Memory Size Computation for Multimedia Processing Applications*

Hongwei Zhu Ilie I. Luican Florin Balasa

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

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 the memory modules. The computation of the memory size is an important step in the process of designing an optimized (for area and/or power) memory architecture for multimedia processing systems. This paper presents a novel non-scalar approach for computing *exactly* the memory size in real-time multimedia algorithms. This methodology uses both algebraic techniques specific to the data-flow analysis used in modern compilers, and also recent advances in the theory of integral polyhedra. In contrast with all the previous works which are only *estimation* methods, this approach performs *exact* memory computations even for applications with a large number of scalar signals.

1 Introduction

In real-time multimedia processing systems – including video and 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 [4]. 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-level (RTL) programs at scalar level [8, 11, 14] and in behavioral specifications at non-scalar level [2, 7, 16, 17]. Good overviews of these techniques can be found in [4, 10].

This paper presents a non-scalar method for computing *exactly* the memory size in real-time multimedia algorithms (assuming the code is procedural). This approach uses both algebraic techniques specific to the data-flow analysis used in modern compilers [9], and recent advances in the theory of integral (n-dimensional) polyhedra. In contrast with previous works which utilize only *approximate* methods due to the size of the problems (in terms of number of scalars and number of array references), *this approach obtains exact determinations even for applications significantly large*. Since the mathematical model is very general, this novel approach is able to handle the entire class of "affine" specifications (see Section 2), therefore practically the entire class of real-time multimedia applications.

The paper is organized as follows. Section 2 explains the problem of memory size computation. The core of the paper – Sections 3 and 4 – presents the technical aspects of this novel approach. Section 5 will briefly discuss implementation aspects and present several experimental results. Section 6 will summarize the main conclusions of this work.

2 The memory size computation problem

The (real-time) multimedia processing algorithms are typically specified in a high-level programming language, where the code is organized in sequences of loop nests having as boundaries linear functions of the outer loop iterators, conditional instructions where the conditions may be both data-dependent or data-independent (relational and/or logical operators of linear functions of loop iterators), and multidimensional signals which array references have (complex) linear indices. This class of specifications is often referred to as *affine*. Sometimes, in image and video processing, there may be also indices containing modulo operators, but these situations can be brought into the affine specification class [4].

Real-time multimedia algorithms describe the processing of streams of data samples. The source code of these algorithms can be imagined as surrounded by an implicit loop having the $time$ as iterator. Consequently, each signal in the algorithm has an *implicit* extra dimension corresponding to the $time$ axis. These algorithms often contain $delayed$ signals, i.e., signals produced (or inputs) in previous data-sample processings, which are consumed during the current sample processing. The delay operator "@" indicates such delayed signals, the following argument signifying the number of previous samples. The delayed signals must be kept "alive" during several *time* iterations, i.e., they must be stored in the background memory during one or several data-sample processings.

An illustrative example, derived from a motion detection algorithm [4], is given below:

$optDlt[0] = 0;$

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

0-7803-9451-8/06/$20.00 ©2006 IEEE.

```
for (i = 8; i ≤ 120; i++)
  for (j = 8; j ≤ 120; j++)
  { Delta[i][j][0] = 0;
    for (k = i − 8; k ≤ i + 8; k++)
      for (l = j − 8; l ≤ j + 8; l++)
        Delta[i][j][17(k − i) + l − j + 145] = A[i][j] − A[k][l]@1
                    +Delta[i][j][17(k − i) + l − j + 144];
    optDlt[113i+j−911] = Delta[i][j][289]+optDlt[113i+j−912];
  }
opt = optDlt[12769];
```

The problem is to determine the *minimum* amount of memory locations necessary to store the signals of a given multimedia algorithm during its execution, or, equivalently, the *maximum* storage occupancy assuming any scalar signal must be stored only during its lifetime. The total number of scalars in the algorithm above is 3,749,063. But due to the fact that scalars having disjoint lifetimes can share the same memory location, the amount of storage can be much smaller than the total number of scalar signals. Actually, only 33,284 memory locations are necessary for this example.

It must be emphasized that image and video processing applications contain deeper loop nests with iterators having typically large ranges, resulting in extremely large numbers of scalar signals. Enumerative techniques or RTL approaches based on the left edge algorithm [8], although appealing by means of simplicity, are too computationally expensive in such cases, often prohibitive to use. For multimedia algorithmic specifications, the algebraic techniques are the only hope.

