

System-Level Exploration Tool for Energy-Aware Memory Management in the Design of Multidimensional Signal Processing Systems

Florin Balasa* Ilie I. Luican† Hongwei Zhu‡ Doru V. Nasui§

* Dept. of Computer Science, Southern Utah University, Cedar City, UT

† Dept. of Computer Science, University of Illinois at Chicago, Chicago, IL

‡ ARM, Inc., Sunnyvale, CA

§ American Int. Radio, Inc., Rolling Meadows, IL

Abstract – Many signal processing systems, particularly in the multimedia and telecom domains, are synthesized to execute data-dominated applications. In such systems, data transfer and storage have a significant impact on both the system performance and the major cost parameters – power consumption and chip area. This paper presents a software tool for system-level exploration, where several memory management tasks are addressed in a common theoretical framework. The tool can compute the minimum storage requirement of a given application and can produce the graph of storage variation during the code execution; it offers memory allocation and signal assignment solutions both for flat and hierarchical organizations and optimizes the dynamic energy consumption in the memory subsystem.

1 Introduction

The storage organization in data-intensive multidimensional signal processing applications, particularly in the multimedia and telecommunication domains, has an important impact on the system-level energy budget. Due to the manipulation of large sets of data – typically organized as multidimensional arrays in the behavioral specification, a multi-layer memory hierarchy is typically used not only to enhance the VLSI system performance but, also, to reduce the overall energy consumption [5].

Despite many advances in memory design techniques over the past two decades, existing computer-aided design (CAD) methodologies are still ineffective in many aspects. Although the reduction of the dynamic energy consumption in hierarchical memory subsystems has been addressed in the past (e.g., [4], [8]), the provided CAD solutions were based on mainly heuristic explorations of the solution space, rather than on a formal methodology. At the same time, the reduction of the static energy in the memory subsystem is a research direction only more recently addressed (e.g., [9]). Several models of mapping the multidimensional signals into the physical memory were proposed in the past (see [7] for a good overview). However, they all failed (a) to provide efficient implementations, (b) to prove their effectiveness in hierarchical memory organizations, and (c) to provide quantitative measures of quality for the mapping solutions.

This paper presents a software tool for system-level exploration, where several memory management tasks are addressed in a common theoretical framework (which distinguishes it from all the

previous works). The tool can compute the minimum storage requirement of a given application and can produce the graph of storage variation during the code execution; it offers memory allocation and signal assignment solutions both for flat and hierarchical organizations and optimizes the dynamic energy consumption in the memory subsystem. (Extensions of this CAD system to the exploitation of memory banking, or dealing with an arbitrary number of memory layers, as well as taking into account the leakage energy consumption, will be considered in the future.)

The rest of the paper is organized as follows. Section 2 briefly presents the polyhedral framework of our CAD system. Section 3 discusses the memory management tasks currently available for system-level exploration: the computation of the minimum data storage of an application and the exploration of functionally-equivalent specifications, the memory allocation and signal assignment, the optimization of the dynamic energy consumption in the memory subsystem. Section 4 discusses implementation aspects and presents experimental results. Finally, Section 5 summarizes the main conclusions of this research.

2 Polyhedral framework for memory management tasks

The algorithmic specifications of telecom and multimedia processing applications are typically organized in sequences of loop nests, containing also conditional instructions, and having as data structures multidimensional arrays, whose array references have (possibly complex) indices, linear functions of the surrounding loop iterators. These input specifications are, typically, considered to be *procedural* (that is, where the loop structure and sequence of instructions induce the execution ordering) and in the *single-assignment* form (that is, each array element is written at most once, but it can be read as an operand an arbitrary number of times). An illustrative example, extracted from the kernel of a motion detection algorithm [6] used in the transmission of real-time video signals on data networks, is shown in Fig. 1.

