Balasa *et al. EURASIP Journal on Embedded Systems* (2017) 2017:6
DOI 10.1186/s13639-016-0043-9

EURASIP Journal on Embedded Systems

RESEARCH Open Access

Energy-aware memory management for embedded multidimensional signal processing applications

Florin Balasa[1*], Noha Abuaesh[1], Cristian V. Gingu[2], Ilie I. Luican[3] and Hongwei Zhu[4]

Abstract

In real-time data-intensive multimedia processing applications, data transfer and storage significantly influence, if not dominate, all the major cost parameters of the design space—namely energy consumption, performance, and chip area. This paper presents an electronic design automation (EDA) methodology for the high-level design of hierarchical memory architectures in embedded data-intensive applications, mainly in the area of multidimensional signal processing. Different from the previous works, the problems of data assignment to the memory layers, of mapping the signals into the physical memories, and of banking the on-chip memory are addressed in a consistent way, based on the same formal model. This memory management framework employs techniques specific to the integral polyhedra based dependence analysis. The main design target is the reduction of the static and dynamic energy consumption in the hierarchical memory subsystem.

Keywords: Memory management, Multidimensional signals, Signal-to-memory mapping, Scratch-pad memory banking, Polytopes and lattices

1 Introduction

In embedded real-time communication and multimedia processing applications, the manipulation of large data sets has a major effect on both power consumption and performance of the system. Due to the significant amount of data transfers between the processing units and the large and energy consuming off-chip memories, these applications are often called *data-dominated* or *data-intensive* [1].

At system level, the power cost can be reduced (and, at the same time, the system performance enhanced) by introducing an optimized custom memory hierarchy [2]. Hierarchical memory organizations reduce energy consumption by assigning the frequently-accessed data to the low hierarchy levels [3], diminishing the *dynamic* energy consumption—which expands due to memory accesses. Moreover, it reduces the *static* energy consumption as well, since this decreases monotonically with the memory size [4].

*Correspondence: fbalasa@aucegypt.edu
[1] Department of Computer Science and Engineering, American University in Cairo, Cairo, Egypt
Full list of author information is available at the end of the article

Within a given memory hierarchy level, power can be reduced by memory partitioning—which principle is to divide the address space in several smaller blocks, and to map these blocks to physical memory banks that can be independently enabled and disabled [5, 6].

The most typical implementation of memory hierarchies makes use of caches. While extremely versatile and fast, caches are not always the best choice in embedded systems. As on-chip storage, the scratch-pad memories (SPMs)—compiler-controlled synchronous random-access memories (SRAMs), more energy-efficient than the hardware-managed caches—are widely used in embedded systems, where caches incur a significant penalty in aspects like area cost, energy consumption, hit latency, and real-time predictability [3].

The SPMs are quite similar to caches in terms of size and speed (typically, one-cycle access time), but without dedicated logic for dynamic swapping of contents with the main memory. Instead, it is the designer's responsibility to explicitly map addresses of external memory to locations of the SPM. While impractical in general-purpose architectures, this process becomes feasible in embedded systems, where designers usually have fine control on

both the software and underlying hardware, and are able to optimally match them. Adding SPMs at the hardware level is not difficult: they usually require an SRAM array and decoders to split the SPM accesses from the external memory (typically, a DRAM) accesses, as shown in Fig. 1.

The research on the assignment of *signals* (multidimensional arrays) to the memory layers [7] focused in part on how to restructure the application code to make better use of the available memory hierarchy [8]. Brockmeyer et al. used the steering heuristic of assigning the arrays having the highest access number over size ratio to the cheapest memory layer, followed by incremental reassignments [9]. Their model takes into account the relative lifetime differences between arrays and between the scalars covered by each array. However, their model operates on entire arrays, not taking into account that the access patterns are, in general, not uniform.

There are rather few research works addressing the problem of signal mapping to the physical memory. De Greef et al. mapped each multidimensional array from the behavioral specification by choosing the canonical linearization which yielded the minimum distance (in memory words) between array elements simultaneously alive [10].

Instead of a linear mapping, Tronçon et al. proposed to compute an m-dimensional mapping window for each m-dimensional array [11]: the sides of a window were computed based on the maximal index difference in each dimension between array elements simultaneously alive. (The bounding-window mapping is also used in *PPCG*—a source-to-source compiler using polyhedral compilation techniques that extracts data-parallelism with a code generator for a modern graphics processing unit [12]). Darte et al. proposed a lattice-based mathematical framework for array mapping, establishing a correspondence between valid linear storage allocations and integer lattices called *strictly admissible* [13].

Partitioning of on-chip memories has been analyzed by several research teams, being typically used as an additional dimension of the memory design space. Shiue and Chakrabarti studied power-efficient partitioned cache organizations, identifying cache sub-banking as an effective approach to reduce cache power consumption [14]. Benini et al. proposed a recursive partitioning of the SPM address space, which achieved a complete exploration of the banking solutions [5]. In [6], the cost function was shown to exhibit properties that allow to apply a dynamic programming paradigm. A leakage-aware approach, based on traces of memory accesses, takes into account that putting a memory block into the dormant state should be done only if the cost of energy overhead and decrease of performance can be amortized [15, 16].

The advances in data-dependence analysis [17] and optimizing compilers [18] have influenced the development of memory management techniques based on the processing and restructuring of behavioral specifications. Ramanujam et al. use data dependence and data reuse to estimate the minimum amount of memory in signal processing codes and, then, reduce the storage requirement through loop-level transformations [19]. However, their approach focuses only on nested loops, and the window sizes for the arrays are determined using only a single linearization—the one induced by the variation of the iterators in the nested loop. De La Luz et al. present a strategy of increasing the memory bank idleness by modifying the execution order of loop iterations [20]. The number of banks and their sizes seem predetermined, though, and it is not clear what happens when the arrays are large, exceeding the size of the banks.

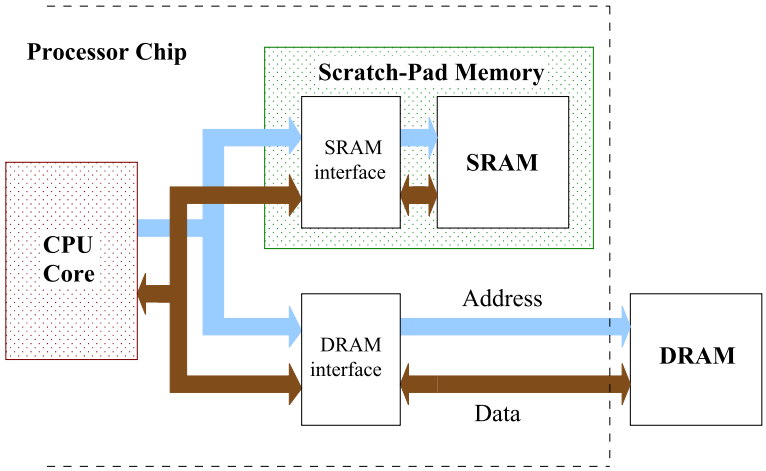

Fig. 1 Simple hierarchical memory architecture with an on-chip SPM and an off-chip DRAM

This paper presents an energy-aware EDA methodology (see Fig. 2) for the high-level design of hierarchical memory architectures in embedded data-intensive applications in the domain of multidimensional signal processing. Different from the previous works, which typically address one memory management task at a time, *three* memory management problems—the data assignment to the memory layers, the mapping of signals to the physical memories, and the banking of the on-chip memory—are addressed in a consistent way, based on the same formal model. This memory management framework employs techniques specific to the integral polyhedra based dependence analysis [21]. The main target is the reduction of the static and dynamic energy consumption in the hierarchical memory subsystem of embedded systems (note that several research works on parallelizing and optimizing compilers—like, for instance, [18]—focused mainly on data-flow optimizations and high-level transformations to improve parallelism and *memory hierarchy performance*: improving the

