IEICE TRANS. FUNDAMENTALS, VOL.E89–A, NO.12 DECEMBER 2006

PAPER *Special Section on VLSI Design and CAD Algorithms*

Memory Size Computation for Real-Time Multimedia Applications Based on Polyhedral Decomposition*

Hongwei ZHU†, ***Nonmember***, **Ilie I. LUICAN**†, ***Student Member***, ***and*** **Florin BALASA**†a), ***Nonmember***

SUMMARY 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. In deriving an optimized (for area and/or power) memory architecture, memory size computation is an important step in the exploration of the possible algorithmic specifications of multimedia applications. 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, more recent advances in the theory of polyhedra. In contrast with all the previous works which are only *estimation* methods, this approach performs *exact* memory computations even for applications significantly large in terms of the code size, number of scalars, and number of array references.
key words: *multimedia processing applications, multi-dimensional signal processing, computation of storage requirements, memory allocation, polyhedra*

1. Introduction

In advanced telecom and 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 [5]. 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 early phase of the design, the system-level exploration. This problem has been tackled in the past both in register-transfer level (RTL) programs at scalar level (e.g., [10]), and in behavioral specifications at non-scalar level (see below).

Common to all scalar-based storage estimation techniques is that the number of scalars is drastically limited. When multi-dimensional arrays are present in the algorithmic specification of the targeted applications, the computation times of these techniques increase dramatically if the arrays are flattened and each array element is treated like a separate scalar.

To overcome the shortcomings of the scalar estimation techniques for high-level algorithmic specifications where the code has an organization based on loop nests and multi-dimensional arrays are present, several research teams proposed different techniques exploiting the fact that, due to the loop structure of the code, large parts of an array can be produced or consumed within a same array reference. These estimation approaches can be basically split in two categories: those requiring a fully-fixed execution ordering [13], [16], [17], and those assuming non-procedural specification where the execution ordering is not (completely) fixed [2], [9]**.

This paper presents a non-scalar method for computing *exactly* the memory size in multi-dimensional signal processing algorithms where the code is procedural, that is, where the loop structure and sequence of instructions induce the (fixed) execution ordering. This approach uses both algebraic techniques specific to the data-flow analysis used in modern compilers [11], and also more recent advances in the theory of polyhedra. In contrast with previous works which utilize only *approximate* methods due to the size of the problems (in terms of number of scalars, number and complexity 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 Sect. 2), therefore practically the entire class of real-time multi-dimensional signal processing applications.

The paper is organized as follows. Section 2 explains the problem of memory size computation. The core of the paper — Sects. 3 and 4 — presents technical aspects of this novel approach. Section 5 will briefly discuss implementation aspects and present several experimental results. Sec-

Manuscript received March 17, 2006.
Manuscript revised June 1, 2006.
Final manuscript received July 25, 2006.
†The authors are with the Dept. of Computer Science, Univ. of Illinois at Chicago, IL, USA.
*An abbreviated version of this paper was presented at ASP-DAC 2006 [18]. This material is based upon work supported by the National Science Foundation under Grant Nr. 0133318.
a) E-mail: fbalasa@cs.uic.edu
DOI: 10.1093/ietfec/e89–a.12.3378

**More information about these non-scalar techniques will be provided in Sect. 5 in a comparative overview with our approach.

Copyright © 2006 The Institute of Electronics, Information and Communication Engineers

tion 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 multi-dimensional signals which array references have (complex) linear indices. This class of specifications is often referred to as *affine* [5].