All the past works, for instance [16, 2, 17, 7], achieved only a *memory size estimation*, rather than an *exact size computation*. The algorithm presented in this paper is the first one – to the best of our knowledge – able to compute *exactly* the storage requirements for multimedia applications, even when the number of scalar signals is very large. The basic reasons of its efficiency are: (a) the use of a relatively recent mathematics advance – the polynomial-time decomposition of an n-dimensional polyhedron into an algebraic sum of unimodular cones [3], (b) the efficient decomposition of the array references of the multidimensional signals in disjoint *linearly bounded lattices* (LBL's) [15], and (c) an efficient mechanism of pruning the code of the algorithmic specification.

Note that the problem of organizing the signals in a distributed (hierarchical) memory architecture, often referred to as the problem of *memory allocation* is beyond the scope of this paper.

3 Computation of array reference size

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

Each *array reference* $M[x_1(i_1, \ldots, i_n)] \cdots [x_m(i_1, \ldots, i_n)]$ of an m-dimensional signal M, in the scope of a nest of n loops having the iterators $i_1, \ldots, i_n$, is characterized by an *iterator space* and an *index space*. The iterator space signifies the set of all iterator vectors $\mathbf{i} = (i_1, \ldots, i_n) \in \mathbf{Z}^n$ in the scope of the array reference. The index space is the set of all index vectors $\mathbf{x} = (x_1, \ldots, x_m) \in \mathbf{Z}^m$ of the array reference. When the indices of an array reference are linear mappings with integer coefficients of the loop iterators, the index space consists of one or several *linearly bounded lattices* (LBL) [15] – the image of an affine vector function over the iterator polytope $\mathbf{A} \cdot \mathbf{i} \geq \mathbf{b}$:

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

where $\mathbf{x} \in \mathbf{Z}^m$ is the index vector of the m-dimensional signal and $\mathbf{i} \in \mathbf{Z}^n$ is an n-dimensional iterator vector (see the example below).

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 in an array reference, that is, how many locations are needed to store one array reference.

If the rank of matrix $\mathbf{T}$ is equal to its number of columns, then the vector function $\mathbf{x} = \mathbf{T} \cdot \mathbf{i} + \mathbf{u}$ between the iterator and index spaces is proven to be a one-to-one mapping [2], and the computation of the number of distinct signal indices (i.e., the amount of memory necessary to store the scalars covered by the array reference) is hence reduced to counting the number of iterator vectors or, equivalently, the number of lattice points (i.e., points having integer coordinates) in the iterator polytope $\mathbf{A} \cdot \mathbf{i} \geq \mathbf{b}$ in (1). In such a situation, a computation technique based on Barvinok's decomposition of a simplicial cone into unimodular cones [3] is used.

An example is given below, illustrating both the concepts and the technique. Note that due to space limitation, several details of the computation had to be skipped, along with part of the theoretical justifications. However, this example succeeds to illustrate well enough the main steps of the computation flow. Moreover, it will show clearly why this approach can handle algorithmic specifications typical to multimedia applications.

Example 1:
```
for (i = 0; i ≤ 4; i++)
    for (j = 0; j ≤ 2i  &&  j ≤ −i + 6; j++)
        ··· A[2i + 3j][5i + j] ···
```

How many memory locations are necessary to store the array reference $A[2i + 3j][5i + j]$? The linearly bounded lattice (LBL) corresponding to this array reference is $\{\mathbf{x} = \mathbf{T}\mathbf{i} + \mathbf{u} \,|\, \mathbf{A}\mathbf{i} \geq \mathbf{b}\} =$

$$\left\{ \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 & 3 \\ 5 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \;\middle|\; \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 2 & -1 \\ -1 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \geq \begin{bmatrix} 0 \\ -4 \\ 0 \\ 0 \\ -6 \end{bmatrix} \right\}$$

(**u=0** here) and the problem is equivalent to computing the size of this set. Since post-multiplying $\mathbf{T}$ with the unimodular matrix[1] $\mathbf{S} = \begin{bmatrix} -1 & 3 \\ 1 & -2 \end{bmatrix}$ results in $\mathbf{T} \cdot \mathbf{S} = \begin{bmatrix} 1 & 0 \\ -4 & 13 \end{bmatrix}$, the rank of matrix $\mathbf{T}$ is 2 – equal to its number of columns; therefore, the vector function **x=Ti + u** is a one-to-one mapping. The computation of the number of scalars covered by $A[2i + 3j][5i + j]$ is equivalent to counting the number of lattice points in the iterator

[1] A square matrix with integer elements having the determinant equal to $\pm$ 1.