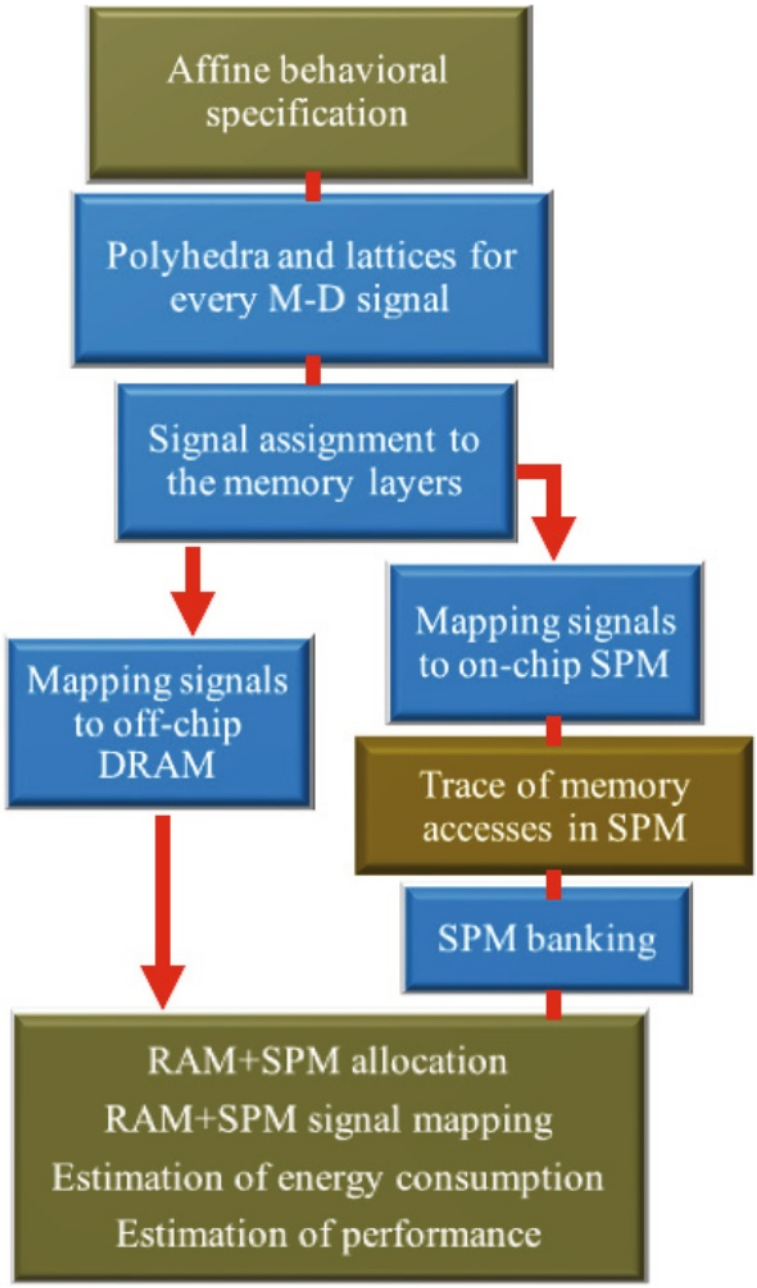

Fig. 2 Flowchart of a memory management system for data-intensive signal processing applications

performance of storage organizations is a target orthogonal to ours).

In contrast to previous EDA works, the advantageous characteristics of the three main tasks are as follows.

- *The data assignment to the memory layers identifies the intensely-accessed parts of the multidimensional arrays*, steering these parts to the energy-efficient storage layer. Such a strategy is thus independent of the number and size of the arrays (being dependent only on the size of the energy-efficient layer) and entails a significant reduction of energy consumption in the memory subsystem.
- *The signal-to-memory mapping is designed to work in hierarchical memory organizations*, being able to operate with *parts* of arrays (rather than entire arrays). *It can provide mapping functions useful for the design of the address generation units* and *it can evaluate metrics of quality*, like the minimum storage requirement of the behavioral specification.
- *Two memory banking techniques are implemented* in our framework: they further reduce the memory energy consumption, *computing very fast near-optimal banking solutions even when the memory address space is large.*

The main input of this memory management framework is the behavioral specification of a data-intensive application. Such a specification is described in a high-level programming language, where the code is typically organized in sequences of loop nests. The loop boundaries are linear functions of the outer loop iterators. The data structures are multidimensional arrays—a characteristic of data-intensive applications [1]; the indices of the array references are linear functions of the loop iterators. The logical expressions in conditional instructions can be either simple or compound. The behavioral specifications describe the processing of *streams* of data samples: different from computer programs, these specifications can be imagined as surrounded by an implicit loop having *time* as iterator. This is why the code can contain *delayed* signals, i.e., signals produced (or inputs) in a previous data processing, which are used as operands during the current execution of the code (for instance, $A[i][j]$ @3 means an array reference produced three *time* iterations in the past).

The specifications supported by this framework are *procedural*[1] and *non-parametric* (for illustration, see the code examples in the text). Our memory management model allows the exploration of various functionally equivalent behavioral specifications by computing the minimum data storage [22] and generating the graph of storage variation during the code execution (see Fig. 3).

The rest of the paper is organized as follows. Section 2 discusses the problem of energy-aware signal assignment based on a case study. Section 3 presents the formal model of the methodology and the algorithm for data assignment to the memory layers. Section 4 describes a storage-efficient mapping approach of multidimensional arrays to the physical memories. Section 5 presents the algorithm for partitioning the on-chip SPMs. Section 6 discusses implementation aspects and presents experimental results. Finally, Section 7 summarizes the main conclusions of this research.

a

```
                                    // All the array elements 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];
  }
```

b

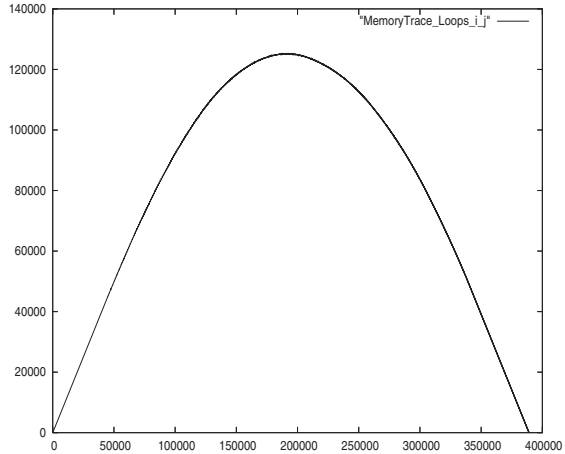

Fig. 3 a Algorithmic specification and **b** the graph of variation of the storage requirement for the signals *A*, *B*, *C*, *D*, and *E*. The abscissae are the numbers of assignment instructions executed at run time; the ordinates are the memory locations (or words, whose width in bytes is known) in use. The minimum data storage is 125,193 locations; interchanging the two loops, the storage requirement significantly decreases to only 576 locations

2 Signal assignment to the memory layers: a case study

Let us consider the illustrative code example in Fig. 4, and assume that each element of the two-dimensional (2-D) array A can be stored in 1 byte (hence, the whole array has a storage requirement of 64 Kbytes). The array is not uniformly accessed during the code execution. Figure 5

```
C[0] = 0 ;                          // int  A[256][256]: input;
for ( i=64; i<192; i++ )
{   for ( j=64; j<192; j++ )
    {  B[i][j][0] = 0 ;
       for ( k=i-64; k<=i+64; k++ )
          for ( l=j-64; l<=j+64; l++ )
             B[i][j][129*k-129*i+l-j+8321] = A[i][j] - A[k][l]
                                  +  B[i][j][129*k-129*i+l-j+8320] ;
       C[128*i+j-8255] = B[i][j][16641] + C[128*i+j-8256] ;
    }
}
opt = C[16384];
```

Fig. 4 Code example of behavioral specification

displays the intensity of *read* accesses of the A-elements: for each pair of possible indexes—between 0 and 255—of the A's elements (the horizontal plane xOy), the number of memory accesses was recorded on the vertical axis Oz. One can see the elements near the center of the array space are accessed with high intensity (for instance, $A[128][128]$ is accessed 33,025 times), whereas the elements at the periphery of the array space are accessed with a significantly lower intensity (for instance, the elements in the four corners of the array space $A[0][0]$, $A[0][255]$, $A[255][0]$, and $A[255][255]$ are accessed only once).

