Mapping Multi-Dimensional Signals into Hierarchical Memory Organizations*

Hongwei Zhu Ilie I. Luican Florin Balasa

Dept. of Computer Science, University of Illinois at Chicago, {hzhu7,iluica2,fbalasa}@uic.edu

Abstract

The storage requirements of the array-dominated and loop-organized algorithmic specifications running on embedded systems can be significant. Employing a data memory space much larger than needed has negative consequences on the energy consumption, latency, and chip area. Finding an optimized storage of the usually large arrays from these algorithmic specifications is an important step during memory allocation. This paper proposes an efficient algorithm for mapping multi-dimensional arrays to the data memory. Similarly to [13], it computes bounding windows for live elements in the index space of arrays, but this algorithm is several times faster. Moreover, since this algorithm works not only for entire *arrays, but also* parts *of arrays – like, for instance, array references or, more general, sets of array elements represented by lattices [11], this signal-to-memory mapping technique can be also applied in multi-layer memory hierarchies.*

1 Introduction

Many signal processing systems, particularly in the multimedia and telecom domains, are synthesized to execute mainly data-dominant applications. Their behavior 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; the indices of the array references are affine functions of loop iterators. The class of specifications with these characteristics are often called *affine* specifications [2].

In these targeted VLSI systems, 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 end product [2]. The optimized mapping of the multi-dimensional signals (arrays) from these behavioral specifications to the data memory is an important step during memory allocation. Employing a data memory space much larger than needed has several negative consequences: (a) the energy consumption per access increases with the memory size [2], as well as (b) the data access latency [9]; in addition, (c) large memories occupy more chip area.

De Greef *et al.* choose one of the canonical linearizations of the array (a permutation of its dimensions), followed by a modulo operation that wraps the set of "virtual" memory locations into a smaller set of actual physical locations [5].

Lefebvre and Feautrier, addressing parallelization of static control programs, developed in [6] an intra-array storage approach based on modular mapping, as well. They first compute the lexicographically maximal "time delay" between the write and the last read operations, which is a super-approximation of the distance between conflicting index vectors (i.e., whose corresponding array elements are simultaneously alive). Then, the modulo operands are computed successively as follows: the modulo operand b_1, applied on the first array index, is set to 1 plus the maximal difference between the first indices over the conflicting index vectors; the modulo operand b_2 of the second index is set to 1 plus the maximal difference between the second indices over the conflicting index vector, when the first indices are equal; and so on.

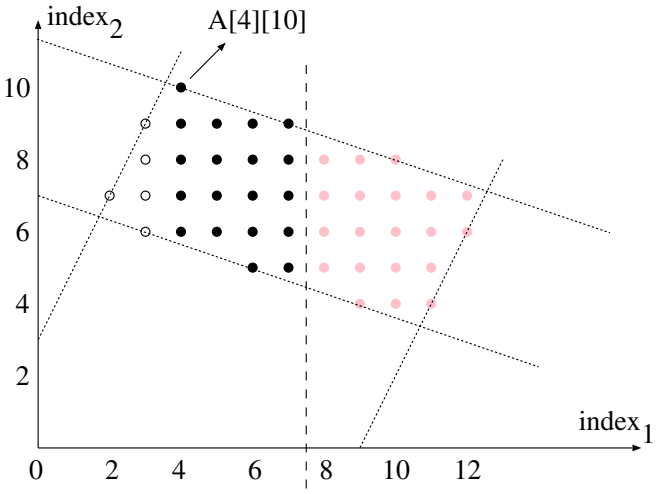
Quilleré and Rajopadhye studied the problem of memory reuse for systems of recurrence equations, a computation model used to represent algorithms to be compiled into circuits [8]. In their model, the loop iterators first undergo an affine mapping (into a linear space of *smallest* dimension – what they call a "projection") before modulo operations are applied to the array indices.

Darte *et al.* proposed a lattice-based mathematical framework for intra-array mapping, establishing a correspondence between valid linear storage allocations and integer lattices called *strictly admissible* relative to the set of differences of the conflicting indices [4]. They proposed two heuristic techniques for building strictly admissible integer lattices, hence building valid storage allocations.

