S} = \begin{bmatrix} 1 & 0 \\ -2 & 7 \end{bmatrix},$$

the new iterator polytope (*Step 2*) is computed:

$$\bar{P} = \{ j \geqslant 2\,,\ -i + 2j \geqslant 1\,,\ i - j \geqslant -1, \\ i - 4j \geqslant -15\}.$$

The maximum (lexicographically) iterator vector in $\bar{P}$ is

$$\begin{bmatrix} i \\ j \end{bmatrix}^{\max} = \begin{bmatrix} 13 \\ 7 \end{bmatrix},$$

which yields

$$\begin{bmatrix} x \\ y \end{bmatrix}^{\max} = \mathbf{H} \begin{bmatrix} i \\ j \end{bmatrix}^{\max} + \mathbf{u} = \begin{bmatrix} 18 \\ 16 \end{bmatrix}.$$

The minimum iterator vector in $\bar{P}$ is

$$\begin{bmatrix} i \\ j \end{bmatrix}^{\min} = \begin{bmatrix} 1 \\ 2 \end{bmatrix},$$

which yields

$$\begin{bmatrix} x \\ y \end{bmatrix}^{\min} = \mathbf{H} \begin{bmatrix} i \\ j \end{bmatrix}^{\min} + \mathbf{u} = \begin{bmatrix} 6 \\ 5 \end{bmatrix}.$$

In the linearization by column concatenation, the array indices are reversed. Matrix $\mathbf{T}$ is thus

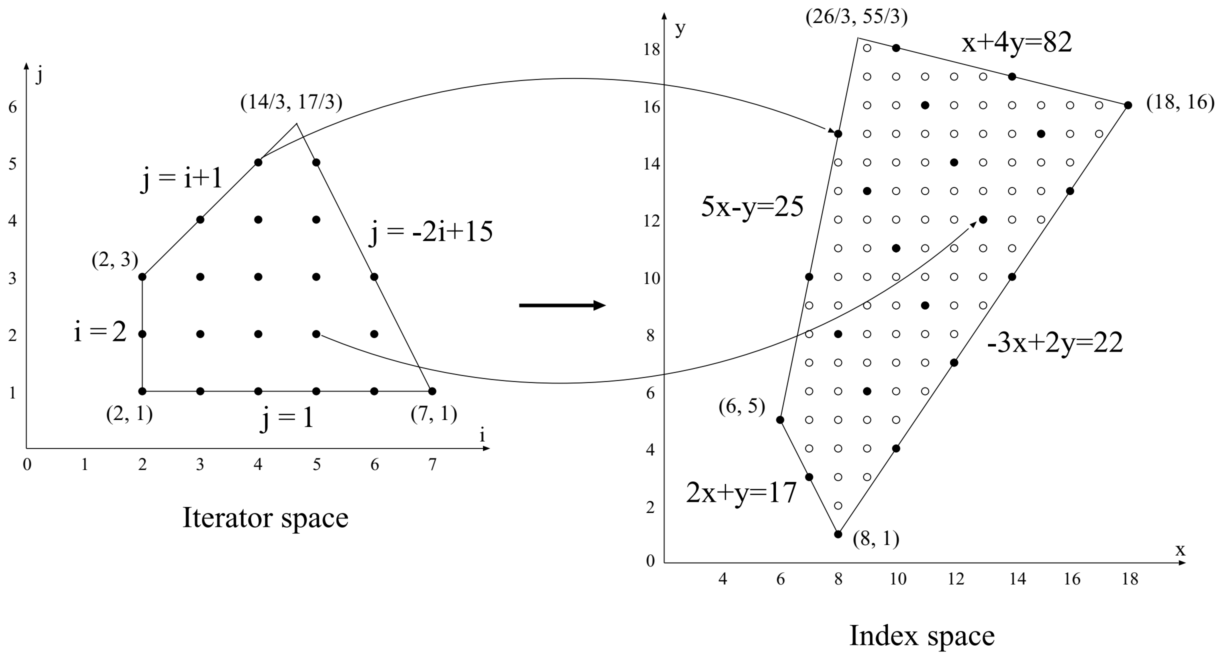


Fig. 8. The iterator and index spaces of the array reference $A[2*i-j+5][3*i+2*j-7]$ from *Example 3*.