Brockmeyer et al. proposed to assign the arrays having the highest access number over size ratio to the on-chip memory layer [9]. Certainly, the array A has a high access ratio—equal to 8320.5 (since there are 545,292,288 accesses to 65,536 array elements): quite obviously, the most desirable scenario – in point of view of both energy consumption and performance—is to store all the signals from the behavioral specification onto the SPM memory layer or, at least, the entire array A. This is usually not possible: quite often, the size of the on-chip memory is a design constraint, usually small relative to the storage requirement of the entire code.

Not only that arrays from the behavioral specification may have storage requirements greater than the SPM size, but also their possible non-uniform pattern of accesses is not taken into account by this past assignment approach [9]. Hu et al. can use *parts* of arrays in the assignment to the memory layers [23]: their illustrative example suggests cuts along one of the array dimensions as the main partitioning heuristic. If this is the case, the approach has a similar shortcoming—the pattern of accesses may have significant variations along these cuts. For instance, in our test case, the A-elements of the row 128 have a range of variation between 128 (for $A[128][0]$) and 33,025 accesses (for $A[128][128]$), with an abrupt increase from 8192 to 24,961 accesses for the neighbor elements $A[128][63]$ and $A[128][64]$ (see Fig. 5).

In data-intensive signal processing applications, the main data structures from the behavioral specification are multi-dimensional arrays. The problem is how to identify the intensely-accessed parts of arrays based on the analysis of the application code, in order to steer their assignment to the energy-efficient data storage layer– the on-chip SPM. Note that a simulated execution of the behavioral specification may be computationally expensive (e.g. when the number of array elements is very large, or when the application code contains deep loop nests); at the same time, such a scalar-oriented technique yields assignment results that cannot be directly used for the design of the address generation units [24].

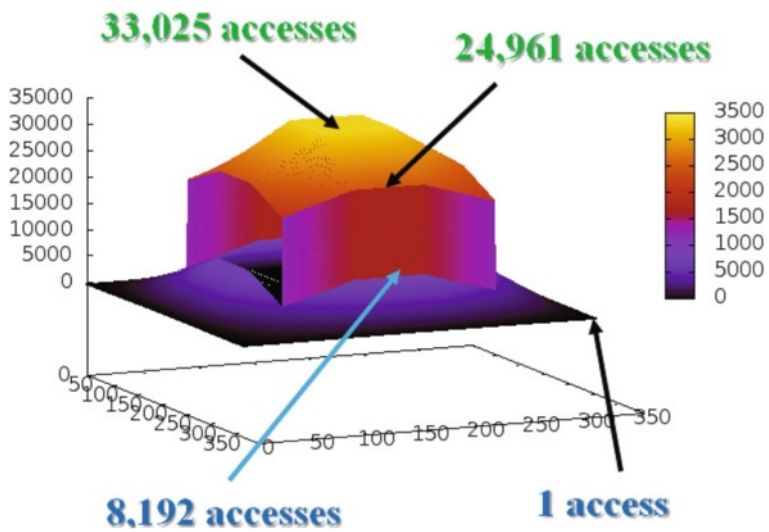

Fig. 5 3-D map of memory accesses to the array A from the code in Fig. 4. A's index (array) space is in the horizontal plane xOy and the numbers of memory accesses are on the vertical axis Oz

An assignment algorithm mapping the array elements from the behavioral specification to the memory layers, targeting the reduction of the energy consumption in the hierarchical memory subsystem, will be described in the next section.

3 Data assignment to the memory layers for the reduction of energy consumption in the hierarchical memory subsystem

Our technique is based on a simple observation: the most intensely-accessed parts of the array space of a multidimensional signal are typically covered by more than one array reference. Actually, in many cases, the more array references cover a certain element, the more accessed that element is. For instance, the most heavily accessed parts of array A (see Fig. 5) from the code in Fig. 4 are the A-elements belonging to both array references $A[i][j]$ and $A[k][l]$. Of course the intensity of memory accesses to them is not uniform but, nevertheless, they are *read* more often than the other A-elements. In order to find out which are the array elements belonging to several array references, we must *intersect* the array references of the signal. This operation is done based on an algebraic model whose principle is briefly explain below.

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 $\boldsymbol{i} = (i_1,\ldots,i_n) \in \boldsymbol{Z}^n$ in the scope of the array reference. The index space is the set of all index vectors $\boldsymbol{x} = (x_1, \ldots, x_m) \in \mathbf{Z}^m$ of the array reference. When the indices of an array reference are linear expressions with integer coefficients of the loop iterators, the index space consists of one or several *linearly bounded lattices* (LBLs) [25]:

$$\left\{\boldsymbol{x} = \boldsymbol{T}\cdot\boldsymbol{i}+\boldsymbol{u} \in \boldsymbol{Z}^m \mid \boldsymbol{A}\cdot\boldsymbol{i} \geq \boldsymbol{b},\ \boldsymbol{i}\in\boldsymbol{Z}^n\right\}$$

where $\boldsymbol{x} \in \boldsymbol{Z}^m$ is an index vector of the m-dimensional signal and $\boldsymbol{i} \in \boldsymbol{Z}^n$ is an n-dimensional iterator vector.

Example 1: In Fig. 4, $B[i][j][129{*}k-129{*}i+l-j+8321]$ is an array reference that can be represented by the lattice:

$$\left\{\begin{bmatrix}x\\y\\z\end{bmatrix} = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ -129 & -1 & 129 & 1\end{bmatrix}\begin{bmatrix}i\\j\\k\\l\end{bmatrix} + \begin{bmatrix}0\\0\\8321\end{bmatrix}\right\},$$

where x, y, and z are the $m = 3$ indexes of the array reference; the $n = 4$ iterators satisfy the inequalities:

$$64 \leq i, j \leq 191,\ i-64 \leq k \leq i+64,\ j-64 \leq l \leq j+64.$$

Let $\mathcal{L}_1 = \{\boldsymbol{x} = \boldsymbol{T}_1\boldsymbol{i}_1 + \boldsymbol{u}_1 \mid \boldsymbol{A}_1\boldsymbol{i}_1 \geq \boldsymbol{b}_1\}$, $\mathcal{L}_2 = \{\boldsymbol{x} = \boldsymbol{T}_2\boldsymbol{i}_2 + \boldsymbol{u}_2 \mid \boldsymbol{A}_2\boldsymbol{i}_2 \geq \boldsymbol{b}_2\}$ be two LBLs derived from the same indexed signal, where $\mathbf{T_1}$ and $\mathbf{T_2}$ have obviously the same number of rows—the signal dimension m. Intersecting the two linearly bounded lattices means, first of all, solving a linear Diophantine system (that is, finding the integer solutions of a linear system with integer coefficients[2]) [26]

$$\boldsymbol{T}_1\boldsymbol{i}_1 - \boldsymbol{T}_2\boldsymbol{i}_2 = \boldsymbol{u}_2 - \boldsymbol{u}_1$$

having the elements of $\mathbf{i_1}$ and $\mathbf{i_2}$ as unknowns. If the system has no solution, then $\mathcal{L}_1 \cap \mathcal{L}_2 = \emptyset$. Otherwise, the solution of the Diophantine system has the form:

$$\begin{bmatrix}\mathbf{i_1}\\ \mathbf{i_2}\end{bmatrix} = \begin{bmatrix}\mathbf{V_1}\\ \mathbf{V_2}\end{bmatrix}\boldsymbol{t} + \begin{bmatrix}\mathbf{v_1}\\ \mathbf{v_2}\end{bmatrix},\ \boldsymbol{t}\in\boldsymbol{Z}^p$$

Replacing $\mathbf{i}_1$ and $\mathbf{i}_2$ in the sets of constraints $\boldsymbol{A}_1\boldsymbol{i}_1 \geq \boldsymbol{b}_1$ and $\boldsymbol{A}_2\boldsymbol{i}_2 \geq \boldsymbol{b}_2$ of the two LBLs, we obtain the set of inequalities:

$$\mathbf{A_1V_1}\cdot\mathbf{t} \geq \mathbf{b_1} - \mathbf{A_1v_1},\ \mathbf{A_2V_2}\cdot\mathbf{t} \geq \mathbf{b_2} - \mathbf{A_2v_2} \quad (1)$$

If (1) has integer solutions, then the intersection is a new LBL:

$$\mathcal{L}_1 \cap \mathcal{L}_2 = \left\{\boldsymbol{x} = \mathbf{T_1V_1}\cdot\mathbf{t} + (\mathbf{T_1v_1} + \mathbf{u_1}) \mid s.t.\ (1),\ \mathbf{t}\in\mathbf{Z}^p\right\}$$

Note that, in geometrical point of view, the set of inequalities (1) represents an integral polytope—a multidimensional polyhedron bounded and closed, restricted to the points having integer coordinates. Checking the existence of integer solutions of a linear system of inequalities is a well-known problem [27].