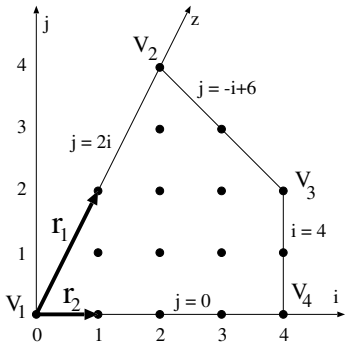

Figure 1: Convex polytope representing the iterator space in *Example 1*

polytope $\mathbf{Ai} \geq \mathbf{b}$ shown in Fig. 1. This latter operation is done as explained below. But, first, a few definitions are necessary.

Definitions Let $r_1, \ldots, r_d \in \mathbf{Z}^d$ be linearly independent integer vectors. The (rational polyhedral) *cone* generated by the *rays* $r_1, \ldots, r_d$ is the set $C(r_1, \ldots, r_d) = \{\sum_1^d \alpha_i r_i, \ \alpha_i \geq 0\}$. For instance, the set of points inside the angle iV_1z is a 2-dimensional cone generated by the rays $r_1 = [1\ 2]^T$ and $r_2 = [1\ 0]^T$ (see Fig. 1). To each vertex of a polyhedron corresponds a *supporting* cone. The supporting cone of the vertex V_1 (Fig. 1), denoted $C(V_1)$, is the one generated by the rays r_1 and r_2. A cone is called *unimodular* if the matrix of the rays $[r_1 \ \cdots \ r_d]$ is unimodular (i.e., its determinant is ± 1).

Step 1 Find the vertices of the iterator polytope $\mathbf{Ai} \geq \mathbf{b}$ and their supporting cones.

Given the inequalities $\{0 \leq i \leq 4,\ 0 \leq j \leq 2i,\ j \leq -i+6\}$ defining the iterator space, the vertices and the rays are computed using the *reverse search* algorithm [1]. The supporting cones corresponding to the vertices $V_1, \ldots, V_4$ of the iterator polytope, as well as their generating rays shown below as column vectors, are:

$$C(V_1) = \left\{ \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right\}, \ C(V_2) = \left\{ \begin{pmatrix} -1 \\ -2 \end{pmatrix}, \begin{pmatrix} 1 \\ -1 \end{pmatrix} \right\},$$

$$C(V_3) = \left\{ \begin{pmatrix} -1 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ -1 \end{pmatrix} \right\}, \ C(V_4) = \left\{ \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}$$

Step 2 Apply Barvinok's algorithm [3] to decompose the supporting cones into unimodular cones.

The first two cones in our example are not unimodular. Their decomposition is given below, without any additional explanation due to lack of space:

$$C(V_1) = \oplus \left\{ \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right\} \oplus \left\{ \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \begin{pmatrix} 0 \\ -1 \end{pmatrix} \right\} \quad (2)$$

$$C(V_2) = \oplus \left\{ \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \begin{pmatrix} -1 \\ -2 \end{pmatrix} \right\} \ominus \left\{ \begin{pmatrix} -1 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ -1 \end{pmatrix} \right\}$$

Step 3 Find out the generating function of each supporting cone.

The above decomposition is performed since any unimodular cone $C(V)$ has associated a generating function [3] of the form

$$F(V) = \frac{z^V}{\Pi_i (1 - z^{r_i})}$$

where $z^V = x^a y^b$ (since a vertex $V(a,b)$ has two coordinates a, b in this case), and the product is over all the generating rays r_i. For instance, if the vertex of the cone is $V = V_1(0,0)$ then $z^V = x^0 y^0 = 1$; if the ray $r_i = [1\ 2]^T$ then $z^{r_i} = x^1 y^2$. The generating function of any cone is obtained making the summation of the functions of all the unimodular component cones. Therefore, from (2), the generating function of the cone $C(V_1)$ is

$$F(V_1) = \frac{1}{(1-y)(1-x)} + \frac{1}{(1-xy^2)(1-y^{-1})}$$

With similar computations, the generating functions for the other supporting cones are:

$$F(V_2) = \frac{x^2y^4}{(1-y^{-1})(1-x^{-1}y^{-2})} - \frac{x^2y^4}{(1-x^{-1}y)(1-y^{-1})}$$

$$F(V_3) = \frac{x^4y^2}{(1-x^{-1}y)(1-y^{-1})}, \quad F(V_4) = \frac{x^4}{(1-x^{-1})(1-y)}$$