In order to avoid the inconvenience of analyzing different linearization schemes like in [5], Tronçon *et al.* proposed [13] to reduce the size of an m-dimensional array A mapped to the data memory, by computing an m-dimensional bounding window $W_A = (w_1, \ldots, w_m)$, whose elements can be used as operands in modulo operations that redirect all accesses to the array A. An access to the element $A[index_1] \ldots [index_m]$ is redirected to $A[index_1 \bmod w_1] \ldots [index_m \bmod w_m]$ (relative to a base address in the data memory). Since the mapping has to ensure a correct execution of the code, two distinct array elements simultaneously alive should not be mapped by the modulo operations

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

978-3-9810801-2-4/DATE07 © 2007 EDAA




```
                                  //  All the A-elements are
for (i=2; i<16; i++)              //  consumed in this loop nest
   for (j=2*i-18; j<=2*i+3; j++)  {
      if ( i+3*j >= 21  &&  i+3*j <= 34 )   A[i][j] = ...
      if ( i+3*j >= 42  &&  i+3*j <= 55 )   ... = A[i-3][j-6] ;
   }
```

Figure 1: The array space of signal A. The points represent the A-elements $A[index_1][index_2]$ which are produced (and consumed as well) in the loop nest. The points to the left of the dashed line represent the elements produced till the end of the iteration ($i = 7, j = 9$), the black points being the elements still alive (i.e., produced and still needed in the next iterations) while the circles representing elements already 'dead' (i.e., not needed as operands any more). The light points to the right of the dashed line are A-elements still unborn (to be produced in the next iterations).

to the same location. Each window element w_i is the maximum difference in absolute value between the i-th indices of any two A-elements simultaneously alive, plus 1. The window W_A determines the memory size required for storing the array.

In the illustrative example shown in Fig. 1 the window corresponding to the 2-D array A is $W_A = (4\ ,\ 6)$. Indeed, it can be easily verified that the maximum distance between the first (respectively, second) indices of two alive elements cannot exceed 3 (respectively, 5). For instance, the simultaneously alive elements $A[4][10]$ and $A[7][5]$ (see Fig. 1) have both indices the farthest apart from each other. Therefore, a memory access to the element $A[index_1][index_2]$ can be safely redirected to $A[index_1 \bmod 4][index_2 \bmod 6]$. Afterwards, this bounding window will be mapped to the memory (starting at some base address) with a typical linearization scheme, like row by row, or column by column.

According to this intra-array mapping model, the storage requirement for the 2-D signal A is $4 \times 6 = 24$ memory locations. It must be noticed that, actually, only 19 locations would be really needed since not more than 19 A-elements can be simultaneously alive: see, again, Fig. 1, where the black dots represent the live elements at the end of the iteration ($i = 7, j = 9$). A minimum array window is typically difficult to use in practical allocation problems (in many cases, requiring a significantly more complex hardware). On the other hand, a signal-to-memory mapping model like the one described above trades-off an excess of storage for a less complex address generation hardware.

This paper proposes an efficient algorithm for mapping multi-dimensional arrays to the data memory. Similarly to [13], it computes bounding boxes for live elements in the index space of arrays, but this algorithm is several times faster. Moreover, since this algorithm works not only for *entire* arrays, but also *parts* of arrays – like, for instance, array references or, more general, sets of array elements represented by lattices [11], this signal-to-memory mapping technique can be also applied in a multi-layer memory hierarchy.

In addition, the software tool implemented based on our model can compute the optimal window for each multi-dimensional array in the specification (therefore, the optimal memory sharing between the elements of the same array, also called the *intra-array in-place mapping* [2]), and also the minimum storage requirement of the entire (procedural) algorithmic specification (therefore, the optimal memory sharing between all the array elements and scalars in the code). In this way, our software tool can provide an accurate evaluation of the *absolute* efficiency of the mapping model. This is not done by [13] or any other works in the field, which only make *relative* evaluations, comparing the performance of one mapping model versus other models.