The intersection of lattices described above can be used to decompose the array space of every multidimensional signal from the application code into disjoint lattices and, also, to compute the number of accesses to the array elements in every partition. A high-level pseudo-code is given below: the decomposition is obtained by recursively intersecting all the array references of a selected signal. The structure of each array reference in terms of component lattices is determined by gradually building a directed acyclic graph (DAG)—each node representing an LBL and each arc denoting an inclusion relation between the respective sets. Initially, this DAG is just a set of nodes, one per each array reference in the code. Gradually, new nodes emerge due to intersections between lattices, and arcs (inclusions) are added between the nodes.

Algorithm 1 Partitioning the index space of a multidimensional array/signal into disjoint lattices

initialize the set of LBLs of the selected signal
with the lattice representing some array reference $\mathcal{A}_0$*;*
for (*all the array references of the given signal*) {
select a reference $\mathcal{A}_1 \neq \mathcal{A}_0$ *and let* $\mathcal{L}_1$ *be its representation;*
for (*the set of LBLs of the signal*) {
select an LBL from the set, let it be called $\mathcal{L}_2$ *;*
compute $\mathcal{L}_1 \cap \mathcal{L}_2$ *;*
if (*the intersection is not empty*)
then *compute* $\mathcal{L}_1 - \mathcal{L}_2$ *and* $\mathcal{L}_2 - \mathcal{L}_1$ *;*
update the set of LBLs and the DAG of lattices;
repeat the above operations till no new LBL is created;
}
}

The *difference* of lattices is a more complex operation, described, for instance, in [22]—where it is used to compute the minimum data storage of behavioral specifications.

Figure 6 shows the resulting directed acyclic graph for the 2-D array A from our illustrative example in Fig. 4. The nodes in the graph are annotated with the amount of storage required by the lattices they represent. The bold nodes without incident arcs denote the nine disjoint lattices partitioning the array space, as displayed in Fig. 7. The number of *read/write* accesses is indicated below these nodes whose names correspond to the partitions in Fig. 7 (where **L** means *Left*, **R** means *Right*, **B** stands for *Bottom* and **T** for *Top*, and **M** means *Middle*). For example, the number of memory accesses to the middle region **M**, as part of the array reference $A[k]\,[l]$, is the size of the LBL $\{i = t_1,\ j = t_2,\ k = t_3,\ l = t_4 \mid 191 \geq t_1,\ t_2,\ t_3,\ t_4 \geq 64,\ t_1 + 64 \geq t_3 \geq t_1 - 64,\ t_2 + 64 \geq t_4 \geq t_2 - 64\}$, which is 152,571,904 [22]. On the other hand, **M** is also included in the array reference $A[i]\,[j]$: the number of memory accesses is the size of the LBL $\{i = t_1,\ j = t_2,\ k = t_3,\ l = t_4 \mid 191 \geq t_1,\ t_2 \geq 64,\ t_1 + 64 \geq t_3 \geq t_1 - 64,\ t_2 + 64 \geq t_4 \geq t_2 - 64\}$, which is 272,646,144 . Hence, the total amount of memory accesses to partition **M** is 152,571,904 + 272,646,144 = 425,218,048.

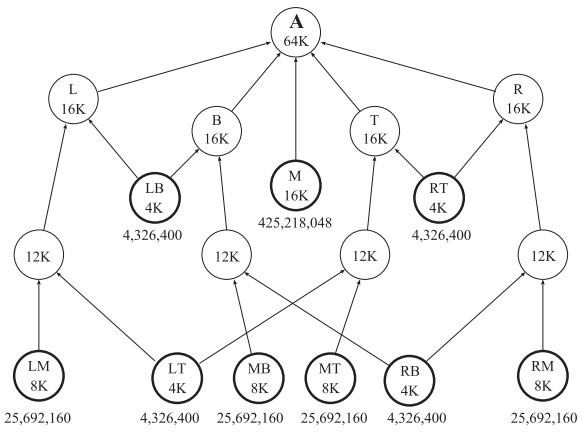

Fig. 6 Directed acyclic graph (DAG) showing the inclusions between the linearly bounded lattices partitioning the array space of signal A

The benefit of the decomposition of the array space for each signal is that it yields access information for steering the signal assignment to the memory layers. The obvious candidates for being stored on-chip are the regions of the array space (LBLs) having the highest ratios between the number of array accesses and their size. Note that Brockmeyer et al. considered similar ratios but at the level of whole arrays [9], whereas our approach localizes the regions heavily accessed in the array space and applies the ratios at the level of these regions. In our illustrative example, the middle region **M** has the highest ratio: 425,218,048 / 16,384 = 25,953.25 (that is, an average of almost 26 thousand accesses per array element). We are using an even more precise metric—the savings of energy, as a percentage, when the lattice is stored on-chip, rather than onto the external DRAM. According to this metric, the *energy benefit* of lattice **M** is 60.88 %—computation will be explained below.

Therefore, storing on-chip the elements in the center of signal A's array space would maximize the benefit in term of energy reduction. However, this central region **M** of the array space requires 16 Kbytes: what is to be done if there is a design constraint limiting the SPM storage to less than 16 Kbytes?

To explain the idea of our assignment algorithm when the SPM size is a design constraint, we shall use again the illustrative example in Fig. 4. The three inner loops are executed for each value of the outer loop iterator i from 64 to 191. If the outer loop were unrolled, then *Algorithm 1* partitioning the array space would yield the 128 lattices partitioning **M**, as displayed in Fig. 8, instead of the linearly bounded lattice representing the partition **M** from Fig. 6. Actually, there is no need to perform any modification of the behavioral specification: the smaller lattices can be obtained by "slicing" the lattice representing **M** for the different values of the first iterator. The partitioning can be continued until a finer level of granularity is reached.

Example 2: The lattice from *Example 1* of the array reference $B[\,i]\,[\,j]\,[\,129 * k - 129 * i + l - j + 8321]$ can be "sliced" (split) into 128 disjoint finer lattices for each value of $i = 64, \ldots, 191$. The first and the last of these 128 lattices are shown below:

$$\left\{ \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ -1 & 129 & 1 \end{bmatrix} \begin{bmatrix} j \\ k \\ l \end{bmatrix} + \begin{bmatrix} 64 \\ 0 \\ 65 \end{bmatrix} \right\},$$

where the iterators j, k, and l satisfy the inequalities:

$$64 \leq j \leq 191\ ,\ 0 \leq k \leq 128\ ,\ j - 64 \leq l \leq j + 64 \text{ and}$$

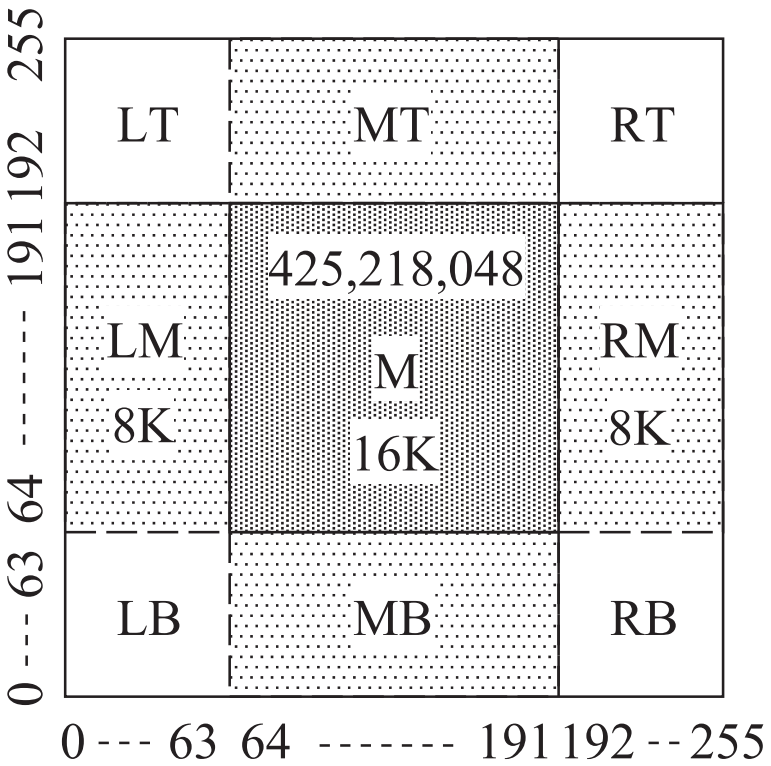

Fig. 7 Partitioning of the array space of signal *A* steered by the intensity of memory accesses: the *darker* the partition, the more it is accessed. The *bold* nodes in the DAG from Fig. 6 represent the nine partitions

