

Algebraic Techniques in the Memory Size Computation of Multimedia Processing Applications*

Hongwei Zhu Karthik Chandramouli Yan Yue Florin Balasa

Dept. of Computer Science, University of Illinois at Chicago

Abstract – In real-time multimedia processing systems a very large part of the power consumption is due to the data storage and data transfer. Moreover, the area cost is often largely dominated by memories. Hence, the optimization of the memory architecture is a crucial step in the design methodology for this type of applications. In deriving an optimized memory architecture, memory size computation is an important step in the data transfer and storage exploration stage. This paper investigates non-scalar methods for computing the memory size in real-time multimedia algorithms. The approach is based on more recent algebraic techniques specific to the data-flow analysis used in modern compilers. In contrast with previous works which utilize only approximate methods due to the size of the problems (in terms of number of scalars) and single-assignment specifications, this research aims to obtain exact determinations even for large applications.

1 Introduction

In real-time multimedia processing systems – including video an image processing, medical imaging, artificial vision, real-time 3D rendering, advanced audio and speech coding – a very large part of the power consumption is due to the data storage and data transfer. A typical system architecture includes custom hardware (application-specific accelerator datapaths and logic), programmable hardware (DSP core and controller), and a distributed memory organization which is usually expensive in terms of power and area cost. Data transfer and memory access operations typically consume more power than a datapath operation. For instance, fetching an operand from an off-chip memory for an addition consumes 33 times more power than the actual computation; even a transfer from an on-chip memory consumes about 4 to 10 times more power than the addition itself [3]. Moreover, the area cost is often largely dominated by memories. Hence, the optimization of the memory architecture is a crucial step in the design methodology for this type of applications. In deriving an optimized memory architecture, memory size estimation/computation is an important step in the data transfer and storage exploration stage. This problem has been tackled in the past both in register-transfer programs at scalar level [11, 8, 18, 15] and in behavioral specifications at non-scalar level [1, 20, 10, 22]. Good overviews of these techniques can be found in [3, 14].

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

This paper investigates non-scalar methods for computing exactly the memory size in real-time multimedia algorithms. This approach uses recent algebraic techniques specific to the data-flow analysis used in modern compilers [13]. In contrast with previous works which utilize only approximate methods due to the size of the problems (in terms of number of scalars), this approach aims to obtain exact determinations even for applications significantly large. Moreover, data-flow analysis enables the study of memory management tasks at the desired level of granularity – between whole array and the scalar level – trading-off computational effort, solution accuracy and optimality. The memory computation approach described in Sections 2 and 3 is intended to handle the entire class of “affine” specifications (therefore, a large class of real-time multimedia applications).

2 Computation of array reference size using algebraic transformations

In order to address the computation of the memory size necessary for the execution of a multidimensional signal processing algorithm, a simpler problem must be addressed first: the computation of the number of distinct scalars covered by a single array reference.

Example for ($i = 0; i \leq 511; i++$)
for ($j = 0; j \leq 511; j++$)
for ($k = 0; k \leq 511; k++$)
... $M[i+k][j+k]$...

How many memory locations are necessary to store the array reference $M[i+k][j+k]$? In spite of its apparent simplicity the problem is difficult. Moreover, it did not receive too much attention although it is the cornerstone of the exact memory computation. The correct answer for the example above is *not* the total number of iterator triplets (i, j, k) , that is $512^3 = 134,217,728$. The reason is that the same scalar signal is obtained for different combinations of the iterators i, j, k . For instance, the iterator vectors $(i, j, k) = (0, 1, 1)$ and $(i, j, k) = (1, 2, 0)$ yield the same scalar $M[1][2]$. The answer is not $1023^2 = 1,046,529$ either ($0 \leq i+k, j+k \leq 1022$) since, e.g., the scalar $M[0][512]$ is not addressed by any iterator vector (i, j, k) .

It must be emphasized that enumerative techniques can always be applied to compute the number of scalars in an array reference. These approaches are obviously simple and extremely efficient for array references with “small” iterator sets.

However, in image and video processing applications most of the array references are characterized by large sets of iterators: an enumerative technique, although very simple, will be too computationally expensive to use in such applications. For the illustrative example shown above, an enumerative algorithm was initially used for testing purpose, but the computation had to be eventually stopped. For such examples, algebraic techniques are the only hope.