The rest of the paper is organized as follows. Section 2 analyzes the mapping model at the level of an array reference. Section 3 describes the global flow of the intra-array mapping approach, focusing on the more significant algorithmic aspects. Section 4 discusses the application of the mapping model in a multi-layer memory hierarchy. Section 5 presents basic implementation aspects and discusses the experimental results. Section 6 summarizes the main conclusions of this work.

2 Computing an array reference window

In this section, a simpler problem will be addressed: given an 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 $i_1, \ldots, i_n$, and where the indices x_i are affine functions of the iterators, compute the dimensions of the bounding window for live elements (as explained in Section 1), assuming that all the M-elements of the array reference are simultaneously alive. The technique for solving this problem will be illustrated on the following

Example 1:
$$\begin{array}{l} for\ (i = 1;\ i \le 3;\ i++) \\ \quad for\ (j = 0;\ j \le 2;\ j++) \\ \quad\quad if(\ i + 2*j \le 5\ )\ \cdots A[2*i+j][i+3*j] \cdots \end{array}$$

The iterators (i,j) satisfy a set of linear inequalities derived from the loop boundaries and conditional expressions: $P = \{\ [i\ j]^T \mid 1 \le i \le 3\ ,\ 0 \le j \le 2\ ,\ i+2j \le 5,\ i, j \in \mathbf{Z}\}$. These in-

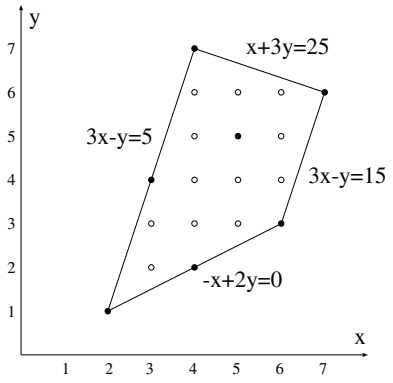


Figure 2: The index space of the array reference $A[2*i+j][i+3*j]$ from *Example 1* (where x and y are the two indices). Only the black dots represent possible indices.

equalities define what is usually called a **Z**-*polytope*.[1] The indices (x,y) of the array reference are given by the affine vector mapping

$$\left\{ \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \middle| \begin{bmatrix} i \\ j \end{bmatrix} \in P \right\}$$

Such a set is usually called a *lattice* [11] (linearly bounded) and it constitutes the *array space* or *index space* of the array reference (see Fig. 2).

More general, the problem to be solved is the following: given the lattice linearly bounded [12]

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

where $\mathbf{x} \in \mathbf{Z}^m$ is an index vector and $\mathbf{i} \in \mathbf{Z}^n$ is an iterator vector, compute the projection spans of the lattice on all the m coordinate axes. The general idea for solving this problem is to find a transformation **S** such that the extreme values of first iterator correspond to the extreme values of, say, the k-th index, for every value of k. In this way, the problem reduces to computing the projection of a **Z**-polytope, this latter problem being well-studied [7, 14].

Algorithm 1

Step 1 The k-th index has the expression: $x_k = \mathbf{t}_k \cdot \mathbf{i} + u_k$, where $\mathbf{t}_k$ is the k-th row of the matrix **T** in (1). Let **S** be a unimodular matrix[2] bringing $\mathbf{t}_k$ to the Hermite Normal Form [11]: $[h_1 \;\; 0 \;\cdots\; 0] = \mathbf{t}_k \cdot \mathbf{S}$. (If the row $\mathbf{t}_k$ is null, then the window reduces to one point: $x_k^{min} = x_k^{max} = u_k$.)

Step 2 After applying the unimodular transformation **S**, the new iterator **Z**-polytope becomes: $\bar{P} = \{ \bar{\mathbf{i}} \in \mathbf{Z}^n \mid \mathbf{A} \cdot \mathbf{S} \cdot \bar{\mathbf{i}} \geq \mathbf{b} \}$.