$$\left\{ \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ -1 & 129 & 1 \end{bmatrix} \begin{bmatrix} j \\ k \\ l \end{bmatrix} + \begin{bmatrix} 191 \\ 0 \\ -16318 \end{bmatrix} \right\},$$

where the iterators j, k, and l satisfy the inequalities:

$$64 \le j \le 191\ ,\ 127 \le k \le 255\ ,\ j - 64 \le l \le j + 64.$$

□

The number of memory accesses for each of the A's smaller 128 lattices (see Fig. 8) can be also computed: then,

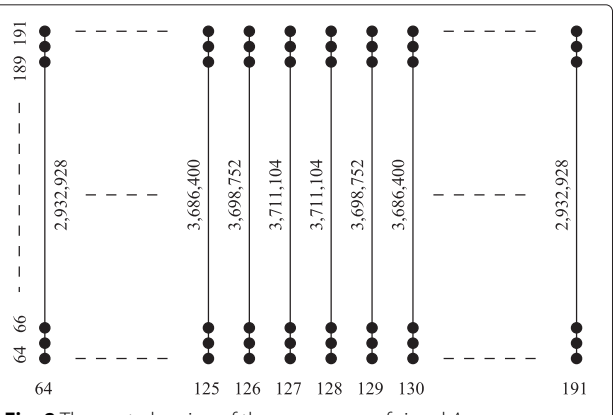

Fig. 8 The central region of the array space of signal *A*

the ratios between these numbers and the lattice sizes (of 128 bytes each) decrease from 28,993 (for lattices 127 and 128) to 22,913.5—for the two lateral ones, 64 and 191. Hence, it is more beneficial to store in the SPM the lattices going from the middle to the periphery of the central region of the array space.

The data assignment tool can be used to explore the impact on energy consumption by various storage distributions between the memory layers. Figure 9 displays the graphs of the energy consumption (both static and dynamic) by the SPM, by the DRAM, and overall when the SPM increases from 0 to 12 Kbytes, while the external DRAM decreases at the same time from 64 to 52 Kbytes. These graphs were obtained using CACTI 6.5 [28] for a technology of 32 *nm*.[3] When the entire signal A is stored onto the DRAM, the energy consumption is 87,947.8 μJ. However, when the DRAM is 56 Kbytes and the SPM is 8 Kbytes, the 64 central lattices of A (numbered 96–159 in Fig. 8) being stored onto the SPM, the energy consumption of the DRAM decreases significantly to 52,301.1 μJ, while the energy consumption of the SPM increases from 0 to only 3,063.6 μJ. The energy benefit for this scenario is

$$\left(1 - \frac{52,301.1 + 3,063.6}{87,947.8}\right) \times 100\ [\,\%] = 37.05\,\%$$

Algorithm 2 Energy-aware data assignment (multidimensional arrays/signals) to the off- and on-chip memory layers

for (*all the multidimensional arrays in the application code*)
  partition its index space in lattices using Algorithm 1;
initially, assign the disjoint lattices of each array to DRAM:
  size (DRAM) = $\sum_{\mathcal{L}}$ *size* ($\mathcal{L}$); *size (SPM) = 0;*
compute the energy benefit of each linearly bounded lattice;
do {
  select the lattice $\mathcal{L}$ *having the highest energy benefit;*
  if (*size* (*SPM*) + *size* ($\mathcal{L}$) $\leq$ *MAX_SPM_SIZE*) {
    assign $\mathcal{L}$ *to the SPM: size (SPM) += size* ($\mathcal{L}$);
    size (DRAM) -= size ($\mathcal{L}$);
    update the benefits of the lattices assigned to DRAM;
  }
  else { *"slice" the lattice* $\mathcal{L}$ *relative to its first iterator;*
    compute the benefits of these finer-granularity lattices;
  }
} **until** (*MAX_SPM_SIZE is reached, OR*
    the maximum level of "slicing" is reached);

Algorithm 2 has, typically, a useful side effect: a decrease of the total access time to the physical memories. The data assignment tool can be also used to explore the access times for various storage distributions. Figure 10 displays the graphs of the total access time to the SPM, to the DRAM, and to the two-layer data memory when the SPM increases from 0 to, say, 8 Kbytes, while the external DRAM decreases at the same time from 64 to 56 Kbytes. These graphs were also obtained using CACTI 6.5 [28] for a technology of 32 *nm*.

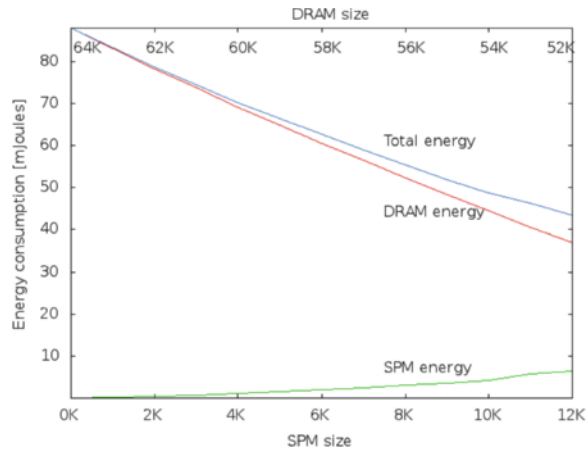

Fig. 9 The graphs of the energy consumption by the SPM, by the DRAM, and by the two-layer data memory storing the array *A* (from the code in Fig. 4) whose footprint is 64 Kbytes. The lower horizontal axis shows the increasing size of the on-chip SPM, and the upper horizontal axis displays the decreasing size of the off-chip DRAM

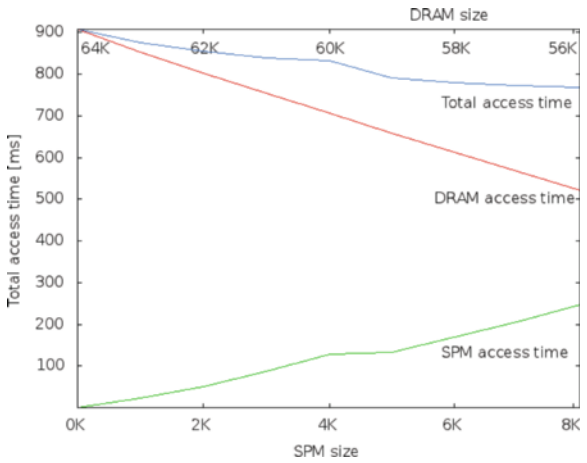

Fig. 10 The graphs of total access time to the SPM, to the DRAM, and to the two-layer data memory storing the array *A* (from the code in Fig. 4) whose footprint is 64 Kbytes. The lower horizontal axis shows the increasing size of the on-chip SPM, and the upper horizontal axis displays the decreasing size of the off-chip DRAM

When the entire signal *A* is stored in the DRAM, the total access time, according to CACTI 6.5, is 907.55 ms; however, when the DRAM is 56 Kbytes and the SPM is 8 Kbytes, the 64 central lattices of *A* (numbered 96–159 in Fig. 8) being stored in the SPM, the DRAM access time decreases to 521.16 ms on account of an increase of the SPM access time to 246.48 ms. Hence, the *time benefit* for this scenario is

$$\left(1 - \frac{521.16 + 246.48}{907.55}\right) \times 100\,[\,\%] = 15.42\,\%$$

When comparing time and energy per access in a memory hierarchy, it may be observed that these two metrics have often similar behavior; namely, they both increase as we move from low to high hierarchy levels. While it sometimes happens that a low-latency memory architecture is also a low-power one, optimizing memory performance does not imply power optimization or vice-versa [14] (although architectural solutions originally devised for performance optimization can be beneficial in terms of energy consumption, as well). There are two basic reasons for this: first, energy consumption and performance do not increase in the same way with memory size and hierarchy level; second, performance is a *worst-case metric*, while power is an *average-case metric*: for instance, the removal of a critical computation that improves performance may be harmful in terms of power consumption.