A *polyhedron* is a set of points $P \subset \mathbb{R}^n$ satisfying a finite set of linear inequalities: $P = \{ \mathbf{x} \in \mathbb{R}^n \mid \mathbf{A} \cdot \mathbf{x} \geq \mathbf{b} \}$, where $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{b} \in \mathbb{R}^m$. If P is a bounded set, then P is called a *polytope*. If $\mathbf{x} \in \mathbb{Z}^n$, then P is called an *integral polyhedron/polytope*. The set $\{ \mathbf{y} \in \mathbb{R}^m \mid \mathbf{y} = \mathbf{A}\mathbf{x}, \mathbf{x} \in \mathbb{Z}^n \}$ is called the *lattice* generated by the columns of matrix \mathbf{A} .

Each array reference $M[x_1(i_1, \dots, i_n)] \dots [x_m(i_1, \dots, i_n)]$ of an m -dimensional signal M , in the scope of a nest of n loops having the iterators i_1, \dots, i_n is characterized by an *iterator space* and an *index space*. The iterator space signifies the set of all iterator vectors $\mathbf{i} = (i_1, \dots, i_n) \in \mathbb{Z}^n$ in the scope of the array reference. The index space is the set of all index vectors $\mathbf{x} = (x_1, \dots, x_m) \in \mathbb{R}^m$ of the array reference.

If the loop boundaries are affine mappings with integer coefficients of the surrounding loop iterators, the increment steps of the loops¹ are ± 1 , and the conditions in the scope of the array reference are relational and/or logical operations between affine mappings of the loop iterators², then the iterator space can be represented by one or several disjoint integral (iterator) polytopes $\mathbf{A} \cdot \mathbf{i} \geq \mathbf{b}$, where $\mathbf{A} \in \mathbb{Z}^{(2n+c) \times n}$ and $\mathbf{b} \in \mathbb{Z}^{2n+c}$. The first $2n$ linear inequalities are derived from the loop boundaries and the last c inequalities are derived from the control-flow conditions (this representation is usually not minimal). The size of the iterator space is $\text{Card} \{ \mathbf{i} \in \mathbb{Z}^n \mid \mathbf{A} \cdot \mathbf{i} \geq \mathbf{b} \}$.

If, in addition, the indices of an array reference are affine mappings with integer coefficients of the loop iterators, the index space consists of one or several *linearly bounded*³ lattices (LBL) [19] – the image of a vector affine function over the iterator polytopes:

$$\{ \mathbf{x} = \mathbf{T} \cdot \mathbf{i} + \mathbf{u} \mid \mathbf{A} \cdot \mathbf{i} \geq \mathbf{b}, \mathbf{i} \in \mathbb{Z}^n \}$$

where $\mathbf{x} \in \mathbb{Z}^m$ is the index (coordinate) vector of the m -dimensional signal. The affine function is characterized by $\mathbf{T} \in \mathbb{Z}^{m \times n}$ and $\mathbf{u} \in \mathbb{Z}^m$. The size of the index space of an array reference is, therefore, $\text{Card} \{ \mathbf{x} = \mathbf{T} \cdot \mathbf{i} + \mathbf{u} \in \mathbb{Z}^m \mid \mathbf{A} \cdot \mathbf{i} \geq \mathbf{b}, \mathbf{i} \in \mathbb{Z}^n \}$. The number of scalars addressed by the array reference is obviously the size of its index space.

Example The index space of the operand $M[i+k][j+k]$ in the example above is the set

$$\left\{ \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right\}$$

¹The loops having increment steps different from ± 1 can be easily “normalized” with the affine transformation $i = i' \cdot \text{Step} + \text{lower_bound}$.

²The polyhedral representation of the iterator space is still valid if the array reference scope also contains data-dependent (but iterator independent) conditions.

³As the definition domain of the mappings is not \mathbb{Z}^n but a polytope – which is bounded by hyperplanes (characterized by linear equations).

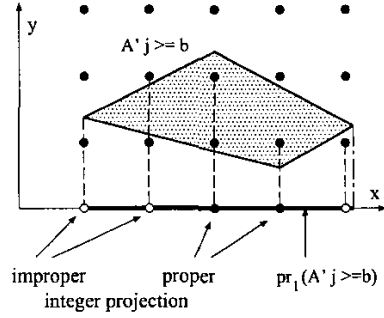


Figure 1: Iterator polytope and its projection

where $0 \leq i, j, k \leq 511$, $i, j, k \in \mathbb{Z}$.

In general, the index space may be a set of linearly bounded lattices. E.g., a conditional instruction $if(i \neq j)$ determines two LBL's for the array references within the scope of the condition – one corresponding to $i \geq j + 1$, and another corresponding to $i \leq j - 1$. As the iterator space of an array reference can be decomposed into a set of disjoint polytopes and the index space – into disjoint LBL's [1], it is assumed in the sequel, without any loss of generality, that each iterator space is represented by a single polytope and, correspondingly, each index space is represented by a single linearly bounded lattice.

