# COMPUTATION OF THE MINIMUM DATA STORAGE FOR MULTI-DIMENSIONAL SIGNAL PROCESSING*

Ilie I. Luican Hongwei Zhu Florin Balasa

Dept. of Computer Science, University of Illinois at Chicago

**Abstract –** The amount of the data storage in signal processing systems, whose behavior is described by loop-organized algorithmic specifications, has an important impact on the overall energy consumption, chip area, as well as system performance. This paper presents a non-scalar approach for computing the minimum storage requirements in high-level procedural specifications, where the main data structures are multi-dimensional arrays. 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 can perform the *exact* computation of the minimum data storage even for applications with numerous loop nests and complex array references.

*Keywords:* memory management, memory size computation, linearly bounded lattice, behavioral specification, array reference.

## 1. INTRODUCTION

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

The behavior of these targeted VLSI systems, synthesized to execute mainly data-dominant applications, is described in a high-level programming language, where the code is typically organized in sequences of loop nests having as boundaries (usually affine) functions of loop iterators, conditional instructions where the arguments may be data-dependent and/or data-independent (relational and/or logic expressions of affine functions of loop iterators). The data structures are multi-dimensional arrays whose indexes in the code are affine functions of surrounding loop iterators. The class of specifications with these characteristics are often called *affine* specifications [3]. An illustrative example of a code in this class is shown in Fig. 1.

The problem addressed in this paper is how to compute the *minimum* amount of memory locations necessary to store the signals during the execution of, e.g., an image processing algorithm, assuming any scalar (array element) must be stored only during its lifetime – from the moment when it is produced till it is used last time. For instance, the total number of array elements in the code from Fig. 1 is 302,498; 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 scalars. Actually, only 5,292 memory locations are necessary (see Section 2).

This problem has been firstly addressed *at scalar level* in register transfer-level (RTL) programs. Good overviews of these techniques can be found in [4, 3]. 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 techniques, several works proposed different *non-scalar* computation models for the estimation of the storage requirements of high-level algorithmic specifications where the code structure was loop-based and multi-dimensional arrays were present. These estimation approaches can be basically split in two categories: those requiring a fully-fixed execution ordering, and those assuming non-procedural specification where the execution ordering is still not (completely) fixed. The techniques falling in the first category will be addressed first.

Verbauwhede *et al.* consider a production axis for each array to model the relative production and consumption time (or date) of the individual array accesses [9]. The difference between these two dates equals the number of array elements produced between them, while the maximum difference gives the storage requirement for the considered array. The time differences are computed based on an integer linear programming model. Zhao and Malik developed a technique based on live variable analysis and integer point counting for intersection/union of mappings of parametrized polytopes [10]. They prove that it is sufficient to find the number of live variables for one statement in each innermost loop of a loop nest in order to get an estimate of the minimum memory size. Ramanujam *et al.* use for each array a reference window containing at any moment during execution the array elements alive (that have already been referenced and will also be referenced in the future) [7]. The maximal window size gives the storage requirement for the corresponding array. Treating the arrays separately, this technique (and [9], as well) does not consider the possibility of inter-array in-place mapping [3].

In contrast to the non-scalar methods described so far, the memory estimation technique presented in [1] does not take execution ordering into account, allowing any ordering not prohibited by data dependencies. The estimation technique described in [5] assumes

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

```
T[0] = 0 ;                          //  A[10][529] : input
for ( j=16 ; j<=512 ; j++)                  // The 1st loop nest
{ S[0][j-16][0] = 0 ;                         // (1)
  for ( k=0 ; k<=8 ; k++)
    for ( i=j-16 ; i<=j+16 ; i++)             // (2) is below
      S[0][j-16][33*k+i-j+17] = A[4][j] - A[k][i] + S[0][j-16][33*k+i-j+16] ;
  T[j-15] = S[0][j-16][297] + T[j-16] ;     // (3)
}
for( j=16 ; j<=512 ; j++)                   // The 2nd loop nest
{ S[1][j-16][0] = 0 ;
  for( k=1 ; k<=9 ; k++)
    for( i=j-16 ; i<=j+16 ; i++)
      S[1][j-16][33*k+i-j-16] = A[5][j] - A[k][i] + S[1][j-16][33*k+i-j-17] ;
  T[j+482] = S[1][j-16][297] + T[j+481] ;
}
out = T[994];                       // out : output
```

Figure 1: Illustrative example of affine specification.

a *partially fixed* execution ordering. The authors employ a data dependence analysis similar to [1], their major improvement being to add the capability of taking into account available execution ordering information, based mainly on loop interchanges.