$$\begin{bmatrix} 3 & 2 \\ 2 & -1 \end{bmatrix}.$$

The unimodular matrix

$$\mathbf{S} = \begin{bmatrix} 1 & 2 \\ -1 & -3 \end{bmatrix},$$

and the Hermite Normal Form:

$$\mathbf{H} = \mathbf{T} \cdot \mathbf{S} = \begin{bmatrix} 1 & 0 \\ 3 & 7 \end{bmatrix}.$$

The new iterator polytope is:

$$\bar{P} = \{i + 2j \geqslant 2\,,\ -i - 3j \geqslant 1\,,\ 2i + 5j \geqslant -1, \quad -i - j \geqslant -15\}.$$

The maximum (lexicographically) iterator vector in $\bar{P}$ is

$$\begin{bmatrix} i \\ j \end{bmatrix}^{\max} = \begin{bmatrix} 25 \\ -10 \end{bmatrix},$$

which yields

$$\begin{bmatrix} x \\ y \end{bmatrix}^{\max} = \mathbf{H} \begin{bmatrix} i \\ j \end{bmatrix}^{\max} + \begin{bmatrix} -7 \\ 5 \end{bmatrix} = \begin{bmatrix} 18 \\ 10 \end{bmatrix}.$$

The minimum iterator vector in $\bar{P}$ is

$$\begin{bmatrix} i \\ j \end{bmatrix}^{\min} = \begin{bmatrix} 8 \\ -3 \end{bmatrix},$$

which yields

$$\begin{bmatrix} x \\ y \end{bmatrix}^{\min} = \mathbf{H} \begin{bmatrix} i \\ j \end{bmatrix}^{\min} + \begin{bmatrix} -7 \\ 5 \end{bmatrix} = \begin{bmatrix} 1 \\ 8 \end{bmatrix}.$$

Since the array indices were reversed, these results correspond to the array elements $A[10][18]$ and $A[8][1]$. □

Algorithm 1 is used in the flow of a more general algorithm computing the size of the memory bounding window *after* the mapping (according to the model [6]) of a multidimensional array (signal) from the application code. The general scheme of this algorithm is given below:

Algorithm 2.
Step 1. Decompose the signal's array references into disjoint lattices.

This step is also used in the computation of the minimum data storage [2]. This is a clear advantage of using the polyhedral framework (Section 2) since partial results from a memory management task can be used in another task, enhancing thus the overall computation efficiency.
Step 2. For every lattice in the decomposition of the array space and for every canonical linearization, compute with *Algorithm 1* the array elements situated at a maximum distance.
Step 3. For every maximal group of simultaneously alive lattices (these can be easily obtained from the

lattices alive at the beginning and at the end of the loop nests), compute the maximum distance between their array elements.

The overall maximum distance plus 1 is the size of the memory window required for mapping the signal. The above steps are sufficient if every loop nest either produces or consumes (but not both!) the signal's elements. Otherwise, a refinement step must be additionally performed:

Step 4. Update the overall maximum distance for the loop nests where lattices of the signal are simultaneously produced and consumed. □

This general scheme can be used to implement other mapping models as well. It is sufficient to replace *Algorithm 1* and to perform some adaptations of the steps 3 and 4 to the characteristics of the new model. For instance, replacing *Algorithm 1* with the algorithm shown below which computes the extreme points of the projection of a given lattice on every axis, we obtain a 1-dimensional window for every index of the array reference (or lattice) and, consequently, an implementation of the mapping model [28].

Algorithm 3.

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 $\mathbf{T}$ of the given lattice.