The basic terminology used in this paper is briefly specified below. A *polyhedron* is a set of points $P \subset \mathbb{R}^n$ satisfying a finite set of linear inequalities: $P = \{ \mathbf{x} \in \mathbb{R}^n \mid \mathbf{A} \cdot \mathbf{x} \geq \mathbf{b} \}$, where $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{b} \in \mathbb{R}^m$. If P is a bounded set, then P is called a *polytope*. If $\mathbf{x} \in \mathbb{Z}^n$, then P is called a *Z-polyhedron/polytope*. The set $\{ \mathbf{y} \in \mathbb{R}^m \mid \mathbf{y} = \mathbf{A}\mathbf{x}, \mathbf{x} \in \mathbb{Z}^n \}$ is called the *lattice* generated by the columns of matrix \mathbf{A} . Each array reference

```

optDelta[0] = 0;           // A[67][161] : input
for ( j=32 ; j<=128 ; j++) // The first loop nest
{ Delta[32][j][0] = 0;
  for ( k=0 ; k<=64 ; k++)
    for ( i=j-32 ; i<=j+32 ; i++)
      Delta[32][j][65*k+i-j+33] = A[32][j] - A[k][i]
      + Delta[32][j][65*k+i-j+32];
  optDelta[j-31] = Delta[32][j][4225] + optDelta[j-32];
}
for( j=32 ; j<=128 ; j++) // The second loop nest
{ Delta[33][j][0] = 0;
  for( k=1 ; k<=65 ; k++)
    for( i=j-32 ; i<=j+32 ; i++)
      Delta[33][j][65*k+i-j-32] = A[33][j] - A[k][i]
      + Delta[33][j][65*k+i-j-33];
  optDelta[j+66] = Delta[33][j][4225] + optDelta[j+65];
}
for( j=32 ; j<=128 ; j++) // The third loop nest
{ Delta[34][j][0] = 0;
  for( k=2 ; k<=66 ; k++)
    for( i=j-32 ; i<=j+32 ; i++)
      Delta[34][j][65*k+i-j-97] = A[34][j] - A[k][i]
      + Delta[34][j][65*k+i-j-98];
  optDelta[j+163] = Delta[34][j][4225] + optDelta[j+162];
}
opt = optDelta[291];      // opt : output

```

Figure 1: Affine algorithmic specification derived from the kernel of a motion detection algorithm [6].

$M[x_1(i_1, \dots, i_n)] \cdots [x_m(i_1, \dots, i_n)]$ of an m -dimensional signal M , in the scope of a nest of n loops having the iterators i_1, \dots, i_n , is characterized by an *iterator space* and an *index* (or *array*) *space*. The iterator space signifies the set of all iterator vectors $\mathbf{i} = (i_1, \dots, i_n) \in \mathbf{Z}^n$ in the scope of the array reference. The index space is the set of all index vectors $\mathbf{x} = (x_1, \dots, x_m) \in \mathbf{Z}^m$ 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) *lattice(s)* [12] linearly bounded, that is, the image of an affine vector function over the iterator \mathbf{Z} -polytope¹ $\mathbf{A} \cdot \mathbf{i} \geq \mathbf{b}$:

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

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. In our context, the elements of all vectors and matrices are considered integers. E.g.,

```

for (i=0; i<=2; i++)
  for (j=0; j<=3; j++)
    for (k=0; k<=4; k++)
      if ( 6*i+4*j+3*k <= 12 ) ... = A[i+2*j+3][j+2*k];

```

The iterator space and the index space of the array reference are:

$$P = \left\{ \begin{bmatrix} i \\ j \\ k \end{bmatrix} \in \mathbf{Z}^3 \mid \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ -6 & -4 & -3 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ -12 \end{bmatrix} \right\}$$

(the inequalities $i \leq 2$, $j \leq 3$, and $k \leq 4$ are redundant), and

¹The iterator space of an array reference is not always a \mathbf{Z} -polytope; it can be a non-convex polyhedron, or even a union of convex and non-convex polyhedra; but, nevertheless, it can be *decomposed* into a finite set of disjoint \mathbf{Z} -polytopes; thus, without lack of generality, the iterator space can be considered one \mathbf{Z} -polytope.

$$\left\{ \begin{bmatrix} x \\ y \end{bmatrix} = \mathbf{T} \cdot \mathbf{i} + \mathbf{u} = \begin{bmatrix} 1 & 2 & 0 \\ 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 3 \\ 0 \end{bmatrix} \mid \begin{bmatrix} i \\ j \\ k \end{bmatrix} \in P \right\}$$

The key step in our framework is the decomposition of the array references for every indexed signal from the given application code into *disjoint* lattices [2]. The benefits of this strategy will become clearer in the next section, but the motivation of partitioning the array references relies on the following intuitive idea: the disjoint lattices which belong to many array references are actually those parts of the arrays more heavily accessed during the code execution. In addition, this decomposition allows to obtain lifetime information on *entire groups* of array elements (which is sufficient for most memory management tasks), without the need of operating with individual array elements (which is a strategy prohibitively time-consuming [5]). The decomposition into disjoint lattices is analytically performed, using intersections and differences of lattices – operations quite complex [2] involving computations of Hermite Normal Forms, solving Diophantine linear systems [12], computing the vertices of \mathbf{Z} -polytopes [1] and their supporting polyhedral cones, counting integral points in \mathbf{Z} -polyhedra [3], and computing integer projections of polytopes [14].