Different from the previous works which are only *approximate* methods, this paper presents a non-scalar technique for *computing exactly* the minimum memory size in multi-dimensional signal processing algorithms, when the specifications are *procedural*, i.e., the execution ordering is induced by the loop structure and it is thus fixed (like in several previous works [9, 10, 7]). This assumption is based on the fact that in present industrial design, the design entry usually includes a full fixation of the execution ordering. Even if this is not the case, the designer can still explore different algorithmic specifications functionally equivalent.

The rest of the paper is organized as follows. Section 2 – the core of the paper – introduces the basic mathematical concepts and presents the flow of memory size computation algorithm. Section 3 discusses implementation aspects and experimental results. Section 4 summarizes the conclusions of this research.

## 2. COMPUTATION OF THE MINIMUM DATA STORAGE

A *polyhedron* is a set of points $P \subset \Re^n$ satisfying a finite set of linear inequalities: $P = \{ \mathbf{x} \in \Re^n \mid \mathbf{A} \cdot \mathbf{x} \geq \mathbf{b} \}$, where $\mathbf{A} \in \Re^{m \times n}$ and $\mathbf{b} \in \Re^m$. If $P$ is a bounded set, then $P$ is called a *polytope*. If $\mathbf{x} \in \mathbf{Z}^n$, then $P$ is called a $\mathbf{Z}$-polyhedron/polytope.

Each *array reference* $M[x_1(i_1, \ldots, i_n)] \cdots [x_m(i_1, \ldots, i_n)]$ of an $m$-dimensional signal $M$, in the scope of a nest of $n$ loops having the iterators $i_1, \ldots, i_n$, is characterized by an *iterator space* and an *index* (or array) *space*. The iterator space signifies the set of all iterator vectors $\mathbf{i} = (i_1, \ldots, i_n) \in \mathbf{Z}^n$ in the scope of the array reference. The index (or array) space is the set of all index vectors $\mathbf{x} = (x_1, \ldots, x_m) \in \mathbf{Z}^m$ of the array reference. When the indices of an array reference are linear mappings with integer coefficients of the loop iterators, the index space consists of one or several *lattices* [8] linearly bounded, i.e., the image of an affine vector function (or mapping) over the iterator $\mathbf{Z}$-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.

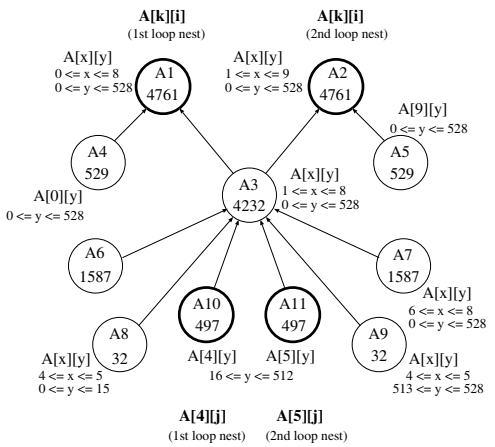


Figure 2: Decomposition of the index space of signal A into disjoint lattices; the arcs in the graph show inclusion relations.

*Example* $for\ (i = 0;\ i \leq 2;\ i++)$
$for\ (j = 0;\ j \leq 3;\ j++) \quad \cdots M[3i+j][5i+2j] \cdots$
The index space of the array reference can be expressed as a lattice:

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

where $x$ and $y$ are the indices of the array reference.

Once these concepts clarified, the memory size computation algorithm will be explained and illustrated with the code in Fig. 1.