Step 1. Let **S** be a unimodular matrix bringing $\mathbf{t}_k$ to the Hermite Normal Form [25]: $[h_1 \ 0 \cdots 0] = \mathbf{t}_k \cdot \mathbf{S}$. (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:

$$\bar{P} = \{ \bar{\mathbf{i}} \in \mathbf{Z}^n \mid \mathbf{A} \cdot \mathbf{S} \cdot \bar{\mathbf{i}} \geqslant \mathbf{b} \}.$$

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 [22]. Then, $x_k^{\min} = h_1 \bar{i}_1^{\min} + u_k$ and $x_k^{\max} = h_1 \bar{i}_1^{\max} + u_k$. □

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 the first index. In this way, the problem reduces to computing the projection of a **Z**-polytope, which is well-studied [22,31].

Concluding, our methodology offers the flexibility to implement and evaluate different mapping models within the same formal framework.

5. Experimental results

A software framework performing memory management tasks (e.g., computation of the minimum data storage for whole applications or for specified signals, evaluation of signal-to-memory mapping techniques) has been implemented in C++, incorporating the ideas and algorithms described in this paper. For the syntax of the algorithmic specifications, we adopted a subset of the C language. The framework can optionally generate the variation of the data storage during the execution of the application code (as illustrated in Figs 4, 5, and 7).

Table 2 summarizes the results of our experiments, carried out on a PC with a 1.85 GHz Athlon XP processor and 512 MB memory. The selected benchmarks are either algebraic kernels or applications from digital signal processing: (1) a real-time regularity detection algorithm used in robot vision; (2) Durbin's algorithm for solving Toeplitz systems with N unknowns; (3) a 2-D Gaussian blur filter from a medical image processing application which extracts contours from tomography images in order to detect brain tumors; (4) a motion detection algorithm used in the transmission of real-time video signals on data networks [4]; (5) the kernel of a motion estimation algorithm for moving objects (MPEG-4).

Columns 2 and 3 display information on the characteristics of the benchmark codes: the numbers of array references and of array elements (scalars). Column 4 shows the minimum sizes of the data storage (obtained with the algorithm described in Section 2) and the corresponding running times. Column 5 displays the amounts of data storage resulted after signal-to-memory mapping, based on the model [6].

Table 2 shows that the computation of the minimum data storage is reasonably fast, but sometimes performing exact computations implies a higher computational effort – as in the case of the 2D Gaussian blur filter application. This happens mainly when the data storage has relative small variations during the code execution, preventing the pruning mechanism of the tool to work efficiently. As explained in [2], the algorithm detects and eliminates from further analysis the blocks of code where the local maxima cannot exceed the overall storage requirement. This "pruning" speeds up the tool, concentrating the analysis on those portions of code where the memory increase is likely to happen. Consequently, the tool is slower for applications having very many local maxima of storage variation. This can be observed when running the 2D Gaussian blur filter ap-

Table 2
Experimental results

Application (parameters)	# Array references	# Array elements	Min. Memory Size / CPU (optimal memory sharing)	Memory Size using mapping model [6]
Regularity detection (MaxGrid = 8, L = 64)	19	4,752	2,304 / <1 sec	3,706
Durbin's algorithm (N = 500)	27	252,499	1,249 / 15 sec	1,502
2D Gaussian blur filter (N = 800, M = 600)	20	5,260,027	480,005 / 137 sec	958,805
Motion detection				
(M = N = 32, $m = n = 4$)	11	72,543	2,740 / 2 sec	2,741
(M = N = 120, $m = n = 8$)	11	3,749,063	33,284 / 16 sec	33,285
MPEG-4 motion estimation	68	265,633	2,465 / 18 sec	3,396

plication, which exhibits a poorer scalability relative to the image parameters M and N.