Algorithm 2 could be extended to an arbitrary number of memory layers if the functions of energy per access and static power versus memory size were available for

each layer. Assuming these functions increase monotonically with the memory size for each layer, and that the value intervals of these functions are disjoint and increase with the hierarchy level, the algorithm can be modified to assign the lattices of larger benefits starting from the lowest level and gradually moving to the higher levels of hierarchy. Our current implementation is dependent on the limitations of CACTI 6.5—the analytical tool used to provide memory information [28].

4 Mapping signals into the physical memory

This design phase decides the memory addresses of the signals from the behavioral specification. The signal-to-memory mapping has the following goals: (a) to map the signals (already assigned to the memory layers) into amounts of data storage as small as possible; (b) to guarantee that scalar signals (array elements) *simultaneously alive* are mapped to distinct storage locations; and (c) to use mapping functions simple enough in order to ensure an address generation hardware of a reasonable complexity.

Different from the previous works [10, 11, 13], this mapping technique is designed to work in hierarchical memory organizations, since it operates with *parts* of arrays (represented by mutually disjoint lattices) that can be assigned to different physical memories. The polyhedral framework, common to all the design phases in our system (data assignment to the memory layers, signal/array mapping onto the external memory and the SPM, followed by the banking of the latter), entails a high computation efficiency since all the phases rely on similar polyhedral operations. We present below the basic ideas of the mapping approach.

For an m-dimensional array, there are $m!$ orderings of the indices. For instance, a 2-D array can be typically linearized concatenating the rows, or concatenating the columns. In addition, the elements in a given dimension can be mapped in the increasing or decreasing order of the respective index. All these $2^m \cdot m!$ possible linearizations are called *canonical* [10]. For any canonical linearization, we compute for every linearly bounded lattice the largest distance (in memory words) between *any two live lattice elements* during the code execution. Based on these results, we compute—for every canonical linearization—the largest distance between *any two live array elements* at any time during the code execution.[4] This distance plus 1 is then the size of the storage *window* required for the mapping of the array into the data memory. More formally, $|W_A| = \min \max \{ dist(A_i, A_j) \} + 1$, where $|W_A|$ is the size of the storage window of a signal A, the minimum is taken over all the canonical linearizations, while the maximum is taken over all the pairs of A-elements (A_i, A_j) simultaneously alive. Even when *parts* of the array are stored in the SPM and the rest of it in the off-chip memory, the sizes of the storage windows can still be computed, since the assignment of data to the memory layers is done at the level of lattices (as explained in Section 3).

Example 3: The mapping model will be illustrated for the loop nest in Fig. 11a. The graph in Fig. 11b represents the *array space* (or *index space*) of the 2-D signal A, that is, the values of the indexes of the array reference $A[i][j]$. Each black point represents the index vector of an A-element $A[i][j]$ which is *produced* (that is, assigned a value) in the loop nest. Assuming these A-elements will be used as operands in a subsequent code, the storage requirement of the loop nest is 38 memory words. However, a *minimum* physical memory window is difficult to use in practical memory management problems: in most of the cases, it would require a significantly complex memory addressing hardware. A signal-to-memory mapping model must trade off an excess of data storage against a less complex address generation unit (AGU), most AGUs needing to compute additions, multiplications, and modulo operations [24]. For instance, a memory window W_A of 50 successive locations (relative to some base address) is sufficient to store the array reference $A[i][j]$ without mapping conflicts between elements simultaneously alive: it suffices that any read/write access to $A[i][j]$ be redirected to the memory word $W_A[(10 * j + i) \bmod 50]$, or to $W_A[(5 * i + j) \bmod 50]$ (since the integer projections [29] of the index space on the two axes are 10 and 5).

By analyzing the canonical linearizations, we try to reduce the memory window even more. This analysis is based on the evaluation of the distance between the minimum and maximum index vectors, relative to the lexicographic order, in a minimal bounding window of the index space (the computation steps being described and illustrated in [30]). In Fig. 11b, these minimum and maximum index vectors are represented by the points M and N, and the distance between them is $dist(M, N) = (11 - 2) \times 5 + (7 - 3) = 49$. Assuming that all the array elements within a linearly bounded lattice are alive, in a canonical linearization, the maximum distance in words between the array elements is the distance between the (lexicographically) minimum and maximum index vectors, providing an index permutation is applied first (in particular, an index interchange for 2-D signals). If in the canonical linearization some dimension is traversed backwards, then a simple transformation reversing the index variation must be also applied. In our example, the interchange of the indexes in Fig. 11c does not reduce the distance between between the points representing the minimum and maximum index vectors, but the reverse of the first index variation—as shown in Fig. 11d—entails a distance reduction: $dist(M, N) = (11 - 2) \times 5 + (5 - 6) = 44$. It follows that the array reference can be stored without mapping conflicts in a memory window W_A of 45 words: it suffices

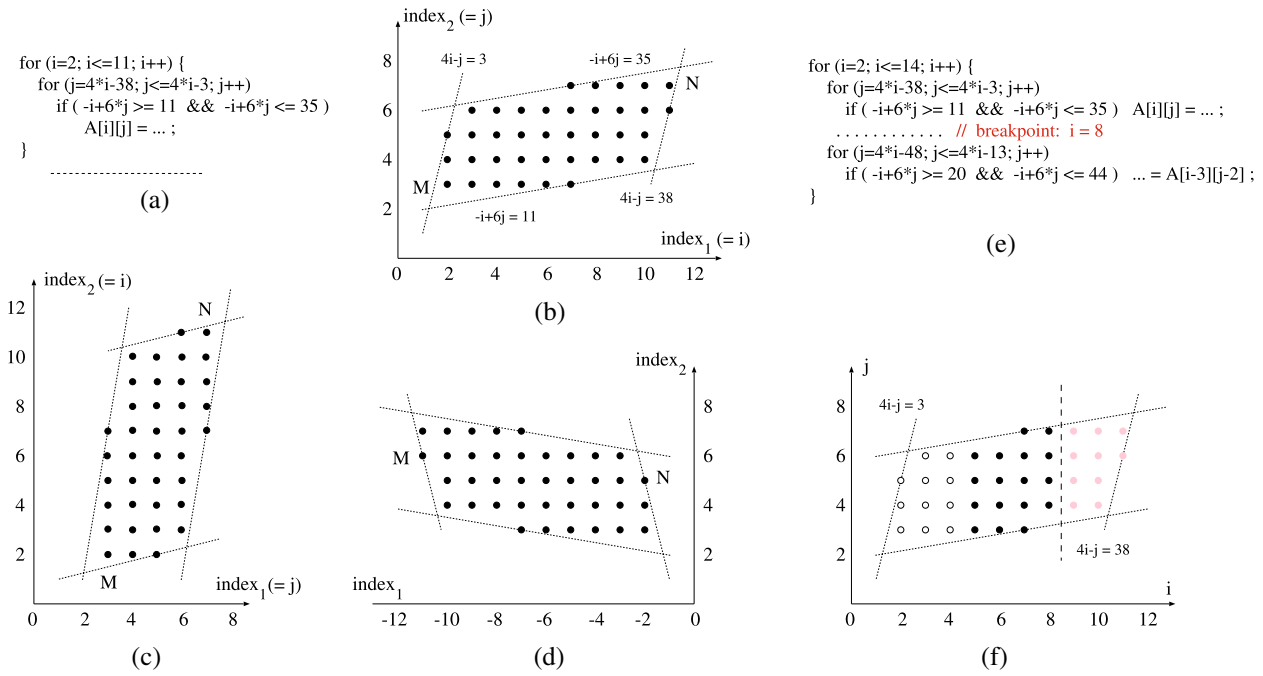

Fig. 11 a Simple code example, illustrating the mapping idea. **b** The index space of the array reference $A[i]\,[j]$. **c** The index space with indexes interchanged. **d** The index space flipped horizontally. **e** Code example with two array references having an identical index space. **f** The index space of the two array references from **e**, where not all the array elements are simultaneously alive

that any read/write access to $A[\,i]\,[j]$ be redirected to the memory word, say, $W_A[\,(5 * (13 - i) + j) \bmod 45]$. To be sure, 45 words represent an excess of storage relative to the minimum storage requirement of 38 words, but the advantage is that there is an easy-to-design function directing the mapping from the index space to the data storage.