3 System-level exploration for memory management

The algebraic framework is the kernel of our CAD system, the memory management tasks (see the next subsections) being implemented on top of it. In this way, any task has access to useful information characterizing the code: the array references and their iterator polytopes, the collections of lattices for every indexed signal, together with complete information on their sizes (the number of array elements they cover), inclusion relations into the array references, and lifetime (the loop nest where a lattice is produced and the one where it is consumed for the last time).

3.1 Computation of the minimum data storage

The minimum storage requirement of an algorithmic specification is actually the maximum number of scalar signals (array elements) simultaneously alive during the code execution. Our tool can compute the minimum data storage for one indexed signal, for a subset of signals, or for the entire code. The mechanism of the computation is identical; the only difference is that, when dealing with only one indexed signal we use only its lattices, whereas when dealing with the entire code, all the lattices are employed. The possibility of memory sharing between the array elements is inherently taken into account (see [2] for a thorough discussion).

Algorithm 1: Computation of the minimum storage requirement for a given array A in the algorithmic specification

Step 1 Compute (an underestimation of) the minimum storage requirement for signal A (denoted Mem_A^{min}) taking into account only the live A -elements at the boundaries between the loop nests.

for (each boundary n between the loop nests n and $n + 1$) {
 let $\mathcal{L}_A(n)$ be the set of disjoint lattices of A , alive at boundary n

```

// The array elements A-E are produced
for ( i=0; i<767; i++) // and consumed in this loop nest
  for ( j=0; j<256; j++) {
    if ( i+j>=127 && i+j<=254 && j<=127 ) A[i][j] = 1 ;
    if ( i+j>=255 && i+j<=382 && j<=127 ) ... = A[i-128][j];
    if ( i+j>=159 && i+j<=318 && j<=159 ) B[i][j] = 2 ;
    if ( i+j>=319 && i+j<=478 && j<=159 ) ... = B[i-160][j];
    if ( i+j>=191 && i+j<=382 && j<=191 ) C[i][j] = 3 ;
    if ( i+j>=383 && i+j<=574 && j<=191 ) ... = C[i-192][j];
    if ( i+j>=223 && i+j<=446 && j<=223 ) D[i][j] = 4 ;
    if ( i+j>=447 && i+j<=670 && j<=223 ) ... = D[i-224][j];
    if ( i+j>=255 && i+j<=510 )           E[i][j] = 5 ;
    if ( i+j>=511 && i+j<=766 )           ... = E[i-256][j];
  }

```

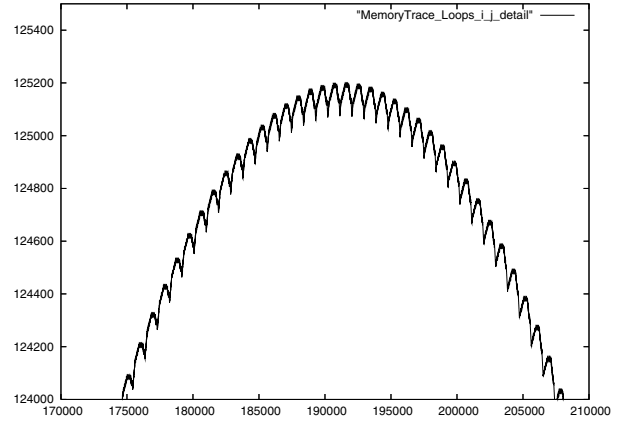


Figure 2: Algorithmic specification and the variation of the storage requirement for the group of signals A, \dots, E . The graph is a detail of the memory trace in a neighborhood of the global maximum. The minimum data storage (corresponding to the *maximum* value of the storage requirement) is 125,193 locations. The abscissae are the number of datapath instructions during the code execution.