When the vectorial function $\mathbf{f} : \mathbb{Z}^n \rightarrow \mathbb{Z}^m$, defined by $\mathbf{f}(\mathbf{i}) = \mathbf{T}\mathbf{i} + \mathbf{u}$ is a one-to-one mapping, the index space and the iterator space $\{ \mathbf{A} \cdot \mathbf{i} \geq \mathbf{b}, \mathbf{i} \in \mathbb{Z}^n \}$ are equal in size, as any n -dimensional point in the iterator space is mapped to a *distinct* m -dimensional point in the index space. In this situation the problem is significantly simpler as it reduces to the computation of the number of iterator vectors. Unfortunately, this is not always the case, as shown in the illustrative example.

The general idea is to transform algebraically the linearly bounded lattice of the given array reference into an equivalent lattice whose size is easier to compute. For instance, by post-multiplying matrix \mathbf{T} with a sequence of unimodular matrices $\mathbf{S} = \mathbf{U}_1 \cdot \mathbf{U}_2 \cdot \dots$, a reduced Hermite normal form [17] is obtained.

Example Post-multiplying matrix \mathbf{T} in the LBL of the array reference $M[i+k][j+k]$ by the unimodular matrix $\mathbf{S} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}$, a new affine mapping $\mathbf{f}'(\mathbf{j}) = \mathbf{T}'\mathbf{j} + \mathbf{u}$ is

obtained, where $\mathbf{T}' = \mathbf{T} \cdot \mathbf{S} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$. The new iterators j_1, j_2, j_3 satisfy the constraints $511 \geq j_1 - j_3 \geq 0$, $511 \geq j_2 - j_3 \geq 0$, $511 \geq j_3 \geq 0$. Therefore, the initial array reference $M[i+k][j+k]$ contains the same number of scalars as $M[j_1][j_2]$ in the scope of the nested loops:

for ($j_3 = 0; j_3 \leq 511; j_3++$)
 for ($j_2 = j_3; j_2 \leq 511 + j_3; j_2++$)
 for ($j_1 = j_3; j_1 \leq 511 + j_3; j_1++$) ... $M[j_1][j_2]$...

Denoting $r = \text{rank } \mathbf{T}'$, the size of the index space (i.e., the number of distinct \mathbf{x} 's) is at most the size of the r -dimensional

polytope $pr_r(\mathbf{A}'\mathbf{j} \geq \mathbf{b})$, where $\mathbf{A}' = \mathbf{A} \cdot \mathbf{S}$, that is, the projection of $\mathbf{A}'\mathbf{j} \geq \mathbf{b}$ on \mathbf{Z}^r along with the first r coordinates. $pr_r(\mathbf{A}'\mathbf{j} \geq \mathbf{b})$ can be easily computed, by eliminating the last $n - r$ variables in $\mathbf{A}'\mathbf{j} \geq \mathbf{b}$ with the Fourier-Motzkin technique [6]. It must be noticed that not necessarily all the points of integer coordinates in $pr_r(\mathbf{A}'\mathbf{j} \geq \mathbf{b})$ represent projections of integer points in $\mathbf{A}'\mathbf{j} \geq \mathbf{b}$ (see Fig. 1). These "improper" integer projections are detected replacing the r coordinates of the projection point in the iterator polytope $\mathbf{A}'\mathbf{j} \geq \mathbf{b}$ and checking whether the resulting $(n - r)$ -dimensional integer polytope is empty or not (proper projection!).

Example (cont'd) The transformed iterator polytope is $\{\mathbf{j} \in \mathbf{Z}^3 \mid \mathbf{A}' \cdot \mathbf{j} \geq \mathbf{b}\} = \{511 \geq j_1 - j_3 \geq 0, 511 \geq j_2 - j_3 \geq 0, 511 \geq j_3 \geq 0\}$. As $r = \text{rank } \mathbf{T}' = 2$, its projection can be obtained eliminating the last iterator j_3 (as $n - r = 1$) from the inequalities above. It follows that $pr_2(\mathbf{A}' \cdot \mathbf{j} \geq \mathbf{b}) = \{(j_1, j_2) \in \mathbf{Z}^2 \mid 511 \geq j_1 - j_2 \geq -511, 1022 \geq j_1 \geq 0, 1022 \geq j_2 \geq 0\}$, which contains 784,897 points (j_1, j_2) , since it can be verified that all these points (j_1, j_2) are proper projections of iterator triplets (j_1, j_2, j_3) in the iterator space $\{511 \geq j_1 - j_3 \geq 0, 511 \geq j_2 - j_3 \geq 0, 511 \geq j_3 \geq 0\}$. For instance, the point $(j_1, j_2) = (1, 2)$ in $pr_2(\mathbf{A}' \cdot \mathbf{j} \geq \mathbf{b})$ corresponds the non-empty 1-dimensional polytope $\{j_3 \in \mathbf{Z} \mid 1 \geq j_3 \geq 0\}$. It follows that the size of the index space of the array reference $M[i + k][j + k]$ and, consequently, the necessary memory to store it, is 784,897 – therefore less than 0.6% of the number of iterator triplets (i, j, k) .