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 a discrete *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 [5], is given below:

```
optDlt[0] = 0;
for (i = 8; i ≤ 120; i++)
  for (j = 8; j ≤ 120; j++)
  { Dlt[i][j][0] = 0;
    for (k = i − 8; k ≤ i + 8; k++)
      for (l = j − 8; l ≤ j + 8; l++)
        Dlt[i][j][17(k − i) + l − j + 145] = A[i][j] − A[k][l]@1
                         +Dlt[i][j][17(k − i) + l − j + 144];
    optDlt[113i+j−911] = Dlt[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 — as computed by hand and confirmed by our tool presented in Sect. 5.

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. The simulated execution of the code or RTL approaches based on the left edge algorithm [10], although appealing by means of simplicity, are too computationally expensive in such cases, often prohibitive to use.

What is fundamentally different from these previous works [13], [16], [17] doing only a *memory size estimation*, the algorithm presented in this paper is able to compute *exactly* the storage requirements for multimedia applications, even when the number of scalar signals is large (typically, 2–3 orders of magnitude larger than in previous works [2], [9], [13], [16], [17]). 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 unimodular cones [3], (b) the efficient decomposition of the array references of the multi-dimensional signals in disjoint *linearly bounded lattices* [15], and (c) an efficient mechanism of pruning the code of the algorithmic specification.

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: $P = \{\mathbf{x} \in \Re^n | \mathbf{A} \cdot \mathbf{x} \geq \mathbf{b}\}$, where $\mathbf{A} \in \Re^{m \times n}$ and $\mathbf{b} \in \Re^m$. If P is a bounded set, then P is called a *polytope*. If $\mathbf{x} \in \mathbf{Z}^n$, then P is called a $\mathbf{Z}$-polyhedron/polytope. Each *array reference* $M[x_1(i_1, \ldots, i_n)] \cdots [x_m(i_1, \ldots, i_n)]$ of an m-dimensional signal M, in the scope of a nest of n loops having the iterators $i_1, \ldots i_n$, is characterized by an *iterator space* and an *index 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* (LBLs) [15], that is, the image of an affine vector function over the iterator polytope[†] $\mathbf{A} \cdot \mathbf{i} \geq \mathbf{b}$:

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

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

In order to address the computation of the memory size necessary for the execution of a multi-dimensional 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.

[†]The iterator space of an array reference is not always a convex polytope; it can be a non-convex polyhedron, or even a union of convex and nonconvex polyhedra. But, nevertheless, it can be *decomposed* into a finite set of disjoint (convex) polytopes, as it will be explained in Sect. 4. This is why, without lack of generality, the iterator space can be considered one (convex) polytope.

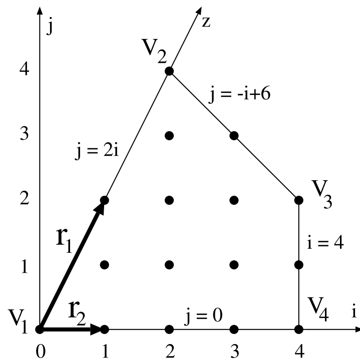

Fig. 1 Quadrilateral (a 2D **Z**-polytope) representing the iterator space in *Example 1*.

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 the decomposition of a simplicial cone into unimodular cones [3] is used[†].

An example is given below, illustrating both the concepts and the technique. 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 the main steps of the computation flow. Moreover, it will clearly show the *scalability* of the technique — which makes it adequate for the multimedia applications, where the array references may cover large sets of scalars.

Example 1: $for\ (i = 0;\ i \leq 4;\ i++)$
$for\ (j = 0;\ j \leq 2i\ \&\&\ j \leq -i+6;\ j++)$
$\cdots A[2i+3j][5i+j] \cdots$

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 $\{\mathbf{x} = \mathbf{Ti} + \mathbf{u} \,|\, \mathbf{Ai} \geq \mathbf{b}\}$ =

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