As explained in Section 4, the minimum data storage can be used to assess the performance of different mapping strategies that trade-off some additional storage for a less complex address generation hardware. In order to illustrate this memory management application, an implementation of the mapping model described in [6] has been carried out. The storage requirements obtained *after* mapping the multidimensional signals to the data memory are displayed in column 5. The results show that there are applications, like the motion detection, where the mapping model [6] gives very good solutions, close to the optimal memory size. On the other hand, there are also applications where the data storage after mapping is significantly larger than the optimal value (e.g., about two times larger for the 2D Gaussian blur filter). In such a situation, the designer should consider the use of other mapping models, in the hope of obtaining better storage results, closer to the minimum data storage. The positive aspect is that our framework is able to measure the *quality* of the mapping, whereas the other works in the field (e.g. [5,6,28]) do not provide similar information. In addition, our experiments suggest that our implementation of the mapping model is faster than the original implementations. A comprehensive assessment is difficult to do in this moment due to the different computation platforms and different benchmark tests. However, a *voice coding* application was processed by [6] in over 27 minutes and by [28] in over 25 minutes (using a 300 MHz Pentium II). In contrast, we did the computations in only 14 seconds. Taking into account the difference of platforms and using a conservative scaling factor [33], our technique seems to be at least two times faster.

6. Conclusions

This paper has presented a methodology operating with polyhedra and images of polyhedra for addressing memory management tasks in multimedia and multi-dimensional signal processing applications. A central application of this methodology is the computation of the minimum data storage in affine, procedural specifications. The paper has discussed two other memory management applications of this technique during system-level exploration. The first application is the evaluation of the impact of the code transformations on the storage requirement. It has been shown that different variants of code of a same application can be compared one against another in storage point of view, without the need of performing a proper memory allocation for each variant. The second application is the assessment of different strategies of mapping multidimensional arrays into the data memory. Moreover, the paper has shown that the polyhedral framework developed in this work can be used to efficiently implement different mapping models.

References

[1] F. Balasa, F. Catthoor and H. De Man, Background memory area estimation for multi-dimensional signal processing systems, *IEEE Trans VLSI Syst* **3**(2) (June 1995), 157–172.

[2] F. Balasa, H. Zhu and I.I. Luican, Computation of storage requirements for multi-dimensional signal processing applications, *IEEE Trans on VLSI Systems* **15**(4) (April 2007), 447–460.

[3] A.I. Barvinok, A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed, *Mathematics of Operations Research* **19**(4) (Nov. 1994), 769–779.

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

[5] A. Darte, R. Schreiber and G. Villard, Lattice-based memory allocation, *IEEE Trans Computers* **54** (Oct. 2005), 1242–1257.

[6] E. De Greef, F. Catthoor and H. De Man, Memory size reduction through storage order optimization for embedded parallel multimedia applications, special issue on "Parallel Processing and Multimedia," in: *Parallel Computing, Elsevier*, A. Krikelis, ed., **23**(12) (Dec. 1997), 1811–1837.

[7] D. Gajski, F. Vahid, S. Narayan and J. Gong, *Specification and Design of Embedded Systems*, Englewood Cliffs, NJ: Prentice Hall, 1994.

[8] C.H. Gebotys and M.I. Elmasry, *Optimal VLSI Architectural Synthesis*, Boston: Kluwer Academic Publ., 1992.

[9] G. Goossens, J. Rabaey, J. Vandewalle and H. De Man, An efficient microcode compiler for custom DSP processors, *Proc. IEEE Int Conf Comp-Aided Design*, Santa Clara CA, Nov. 1987, 24–27.

[10] G. Goossens, *Optimization Techniques for Automated Synthesis of Application-specific Signal-processing Architectures*, Ph.D. thesis, K.U. Leuven, Belgium, 1989.

[11] P. Grun, F. Balasa and N. Dutt, Memory size estimation for multimedia applications, in *Proc. 6th Int. Workshop on Hardware/Software Co-Design*, Seattle WA, March 1998, 145–149.

[12] A. Hashimoto and J. Stevens, Wire routing by optimizing channel assignment within large apertures, in *Proc. 8th Design Automation Workshop*, 1971, 155–169.

[13] Q. Hu, A. Vandecappelle, M. Palkovic, P.G. Kjeldsberg, E. Brockmeyer and F. Catthoor, Hierarchical memory size estimation for loop fusion and loop shifting in data-dominated applications, in *Proc. Asia & S.-Pacific Design Automation Conf.*, Yokohama, Japan, Jan. 2006, 606–611.