(i.e., produced *before* the boundary and consumed *after* it);

$$Mem_A^{min}(n) = \sum_{L \in \mathcal{L}_A(n)} size(L);$$

// The minimum storage requirement of A at boundary n

$$Mem_A^{min} = \max_n \{ Mem_A^{min}(n) \};$$

// min. storage requirement of A over all loop boundaries

Step 2 Compute the exact value of Mem_A^{min} .

Mem_A^{min} after *Step 1* is the exact value of the minimum storage when all the A -elements are either produced or consumed (but not both!) in the loop nests. If this is not the case, the value Mem_A^{min} obtained at *Step 1* is a lower bound of the storage requirement and may need to be adjusted upwards. Let \mathcal{L}_A be the set of disjoint lattices of the indexed signal A , and let $L \in \mathcal{L}_A$ be a lattice which is consumed in a (normalized) loop nest where A -elements are produced as well. Then,

for (each A -element covered by the lattice L) {
 compute the iteration vector \mathbf{i} when the A -element is read for the last time;²
 determine the disjoint lattices \mathcal{L}_p of A partially produced until the A -element is consumed in iteration \mathbf{i} ;
 determine the disjoint lattices \mathcal{L}_c of A partially consumed until the A -element is consumed in iteration \mathbf{i} ;
if ($\sum_{L' \in \mathcal{L}_p} size(L') > \sum_{L'' \in \mathcal{L}_c} size(L'')$) $Mem_A^{min} = \max\{ Mem_A^{min}, Mem_A^{min}(n) + \sum_{L' \in \mathcal{L}_p} size(L') - \sum_{L'' \in \mathcal{L}_c} size(L'') \}$;
}

The guiding idea is that a local or global maximum of Mem_A^{min} is reached immediately before the consumption of some A -element. *Step 2* actually investigates all the local maxima within that loop nest.

The computation of the minimum data storage can be employed to assess the quality in terms of storage amount of the memory allocation solution. It is also useful in evaluating the impact of dif-

²This requires the computation of the maximum iterator vector relative to the lexicographic order [2]. If the lattice L is included in several array references in that loop nest, the *overall* maximum iterator vector is considered.

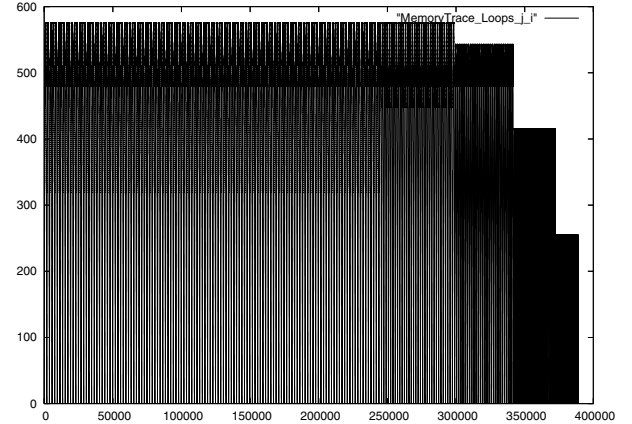


Figure 3: Memory trace of the code in Fig. 2 after a loop interchange: the minimum data storage is only 576 locations.

ferent code (and, in particular, loop) transformations on the data storage. For instance, the minimum memory size needed for the execution of the code from Fig. 2 is 125,193 locations – as computed by the software tool. By interchanging the loops, the storage requirement decreases to only 576 locations – a drastic reduction with over 99.5%, the new memory trace being the graph shown in Fig. 3. Different functionally-equivalent code versions of a same application can be compared one against another in storage point of view, without the need of performing a proper memory allocation for each code version – which would be a significantly more expensive solution.

3.2 Mapping multidimensional signals into the physical memory

The signal-to-memory assignment algorithm employed in our CAD system has a similar general goal as Tronçon's mapping

model [13]: for each m -dimensional array A in the specification code, it computes a maximal bounding window W_A whose dimensions w_i ($1 \leq i \leq m$) are used to redirect the accesses to the array. Any array element is mapped into the window W_A : $A[index_1] \dots [index_m] \mapsto W_A[index_1 \bmod w_1] \dots [index_m \bmod w_m]$. Each window dimension w_i is computed as the maximum difference in absolute value between the i -th indices of any two array elements simultaneously alive, plus 1. This model ensures that any two array elements simultaneously alive are mapped to distinct memory locations. In its turn, the window W_A is mapped into the physical memory (relative to a base address) by a typical canonical linearization, like row or column concatenation for 2-dimensional arrays. The amount of data memory required for storing (after mapping) the array A is the volume of the window W_A , that is, $\prod_{i=1}^m w_i$.