($\mathbf{u}$=$\mathbf{0}$ here) and the problem is equivalent to computing the size of this set. Since the rank of matrix $\mathbf{T}$ is 2 — equal to its number of columns; therefore, the vector function $\mathbf{x}$=$\mathbf{Ti}$ + $\mathbf{u}$ is a one-to-one mapping [2]. 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 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*, pointed in the origin, 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 $F(V_i)$ of each supporting cone and the generating function $F = \sum_i F(V_i)$ of the whole iterator polytope P.

Given a polytope P in the d-dimensional space, for each lattice point $\mathbf{p} = (p_1, p_2, \ldots, p_d)$ in P, a term $z^{\mathbf{p}} = {x_1}^{p_1} {x_2}^{p_2} \ldots {x_d}^{p_d}$ can be introduced. The function $F(P) = \sum_p z^{\mathbf{p}}$ (or, simply, F) is defined as the generating function of the polytope P. For example, the generating function of the triangle with the vertices (0, 0), (1, 2), (2, 0) is $F = x^0y^0 + x^1y^0 + x^1y^1 + x^1y^2 + x^2y^0 = 1 + x + xy + xy^2 + x^2$, each monomial term corresponding to one of the lattice points inside or on the border of the triangle: e.g., x^1y^2 corresponds to the point (1,2), and so on. By evaluating F at $z = \mathbf{1}$, we get the number of lattice points in P [3]. For instance, if $x = 1, y = 1$ then $F = 5$, which is the number of lattice points in the triangle.

Writing F as a sum of monomial terms would be impractical for large polytopes. Fortunately, F can be com-

[†]There are also other methods, for instance, based on Ehrhart polynomials [6], or even simpler, by adapting the Fourier-Motzkin elimination [7]. The current technique was chosen because of its scalability, as explained later.

pactly written as an algebraic sum of rational functions, each term corresponding to one of the unimodular cones in the decomposition of the supporting cones $C(V_i)$ $(i = 1, \ldots, 4)$ of the polytope vertices (*Steps 1* and *2*). According to [3], each supporting cone $C(V)$ has associated a generating function $F(V)$ of the form

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

where $z^V = x^a y^b$ (since a vertex $V(a, b)$ has two coordinates a, b in this case), the sum is over all the unimodular component cones, and the product at the denominator is over all the generating rays r_i.

For instance, the generating function $F(V_1)$ has two terms, one for each unimodular cones in the decomposition (2) of $C(V_1)$. Since $V = V_1(0, 0)$, then the two numerators are $z^V = x^0 y^0 = 1$. The rays $[0\ 1]^T$ and $[1\ 0]^T$ of the first unimodular cone (2) yield the first denominator: $\Pi_i(1 - z^{r_i}) = (1 - x^0 y^1)(1 - x^1 y^0) = (1 - y)(1 - x)$. The rays $[1\ 2]^T$ and $[0\ -1]^T$ of the second unimodular cone (2) yield the denominator: $(1 - x^1 y^2)(1 - x^0 y^{-1}) = (1 - xy^2)(1 - y^{-1})$. Therefore, 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:

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

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

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 polytope.

In order to evaluate $F = \sum_i F(V_i)$ at $z = \mathbf{1}$, F must be transformed into a one-variable function. This is done with a substitution $z \longrightarrow t^\lambda$, where $\lambda = [\lambda_1\ \lambda_2]^T$ is an integer vector chosen such that the substitution $x = t^{\lambda_1}$, $y = t^{\lambda_2}$ would not make any denominator equal to zero [8]. In our example, choosing $\lambda_1 = 0$ is not possible since the denominator factor $1 - x$ of $F(V_1)$ would be zero. Similarly, we must have $\lambda_2 \neq 0$ and $\lambda_1 + 2\lambda_2 \neq 0$. We choose, for instance, $\lambda = [1\ -1]^T$†. With the substitution $x = t$, $y = t^{-1}$, the generating function of the iterator polytope becomes:

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

After eliminating the negative exponents in the denominators, we factorize t^{-2} in order to eliminate all the negative exponents in F; this makes the further computations simpler, without changing the final result — the evaluation of F in $t = 1$. Computationally, this is accomplished by substituting $t = s + 1$ and doing a Taylor expansion about $s = 0$ via a polynomial division. After the substitution $t = s + 1$, we obtain 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 [8]: $c_0 = \frac{a_0}{b_0}$ and