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

Figure 2 shows the result of this decomposition for the 2-dimensional signal $A$ in the illustrative example. The graph displays the inclusion relations ($arcs$) between the lattices of $A$ ($nodes$). The 4 "bold" nodes are the 4 array references of signal $A$ in the code (see Fig. 1). For instance, the node $A1$ represents the lattice of $A[k][i]$ in the first loop nest. The nodes are also labeled with the size of the corresponding lattice – that is, the number of points having integer coordinates in those sets. The inclusion graph is gradually constructed by partitioning analytically the (four) array references using lattice *intersections* and *differences*. While the *intersection* of two non-disjoint lattices is a lattice as well (e.g., [1]), the *difference* operation is not closed. Denoting $A1 \cap A2 = A3$, $A1 - A3$ and $A2 - A3$ are also lattices (denoted $A4$, $A5$ in Fig. 2). However, the difference $A3 - A10$ is not a lattice due to the non-convexity of this set.

At the end of the decomposition, the nodes without any incident arc represent non-overlapping lattices. Every array reference in the code is now either a disjoint lattice itself (like $A10$ and $A11$), or a union of disjoint lattices (e.g., $A1 = A4 \cup A3 = A4 \cup \bigcup_{i=6}^{11} A_i$).

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

The algorithmic specification is, basically, a sequence of nested loops. (Single instructions outside nested loops are actually nests of depth zero.) We refer to these loop nests as *blocks* of code.

After the decomposition of the array references in the specification code, for each disjoint lattice it is determined the block where it 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 lattice can be computed exactly. For instance, the data storage at the beginning of the code in Fig. 1 is $size(\bigcup_{i=4}^{11} A_i) = 5,290$ since all the array elements of $A$ are necessary for the code execution. However, after the execution of the first loop nest, the storage requirement decreases since the lattice $A4$ of signal $A$ is no longer necessary in the second loop nest. The memory size after the first loop nest is $size(\bigcup_{i=5}^{11} A_i) + 1 = 4,762$ , where 1 stands for $T[497]$ produced in the last loop iteration.

**Step 3** *Find the maximum storage requirement inside each block.*

**Step 3.1** *Determine the* characteristic *memory variation for each assignment instruction in the current block of code.*

Take, for instance, the first loop nest from the illustrative example in Fig. 1. The assignment (1) produces at each iteration a new element $S[0][j-16][0]$ of the array $S$. We say that the *characteristic* memory variation of this assignment is +1 since each time the instruction is executed the memory size will increase by one location. Similarly, the assignment (3) has a characteristic memory variation of -1 (i.e., +1-1-1) since at each iteration one scalar signal $T[j-15]$ is produced and two other scalars – $S[0][j-16][297]$ and $T[j-16]$ – are consumed (used for the last time).

In general, the characteristic memory variation of an assignment is easily computed: a *produced* array reference having a bijective vector function has a contribution of +1; an entirely *consumed* array reference having a one-to-one mapping has a contribution of -1; an array reference having no component lattice consumed in the block has a zero contribution, independent of its mapping. The rest of the array references in the assignment are ignored for the time being, being dealt with at *Step 3.3*. For instance, at assignment (2), the two array references $S$ bring a contribution of +1 and -1, respectively; the array reference $A[4][j]$ brings a zero contribution since it contains only the lattice $A10$ (see Fig. 2) which is not consumed in this block ($A10$ is also covered by the array reference $A[k][i]$ from the second loop nest, so it will be consumed there). Only part of the array reference $A[k][i]$ is consumed in this block, that is, the lattice $A4$. Therefore, the characteristic memory variation of assignment (2) is zero, meaning that the *typical* variation is of zero locations; however, for some iterations the memory variation may be -1 due to the consumption of the elements of $A4$ covered by the array reference $A[k][i]$.

**Step 3.2** *Check whether the maximum storage requirement could occur in the current block or not.*

The memory size at the beginning of the first loop nest is 5,291 (and this is the largest value among the memory sizes at the block boundaries). The maximum possible memory increase is due only to assignment (1), having a characteristic memory variation of +1, executed 497 times. So, theoretically, the memory size could reach (but not exceed) the value 5,291+497 (although it will not). The maximum memory size could occur in this block, so its analysis should continue.