Before describing the global flow of the mapping algorithm, we are going to explain how the window sides w_k , for $k = 1, \dots, m$, are computed for just one (linearly bounded) lattice (1).

Algorithm 2: Computation of the k -th window side w_k of a lattice

The general idea for solving this problem is to find a transformation \mathbf{S} such that the extreme values of first iterator correspond to the extreme values of the k -th index. In this way, the problem reduces to the computation of the projection of a \mathbf{Z} -polytope, this latter problem being well-studied (e.g., [14]).

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 \mathbf{T} in (1). Let \mathbf{S} be a unimodular matrix³ bringing \mathbf{t}_k to the Hermite Normal Form [12]: $[h_1 \ 0 \ \dots \ 0] = \mathbf{t}_k \cdot \mathbf{S}$. (If the row \mathbf{t}_k is null, then the window reduces to one point: $w_k=1$ since $x_k^{min} = x_k^{max} = u_k$.)

Step 2 After applying the unimodular transformation \mathbf{S} , the new iterator \mathbf{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 \mathbf{Z} -polytope \bar{P} on the first axis. 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 k -th window dimension $w_k = x_k^{max} - x_k^{min} + 1$. \square

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. The overall size of these windows yields the (one-layer) data memory allocation. Moreover, the window sides w_k are used to map the array elements to the physical memory locations using modulo operations – as explained above, assignment solution complementing the allocation.

Algorithm 3: Computation of the bounding windows of all the arrays in the algorithmic specification

Step 1 Compute preliminary windows (underestimations of w_k) for each indexed signal taking into account the live elements at the boundaries between the loop nests.

Let A be an indexed signal in the algorithmic specification and \mathcal{L}_A the set of its disjoint lattices. A high-level pseudo-code of the computation of A 's preliminary window is given below:

for (each disjoint lattice $L \in \mathcal{L}_A$)

for (each dimension k of signal A)

compute x_k^{min} and x_k^{max} of L using Algorithm 2 ;

for (each boundary n between the loop nests n and $n + 1$) {
let $\mathcal{L}_A(n)$ be the set of disjoint lattices of A , alive at boundary n
(i.e., produced before the boundary and consumed after it);

for (each dimension k of signal A) {

$X_k^{min} = \min_{L \in \mathcal{L}_A(n)} \{x_k^{min}\}$; $X_k^{max} = \max_{L \in \mathcal{L}_A(n)} \{x_k^{max}\}$;
 $w_k(n) = X_k^{max} - X_k^{min} + 1$; // $w_k(n)$ is the k -th side
// of A 's bounding window at boundary n

}
}

for (each dimension k of signal A) $w_k = \max_n \{w_k(n)\}$;

// w_k is now the k -th side of A 's window over all boundaries

The window sides w_k computed for each index k at Step 1 are maximal when every loop nest in the code either produces or consumes (but not both!) the array's elements. This works, e.g., for signal A in the illustrative example (see Fig. 1), where every loop nest only consumes A -elements. On the other hand, this does not work for Δ and $opt\Delta$ whose elements are both produced and consumed in the same loop nest. In this situation, their values of w_k after Step 1 are only underestimates and, hence, may need to be increased. This is done in the next step.

Step 2 Update the bounding window (compute the exact values of w_k) for each indexed signal.

Let $L \in \mathcal{L}_A$ be a disjoint lattice of an indexed signal A which is consumed in a certain (normalized) loop nest where A -elements are produced as well. Then,

for (each A -element covered by the lattice L) {

compute the iteration vector \mathbf{i} when the A -element is read for the last time;

determine the disjoint lattices \mathcal{L}_p of A partially produced until the A -element is consumed in iteration \mathbf{i} ;

determine the disjoint lattices \mathcal{L}_c of A partially consumed until the A -element is consumed in iteration \mathbf{i} ;

for (each dimension k of signal A)

for (each lattice in $\mathcal{L}_p \cup \mathcal{L}_c$) {

compute x_k^{min} and x_k^{max} using Algorithm 2 ;

$w_k = \max \{ w_k, x_k^{max} - x_k^{min} + 1 \}$; // w_k may increase

}
}

The guiding idea is that local or global maxima of w_k are reached immediately before the consumption of an A -element, which may entail a shrinkage of some side of the bounding window encompassing the live elements.