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

After polynomial divisions in all the terms of F, the algebraic sum of the coefficients c_2 (since the space dimension $d = 2$) is the evaluation of F in $s = 0$, that 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 necessary 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,

†There is a polynomial-time algorithm for λ selection in [4]. However, [8] suggests a much simpler approach, trying random vectors with small integer entries, allowing small increments if necessary, until λ is found.

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 [7]. 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 $\mathbf{Z}$-polytope is empty.

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

The general idea is to bring, first, the matrix $\mathbf{T}$ of the mapping to the Hermite Normal Form (HNF) [14]. For instance, post-multiplying the matrix of the vector mapping $\mathbf{T} = \begin{bmatrix} 3 & 1 \end{bmatrix}$ by the unimodular matrix $\mathbf{S} = \begin{bmatrix} 0 & 1 \\ 1 & -3 \end{bmatrix}$ yields the HNF: $\mathbf{T} \cdot \mathbf{S} = \begin{bmatrix} 1 & 0 \end{bmatrix}$. Note that while HNF is a canonical form, the factorizing matrix $\mathbf{S}$ is not necessarily unique when $\mathbf{T}$ is not square and non-singular. Most of the algorithms computing HNF propose also techniques to build the factorizing (unimodular) matrix. In our context, any such an approach will do, provided it has polynomial complexity for reason of efficiency[†].

The rank of matrix $\mathbf{T}$ is $r = 1$, hence less than the number of columns $n = 2$ of $\mathbf{T}$; in this case, the vector function $\mathbf{x}=\mathbf{Ti} + \mathbf{u}$ 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 initial iterator space (which is the same as in *Example 1*) into $\{\mathbf{AS} \cdot [k\ \ l]^T \geq \mathbf{b}\}=\{\ 3l \leq k \leq 5l,\ k \leq 2l + 6,\ l \leq 4\ \}$ (where $[k\ \ l]^T = \mathbf{S}^{-1}[i\ \ j]^T$ is the new iterator vector after the transformation $\mathbf{S}$) whose exact 1D projection (since $r = 1$) is $\{\ 0 \leq k \leq 14\ \}$, obtained eliminating l in the inequalities above [7]. The points in the "dark shadow" [12] $4 \leq k \leq 14$ correspond to lattice points (k,l) in the modified iterator space. The other values of k are individually checked. k=1 and 2 result to be invalid projections: replacing these values in the modified iterator space, no integer solution for l can be found. Conversely, k=0 and 3 are valid projection, since (k,l)=(0,0) and (3,1) belong to the modified iterator space. Therefore, storing $A[3i + j]$ requires $15 - 2 = 13$ locations. Indeed, the index $3i + j$ can take all the values between 0 and 14, except 1 and 2.

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 LBLs can be performed by a recursive intersection, starting from the array references in the code. Let $Lbl_1 = \{\mathbf{x} = \mathbf{T}_1\mathbf{i}_1 + \mathbf{u}_1 \mid \mathbf{A}_1\mathbf{i}_1 \geq \mathbf{b}_1\}$ $Lbl_2 = \{\mathbf{x} = \mathbf{T}_2\mathbf{i}_2 + \mathbf{u}_2 \mid \mathbf{A}_2\mathbf{i}_2 \geq \mathbf{b}_2\}$ be two LBLs of an indexed signal in the algorithmic specification, where $\mathbf{T_1}$ and $\mathbf{T_2}$ have the same number of rows (the signal dimension). Intersecting the two linearly bounded lattices means, first of all, solving a linear Diophantine system[††] $\mathbf{T}_1\mathbf{i}_1 - \mathbf{T}_2\mathbf{i}_2 = \mathbf{u}_2 - \mathbf{u}_1$ 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 [14]. If the set of coalesced constraints of the two LBLs

$$\begin{aligned} \mathbf{A_1 V_1} \cdot \mathbf{i} &\geq \mathbf{b_1} - \mathbf{A_1 v_1} \\ \mathbf{A_2 V_2} \cdot \mathbf{i} &\geq \mathbf{b_2} - \mathbf{A_2 v_2} \end{aligned} \tag{3}$$