The sum of these rational functions yields the generating function F of the whole quadrilateral in Fig. 1.

Step 4 Compute the number of lattice points from the generating function $F = \sum_i F(V_i)$ of the whole polytope.

In order to obtain a one-variable generating function F, we make the substitution $z \longrightarrow t^\lambda$, where λ is an integer vector chosen such that no dot product of λ with any generating ray is zero [6]. In this example, we choose $\lambda = [1\ -1]^T$ (there is an algorithm for this as well). With the substitution $x = t$, $y = t^{-1}$, the generating function of the iterator polytope becomes:

$$F = \begin{array}{l} \frac{1}{(1-t^{-1})(1-t)} + \frac{1}{(1-t^{-1})(1-t)} + \frac{t^{-2}}{(1-t)^2} \\ -\frac{t^{-2}}{(1-t^{-2})(1-t)} + \frac{t^2}{(1-t^{-2})(1-t)} + \frac{t^4}{(1-t^{-1})^2} \end{array}$$

After eliminating the negative exponents in the denominators, and after factorizing t^{-2}, we substitute $t = s+1$, obtaining rational terms of the form $\frac{P(s)}{s^d Q(s)}$, where $P(s)$ and $Q(s)$ are polynomials, and $d = 2$ is the dimension of the iterator space:

$$F = \frac{-(s+1)^3}{s^2} + \frac{-(s+1)^3}{s^2} + \frac{1}{s^2} - \frac{-(s+1)^2}{s^2(s+2)} + \frac{-(s+1)^6}{s^2(s+2)} + \frac{(s+1)^8}{s^2}$$

If $P(s) = a_0 + a_1 s + a_2 s^2 + \ldots$ and $Q(s) = b_0 + b_1 s + b_2 s^2 + \ldots$, the coefficients of the quotient $P(s)/Q(s) = c_0 + c_1 s + c_2 s^2 + \ldots$ can be obtained recursively as follows [6]:

$$c_0 = \frac{a_0}{b_0} \text{ and } c_k = \frac{1}{b_0}(a_k - b_1 c_{k-1} - b_2 c_{k-2} - \ldots - b_k c_0) \text{ for } k \geq 1$$

The algebraic sum of the coefficients c_2 (since here the space dimension is 2) after the polynomial divisions in all the terms of F is the number of lattice points [3]. In this example, the 6 coefficients c_2 (one for each term of F) are $\{-3, -3, 0, \frac{1}{8}, -\frac{49}{8}, 28\}$. Their sum yields 16, which is indeed the number of lattice points inside (or on the border of) the iterator polytope in Fig. 1, and it is also the number of memory locations to store the array reference $A[2i+3j][5i+j]$ since the vector function **x=Ti+u** from the iterator to the index space is a one-to-one mapping.

Assume now that the range of the first iterator in *Example 1* is 0 to 400 (rather than 0 to 4) and in the second loop the condition $j \leq -i + 6$ is replaced by $j \leq -i + 600$. The iterator polytope is a quadrilateral similar with the one in Fig. 1, but much larger, the similarity ratio being 100. The computation effort necessary to find the number of memory locations for the array reference $A[2i + 3j][5i + j]$ is not affected by the very significant increase in size of the iterator space. Indeed, the 4 supporting cones are generated by the same rays, the decompositions are the same, the generating functions are almost the same. The only difference appears at the numerators of $F(V_2)$, $F(V_3)$, and $F(V_4)$ due to the modifications of the coordinates of these vertices. For instance, the numerator of $F(V_3)$ becomes $x^{400}y^{200}$ since the new coordinates of V_3 are (400,200). The storage requirement for this case is 100,501 locations. Note that the number of lattice points does not scale up with the square of the similarity ratio like, for instance, the area of the quadrilateral.