Step 3 Compute the extreme values of $\bar{i}_1$ (denoted $\bar{i}_1^{min}$ and $\bar{i}_1^{max}$) by projecting the **Z**-polytope $\bar{P}$ on the first axis [7]. Then, $x_k^{min} = h_1 \bar{i}_1^{min} + u_k$ and $x_k^{max} = h_1 \bar{i}_1^{max} + u_k$. □

The algorithm will be illustrated projecting the array reference from *Example 1* on the first axis and finding the extreme values of the first index x. From *Example 1*, $[x] = \begin{bmatrix} 2 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + [0]$. The unimodular matrix $\mathbf{S} = \begin{bmatrix} 0 & 1 \\ 1 & -2 \end{bmatrix}$ (see, e.g., [11] for building **S**) brings $\mathbf{t}_1 = \begin{bmatrix} 2 & 1 \end{bmatrix}$ to the Hermite Normal Form: $\mathbf{t}_1 \cdot \mathbf{S} = \begin{bmatrix} 1 & 0 \end{bmatrix}$. Since $\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -2 \end{bmatrix} \begin{bmatrix} \bar{i} \\ \bar{j} \end{bmatrix}$, the initial iterator space P (see *Example 1*) becomes $\bar{P} = \{1 \leq \bar{j} \leq 3 \, , \; 0 \leq \bar{i} - 2\bar{j} \leq 2 \, , \; 2\bar{i} - 5 \leq 3\bar{j}, \;\; \bar{i}, \bar{j} \in \mathbf{Z}\}$. Eliminating $\bar{j}$ from these inequalities with a Fourier-Motzkin technique [3], the extreme values of the *exact shadow* [7] of $\bar{P}$ on the first axis are $\bar{i}^{min} = 2$ and $\bar{i}^{max} = 7$, and those extreme points are valid projections (i.e., there exists $\bar{j}$'s such that $(\bar{i}^{min}, \bar{j})$ and $(\bar{i}^{max}, \bar{j})$ satisfy the inequalities defining $\bar{P}$). Since $h_1 = 1$ and $u_1 = 0$, it follows immediately that $x^{min} = 2$ and $x^{max} = 7$, which can be easily observed from Fig. 2. Projecting the lattice on the second axis, the second row of the affine mapping is $\mathbf{t}_2 = \begin{bmatrix} 1 & 3 \end{bmatrix}$ and the unimodular matrix is $\mathbf{S} = \begin{bmatrix} 1 & -3 \\ 0 & 1 \end{bmatrix}$. With a similar computation, it follows that $y^{min} = 1$ and $y^{max} = 7$. Therefore, the bounding window of the array reference $A[2*i+j][i+3*j]$ is W_A=(6 , 7).

3 Global flow of the mapping algorithm

In the previous section, we computed the bounding window for the live array elements covered by a given array reference. However, the assumption we made – that all the elements in the array reference are alive – does not always hold during the execution of the algorithmic specification. In the illustrative example from Fig. 3(a), the array reference $A[k][i]$ from the assignment marked with (*) is entirely alive (4761 elements) at the beginning of the loop nest, but it is not any more when the execution of the loop nest is over: since 529 of its elements ($A[0][i]$, $0 \leq i \leq 528$) were consumed for the last time during the loop nest execution, only 4232 A-elements are still alive at the end, needing thus storage.

Even more revealing, the array reference $S[0][j-16][33*k+i-j+17]$ from the same starred assignment covers 147,609 S-elements. If all the elements were simultaneously alive, the necessary bounding window would be $W_S = (1,\ 497,\ 297)$. However, a closer look shows that, actually, only one single element is alive at a time: each S-element produced in a certain iteration is immediately consumed in the next iteration, being also covered by the array reference $S[0][j-16][33*k+i-j+16]$. So, a window $W_S = (1,\ 1,\ 1)$ would suffice: each access to the two array references could be safely redirected to the same memory location.

The global flow of the mapping algorithm, computing the bounding windows of the arrays while taking into account the life span of the elements, is described below.

Algorithm 2

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

The decomposition of the array references will allow to find out which parts of an array reference are staying alive during the execution of a loop nest and which parts are consumed. This decom-

[1] In general, this iterator space can be a finite set of **Z**-polytopes; these can be considered disjoint (since, otherwise, a polytope decomposition can be applied).

[2] A square matrix whose determinant is ± 1.