has integer solutions, then the intersection is a new LBL: $Lbl_1 \cap Lbl_2 = \{\mathbf{x} = \mathbf{T_1 V_1} \cdot \mathbf{i} + \mathbf{T_1 v_1} + \mathbf{u_1} \mid$ s.t. constraints (3)$\}$.

Example 3: The disjoint LBLs of signal *Dlt* from the illustrative example in Sect. 2 are (in non-matrix format):
$Dlt_1 = \{\ x = i,\ y = j,\ z = 0 \mid 120 \geq i,\ j \geq 8\ \}$
$Dlt_2 = \{\ x = i,\ y = j,\ z = 289 \mid 120 \geq i,\ j \geq 8\ \}$
$Dlt_3 = \{\ x = i,\ y = j,\ z = 17(k - i) + l - j + 145 \mid 120 \geq i,\ j \geq 8$
$8 \geq k - i,\ l - j \geq -8$ and $143 \geq 17(k - i) + l - j\ \}$

Figure 2 shows a polyhedral dependence graph built from the illustrative example in Sect. 2, where the nodes are the disjoint LBLs 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 Sect. 3).

If the iterator space of an array reference is not a (convex) polytope, it can be partitioned into *disjoint* polytopes with the same decomposition algorithm presented above. Note that the convex polytope is a particular case of LBL when in the vector mapping $\mathbf{x} = \mathbf{T} \cdot \mathbf{i} + \mathbf{u}$ the matrix $\mathbf{T}$ is $\mathbf{I}_n$

[†]These algorithms are designed to prevent the so-called "coefficient swell" [14].

[††]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 [14].

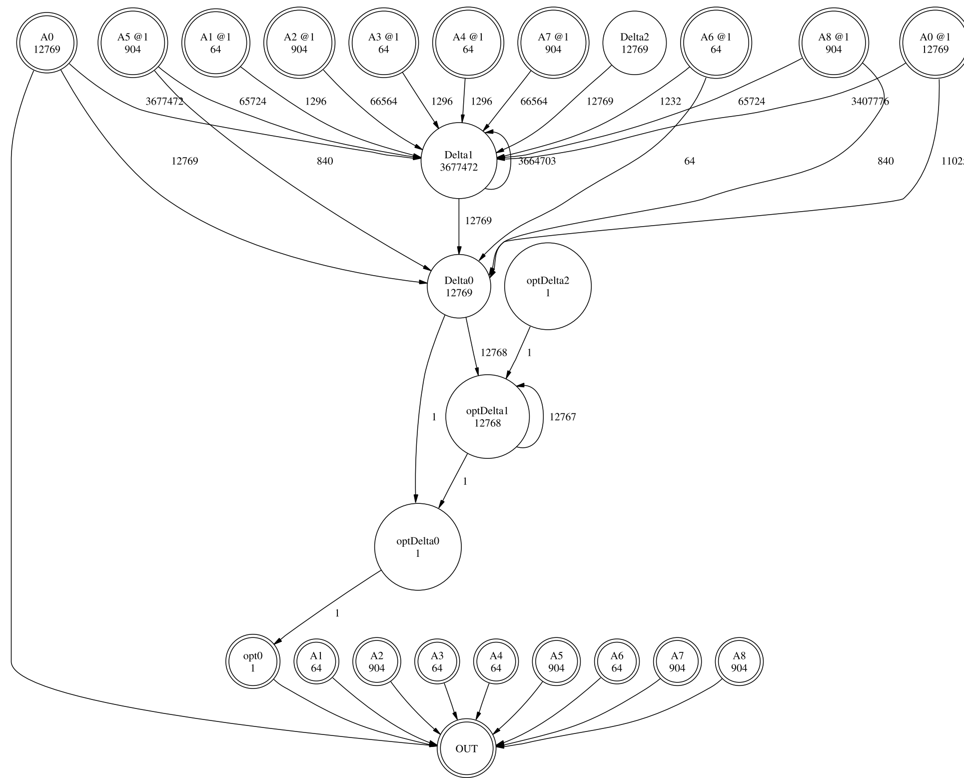

Fig. 2 Polyhedral dependence graph having as nodes the disjoint lattices derived from the illustrative example (Sect. 2).

(the identity matrix of order n) and $\mathbf{u}$ is a zero-column vector. Therefore, the same decomposition algorithm (actually, a simplified version of it) can be applied. The principle is the same: a recursive intersection of the polytopes composing the iterator space.

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 Sect. 3.

Step 3 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). The 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 4: $for\ (i = 0;\ i \le 6;\ i++)$
$for\ (j = 0;\ j \le 2;\ j++)$
$for\ (k = 0;\ k \le 4;\ k++)\ A[i + j + k] \cdots$

The max iterator vector addressing the scalar, say, $A[7]$ is $[i\ j\ k]^T_{max} = [6\ \ 1\ \ 0]^T$, while the min iterator vector is $[1\ \ 2\ \ 4]^T$.

Our algorithm finds the LBLs produced and consumed in the current block, computing the min and, respectively, max iterator vectors for the scalar signals covered by these LBLs since these iterator vectors correspond to the increase and, respectively, decrease of the memory. Knowing the number of elementary iterations in the loop nest (by counting the lattice points in the iterator spaces with the algorithm in Sect. 3), one can then determine *exactly* the memory variation and, in particular, the maximum storage amount in each of the blocks.

Note that the number of array references in the current block of code is arbitrary. Also, it is possible to have array references with distinct vector functions mapping the same array elements (their LBLs are equal sets, although their mappings are different).

Example 5: $for\ (i = 0;\ i \le 10;\ i++)$
$for\ (j = 0;\ j \le 10;\ j++)$
$= \cdots A[i + j] \cdots \qquad //\ Assign.\ (1)$

[†]The loops can be easily normalized with simple linear transformations.

$$= \cdots A[20 - i - j] \cdots \quad // \; Assign. (2)$$

Notice that the two array references are mapping the same array elements, from $A[0]$ to $A[20]$. Suppose all these elements of signal A are consumed in the loop nest above. The elements $A[t]$, $t = 1, 10$ are consumed in the iteration given by their max iterator vector $[i \; j]^T_{max} = [10 \;\; 10 - t]^T$ at assignment (2); similarly, the elements $A[t]$, $t = 11, 20$ are consumed in the iteration $[i \; j]^T_{max} = [10 \;\; t - 10]^T$ at assignment (1). There are, in total, 242 assignments executed in this loop nest. The first consumed element is $A[10]$ in the iteration (10,0), at assignment (2). Hence, after the 222-nd assignment one memory location (the one storing $A[10]$) becomes free. Then, after the 223-rd assignment the location storing $A[11]$ becomes free, followed by the one storing $A[9]$ after the 224-th assignment, etc.