Moreover, the technique sketched above, although illustrated for a 2-dimensional signal in the scope of an iterator space of dimension 2, works for arbitrary numbers of dimensions of both the index and iterator spaces. Therefore, it is well-suited to address the size of array references typical to multimedia applications. □

The example above illustrated the case when there is a one-to-one mapping between the iterator and index spaces. But this is not always true. When the rank r of matrix $\mathbf{T}$ is smaller than n, the number of columns of $\mathbf{T}$, the memory occupied by the array reference is upper bounded by the number of lattice points in the r-dimensional polytope $pr_r(\mathbf{Ai} \geq \mathbf{b})$ – the real projection of $\mathbf{Ai} \geq \mathbf{b}$ on $\mathcal{R}^r$ along the first r coordinates. $pr_r(\mathbf{Ai} \geq \mathbf{b})$ can be easily computed by eliminating the last $n - r$ iterators in $\mathbf{Ai} \geq \mathbf{b}$ with the Fourier-Motzkin technique [5]. It must be noticed that not necessarily all the lattice points in $pr_r(\mathbf{Ai} \geq \mathbf{b})$ represent projections of lattice points from $\mathbf{Ai} \geq \mathbf{b}$ [12]. These invalid projections are detected by replacing the r coordinates of the projection point under question in the polytope $\mathbf{Ai} \geq \mathbf{b}$ and checking if the resulting $(n - r)$-dimensional integral polytope is empty.

Example 2: $for\ (i = 0;\ i \leq 4;\ i++)$
$for\ (j = 0;\ j \leq 2i\ \&\&\ j \leq -i + 6;\ j++)\ \cdots A[3i + j]$

Since post-multiplying $\mathbf{T} = \begin{bmatrix} 3 & 1 \end{bmatrix}$ with the unimodular matrix $\mathbf{S} = \begin{bmatrix} 0 & 1 \\ 1 & -3 \end{bmatrix}$ results in $\mathbf{T} \cdot \mathbf{S} = \begin{bmatrix} 1 & 0 \end{bmatrix}$, the rank of matrix $\mathbf{T}$ is $r = 1$ – less than the number of columns $n = 2$ of $\mathbf{T}$; in this case, the vector function **x=Ti + u** may not be a one-to-one mapping. Indeed, the iterator vectors $[i\ j]^T = [2\ 3]^T$ and $[3\ 0]^T$ from the iterator space in Fig. 1 are mapped to the same index $3i + j = 9$. The transformation $\mathbf{S}$ modifies the iterator space into $\{\mathbf{AS} \cdot \mathbf{j} \geq \mathbf{b}\}$=$\{\ 3l \leq k \leq 5l,\ k \leq 2l + 6,\ l \leq 4\ \}$ ($\mathbf{j} \stackrel{not}{=} [k\ l]^T = \mathbf{S}^{-1}[i\ j]^T$ is the new iterator vector after the transformation $\mathbf{S}$) which real projection is $\{\ 0 \leq k \leq 14\ \}$, obtained eliminating l in the inequalities above. But not all these 15 points are valid projections. $k = 1,\ 2$ result to be invalid: replacing these values in the modified iterator space, no integer solution for l can be found. Therefore, storing $A[3i + j]$ requires 15-2=13 locations.

The algorithm described above is implemented in our memory computation tool (see Section 5).

4 Memory size computation algorithm based on data-dependence analysis

The main steps of the memory size computation algorithm will be discussed below.

Step 1 Extract the array references from the given algorithmic specification of the multimedia application and decompose the array references for every indexed signal into *disjoint* linearly bounded lattices.

The analytical partitioning of the array references of every signal into disjoint LBL's can be performed by a recursive intersection, starting from the array references in the code. 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 system[2] $\mathbf{T}_1\mathbf{i}_1 - \mathbf{T}_2\mathbf{i}_2 = \mathbf{u}_2 - \mathbf{u}_1$ 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 coalesced constraints of the two LBL's (denoted Lbl_1 and Lbl_2)

$$\begin{aligned} \mathbf{A_1V_1} \cdot \mathbf{i} &\geq \mathbf{b_1} - \mathbf{A_1v_1} \\ \mathbf{A_2V_2} \cdot \mathbf{i} &\geq \mathbf{b_2} - \mathbf{A_2v_2} \end{aligned} \tag{3}$$

has integer solutions, then the intersection is a new LBL:

$Lbl_1 \cap Lbl_2 = \{\mathbf{x} = \mathbf{T_1V_1} \cdot \mathbf{i} + \mathbf{T_1v_1} + \mathbf{u_1} |$ s.t. constraints (3)$\}$

However, the real difficulty is the decomposition of the differences $Lbl_1 - (Lbl_1 \cap Lbl_2)$ and $Lbl_2 - (Lbl_1 \cap Lbl_2)$, and the reason is that the difference of two LBL's is not necessarily an LBL. Due to the present space limitation, the full LBL decomposition algorithm will be published elsewhere.

Example: The disjoint LBL's of signal $Delta$ from the illustrative example in Section 2 are (in non-matrix format):

$Delta_1 = \{\ x = i,\ y = j,\ z = 0 \mid 120 \geq i,\ j \geq 8\ \}$
$Delta_2 = \{\ x = i,\ y = j,\ z = 289 \mid 120 \geq i,\ j \geq 8\ \}$
$Delta_3 = \{\ x = i,\ y = j,\ z = 17(k-i)+l-j+145 \mid 120 \geq i,\ j \geq 8$
$8 \geq k - i,\ l - j \geq -8$ and $143 \geq 17(k - i) + l - j\ \}$

Figure 2 shows a polyhedral dependence graph built from the illustrative example in Section 2, where the nodes are the disjoint LBL's determined at this step and the arcs are the dependence relations between them derived from the code. The nodes are labeled with the number of scalar signals they cover and the arcs are labeled with the number of dependencies (both computed using the algorithm from Section 3).

[2] 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 [13].

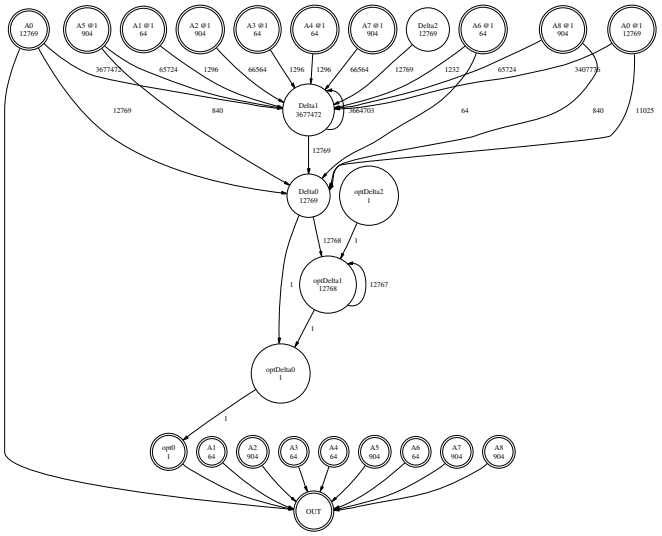

Figure 2: Polyhedral dependence graph having as nodes the disjoint linearly bounded lattices from the example in Section 2

Step 2 Determine the memory size at the boundaries between the blocks of code.

The algorithmic specification is a sequence of nested loops, referred also as *blocks*. After the decomposition of the array references, for each disjoint LBL it is determined the block where the LBL is created (i.e., *produced*), and the block where it is used as an operand for the last time (i.e., *consumed*). Based on this information, the memory size between the blocks can be determined *exactly*, since the storage requirement of each disjoint LBL can be computed *exactly* – using the algorithm explained in Section 3.

Step 3 Pruning the algorithmic specification.

If in a block signals are produced but no signal is last time consumed, that block is irrelevant in memory size point of view and can be skipped from further analysis since the memory will only increase to the amount at the end of the block – which is already known from *Step 2*. Similarly, if the amount of storage required by the newly created LBL's, together with the amount of memory at the beginning of the block, is not larger than the maximum storage at the boundary level, then that block can be pruned as well. This pruning speeds up the tool, concentrating the analysis on those portions of code where the memory increase is likely to happen.

Step 4 For each of the remaining blocks of code, compute the maximum memory size inside the block. This operation is based on the computation of min/max iterator vectors relative to the lexicographic order.