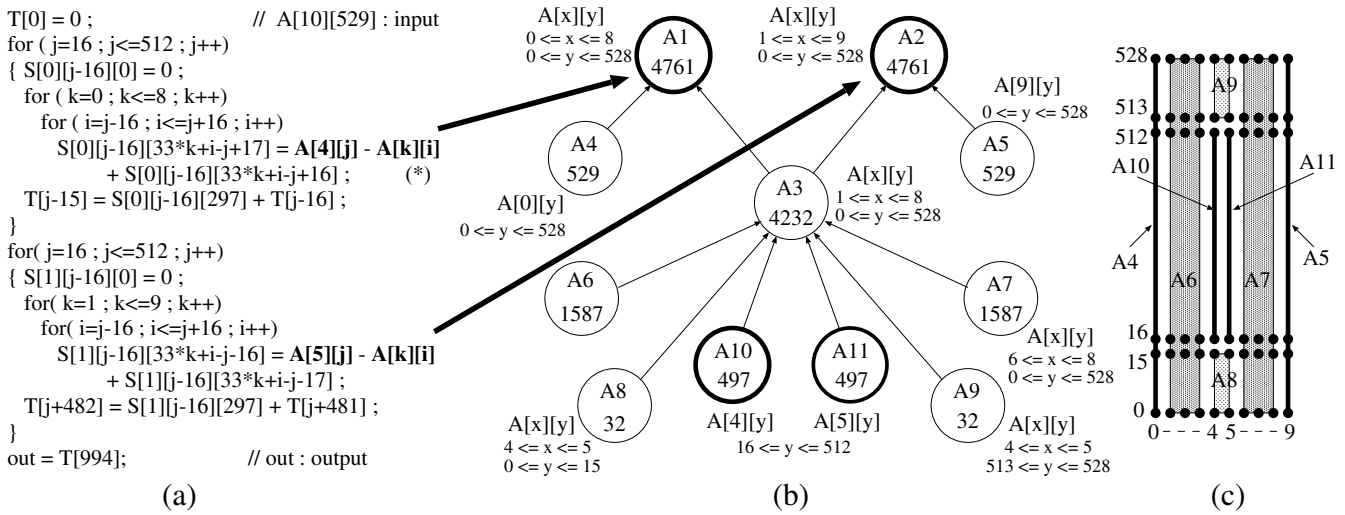


Figure 3: (a) *Example 2*. (b) Decomposition of the array (index) space of signal A into disjoint lattices; the arcs in the graph show the inclusion relations between lattices. (c) The partitioning of A's array space according to the decomposition (b).

position into disjoint lattices – used also in [15] – can be performed analytically, by recursively intersecting the array references of every multi-dimensional signal in the code. Two operations are relevant in our context: the *intersection* and the *difference* of two lattices. While the intersection of lattices was addressed also by other works (in different contexts, though) as, for instance, [12], the *difference* operation is far more difficult. (Because of lack of space, this operation will be described elsewhere.)

An inclusion graph is gradually built during the decomposition. This is a directed acyclic graph whose nodes are lattices and whose arcs denote inclusion relations between the respective sets. This graph is used on one hand to speed up the decomposition (for instance, if the intersection $L_1 \cap L_2$ results to be empty, there is no sense of trying to intersect L_1 with the lattices included in L_2 since those intersections will be empty as well), and on the other hand, to determine the structure of each array reference in terms of disjoint lattices.

Figure 3(b) shows such an inclusion graph and the result of the decomposition of the four ("bold") array references of the 2-dimensional signal A in the illustrative example from Fig. 3(a). The graph displays the inclusion relations ($arcs$) between the lattices of A ($nodes$). The four "bold" nodes represent the four array references of signal A in the code. For instance, the node $A1$ represents the lattice of $A[k][i]$ from the first loop nest. The nodes are also labeled with the size of the corresponding lattice, that is, the number of array elements covered by it. In this example, $A1 \cap A2 = A3$ and $A1 - A3$, $A2 - A3$ are also bounded lattices (denoted $A4$, $A5$ in Fig. 3(b)). However, the difference $A3 - A10$ is not (due to the non-convexity of this set), so the resulting set had to be decomposed further. At the end of the decomposition, the nodes without any incident arc ($A4, \ldots, A11$) represent non-overlapping lattices (they are displayed in Fig. 3(c)). Every array reference in the code is now either a disjoint lattice itself (like $A[4][j]$ is $A10$ and $A[5][j]$ is $A11$), or a union of disjoint lattices (e.g., the first array reference $A[k][i]$ is $A1 = A4 \cup A3 = A4 \cup \bigcup_{i=6}^{11} A_i$).