The current algorithm is very efficient due to the use of the *exact projection* concept [16] during the computation of the projection of the transformed iterator polytope $pr_r(\mathbf{A}'\mathbf{j} \geq \mathbf{b})$. A projection along a coordinate is called *exact* if the eliminated variable have only coefficients with magnitude one. If the projections along all the eliminated variables are exact, then all the points in $pr_r(\mathbf{A}'\mathbf{j} \geq \mathbf{b})$ are proper projections of the points in the transformed iterator polytope $\{\mathbf{j} \in \mathbf{Z}^n \mid \mathbf{A}' \cdot \mathbf{j} \geq \mathbf{b}\}$. This is the case in the illustrative example, as the only variable eliminated, i.e. j_3 , has coefficients of magnitude one in $\{\mathbf{j} \in \mathbf{Z}^3 \mid \mathbf{A}' \cdot \mathbf{j} \geq \mathbf{b}\} = \{511 \geq j_1 - j_3 \geq 0, 511 \geq j_2 - j_3 \geq 0, 511 \geq j_3 \geq 0\}$. In this situation, the number of scalars within the array reference $M[i + k][j + k]$ is exactly $\text{Card}\{pr_2(\mathbf{A}'\mathbf{j} \geq \mathbf{b})\} = \text{Card}\{(j_1, j_2) \in \mathbf{Z}^2 \mid 511 \geq j_1 - j_2 \geq -511, 1022 \geq j_1 \geq 0, 1022 \geq j_2 \geq 0\} = 784,897$, as already mentioned above.

When inexact projections (in the sense described) occur, only a subset of of the projection polytope $pr_r(\mathbf{A}'\mathbf{j} \geq \mathbf{b})$ contains with certainty proper projection points. This subset can be built intersecting $pr_r(\mathbf{A}'\mathbf{j} \geq \mathbf{b})$ with inequalities constructed as explained below.

Suppose that, while eliminating the iterator j_n , the bounds f, g are affine functions of the other iterators

$$g(j_1, \dots, j_{n-1}) \leq \beta j_n, \quad \alpha j_n \leq f(j_1, \dots, j_{n-1}) \quad (1)$$

where at least one of the coefficients α, β are not ± 1 . To eliminate j_n , the two inequalities are normalized, to get $\alpha g(j_1, \dots, j_{n-1}) \leq \alpha \beta j_n \leq \beta f(j_1, \dots, j_{n-1})$. The inequality $\alpha g(j_1, \dots, j_{n-1}) \leq \beta f(j_1, \dots, j_{n-1})$ will eliminate j_n ,

but there may be no multiple of $\alpha\beta$ in that range, in which case there is no integer j_n satisfying the inequalities (1). In order to be assured that there must be a multiple of $\alpha\beta$ between the bounds $\alpha g(j_1, \dots, j_{n-1})$ and $\beta f(j_1, \dots, j_{n-1})$, it is sufficient to tighten the constraint $0 \leq \beta f(j_1, \dots, j_{n-1}) - \alpha g(j_1, \dots, j_{n-1})$, replacing it by $(\alpha - 1)(\beta - 1) \leq \beta f(j_1, \dots, j_{n-1}) - \alpha g(j_1, \dots, j_{n-1})$. The points satisfying these inequalities are all proper projections of iterator vectors.

The other points in $pr_r(\mathbf{A}'\mathbf{j} \geq \mathbf{b})$ which do not satisfy the set of inequalities constructed as shown above must be checked replacing their first r coordinates in the iterator polytope $\mathbf{A}'\mathbf{j} \geq \mathbf{b}$ and verifying whether the resulting $(n - r)$ -dimensional integer polytope is empty or not.

More research is needed regarding the possible application of Ehrhart polynomials [5] or even Barvinok's polynomial-time algorithm [2] proposed for counting the size of integer polytopes to counting their affine images.