The memory windows of the signals in the illustrative example from Fig. 1 are: $W_A = (67, 161)$, $W_{\Delta} = (1, 1, 1)$, and $W_{opt\Delta} = (1)$. (Here, W_A results equal to the entire array space since A is an input signal.) The storage requirement is obtained by summing up the individual window sizes: $|W_A| + |W_{\Delta}| + |W_{opt\Delta}| = 67 \times 161 + 1 + 1 = 10,789$ memory locations. The data storage after mapping for this example is equal to the minimum memory size computed by Algorithm 1 in Section 3.1 (although this is not always true); therefore, the mapping model employed – based on the bounding windows of simultaneously alive array elements – is very effective for this code.

³A square matrix with integer elements whose determinant is ± 1 .

3.3 Optimization of the dynamic energy consumption in a 2-layer memory subsystem

In embedded communication and multimedia processing system, the on-chip memory is often complemented by an external (off-chip) memory for storing the large amounts of data during the execution of the application. The memory subsystem is typically a major contributor to the overall energy budget of the entire system. 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 [5]. Savings of dynamic energy (which expands only when memory accesses occur) at the level of the whole memory subsystem can be mainly obtained by accessing frequently used data from smaller on-chip memories rather than from large background (off-chip) memories. As on-chip storage, the scratch-pad memories (SPMs) – software-controlled static or dynamic random-access memories, more energy-efficient than caches – are widely used in embedded systems, in which the flexibility of caches in terms of workload adaptability is often unnecessary, whereas power consumption and cost play a much more critical role. Different from caches, the SPM occupies one distinct part of the virtual address space with the rest of the address space occupied by the main memory. The consequence is that there is no need to check for the availability of the data in the SPM. Hence, the SPM does not possess a comparator and the miss/hit acknowledging circuitry. This contributes to a significant energy (as well as area) reduction. Another consequence is that in cache memory systems, the mapping of data to the cache is done during run-time, whereas in SPM-based systems this can be done either manually by the designer, or automatically – by a compiler, using a suitable algorithm.

Different from previous works, our CAD tool identifies with precision those parts of arrays more intensely accessed [10]. These parts – represented as lattices – are assigned to the on-chip scratch-pad memory, achieving a reduction of the dynamic energy consumption due to memory accesses. Since the bounding window of any lattice can be computed using *Algorithm 2*, those lattices covering the most accessed elements of the arrays are mapped to the scratch-pad memory (SPM) and *Algorithm 3* can find their bounding windows. The same *Algorithm 3* will compute the bounding windows for the off-chip memory as well.

Algorithm 4: Hierarchical (two-layer) energy-aware memory allocation for a given array A in the algorithmic specification

```

for ( each disjoint lattice  $L \in \mathcal{L}_A$  )
  compute the total number of memory accesses to  $L$  ;
  assign to the SPM a subset of disjoint lattices  $\mathcal{S}_A \subset \mathcal{L}_A$ 
    having the highest average access values, such that
     $\sum_{L \in \mathcal{S}_A} size(L) \leq MaxSizeSPM$ ; // the total size of the
    lattices assigned to SPM should not exceed its maximum size
  compute using Algorithm 3 the bounding window  $W_A^{SPM}$ 
    of the lattices in the subset  $\mathcal{S}_A$ ; // Mapping solution for SPM
  compute using Algorithm 3 the bounding window  $W_A^{offchip}$ 
    of the lattices in  $\mathcal{L}_A - \mathcal{S}_A$ ; // Mapping for the off-chip mem.

```

Applying *Algorithm 4* for all the indexed signals in the algorithmic

specification, we obtain a 2-layer (SPM and off-chip) memory allocation and assignment solution for the entire application. Using the CACTI v4.2 power model [11], estimates of the dynamic energy consumption in the memory subsystem can be provided.