On the other hand, the memory size at the beginning of the second loop nest is 4,762. The memory size within that block could reach the value 4,762+497=5,259 which is already smaller than 5,291. It follows that the maximum storage requirement cannot occur in the second loop nest, so this block can be skipped from further analysis. This code pruning enhances the running times, concentrating the analysis on those portions of code where the memory increase is likely to happen.

**Step 3.3** *Compute the maximum (*lexicographic*) iterator vectors of the lattice elements consumed in the block, but of only those not covered by the array references that contributed to the characteristic memory variations (see* Step 3.1*) of the assignment instructions.*

Distinct iterator vectors can access a same array element, whereas we are only interested in that unique iterator vector accessing the array element *for the last time*. This is what we call the *maximum (lexicographic) iterator vector*.

*Definition* Let $\mathbf{i} = [i_1, \ldots, i_n]^T$ and $\mathbf{j} = [j_1, \ldots, j_n]^T$ 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 $maximum$ iterator vector from a set of such vectors is the largest vector in the set relative to the lexicographic order.

*Example*
$$for\ (i = 0;\ i \leq 5;\ i++)$$
$$\quad for\ (j = 0;\ j \leq 5;\ j++)$$
$$\quad\quad for\ (k = 0;\ k \leq 5;\ k{+}{+}) \quad \cdots M[i-3j+2k] \cdots$$

The $maximum$ iterator vector such that the element, say, $M[5]$ is accessed is $[i\ j\ k]^T_{max} = [5\ 2\ 3]^T$.

The algorithm computing these vectors is briefly presented. Given an array reference $M[x_1(i_1, \ldots, i_n)]\ \cdots\ [x_m(i_1, \ldots, i_n)]$ in the scope of the iterator polytope $\mathbf{A}\cdot[i_1\ \ldots\ i_n]^T \geq \mathbf{b}$ and an array element $M[x^0_1]\ \cdots\ [x^0_m]$:

1. *Solve the Diophantine system* [8] $x_j(i_1, \ldots, i_n) = x^0_j$ , $j = 1, \ldots, m$. *Let its solution (assuming it does exist) be* **i=V·t+u**.

2. *Bring matrix* $\mathbf{V}$ *to a reduced Hermite form* [8] *by post-multiplying it with a unimodular matrix* $\mathbf{S}$ *(less a possible row permutation):* $\mathbf{V'} = \mathbf{V} \cdot \mathbf{S} = \begin{bmatrix} \mathbf{V_1} \\ \mathbf{V_2} \end{bmatrix}$ *, where matrix* $\mathbf{V_1}$ *is lower-triangular, with positive diagonal elements.*

3. *Project the new* $\mathbf{Z}$*-polytope* $\mathbf{A}(\mathbf{V't'} + \mathbf{u}) \geq \mathbf{b}$ *on the first coordinate axis and find the maximum coordinate* $t'_1 \in \mathbf{Z}$ *in the "exact shadow"* [6]. *Replace its value in the* $\mathbf{Z}$*-polytope and repeat this operation, finding* $t'_2,\ t'_3,\ \ldots$ *. Then,*
$\mathbf{i}_{max} = \mathbf{V't'} + \mathbf{u}$*, where* **t'** *is the vector of the* $t'_k$ *values.* □

From Fig. 2, $A4$ is the only lattice consumed in the first loop nest satisfying the conditions of *Step 3.3*. It is part of the array reference $A[k][i]$, the elements of $A4$ having the indexes in the set $\{x = 0\ ,\ 0 \leq y \leq 528\}$. The maximum iterator vectors of these elements indicate the iterations when they are consumed. For instance, the element $A[0][0]$ is consumed in the iteration $[j\ k\ i]^T = [16\ \ 0\ \ 0]^T$; $A[0][1]$ is consumed in $[17\ \ 0\ \ 1]^T$, while $A[0][528]$ is consumed in the iteration $[512\ \ 0\ \ 528]^T$. In general, the maximum iterator vectors of $A4$'s elements are: $[j\ k\ i]^T_{max} = [t+16\ \ 0\ \ t]^T$ when $0 \leq t \leq 496$, and $[512\ \ 0\ \ t]^T$ when $497 \leq t \leq 528$.