3 Memory size computation using data-dependence analysis

Once the collections of array references for each multidimensional signal are extracted from the signal processing algorithm, an analytical partitioning into disjoint signal groups is performed for each collection. The aim of the partitioning process is to determine which parts of an array operand are not needed any more after the computation of the signals in a resulting array reference, assuming a given loop hierarchy and a certain data-flow. In other words, which are the scalar signals consumed for the last time, and what is their number, when the signals within a definition domain are produced. The last question is related directly to the evaluation of the storage requirements, as it allows to compute exactly how many memory locations are needed and how many can be freed when a certain group of signals is produced

The analytical partitioning of the array references in disjoint groups of signals can be performed by recursively intersecting the linearly bounded lattices (LBL's) representing the indexes of the multidimensional signals in the algorithm. Let $\{\mathbf{x} = \mathbf{T}_1 \mathbf{i}_1 + \mathbf{u}_1 \mid \mathbf{A}_1 \mathbf{i}_1 \geq \mathbf{b}_1\}$, $\{\mathbf{x} = \mathbf{T}_2 \mathbf{i}_2 + \mathbf{u}_2 \mid \mathbf{A}_2 \mathbf{i}_2 \geq \mathbf{b}_2\}$ be two LBL's derived from the same indexed signal, where \mathbf{T}_1 and \mathbf{T}_2 have obviously the same number of rows – the signal dimension. Intersecting the two linearly bounded lattices means, first of all, solving a linear Diophantine system⁴ $\mathbf{T}_1 \mathbf{i}_1 - \mathbf{T}_2 \mathbf{i}_2 = \mathbf{u}_2 - \mathbf{u}_1$ having the elements of \mathbf{i}_1 and \mathbf{i}_2 as unknowns. If the system has no solution, the intersection is empty. Otherwise, let

$$\begin{bmatrix} \mathbf{i}_1 \\ \mathbf{i}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{V}_1 \\ \mathbf{V}_2 \end{bmatrix} \mathbf{i} + \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \end{bmatrix}$$

be the solution of the Diophantine system. If the set of coa-

⁴Finding 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 [17].

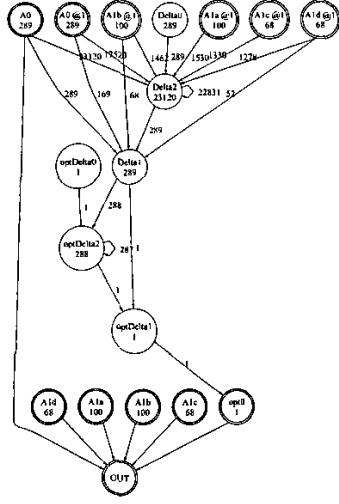


Figure 2: Polyhedral data-dependence graph

lesced constraints

$$\begin{aligned} \mathbf{A}_1 \mathbf{V}_1 \cdot \mathbf{i} &\geq \mathbf{b}_1 - \mathbf{A}_1 \mathbf{v}_1 \\ \mathbf{A}_2 \mathbf{V}_2 \cdot \mathbf{i} &\geq \mathbf{b}_2 - \mathbf{A}_2 \mathbf{v}_2 \end{aligned} \quad (2)$$

has at least one integer solution, then the intersection is a new LBL defined by

$$\{\mathbf{x} = \mathbf{T}_1 \mathbf{V}_1 \cdot \mathbf{i} + \mathbf{T}_1 \mathbf{v}_1 + \mathbf{u}_1 \mid \text{s.t. constraints (2)}\} \quad (3)$$

After the partitioning of the array references, a polyhedral data-dependence graph with exact dependence relations can be produced (Fig. 2). The nodes in the graph correspond to the groups of signals determined analytically as shown above. These nodes are weighted with the number of scalar signals in the group (computed as explained in Section 2). The arcs are weighted with the exact number of dependences between the groups of scalar signals corresponding to the nodes. The data-dependence graphs can be built at the desired level of granularity, depending on the loop nesting level.

Example The graph derived from this code is shown in Fig. 2:

```

optDelta[0] = 0;
for (i = 4; i ≤ 20; i++)
  for (j = 4; j ≤ 20; j++) {
    Delta[i][j][0] = 0;
    for (k = i - 4; k ≤ i + 4; k++)
      for (l = j - 4; l ≤ j + 4; l++)
        Delta[i][j][9*(k-i)+l-j+41] = A[i][j]-A[k][l]@1
          +Delta[i][j][9*(k-i)+l-j+40];
    optDelta[17*i+j-71] = Delta[i][j][81]
      +optDelta[17*i+j-72]; }
opt = optDelta[289];

```

These polyhedral dependence graphs allow the computation of the memory size by performing a relative lifetime analysis, without simulating the computation of the code and without decomposing the arrays into scalars (which is beneficial, as

in most applications the number of scalars is extremely large). When the algorithmic specification is nonprocedural – the computation ordering being basically free, constrained only by dependence relations – these graphs can be used to estimate efficiently the memory size [1]. When the specification is procedural, the case assumed along this paper, the polyhedral dependence graph can be used to compute exactly the memory size necessary for the algorithm execution using a technique similar to the left-edge algorithm [11] operating with the groups of scalars represented by the nodes of the graph. In the example above the memory size is exactly 627 locations.