For the illustrative example in Fig. 1, a flat allocation solution requires 10,787 (off-chip) memory locations for storing signal A (Δ and $opt\Delta$ can be stored in datapath registers since they require 1 location each). Signal A has 3 disjoint lattices, each one covering 97 A -elements and receiving 425,572 memory *read* accesses (i.e., an average of 4387.34 accesses per array element). If these 3 lattices are stored in the SPM ($3 \times 97 = 291$ locations), the reduction of the dynamic energy consumption versus the 1-layer (off-chip) allocation is about 51.5% (relative to CACTI model), since the energy consumed per access by the SPM is 1-2 orders of magnitude smaller. The designer can use the tool to explore also other solutions imposing different size limits for the SPM.

4 Experimental results

A CAD system performing the memory management tasks referred in this paper has been implemented in C++ under a Linux OS. The input algorithmic specifications processed by the system are expressed in a subset of the C language.

Table 1 summarizes part of the results of our experiments. The benchmarks used are: (1) a motion detection algorithm ($M = N = 64, m = n = 4$) used in the transmission of real-time video signals on data networks [5]; (2) a real-time regularity detection algorithm used in robot vision; (3) Durbin's algorithm for solving Toeplitz systems (with $N = 500$ unknowns); (4) the kernel of a motion estimation algorithm for moving objects (MPEG-4); (5) a singular value decomposition (SVD) updating algorithm ($n = 100$) used in spatial division multiplex access (SDMA) modulation in mobile communication receivers, in beamforming, and Kalman filtering; (6) the kernel of a voice coding application – essential component of a mobile radio terminal. Columns 2 and 3 display the numbers of array references and array elements in the specification codes. Columns 4 and 5 show the minimum data memory sizes (optimal memory sharing), computed as explained in Section 3.1, and the corresponding CPU times (obtained on a PC with a 1.85 GHz Athlon XP processor and 512 MB memory). Columns 6 and 7 display the allocation solutions (the amounts of data memory *after* the signal-to-memory mapping, computed as explained in Section 3.2) and the corresponding CPU times. The last four columns show, respectively, the dynamic energy consumptions assuming only one (off-chip) memory layer; the SPM sizes and the savings of dynamic energy (when steering the SPM assignment based on the highest average access values of the lattices) versus the single-layer memory scenario; the CPU times.

Different from all the other signal-to-memory mapping techniques (see [7]), this computation methodology allows to determine the additional storage implied by the mapping model by computing the minimum window (storage requirement) of each individual array and the minimum data storage for the execution of the whole code, providing thus useful information for the ini-

| Application | #Array refs. | #Array elements | Min. data memory | CPU ₁ [sec] | Mem. after mapping | CPU ₂ [sec] | Dyn. energy 1-layer [μJ] | SPM size | Energy saved | CPU ₃ [sec] |
|----------------------|--------------|-----------------|------------------|------------------------|--------------------|------------------------|---------------------------------|----------|--------------|------------------------|
| Motion detection | 11 | 318,367 | 9,524 | 6 | 9,525 | 7 | 2,152 | 1,560 | 38.1% | 9 |
| Regularity detection | 19 | 4,752 | 572 | 1 | 657 | 3 | 353 | 1,024 | 68.2% | 1 |
| Durbin alg. | 27 | 252,499 | 1,249 | 5 | 1,702 | 7 | 3,588 | 764 | 73.2% | 12 |
| Motion estim. | 68 | 265,633 | 2,465 | 18 | 2,680 | 21 | 3,088 | 1,416 | 50.7% | 23 |
| SVD updating | 87 | 3,045,447 | 34,950 | 26 | 70,203 | 35 | 22,231 | 12,672 | 46.0% | 37 |
| Voice coder | 236 | 33,619 | 11,890 | 2 | 13,104 | 14 | 714 | 3,879 | 39.5% | 8 |

Table 1: Experimental results.

tial design phase of system-level exploration. By means of the signal-to-memory mapping model, an excess of storage in memory allocation is traded-off against a less complex address generation hardware (practically all the mapping models using modulo computations). By computing the individual minimum windows and the minimum data storage, this tool has the unique feature of yielding a *measure* of this compromise – the amount of extra storage *after mapping*.

Although our approach is based on a mapping strategy similar to [13], the computation methodology employed is entirely different. Tronçon *et al.* perform first a “liveness analysis”. During this phase, a set of program points is firstly decided. For instance, if the code contains n nested loops, $2n + 2$ program points are considered only for that nest. In contrast, our lattice-oriented methodology requires significantly fewer program points – only between loop nests. In addition, our computation of the window sides w_k is mostly based on lattice projections rather than on incremental adjustments and enumerations of the integer points in \mathbf{Z} -polyhedra. These are the reasons why our computation methodology is significantly faster and it has a better scalability in terms of number of array elements. (Due to lack of space, Table 1 displays only a part of our experimental results.)