Definition Let $\mathbf{i} = [i_i, \ldots, i_n]^T$ and $\mathbf{j} = [j_i, \ldots, j_n]^T$ be two iterator vectors in the scope of n nested loops, which may be assumed "normalized" (i.e., all the iterators are increasing with the step 1). Iterator vector $\mathbf{j}$ is larger lexicographically than $\mathbf{i}$ (written $\mathbf{j} \succ \mathbf{i}$) if ($j_1 > i_1$), or ($j_1 = i_1$ and $j_2 > i_2$), or ... ($j_1 = i_1, \ldots, j_{n-1} = i_{n-1}$, and $j_n > i_n$). The min/max iterator vector from a set of such vectors is the smallest/largest vector in the set relative to the lexicographic order.

Example 3:
$$for\ (i = 0;\ i \leq 3;\ i++)$$
$$for\ (j = 0;\ j \leq 3;\ j++)$$
$$for\ (k = 0;\ k \leq 3;\ k++) \quad \cdots A[i+j+k] \cdots$$

The max iterator vector addressing the scalar, say, $A[5]$ is $[i\ j\ k]^T_{max} = [3\ 2\ 0]^T$, while the min iterator vector is $[0\ 2\ 3]^T$.

Our algorithm finds the LBL's produced and consumed in the current block, computing the min and, respectively, max iterator vectors for the scalar signals covered by these LBL's since these iterator vectors correspond to the increase and, respectively, decrease of the memory. Knowing the number of flops (i.e., elementary iterations) in the loop nest (by counting the lattice points in the iterator spaces with the algorithm in Section 3), one can then determine *exactly* the memory variation and, in particular, the maximum storage amount in each of the blocks. Actually, part of the LBL's produced or consumed in the block can be conveniently skipped if their effect on the memory variation can be taken into account without generating the scalars they cover. For instance, in the illustrative example from Section 2, each iterator vector $[i\ j\ k\ l]^T$ corresponds to a unique produced scalar $Delta[i][j][17(k-i)+l-j+145]$ and a unique consumed scalar $Delta[i][j][17(k-i)+l-j+144]$. The effect of the two array references on the memory variation is +1-1=0 in each iteration and, therefore, these operands can be skipped from further analysis, pruning that increases significantly the computation speed.

5 Experimental results

A memory size computation tool (named $K2$ after the famous peak which climbing adversity intends to suggest the difficulty of its implementation) 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 (see, e.g., the illustrative example in Section 2). This is not a restrictive feature of the theoretical model since any modification in the specification language would affect only the front-end of the tool. In addition to the computation of the minimum memory size requirements and different statistical data on the memory usage by the multidimensional signals in the multimedia specification, the tool can optionally generate dependence graphs (like the one in Fig. 2) at different granularity levels, which provide information about the relations between different groups of signals, and also the trace of the memory occupancy during the execution of the input specification. Such a memory trace is shown in Fig. 3.

Table 1 summarizes the results of our experiments, carried out on a Sun Blade 100 workstation. The benchmarks used are: (1) Durbin's algorithm which solves a Toeplitz system with N unknowns [16], (2) a real-time regularity detection algorithm used in robot vision, (3) a 2D Gaussian blur filter from a medical image processing application which extracts contours from tomograph 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], and (5) the kernel of a voice coding application – essential component of a mobile radio terminal.