3.1 Dealing with parametric specifications

Example The cavity detection algorithm [4] is a medical image processing application which extracts contours from images to help physicians detect brain tumors. The algorithm consists of a number of functions, each of which has an image frame as input and one as output. In the first function, a horizontal and Gauss-blurring step is performed, in which each pixel is replaced by a weighted average of itself and its neighbors. In the second function, for each pixel, the difference with all eight neighbors is computed, and the pixel is replaced by the maximum of these differences. Afterwards, the image is first reversed. To this end, the maximum value of the image is computed and each pixel is replaced by the difference between this maximum value and itself. Next, for each pixel, we look whether a neighbor pixel is larger than itself; if this is the case the output is false, otherwise it is true. The Gauss-blurring function is sketched below:

```

void GaussBlur (image_in[M][N], gauss_xy[M][N]) {
  // Perform horizontal and vertical gauss-blurring on each pixel
  unsigned char gauss_xy[M][N];
  for (y = 0; y < M; ++y)
    for (x = 0; x < N; ++x)
      gauss_xy[y][x] = ... // Apply horizontal gauss-blurring
  for (y = 0; y < M; ++y)
    for (x = 0; x < N; ++x)
      gauss_xy[y][x] = ... // Apply vertical gauss-blurring
}

```

Many algorithmic specifications in multidimensional signal processing contain parameters, like the dimensions M , N of the image frames in the Gauss-blurring function. Although it is usually impossible to determine the memory size as a function of the parameters in the algorithmic specification, it is still possible to perform a large part of the computations independent on the parameter values. In this way, the core of the computations can be reused when the values of the parameters are modified without creating a completely new problem to be solved.

Our memory size computation model addresses the applications where the iterator space of all the array references in the specification can be represented by integral (eventually, parametric) polytopes of the form $\mathbf{A} \cdot \mathbf{i} \geq \bar{\lambda}$, where the elements of the vector $\bar{\lambda}$ are parameters λ_i or integer constants. Most digital signal processing applications as, for instance, the cavity detection algorithm [4], satisfy this condition.

As explained in Section 3, the construction of the polyhe-

dral data-dependence graph starts with an analytical partitioning of the array references, where the nodes of the graph are obtained by recursively intersecting the array indexes of each multidimensional signals. An intersection is empty if the resulting (parametric) linear system (2) is inconsistent. This can be determined by applying a pairwise elimination [6] to exclude one by one all the iterators from the system. Finally, a *solvent* system of the form $\mathbf{0} \geq \mathbf{Y} \cdot \bar{\lambda}$ is obtained.

Example When the variables x_i are eliminated from the system

$$\begin{array}{rcl} -5x_1 - x_2 - 3x_3 + x_4 & \geq & \lambda_1 \\ 2x_1 - x_2 - x_3 + x_4 & \geq & \lambda_2 \\ -2x_1 + 2x_2 + 4x_3 - 2x_4 & \geq & \lambda_3 \\ 3x_1 + x_2 + 2x_3 - x_4 & \geq & \lambda_4 \\ x_1 & & \geq \lambda_5 \\ -6x_1 + x_2 + 5x_3 + x_4 & \geq & \lambda_6 \end{array}$$

the following solvent system is obtained:

$$\begin{array}{l} 0 \geq 24\lambda_1 + 44\lambda_2 + 24\lambda_3 + 20\lambda_4 + 20\lambda_5 \\ 0 \geq 32\lambda_1 + 32\lambda_2 + 12\lambda_3 + 40\lambda_4 \\ 0 \geq 16\lambda_1 + 56\lambda_2 + 36\lambda_3 + 40\lambda_5 \end{array}$$

The solvent system can be simplified, eliminating all the redundant inequalities. It can be proven (e.g., [7]) that a linear inequality is redundant if and only if it is a non-negative combination of the other inequalities. In the solvent system above, the first inequality can be eliminated as it is equal to the half of the sum of the other two inequalities. Note that if $\lambda_i > 0$, then solvent system has no solution and, consequently, the given system is inconsistent.

If the solvent system results to be empty, then the corresponding intersection of linearly bounded lattices is empty. Otherwise, the resulting LBL (3) will be conditioned by the solvent system $\mathbf{0} \geq \mathbf{Y} \cdot \bar{\lambda}$.

As a result of the analytical partitioning, the nodes in the polyhedral data-dependence graph will be obtained. However, different from Section 3, part of the nodes will have a conditional existence: in addition to their LBL-based analytical expression, they will have a parametric solvent system attached. In a similar way, some of the arcs in the data-dependence graph will be “conditional” arcs, having a parametric solvent system associated with them.