[14] P.G. Kjeldsberg, F. Catthoor and E.J. Aas, Data dependency size estimation for use in memory optimization, *IEEE Trans CAD of IC's and Syst* **22**(7) (July 2003), 908–921.

[15] F.J. Kurdahi and A.C. Parker, REAL: A program for register allocation, in *Proc. 24th ACM/IEEE Design Automation Conf.*, 1987, 210–215.

[16] V. Lefebvre and P. Feautrier, Automatic storage management for parallel programs, *Parallel Computing* **24** (1998), 649–671.

[17] B. Le Gal, E. Casseau, S. Huet and E. Martin, Dynamic memory access management for high-performance DSP applications using high-level synthesis, to be published in *IEEE Trans. on VLSI Systems*, 2008.

[18] S.Y. Ohm, F.J. Kurdahi and N. Dutt, Comprehensive lower bound estimation from behavioral descriptions, in *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, 1994, 182–187.

[19] K.K. Parhi, Calculation of minimum number of registers in arbitrary life time chart, *IEEE Trans Circ & Syst - II: Analog and Digital Signal Processing* **41**(6) (1994), 434–436.

[20] J. Park and P.C. Diniz, Synthesis of pipelined memory access controllers for streamed data applications on FPGA-based computing engines, in *Proc. Int. Symposium on Systems Synthesis*, 2001, 221–226.

[21] P.G. Paulin and J.P. Knight, Force-directed scheduling for the behavioral synthesis of ASIC's, *IEEE Trans on Comp.-Aided Design of ICs and Syst* **8**(6) (June 1989), 661–679.

[22] W. Pugh, A practical algorithm for exact array dependence analysis, *Comm of the ACM* **35**(8) (Aug. 1992), 102–114.

[23] F. Quilleré and S. Rajopadhye, Optimizing memory usage in the polyhedral model, *ACM Trans Programming Languages and Syst* **22**(5) (2000), 773–815.

[24] J. Ramanujam, J. Hong, M. Kandemir and A. Narayan, Reducing memory requirements of nested loops for embedded systems, *Proc. 38th ACM/IEEE Design Automation Conf.*, June 2001, 359–364.

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

[26] L. Stok and J. Jess, Foreground memory management in data path synthesis, *Int J Circuit Theory and Appl* **20** (1992), 235–255.

[27] L. Thiele, Compiler techniques for massive parallel architectures, in: *State-of-the-art in Computer Science*, P. Dewilde, ed., Kluwer Acad. Publ., 1992.

[28] R. Tronçon, M. Bruynooghe, G. Janssens and F. Catthoor, Storage size reduction by in-place mapping of arrays, in: *Verification, Model Checking and Abstract Interpretation*, A. Coresi, ed., 2002, 167–181.

[29] C.J. Tseng and D. Siewiorek, Automated synthesis of data paths in digital systems, *IEEE Trans on Comp.-Aided Design of ICs and Syst* **CAD-5**(3) (July 1986), 379–395.

[30] I. Verbauwhede, C. Scheers and J.M. Rabaey, Memory estimation for high level synthesis, in: *Proc. 31st ACM/IEEE Design Automation Conf.*, June 1994, 143–148.

[31] S. Verdoolaege, K. Beyls, M. Bruynooghe and F. Catthoor, Experiences with enumeration of integer projections of parametric polytopes, in: *Compiler Construction: 14th Int. Conf.*, (Vol. 3443), R. Bodik, ed., Springer, 2005, 91–105.

[32] Y. Zhao and S. Malik, Exact memory size estimation for array computations, *IEEE Trans VLSI Syst* **8**(5) (2000), 517–521.

[33] * * * , *Benchmark Marathon: 65 CPUs from 100 MHz to 3066 MHz* [Online]. Available: http://www.tomshardware.com/2003/02/17/benchmark_marathon/index.html.