Note that Brockmeyer *et al.* [4] perform the assignment of the multidimensional signals to the memory layers at the level of entire arrays. The quantitative measure of their assignment approach is the average number of accesses per array element. Our model clearly offers a higher flexibility, identifying those parts of arrays that are heavily accessed, which can be significantly smaller than the whole array space (only 291 locations out of an array space of $67 \times 161 = 10,787$, i.e. about 2.7% in the example from Fig. 1). This is why, although previous models (e.g., [4], [8]) produce important dynamic energy savings as well, our model led to 20%-33% better savings than them.

5 Conclusions

This paper has presented an integrated CAD methodology for system-level exploration, focused on memory management tasks for the design of multidimensional signal processing systems. All these tasks are efficiently addressed within a common algebraic framework.

References

- [1] D. Avis, “Irs: A revised implementation of the reverse search vertex enumeration algorithm,” in *Polytopes – Combinatorics and Computation*, G. Kalai (ed.), Birkhauser-Verlag, pp. 177-198, 2000.
- [2] F. Balasa, H. Zhu, I.I. Luican “Computation of storage requirements for multi-dimensional signal processing applications,” *IEEE Trans. VLSI Syst.*, vol. 14, 2007.
- [3] A.I. Barvinok and J. Pommersheim, “An algorithmic theory of lattice points in polyhedra,” in *New Perspectives in Algebraic Combinatorics*, Cambridge Univ. Press, 1999, pp. 91-147.
- [4] E. Brockmeyer, M. Miranda, H. Corporaal, and F. Catthoor, “Layer assignment techniques for low energy in multi-layered memory organisations,” in *Proc. 6th ACM/IEEE Design and Test in Europe Conf.*, Munich, Germany, March 2003, pp. 1070-1075.
- [5] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P. G. Kjeldsberg, T. Van Achteren, and T. Omes, *Data Access and Storage Management for Embedded Programmable Processors*, Boston: Kluwer Acad. Publ., 2002.
- [6] E. Chan and S. Panchanathan, “Motion estimation architecture for video compression,” *IEEE Trans. Consumer Electr.*, vol. 39, pp. 292-297, Aug. 1993.
- [7] A. Darte, R. Schreiber, and G. Villard, “Lattice-based memory allocation,” *IEEE Trans. Computers*, vol. 54, pp. 1242-1257, Oct. 2005.
- [8] Q. Hu, A. Vandecappelle, M. Palkovic, P.G. Kjeldsberg, E. Brockmeyer, F. Catthoor, “Hierarchical memory size estimation for loop fusion and loop shifting in data-dominated applications,” *Proc. Asia-South Pacific Design Automation Conf.*, Yokohama, Japan, Jan. 2006, pp. 606-611.
- [9] M. Kandemir, M.J. Irwin, G. Chen, and I. Kolcu, “Compiler-guided leakage optimization for banked scratch-pad memories,” *IEEE Trans. VLSI Systems*, vol. 13, no. 10, pp. 1136-1146, Oct. 2005.
- [10] I.I. Luican, H. Zhu, and F. Balasa, “Formal model of data reuse analysis for hierarchical memory organizations,” in *Proc. IEEE/ACM Int. Conf. Comp.-Aided Design*, San Jose CA, 2006.
- [11] G. Reinman and N. Jouppi, “CACTI: An integrated cache timing and power model,” COMPAQ Western Research Lab, Palo Alto, 1999.
- [12] A. Schrijver, *Theory of Linear and Integer Programming*, New York: John Wiley, 1986.
- [13] R. Tronçon, M. Bruynooghe, G. Janssens, and F. Catthoor, “Storage size reduction by in-place mapping of arrays,” in *Verification, Model Checking and Abstract Interpretation*, A. Coresi (ed.), 2002, pp. 167-181.
- [14] S. Verdoolaege, K. Beyls, M. Bruynooghe, and F. Catthoor, “Experiences with enumeration of integer projections of parametric polytopes,” in *Compiler Construction: 14th Int. Conf.*, R. Bodik (ed.), vol. 3443, Springer, 2005, pp. 91-105.