The resulting *parametric* data-dependence graph will be ultimately simplified when the values of the parameters are known. Each solvent system associated with the nodes and arcs will be checked for consistency: some of the “conditional” nodes and arcs will disappear if inconsistency of the respective solvent systems is ascertained. Note that the inconsistency of a system associated with a node entails the inconsistency of the systems attached to all the incident arcs to/from the node. Consequently, a “conditional” node will disappear together with all the arcs connected to it. Therefore, the computation is more effective if the “conditional” nodes are processed first, followed by the “conditional” arcs afterwards.

The memory size computation can be subsequently performed using the final data-dependence graph, similar to the “non-parametric” case (Section 3).

3.2 Specifications with parallelism instructions

In order to handle appropriately high throughput applications, any design space exploration system should be extended with a preprocessing stage: exploiting the (initially unknown) parallelism existent in the source code of the specification. Two steps should be performed when dealing with high throughput applications:

1. Extracting the (largely hidden) parallelism from the initially specified code. This can be done either manually – using parallelizing code transformations [21, 13], or automatically – using parallelizing compilers.
2. Finding the lowest degree of parallelism in the specification, sufficient to meet the throughput and hardware constraints of the design.

Although the two topics above are beyond the scope of this paper, we believe that a good memory size computation approach should be able to deal with explicit parallel instructions. A first attempt in this direction was reported by Grun *et al.* [9] who proposed an estimation technique computing lower and upper memory bounds with an gradual precision increase.

The algebraic memory size computation approach presented in this paper – based on evaluating the memory required by the computation of a polyhedral data-dependence graph derived from the algorithmic specification – can be naturally extended to handle parallel loops of the following types:

- a) *forall* loop. This type of loop is intended mainly to capture the array assignment behavior. If a *forall* loop contains a single assignment, then it is executed just like an array assignment. If the *forall* loop contains several assignments, then each statement is executed completely for all values of the iterator before the next statement is started.
- b) *dopar* loop. This type of loop is designed for multiprocessor execution. Each iteration of the loop is executed in parallel by a different processor, the code within each iteration being executed sequentially. If two different iterations change the same variable, each iteration sees only its own modifications.
- c) *doall* loop. This type is actually a special case of *dopar* loops without conflicts between iterations, i.e., where an assigned variable is used only in that iteration. In such a case, executing the iterations sequentially or in parallel in any order is legal, since the result does not depend on the computation order. The strategy of dealing with *dopar* loops, for instance, is to refine the polyhedral data-dependence graph by decomposing the nodes corresponding to the array references within the *dopar* loop. This decomposition can be carried out by slicing the n -dimensional iterator polytopes $\mathbf{A} \cdot \mathbf{i} \geq \mathbf{b}$ into sets of $(n-1)$ -dimensional polytopes (where n is the depth of the loop nest). The slicing operation must be done along the direction of the *dopar* loop iterator, using the pairwise (Fourier-Motzkin) elimination technique [6]. The data-dependence graph will be expanded, containing disjoint groups of nodes corresponding to each iteration in the *dopar* loop. The memory size computation algorithm can afterwards operate separately on these disjoint groups of nodes.

Application	Scalars	Memory	CPU [m:s]
SVD updating	25,717	1410	2:05
Motion detection	72,543	1371	8:28
Vocoder kernel	28,559	12144	6:55

Table 1: Experimental results for signal processing applications

4 Experimental results

The implementation of the presented approach was done in C++ and was tested on a Sun Blade 100 workstation. For the time being, the extensions to handle parametric specifications and explicit parallelism are still under development. The evaluation of the novel technique has been carried out on several multidimensional signal processing applications like, for instance: (1) A singular value decomposition (SVD) updating algorithm [12] for use in spatial division multiplex access (SDMA) modulation in mobile communication receivers. The SVD updating is an important algebraic kernel used, for instance, in beamforming, recursive least squares estimation, and Kalman filtering. For mobile telephony, signals on the same carrier frequency but arriving from different locations can be separated by means of antenna arrays and advanced computational techniques (as, e.g., SVD updating), which allows to enlarge significantly the network capacity. (2) A motion detection algorithm for use in the transmission of real-time video signals on data networks. The video signals must be compressed by video-coders/decoders to fit in the (limited) bandwidth of the network. An approximate but effective algorithm uses the idea of motion compensation – the motion estimation of part of the image between consecutive frames. (3) The kernel of a voice coding application – essential component of a mobile radio terminal.

Table 1 summarizes the results of our experiments. The computation times are quite large due to the exponential complexity of some of the techniques employed (e.g., the Fourier-Motzkin elimination [6]). However, a scalar approach based on computation simulation took several hours to complete.