Figure 11e shows another code example, the array elements produced by the array reference $A[i]\,[j]$ are consumed by the array reference $A[i-3]\,[j-2]$. The points to the left of the dashed line represent the iterator vectors of the elements produced till the breakpoint indicated in the code, the black points representing the elements still alive (i.e., produced and still used as operands in the next iterations), while the circles representing A-elements already "dead" (i.e., not needed as operands any more). The light grey points to the right of the dashed line represent the index vectors of A-elements still unborn (to be produced in the next iterations). There is a canonical linearization in which the distance between the index vectors of simultaneously alive elements is 17 (which entails a memory window of 18 words), very close to the minimal storage requirement of 17 words. □

The computation of distances are performed for each disjoint lattice extracted from the code [30]. The overall mapping results are assembled, taking into account the lifetimes of lattices, as well as the lifetimes of the array elements they contain.

In order to avoid the inconvenience of analyzing different linearization schemes (whose number grows fast with the signal's dimension), we also use a second mapping technique based on integer projections: although it often yields slightly worse storage results than the linearization approach, it has the advantage of being faster.

We compute a maximal m-dimensional bounding box $BB_A = (w_1, \ldots, w_m)$ large enough to encompass at any time during code execution the simultaneously alive (m-dimensional) A-elements. As already mentioned in Section 1, this bounding-box technique was also used in—a polyhedral parallel code generator for CUDA [12]. An access to the element $A[\,index_1] \ldots [\,index_m]$ can then be redirected without any conflict to the bounding box element $BB_A[\,index_1 \bmod w_1] \ldots [\,index_m \bmod w_m]$.

Each window side w_k is computed as the maximum difference in absolute value between the kth indexes of any two A-elements (A_i, A_j) simultaneously alive, plus 1. More formally, $w_k = \max\{\,|x_k(A_i) - x_k(A_j)|\,\} + 1$, for $k = 1, \ldots, m$. This ensures that any two array elements simultaneously alive are mapped to distinct memory locations. Then, the bounding box BB_A can be mapped one-to-one to a memory window W_A. The amount of data memory required for storing the

array is the volume of the bounding box BB_A, that is, $|W_A| = \Pi_{k=1}^{m} w_k$.

This mapping approach can be independently applied to each memory layer, providing mapping functions for all the signals in the specification and a complete storage allocation/assignment solution for distributed memory organizations. In addition, it can generate the traces of memory accesses for each memory layer, the trace to the SPM being particularly useful for energy-aware memory banking (see the next section). Our memory management software also computes the minimum storage requirement of each multidimensional signal in the specification [22] (therefore, the optimal memory sharing between the elements of each array), as well as the minimum data storage for the entire algorithmic specification—therefore, the optimal memory sharing between all the array elements and scalars in the code. These lower-bounds are used as metrics of quality for the mapping solution, since they show how much larger the mapping windows are versus the minimum storage requirements: no prior technique provides such metrics of quality for their mapping solutions.

5 Scratch-pad memory banking for the reduction of energy consumption

After being assigned to the off- and on-chip memory layers, the linearly bounded lattices are mapped to the external DRAM and SPM; so, the distribution of the memory accesses to the SPM address space is known. Let us assume that the range of contiguous addresses mapped to the on-chip SPM is $\{0, 1, \ldots, N-1\}$, that memory is word-addressable and the word width is known (being imposed by the chosen core processor). The dynamic energy $E_1^{dyn}(0,N)$ (where the arguments are the start address and the number of words, the subscript being the number of banks) consumed by a monolithic SPM is [5]

$$E_1^{dyn}(0,N) = E_R(N) \cdot \sum_{i=0}^{N-1} read[i] + E_W(N) \cdot \sum_{i=0}^{N-1} write[i],$$

where $E_R(N)$ and $E_W(N)$ are the energies consumed per *read*, respectively *write*, access to an SPM of N words; they are technology-dependent metrics. In addition, $read[i]$ and $write[i]$ represent the number of accesses to word i and, consequently, the sums represent the total numbers of *read/write* accesses to the on-chip memory locations $0, 1, \ldots, N-1$.

If the address space of the on-chip SPM is arbitrarily partitioned in two ranges $\{0, 1, \ldots, k-1\}$ and $\{k, k+1, \ldots, N-1\}$, then the dynamic energy consumed in the two-bank SPM becomes:

$$E_2^{dyn}(0,k,N) = E_1^{dyn}(0,k) + E_1^{dyn}(k,N-k)$$

The first two arguments of E_2^{dyn} are the start addresses in words of the two banks, the third being the total size.

The static energy consumed in the two-bank SPM, having the address space partitioned as above, is the sum of the static energies in each bank: $E_2^{st}(0,k,N) = E_1^{st}(0,k) + E_1^{st}(k,N-k)$. Neither term depends on the number of memory accesses.

The partitioning is energetically beneficial if $E_2^{dyn}(0,k,N) + E_2^{st}(0,k,N) + \Delta E_{12} < E_1^{dyn}(0,N) + E_1^{st}(0,N)$, where ΔE_{12} is the energy overhead required by the extra logic (usually, a decoder) and interconnections necessary to move from the monolithic SPM to a two-bank architecture. Figure 12 shows the more complex architecture of a multi-bank versus the monolithic architecture: the additional components and interconnects—the address and data buses, the decoder, the control signals—may introduce a non-negligible overhead on power consumption that must be compensated by the savings entailed by bank partitioning. These savings are caused by the average power decrease in accessing the memory hierarchy, because a large fraction of accesses is typically concentrated on a smaller, more energy-efficient bank. In addition, the memory banks that stay idle long enough can be disabled through their chip-select (CS) pins. Equivalently, the partitioning is energetically beneficial if the *energy benefit* of the two-bank solution

$$\left(1 - \frac{E_2^{dyn}(0,k,N) + E_2^{st}(0,k,N) + \Delta E_{12}}{E_1^{dyn}(0,N) + E_1^{st}(0,N)}\right) \times 100\,[\%]$$

versus a monolithic SPM is positive.

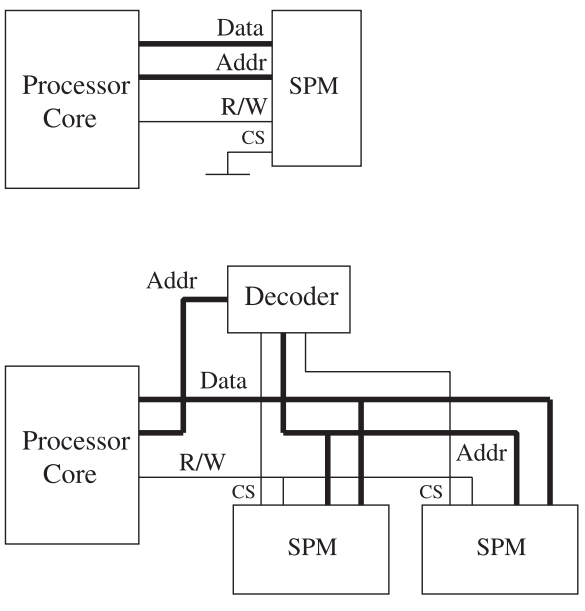

Fig. 12 Architectures with monolithic and two-bank SPMs

The solution space of two-way memory banking can be exhaustively explored (and, hence, optimally solved in energy point of view) by iteratively moving the upper bound k of the first bank from 1 to $N-1$, and finding the global minimum: $\min_k \{E_2^{dyn}(0,k,N) + E_2^{st}(0,k,N)\} + \Delta E_{12}$.