Actually, part of the LBLs 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 Sect. 2, each iterator vector $[i \; j \; k \; l]^T$ corresponds to a unique produced scalar $Dlt[i][j][17(k - i) + l - j + 145]$ and a unique consumed scalar $Dlt[i][j][17(k - i) + l - j + 144]$. 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 Sect. 2) "enriched" with the delay operator @. In addition to the computation of the minimum memory size requirements and different statistical data on the memory usage by the multi-dimensional signals in the multimedia specification, the tool can optionally generate dependence graphs (like the one in Fig. 2) at different granularity levels, which provides 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. It must be stressed that the tool *does not simulate the execution of the specification code*; the tool finds the points where the memory occupancy changes value, this number of points being 14,806 for the illustrative example, which represents only 0.4% of the number of datapath instructions when the illustrative code is executed.

Table 1 summarizes the experimental results. The benchmarks used are: (1) a real-time regularity detection algorithm used in robot vision; (2) the kernel of a voice coding application — essential component of a mobile radio terminal; (3) a singular value decomposition (SVD) updating algorithm used in spatial division multiplex access (SDMA) modulation in mobile communication receivers, in beamforming, and Kalman filtering; (4) a 2D Gaussian blur filter from a medical image processing application which extracts contours from tomograph images in order to detect brain tumors; (5) a motion detection algorithm used in the transmission of real-time video signals on data networks [5]; (6) a dynamic programming algorithm used in several multimedia applications.

Columns 2 and 3 display the numbers of array references in the code and, respectively, of the disjoint LBLs derived in the partitioning process (Sect. 4). Column 4 displays the numbers of scalar signals, column 5 shows the storage requirements (number of memory locations), and column 6 gives a measure of the memory sharing due to the disjoint lifetimes of the scalar signals. The last column displays the running times of our experiments carried out on a PC with a 1.85 GHz Athlon XP processor and 512 MB memory.

Since there is no other similar tool able to validate the memory size results, we adopted *indirect* validation ways. We built a large number of artificial test programs, complex enough to cover all the relevant situations, but not so large to prevent us from verifying the memory size variation by sim-

[†]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.

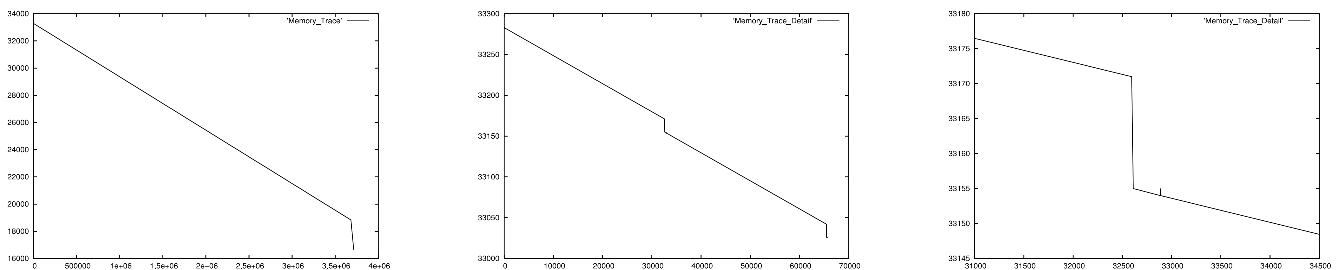

Fig. 3 Memory trace for the illustrative example in Sect. 2. The abscissae are the numbers of datapath instructions in the code, the ordinates are memory locations. The first graphic represents the entire trace. The second graphic is a detailed trace in the interval [0 : 65767], which corresponds to 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).

Table 1 Experimental results.

Application (params.)	Array refs.	Disjoint LBLs	Scalars	Memory Size	Memory / Scalars (%)	CPU (s)
Regularity detection (MaxGrid=8, L=64)	19	17	4,752	2,304	48.48	< 1
Vocoder kernel	232	153	33,619	11,890	35.37	2
SVD updating (N=100)	8,472	15,027	3,045,447	34,950	1.15	122
2D Gauss. blur filter (M=N=500)	1,594	4,491	2,735,027	250,005	9.14	103
Motion detection (M=N=64, m=n=4)	491	498	318,367	9,524	2.99	11
Dyn. prog. (L=500)	3,994	3,489	21,082,751	124,751	0.59	78

ulated execution of the code[†]. Subsequently, we compared these memory traces with the results provided by the tool. A second validation approach was to study thoroughly the code of the applications and try to find *theoretically* the minimum memory size. This second strategy had, obviously, a limited success due to the inherent complexity of most of the applications. However, we succeeded in a few cases to find by hand computation the minimum memory size (like, e.g., for the motion detection kernel).