5 Conclusions

This paper has presented a non-scalar model for computing the memory size required in real-time multimedia algorithms, where the storage of large multidimensional signals causes a significant cost in terms of both area and power consumption. The approach is based on recent algebraic techniques specific to the data-flow analysis used in modern compilers. This model will be used in the synthesis of a multilevel memory architecture optimized for area and/or power, subject to timing constraints.

References

- [1] F. Balasa, F. Catthoor, H. De Man, "Exact evaluation of memory area for multi-dimensional signal processing systems," *Proc. IEEE Int. Conf. on CAD*, pp. 669-672, Santa Clara, Nov. 1993.
- [2] A.I. Barvinok, "A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed," *Mathematics of Operations Research*, vol. 19, no. 4, pp. 769-779, Nov. 1994.
- [3] 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.
- [4] F. Catthoor, E. Brockmeyer, K. Danckaert, C. Kulkarni, L. Nachtergaele, A. Vandecappelle, "Custom memory organization and data transfer: architecture issues and exploration methods," chapter in *VLSI section of Electrical and Electronics Engineering Handbook* (ed. M. Bayoumi), Academic Press, 2000.
- [5] Ph. Clauss, "Counting solutions to linear and nonlinear constraints through Ehrhart polynomials," *Proc. ACM Int. Conf. on Supercomputing*, pp. 278-285, 1996.
- [6] G.B. Dantzig, B.C. Eaves, "Fourier-Motzkin elimination and its dual," *J. Combinatorial Theory (A)*, vol. 14, pp. 288-297, 1973.
- [7] R.J. Duffin, "On Fourier's analysis of linear inequality systems," *Mathematical Programming Study*, vol. 1, pp. 71-95, 1974.
- [8] G. Goossens, *Optimization Techniques for Automated Synthesis of Application-specific Signal-processing Architectures*, Ph.D. thesis, K.U. Leuven, Belgium, 1989.
- [9] P. Grun, F. Balasa, N. Dutt, "Memory size estimation for multimedia applications," *Proc. 6th Int. Workshop on Hardware/Software Co-Design*, pp. 145-149, Seattle WA, March 1998.
- [10] P.G. Kjeldsberg, F. Catthoor, E.J. Aas, "Storage requirement estimation for data-intensive applications with partially fixed execution ordering," *Proc. 8th Int'l Workshop on Hardware/Software Co-Design*, pp. 56-60, San Diego CA, May 2000.
- [11] F.J. Kurdahi, A.C. Parker, "REAL: A program for register allocation," *Proc. 24th Design Automation Conf.*, pp. 210-215, 1987.
- [12] M. Moonen, P. Van Dooren, J. Vandewalle, "SVD-updating for tracking slowly time-varying systems," *Advanced alg. and architectures for signal processing*, *Proc. SPIE*, Vol. 1152, 1989.
- [13] S.S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Francisco, 1997.
- [14] P.R. Panda *et al.*, "Data and memory optimization techniques for embedded systems," *ACM Trans. on Design Automation of Electronic Systems*, Vol. 6, No. 2, pp. 149-206, April 2001.
- [15] K.K. Parhi, "Calculation of minimum number of registers in arbitrary life time chart," *IEEE Trans. Circ. & Syst. - II: Analog and Digital Signal Processing*, Vol. 41, No. 6, pp. 434-436, 1994.
- [16] W. Pugh, "A practical algorithm for exact array dependence analysis," *Comm. of the ACM*, vol. 35, no. 8, pp. 102-114, Aug. 1992.
- [17] A. Schrijver, *Theory of Linear and Integer Programming*, John Wiley, New York, 1986.
- [18] L. Stok, J. Jess, "Foreground memory management in data path synthesis", *Int. Journal on Circuit Theory and Appl.*, Vol. 20, pp. 235-255, 1992.
- [19] L. Thiele, "Compiler techniques for massive parallel architectures," in *State-of-the-art in Computer Science* (ed. P. Dewilde), Kluwer Acad. Publ., 1992.
- [20] I. Verbaauwhede, C. Scheers, J.M. Rabaey, "Memory estimation for high level synthesis," *Proc. 31st ACM/IEEE Design Automation Conf.*, pp. 143-148, June 1994.
- [21] M.E. Wolf, M.S. Lam, "A loop transformation theory and algorithm to maximize parallelism," *IEEE Trans. on Parallel and Distributed Syst.*, Vol. 2, No. 4, pp. 452-471, Oct. 1991.
- [22] Y. Zhao, S. Malik, "Exact memory size estimation for array computations," *IEEE T-VLSI Syst.*, Vol. 8, No. 5, pp. 517-521, 2000.