Step 2 *Using* Algorithm 1*, compute the extreme values of each signal's index for the live elements at the boundaries between blocks of code.*

The algorithmic specification is a sequence of nested loops, referred also as *blocks* of code. (Single instructions outside nested loops are actually nests of depth zero.) After the decomposition of the array references, one can easily find out the blocks where each disjoint lattice was *produced* and *consumed* (i.e., used as an operand for the last time).[3] For instance, the disjoint lattice $A4 = \{x = 0, y = j \mid 0 \leq j \leq 528\}$ (see Fig. 3(b)) belongs to the input signal A, so it is alive when the execution of the code starts. $A4$ is consumed in the first loop nest since it is included only in the array reference $A[k][i]$ (the lattice $A1$) in the assignment line marked with (*) in Fig. 3(a).

Based on the live lattices between the blocks of code, one can compute using *Algorithm 1* preliminary windows for each signal (array). For instance, since the disjoint lattices $A5, \ldots, A11$ are alive before the start of the second loop nest, the extreme values of the two indices of signal A are 1 and 9, and respectively, 0 and 528 (see Fig. 3(c)), A's bounding window relative to that block boundary being W_A=(9 , 529). But since all the disjoint lattices $A4, \ldots, A11$ are alive before the start of the first loop nest, then the bounding window for that boundary is larger: W_A=(10 , 529). Obviously, the maximal window must be selected. Similarly, signal T will have a window W_T=(1) since only one T-element is alive between the blocks of code ($T[0]$, $T[497]$, or $T[994]$). The window of the 3-dimensional signal S is W_S=(0, 0, 0) for the time being since no S-element is alive at any boundary between blocks.

The index windows computed here are the final ones when every

[3]The algorithmic specifications are in the *single-assignment* form, that is, each array element is written at most once, but it can be read an arbitrary number of times.

block of code either produces or consumes (but not both!) the array's elements. This works for signal A in *Example 2*, but it does not for S and T (see Fig. 3(a)).

Step 3 *Update the extreme values of the signal's indices for the live elements within each block of code where array elements are both produced and consumed.*

In the illustrative example from Fig. 1, the A-elements are both produced ($A[i][j] = \ldots$) and consumed ($\ldots = A[i-3][j-6]$) in the loop nest. In such a situation, the basic idea is to compute the iterator vectors when array elements are accessed for the last time (due to lack of space, this computation will be described elsewhere) and, subsequently, apply *Algorithm 1* to the live lattices corresponding to those iterations.

There are quite frequently situations when array elements are produced and consumed at every iteration. Such cases, conveniently exploited, occur when array references have one-to-one correspondences between the iterators and the indices. For instance, in *Example 2*, each iterator vector $[j\ k\ i]^T$ corresponds to a unique (produced) array element $S[0][j-16][33*k+i-j+17]$ and a unique (consumed) element $S[0][j-16][33*k+i-j+16]$ (see the assignment marked with (*) in the first loop nest). Since each iteration will produce and consume one element, the bounding window of S is W_S=(1, 1, 1). Similarly, each iterator j corresponds to unique T-elements $T[j-15]$ and $T[j-16]$ which yields a window W_T=(1).

The storage requirement for *Example 2* is obtained by summing up the individual window sizes: $|W_A| + |W_S| + |W_T|$= 5290+1+1=5292 memory locations. It can be shown that this storage requirement is really the minimum one, therefore, the mapping model we adopted – based on the bounding windows of simultaneously alive array elements – was very effective for this code. However, this is not always the case: as already mentioned in Section 1, signal A's window in the illustrative example from Fig. 1 is (4, 6), whereas the minimum storage requirement is 19 locations.