This tool can process large specifications in terms of number of loop nests, lines of code, number of array references. For instance, the voice coding application contains 232 array references organized in 40 loop nests. In one of our experiments, the tool processed in less than 8 minutes a code with 113 loop nests 3 level deep, containing 906 array references, many having very complex indices.

A comparative evaluation with the previous non-scalar works reveals the following notable differences:

1. Part of the previous works impose important constraints on the properties of the algorithmic specifications they can process.

For instance, Ramanujam *et al*. address only specifications with *perfectly* nested loops (i.e., in which every statement is inside the innermost loop) [13]. Verbauwhede *et al*. consider loops with non-constant boundaries, but in such cases the varying boundaries are internally replaced with upper- and lower-bounds in order to fit their computational model [16].

In comparison, our model handles the entire class of "affine" specifications — as described in Sect. 2 — without any constraints.

2. Part of the previous works do not report running times [9], [13].

Although [9] is the only previous work — to the best of our knowledge — reporting results on a complex application (the MPEG-4 motion estimation kernel) with a significant amount of scalars (262,400), the absence of running time information is a shortcoming.

3. All the previous works except [9] use benchmarks with a relatively small number (up to tens of thousands) of scalars. These works do not offer any information on the scalability of their techniques.

From our past experience, even if a memory size estimation technique behaves reasonably well when dealing with examples containing thousands of scalars, the computation time can sharply increase till becoming ineffectual for examples where the number of scalars is larger by 1–2 orders of magnitude.

In comparison, our approach can handle applications with millions of scalars in acceptable times. The running times of our approach are also increasing significantly with the size of the problem, but our tool is still effective when processing examples with at least one (but often 2–3) order(s) of magnitude more scalars.

4. The previous memory estimation techniques yield storage results that may be highly inaccurate.

Verbauwhede *et al.* state that their determinations are exact when the loop boundaries are constant, but overestimates occur when they are not [16]. However, they do not report any concrete results on the amount of the overestimates they experienced. Ramanujam *et al.* report exact determinations for all their benchmarks, except one exhibiting an estimation error of 13% [13]. Zhao and Malik obtained an estimation of 1372 memory locations for the motion detection kernel, when the set of parameters is $M = N = 32$, $m = n = 4$ [17]. This is a rather poor estimation since the correct result (computed by our tool, but also theoretically confirmed) for the same set of parameters is 2740 storage locations.

In contrast, our tool performs only exact determinations. One could argue that it may be better to obtain fast estimations, even not very accurate, rather than exact determinations with a significantly higher computation effort. This argument is fair enough, but it does not apply to the present situation. The aforementioned estimation result [17] was obtained in 21 seconds on a Sun Enterprise 4000 machine with 4 (336 MHz UltraSparc) processors and 4 GB memory, whereas our computation time was of only 2 seconds on the Athlon XP PC and 7 seconds on a Sun Ultra 20. Therefore, not only our approach found the *exact* result, but it did it *faster* than the estimation technique [17].

6. Conclusions

This paper has presented a non-scalar approach for com-

[†]Such tools exhibit, in general, a poor scalability, being ineffectual for large programs, with numerous scalars and deep loop nests.