This tool can process large specifications in terms of number

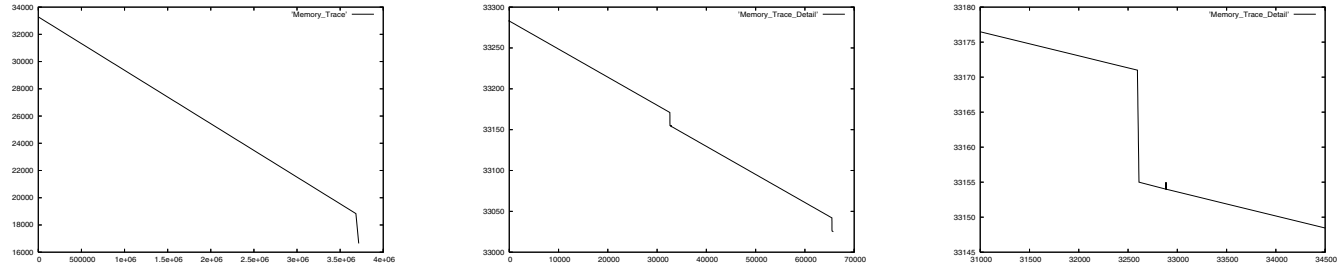

Figure 3: Memory trace for the illustrative example in Section 2. The abscissae are the numbers of datapath instructions in the code, the ordinates are memory locations. The first graphic represents the entire trace. The second graphic is a detailed trace in the interval [0 : 65767], which corresponds to the first two iterations of the outer loop ($i = 8$ and 9). The third graphic is a detailed trace in the zone covering the end of the first outer-loop iteration and the start of the second one. The global maximum is at the point (x=2, y=33284).

Application	Parameters	Memory	CPU
Durbin alg.	N=100	249	$<$ 1s
Regularity detection	MaxGrid=5, L=64	960	$<$ 1s
2D Gauss. blur filter	M=N=50	14451	2s
Motion detection	M=N=32, m=n=4	2740	16s
Vocoder kernel	–	11890	7s

Table 1: Experimental results (col. 3 shows the memory locations)

of loop nests, lines of code, number of array references. For instance, the voice coding application contains 236 array references organized in 40 loop nests. In one of our experiments, the illustrative example (Section 2) was unrolled one loop level, resulting in a code with 113 loop nests 3 level deep, and a total of 906 array references, many having complex indices. The tool processed this example in about 8 minutes.

A comparative evaluation with previous works performing memory size *estimation* is hard to do in the absence of their benchmark algorithms. For instance, the test of the motion detection kernel is referred also in [17]. If the authors used like us the algorithm given in [4], then their result (of 1372 memory locations) is a poor estimation since the correct *exact* result for the same parameters is 2740 (see Table 1).

6 Conclusions

This paper has presented a non-scalar approach for computing the memory size in real-time multimedia algorithms, where the storage of large multidimensional signals causes a significant cost in terms of both area and power consumption. This method uses modern elements in the theory of polyhedra and algebraic techniques specific to the data-flow analysis used nowadays in compilers. Different from past works which were only performing a *memory size estimation*, our approach does *exact* computations and it is the first one to do so.

References

[1] D. Avis, "lrs: A revised implementation of the reverse search vertex enumeration algorithm," in *Polytopes – Combinatorics and Computation*, G. Kalai, G. Ziegler, Birkhauser-Verlag, pp. 177-198, 2000.

[2] F. Balasa, F. Catthoor, H. De Man, "Background memory area estimation for multi-dimensional signal processing systems," *IEEE Trans. VLSI Syst.*, vol. 3, no. 2, pp. 157-172, June 1995.

[3] 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.

[4] 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.

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

[6] J.A. De Loera, R. Hemmecke, J. Tauzer, R. Yoshida, "Effective lattice point counting in rational convex polytopes," *http://www.math.ucdavis.edu/∼latte/pdf/lattE.pdf*, 2003.

[7] P.G. Kjeldsberg, F. Catthoor, E.J. Aas, "Data dependency size estimation for use in memory optimization," *IEEE Trans. CAD of IC's and Syst.*, vol. 22, no. 7, pp. 908-921, July 2003.

[8] F.J. Kurdahi, A.C. Parker, "REAL: A program for register allocation," *Proc. 24th Design Automation Conf.*, pp. 210-215, 1987.

[9] S.S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Francisco, 1997.

[10] P.R. Panda, F. Catthoor, N. Dutt, K. Dankaert, E. Brockmeyer, C. Kulkarni, P.G. Kjeldsberg, "Data and memory optimization techniques for embedded systems," *ACM Trans. Design Automation of Electronic Syst.*, vol. 6, no. 2, pp. 149-206, April 2001.

[11] 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.

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

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

[14] L. Stok, J. Jess, "Foreground memory management in data path synthesis", *Int. J. Circuit Theory and Appl.*, vol. 20, pp. 235-255, 1992.

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

[16] I. Verbauwhede, C. Scheers, J.M. Rabaey, "Memory estimation for high level synthesis," *Proc. 31st ACM/IEEE Design Automation Conf.*, pp. 143-148, June 1994.

[17] Y. Zhao, S. Malik, "Exact memory size estimation for array computations," *IEEE Trans. VLSI Syst.*, vol. 8, no. 5, pp. 517-521, 2000.