4 Mapping signals to multi-layer memories

In embedded communication and multimedia processing applications, the manipulation of large data sets has a major effect on both power consumption and performance of the system. This is due to the significant amount of data transfers to/from large, high-latency, energy-consuming off-chip data memories. The energy cost can be reduced and the system performance enhanced by introducing an optimized custom memory hierarchy that exploits the temporal locality of the data accesses [2]. The optimization of the hierarchical memory architecture implies the addition of layers of smaller memories to which heavily used data can be copied. Savings of dynamic energy can thus be obtained by accessing frequently used data from smaller on-chip memories rather than from large off-chip memories [1].

Our data reuse model identifies those parts of arrays which are more intensely accessed. These parts of arrays – represented as lattices as well – are copied in a scratch-pad memory (software-controlled SRAM or DRAM, more energy-efficient than a cache), achieving a reduction of the dynamic energy consumption due to memory accesses. Since the bounding window of any lattice (rather than whole array) can be computed as shown in this paper (e.g., using *Algorithm 1* if all its array elements are alive), those lattices covering the most accessed elements of the arrays are copied in the scratch-pad memory.[4]

For instance, in *Example 2* from Fig. 3(a), the lattices $A10 = \{x = 4,\ y = j \mid 16 \leq j \leq 512\}$ and $A11 = \{x = 5,\ y = j \mid 16 \leq j \leq 512\}$, covering roughly the center of the array space of signal A (see Fig. 3(c)), are the most accessed array parts: 179,867 memory accesses for each of the two lattices $A10$ and $A11$, therefore, an average of 361.90 accesses for each A-element covered by them. Assuming $A10$ and $A11$ are copied in the scratch-pad memory, their bounding windows computed with *Algorithm 1* (since all their elements are simultaneously alive) are both W=(1, 497). Moreover, since both lattices are simultaneously alive, the overall bounding window is W=(2, 497). Obviously, this window is smaller than the window (10,529) of the whole array A. This allocation solution with two memory layers implies, besides the background memory of 5292 locations, another memory of 2×497=994 locations closer to the processor. This increase of data memory space is compensated here by savings of dynamic energy of over 30%, according to the CACTI power model [10].

5 Experimental results

A software tool performing the mapping of the multi-dimensional arrays from a given algorithmic specification (expressed in a subset of the C language) into the data memory has been implemented in C++, incorporating the algorithms described in this paper.

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. Columns 2 and 3 display the numbers of array references and array elements in the specification code. Columns 4 and 5 show the data memory sizes (i.e., the total window sizes of the arrays[5]) after the signal-to-memory mapping and the CPU times. Column 6 displays the sum of the *minimum* array windows (optimized intra-array memory sharing). Column 7 shows the minimum storage requirements of the application codes, when the memory sharing between elements of different arrays is optimized as well. For a better understanding of the meaning of the data shown in this table, the first test is the illustrative example in Fig. 1. The rest of the benchmark tests are multimedia applications and typical algebraic kernels used in signal processing.

Although our approach employs a mapping strategy similar to [13], the computation methodology is entirely different. Tronçon *et al.* use, basically, sequences of emptiness checks for **Z**-polytopes derived from the code [13], whereas our algorithm

[4] A more detailed presentation of our data reuse model is beyond the scope of this paper which focuses on the signal-to-memory mapping.

[5] If two entire arrays have disjoint lifetimes, their bounding windows can be mapped starting at the same base address. Then, the maximum of the two window sizes is added to the other window sizes. Such situations, or even more general ones, do not occur in the current tests; so, the sum of the window sizes is a reasonable measure for the total storage requirement.