puting the memory requirements for real-time multimedia applications, where the storage of large multi-dimensional signals causes a significant cost in terms of both area and power consumption. This method uses modern elements of 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.

References

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

[2] F. Balasa, F. Catthoor, and H. De Man, "Background memory area estimation for multi-dimensional signal processing systems," IEEE Trans. Very Large Scale Integr. (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," Math. Oper. Res., vol.19, no.4, pp.769–779, Nov. 1994.

[4] A.I. Barvinok and J. Pommersheim, "An algorithmic theory of lattice points in polyhedra," in New Perspectives in Algebraic Combinatorics, pp.91–147, Cambridge Univ. Press, 1999.

[5] 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, Kluwer Academic Publ., Boston, 1998.

[6] Ph. Clauss and V. Loechner, "Parametric analysis of polyhedral iteration spaces," J. VLSI Signal Process., vol.19, no.2, pp.179–194, 1998.

[7] G.B. Dantzig and B.C. Eaves, "Fourier-Motzkin elimination and its dual," J. Comb. Theory A, vol.14, pp.288–297, 1973.

[8] J.A. De Loera, R. Hemmecke, J. Tauzer, and R. Yoshida, "Effective lattice point counting in rational convex polytopes," J. Symbol. Comput., vol.38, no.4, pp.1273–1302, 2004.

[9] P.G. Kjeldsberg, F. Catthoor, and E.J. Aas, "Data dependency size estimation for use in memory optimization," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.22, no.7, pp.908–921, July 2003.

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

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

[12] W. Pugh, "A practical algorithm for exact array dependence analysis," Commun. ACM, vol.35, no.8, pp.102–114, Aug. 1992.

[13] 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., pp.359–364, June 2001.

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

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

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

[17] Y. Zhao and S. Malik, "Exact memory size estimation for array computations," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.8, no.5, pp.517–521, 2000.

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

Hongwei Zhu received the B.S. degree in Electrical Engineering from Xi'an Jiaotong University, P.R. China, in 1996, and his M.S. degree in Electrical & Electronic Engineering from Nanyang Technological University, Singapore, in 2001. Currently, he is a Ph.D. candidate in the Dept. of Computer Science, University of Illinois at Chicago (UIC). Before joining UIC, he worked as a senior R&D engineer at JVC Asia Pte. Ltd., Singapore. His research interests are memory management for real-time multi-dimensional signal processing and combinatorial optimization in VLSI CAD.

Ilie I. Luican received the B.S. and M.S. degrees in Computer Science from the Polytechnical University of Bucharest (PUB), Romania, in 2002 and 2003, respectively. From 2003 to 2004, he was with the Automatic Control Department of PUB. Currently, he is a Ph.D. candidate in the Dept. of Computer Science, University of Illinois at Chicago. His research interests are high-level synthesis and memory management for real-time multi-dimensional signal processing.

Florin Balasa received the M.S. and Ph.D. degrees in Computer Science from the Polytechnical University of Bucharest in 1981 and 1994, respectively. He also received the Ph.D. degree in Electrical Engineering from the Katholieke Universiteit Leuven, Belgium, in 1995. Dr. Balasa is currently an Assistant Professor of Computer Science at the University of Illinois at Chicago (UIC). From 1990 to 1995, he worked at the Interuniversity Microelectronics Center (IMEC), Leuven, Belgium. From the fall of 1995 to 1999, he worked at Conexant Systems Inc., Newport Beach, CA. Dr. Balasa is a recipient of the National Science Foundation Career Award. His research interests are mainly focused on high-level synthesis and physical design.