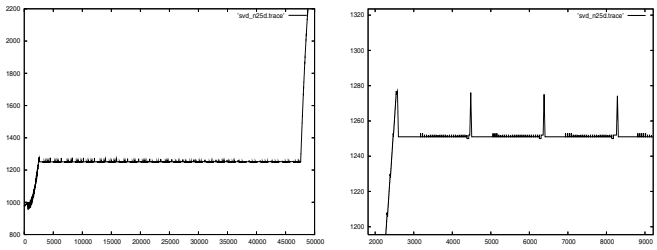


Figure 3: Memory trace for the SVD updating ($n = 25$) algorithm. The abscissae are the numbers of datapath instructions executed, the ordinates are memory locations. The first graph represents the entire trace. The second graph is a detailed trace in the interval [2200 : 9100], which corresponds to the end of the QR update and the start of the SVD diagonalization [3]. The global maximum is at the point (x=48848, y=2175), the rightmost on the first trace.

**Step 3.4** *Compute the maximum storage requirement in the block.*

In this moment, there is enough information to compute the exact memory size after any assignment increasing the memory (with positive characteristic memory variation). The initial memory at the beginning of the first loop nest is 5,291; the characteristic memory variations of the three assignments are +1, 0, and -1. Finding the number of times a certain assignment is executed reduces to the computation of the size of a **Z**-polytope [2]. The maximum storage requirement (5,292) is reached in the first iteration ($j = 16$) after assignment (1). It is the maximum value for the entire code and, therefore, this is the minimum amount of the data storage.

### 3. EXPERIMENTAL RESULTS

A software framework for the computation of the data memory size 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 Fig. 1). 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 algorithmic specification, the framework can optionally generate the variation 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 PC with a 1.85 GHz Athlon XP processor and 512 MB memory. The benchmarks used are: (1) a motion detection algorithm used in the transmission of real-time video signals on data networks [3]; (2) a real-time regularity detection algorithm used in robot vision; (3) Durbin's algorithm for solving Toeplitz systems with $n$ unknowns [9]; (4) the kernel of a motion estimation algorithm for moving objects (MPEG-4); (5) a singular value decomposition (SVD) updating algorithm used in spatial division multiplex access (SDMA) modulation, in beamforming, and Kalman filtering; (6) the kernel of a voice coding application.

This tool can process large specifications in terms of number of loop nests, lines of code, number of array references. In one of our experiments, the tool processed a difficult example of about 900 lines of code, with 113 loop nests 3-level deep, and a total of 906

| Application (parameters) | #Array refs. | #Scalars | Memory size | CPU (sec) |
|---|---|---|---|---|
| Motion detection | | | | |
| (M=N=32, m=n=4) | | 72,543 | 2,740 | 2 |
| (M=N=120, m=n=8) | 11 | 3,749,063 | 33,284 | 16 |
| Regularity detection | 19 | 4,752 | 2,304 | $< 1$ |
| Durbin alg. (n=500) | 27 | 252,499 | 1,249 | 15 |
| Motion estimation | 68 | 265,633 | 2,465 | 18 |
| SVD (n=100) | 87 | 3,045,447 | 34,950 | 26 |
| Vocoder kernel | 232 | 33,706 | 11,890 | 2 |

Table 1: Experimental results.

array references (most of them having complex indices), covering about 20 million scalars, and yielding a total of 3,159 lattices, in less than 6 minutes. Unlike the previous estimation techniques, this tool performs exact determinations for procedural codes. Zhao and Malik obtained an estimation of 1372 memory locations for the motion detection kernel ($M = N = 32$, $m = n = 4$) [10], whereas the correct result is 2,740 storage locations (see Table 1).

### 4. CONCLUSIONS

This paper has presented a non-scalar approach for computing the minimum data storage in multi-dimensional signal processing applications. Different from past works performing only memory size *estimations*, our approach can do *exact evaluations* even for complex algorithmic specifications.

## References

[1] 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.

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

[5] 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.

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

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

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

[9] 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.

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