Application	# Array references	# Array elements	Memory size after mapping	CPU [sec]	Memory size (min. array windows)	Min. memory size (optimal memory sharing)
Example Figure 1	2	44	**24**	$<$ **1**	19	19
Motion detection	11	318,367	**9,525** (+1952)	**12**	9,525	9,524
Regularity detection	19	4,752	**4,353** (+875)	**3**	2,449	2,304
SVD updating	87	386,472	**17,554** (+1654)	**18**	10,204	8,725
Voice coder	232	33,619	**13,104** (+912)	**14**	12,963	11,890

Table 1: Experimental results. Column 4 displays with **bold fonts** the total window sizes if one memory layer is used; it also displays in parentheses the total window sizes of the array elements copied in the scratch-pad when two memory layers are used (see Section 4).

is based on the decomposition of the array references in disjoint bounded lattices. As a result, our approach can be used in multi-layer memory organizations and, in addition, it is several times faster. The computation times reported in [13] are typically of the order of minutes, whereas our implementation runs for the same examples, or of similar complexity, in tens of seconds at most. For instance, the *voice coding* application – component of a mobile radio terminal – was processed by [13] in over 25 minutes (using a 300 MHz Pentium II); in contrast, our running time is significantly shorter (in spite of the different computation platforms): only 14 seconds (with a 1.85 GHz Athlon XP). Another common benchmark is a singular value decomposition (SVD) updating algorithm, an algebraic kernel used in spatial division multiplex access (SDMA) modulation in mobile communication receivers. Tronçon *et al.* used, probably, in this test matrices of a smaller order (or only part of the code) since their test has only 6,038 array elements; their reported run time is 87 seconds. In our test with 386,472 array elements, the CPU time was 18 seconds.

Different from all the other signal-to-memory mapping techniques, our computation methodology allows to determine the additional storage implied by the mapping model by computing the minimum bounding window of each array and the minimum storage requirement of the whole code (the last two columns in the result table), providing thus useful information for the initial design phase of system-level exploration. Table 1 shows that there are applications, like the motion detection, where this mapping model gives very good solutions, close (or, even, equal) to the optimal array windows. On the other hand, there are also applications where the window sizes result significantly larger than the optimal ones (e.g., 72% larger for the SVD updating). The positive aspect is that our tool is able to measure the quality of the mapping, whereas the other works in the field do not provide similar information.

6 Conclusions

This paper has addressed the problem of intra-array mapping of the multi-dimensional signals from multimedia behavioral specifications to the data memory. The paper proposes a mapping algorithm based on the computation of bounding windows for the simultaneously alive array elements, technique that can be also applied in multi-layer memory hierarchies. Moreover, the software tool implementing this algorithm can compute the optimal window for each multi-dimensional array in the specification, providing an accurate evaluation of the efficiency of the mapping model.

References

[1] E. Brockmeyer, M. Miranda, H. Corporaal, F. Catthoor, "Layer assignment techniques for low energy in multi-layered memory organisations," *Proc. 6th ACM/IEEE Design and Test in Europe Conf.*, pp. 1070-1075, Munich, Germany, March 2003.

[2] 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.

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

[4] A. Darte, R. Schreiber, G. Villard, "Lattice-based memory allocation," *IEEE Trans. Computers*, vol. 54, pp. 1242-1257, Oct. 2005.

[5] E. De Greef, F. Catthoor, H. De Man, "Memory size reduction through storage order optimization for embedded parallel multimedia applications", special issue on "Parallel Processing and Multimedia" (ed. A. Krikelis), in *Parallel Computing*, Elsevier, vol. 23, no. 12, pp. 1811-1837, Dec. 1997.

[6] V. Lefebvre, P. Feautrier, "Automatic storage management for parallel programs," *Parallel Computing*, vol. 24, pp. 649-671, 1998.

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

[8] F. Quilleré, S. Rajopadhye, "Optimizing memory usage in the polyhedral model," *ACM Trans. Programming Languages and Syst.*, vol. 22, no. 5, pp. 773-815, 2000.

[9] 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.

[10] G. Reinman, N.P. Jouppi, "CACTI2.0: An integrated cache timing and power model," COMPAQ Western Research Lab, 1999.

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

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

[13] R. Tronçon, M. Bruynooghe, G. Janssens, F. Catthoor, "Storage size reduction by in-place mapping of arrays," *Verification, Model Checking and Abstract Interpretation*, pp. 167-181, 2002.

[14] S. Verdoolaege, K. Beyls, M. Bruynooghe, F. Catthoor, "Experiences with enumeration of integer projections of parametric polytopes," in *Compiler Construction: 14th Int. Conf.*, pp. 91-105, 2005.

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