A similar cost metric can be used to explore multi-way banking solutions: any possible partition into M (≥ 2) banks is defined by a set of M-1 addresses identifying the memory bank boundaries. Based on this idea, Benini et al. implemented a recursive algorithm [5] where the solution space is exhaustively explored (their main input is the graph of the distribution of memory accesses to the SPM address space, rather than the behavioral specification of the application). This search for an energetically-optimal solution proves to be computationally expensive (see Section 6), even infeasible, for larger values of M—the maximum number of banks—and/or larger values of the SPM size. Angiolini et al. carried out more efficiently a similar exploration using dynamic programming [6]. Although the time complexity is polynomial, our experiments found that the running times of their method exhibit a fast increase with the sizes of the SPM and the execution trace (the computation time was over 8 h for our illustrative example in Fig. 4, and an SPM size of 8 Kbytes, when the memory word was 1 byte. For SPM sizes smaller than 2 Kbytes, the technique can be effective, though).

The banking algorithms we propose are consistent with our model of partitioning the array space of signals into disjoint lattices (see Section 3). For $M < 4$, these algorithms are basically identical to the exploration algorithm presented in [5] since this approach yields optimal solutions. For $M \geq 4$, as the running times may be extremely large, we introduce a constraint that significantly reduces the exploration space: no SPM-assigned lattice can cross a bank boundary. This constraint ensures the effectiveness of our approach in point of view of speed and near-optimality of the results—as *Example 4* will show.

5.1 Lattice-based recursive algorithm

In addition to M, the maximum number of SPM banks, the inputs of the SPM partitioning algorithm are:

Input 1: An array $\mathcal{A} = [addr_0,\ addr_1,\ \ldots\ ,\ addr_n]$ of ordered addresses such that a linearly bounded lattice L_k, $k = 1, \ldots, n$, assigned to the on-chip memory layer be mapped at the SPM successive addresses $\{addr_{k-1}, \ldots,\ addr_k - 1\}$.

Input 2: An array $\mathcal{RW} = [rw_1,\ \ldots,\ rw_n]$ which elements represent the numbers of *read/write* accesses for each lattice mapped onto the SPM (notice that the numbers of *read/write* accesses for each lattice mapped onto the SPM are already known from Section 3).

Input 3: An array $\mathcal{E} = [\Delta E_{12},\ \Delta E_{23},\ \ldots\ ,\ \Delta E_{M-1,M}]$, which elements $\Delta E_{k,k+1}$ are the energy overheads resulting from moving from an on-chip SPM with k banks to one with $k+1$ banks. The decoding circuitry was synthesized using the ECP family of FPGA's from lattice semiconductor [31] and, for the energy overheads, we used the power calculator from Lattice Diamond [31].

Output: The energetically-optimal SPM partitioning, i.e., an array of SPM addresses delimiting the banks, and the minimum value of the total (static and dynamic) energy consumption for this optimal SPM banking solution.

The algorithm starts from the monolithic architecture and searches for the energetically-optimal partitioning of the SPM in no more than M memory banks, such that the borderlines between banks are addresses in the array $\mathcal{A}$ (hence, ensuring that any lattice of signals is entirely stored in one bank). A variable `crtBestSolution` records the set of addresses in $\mathcal{A}$ corresponding to the most energetically-efficient partition reached in any moment of the exploration; initially, the SPM being monolithic, this set is $\{addr_0,\ addr_n\}$. A variable `crtMinEnergy` registers the total energy consumption of the best SPM banking solution encountered during the exploration. A function `SPM_energy(bank_size, number_accesses)` uses CACTI 6.5 [28] and the number of *read/write* accesses in order to compute the total energy (both static and dynamic) consumed in a bank of the specified size. A recursive function `Multi_Bank`, whose first formal parameter m (initially equal to 2) is the current number of banks, searches for the optimal solution such that the first bank ends at $addr_k$. This function is successively called for $k = 1,\ 2,\ \ldots\ ,\ n-1$. `EnergyConsumed` registers the amount of energy consumed from the start of the SPM till the borderline $addr_k$. If its value exceeds the best energy already recorded (`crtMinEnergy`), there is no need to continue the exploration since all the next solutions will be energetically-worse—due to the monotonic increase of the energy consumption with the SPM size.

Algorithm 3 Energy-aware recursive SPM banking

```
crtBestSolution = {addr_0 , addr_n};
crtMinEnergy =
      SPM_energy (addr_n–addr_0 , Σ_{i=0}^{n} rw_i);
push (SolutionStack, crtBestSolution);
for k = 1 to n-1 do {
  EnergyConsumed =
      SPM_energy (addr_k–addr_0 , Σ_{i=0}^{k} rw_i);
  if (EnergyConsumed ≥ crtMinEnergy) break; // no chance
   // finding a better solution for any larger k: exploration over!
  Multi_Bank (2, M, k, EnergyConsumed);
// explore solutions with first bank [addr_0 , addr_k]
}
pop (SolutionStack);
```

Outputs: `crtBestSolution` – an ordered set of SPM addresses from $\mathcal{A}$ delimiting the banks, and the corresponding energy consumption `crtMinEnergy`.

The recursive function `Multi_Bank` searches for the best banking solution starting from $addr_k$ till the end of the SPM at $addr_n$. From the beginning of the SPM ($addr_0$) till the address $addr_k$ there are already $m-1$ banks. At the beginning, the function considers $[addr_k\ ,\ addr_n]$ as the mth bank of the SPM: if this banking configuration is energetically better than all previous solutions, it is duly recorded as the best solution reached during the exploration. If the maximum number of banks is not reached yet ($m < M$), then the function explores solutions with $m+1$ banks or more, considering the mth bank to be $[addr_k\ ,\ addr_j]$, for $j = k+1,\ k+2,\ \ldots\ ,\ n-1$.

A solution stack is used and typical stack functions (`push, pop, top`) to record and resume the partial banking solutions. For instance, the `push` instruction in the body of the `Multi_Bank` function takes the set of memory addresses on the top of the stack, adds the new element $addr_k$ to it, and the new set is pushed back on the stack.

Since the energy cost is monotonically increasing with the SPM size, a backtracking mechanism is incorporated before the recursive call to prevent the search towards more energetically-expensive partitions. The *output* of the algorithm is an array of SPM addresses delimiting the banks, and the corresponding energy consumption.

In addition, for each LBL in the decomposition of the array space, we compute the time intervals (in clock cycles) when the lattice is *not* accessed. This *idleness analysis* cannot be done directly in terms of *time*: first, it is done in terms of loop iterators. For instance, we must determine the iterator vectors in the loop nests when a disjoint lattice is accessed for the first time and for the last time.[5] Only afterwards, we compute the clock cycles during the code execution corresponding to those iterator vectors. When the recursive function `Multi_Bank` investigates the case when the m-th bank is between $Addr[k]$ and $Addr[n]$ (the end of the SPM), the idleness intervals of the lattices $L_{k+1},\ \ldots\ , L_n$ assigned to this m-th bank are intersected in order to determine whether there are idleness intervals at the bank level. If this is the case, the bank can be switched to the *sleep* state during the idleness intervals that are large enough. (A time overhead of one clock cycle for the transition from the *sleep* to the *active* state is also applied, in accordance with simulated data on caches reported in [32]). In order to overcome the energy overhead entailed by the transition of a memory bank from the *active* state into the *sleep* state and back to the *active* state, the bank must remain in the *sleep* state at least a minimum number of clock cycles (otherwise, the economy of static energy is lesser than the energy overhead of the transitions). This idleness threshold in cycles can be estimated; typical values are in the order of hundreds of cycles [16]. So, if the idleness of a bank (resulted from the intersection of the idleness intervals of the lattices assigned to the bank) exceeds the idleness threshold, then the energy cost of the bank is computed taking into account the switches to the *sleep* state and back. The idleness intervals of each lattice are organized into an *interval tree* [33] as the depth of this data structure is $O(log\ n)$ for n intervals, and typical interval operations have logarithmic complexity.

A high-level pseudo-code of the recursive function `Multi_Bank` is given below:

```
  void Multi_Bank (m, M, k, EnergyConsumed) {
  if (crtMinEnergy ≤ EnergyConsumed+ΔE_{m−1,m})
return;
  EnergyConsumed += ΔE_{m−1,m} ;
  EnergySPM = EnergyConsumed +
      SPM_energy (addr_n−addr_k , Σ_{i=k}^{n} rw_i);
//  a new partitioning solution of m banks is ready
  if (EnergySPM < crtMinEnergy) {
    crtMinEnergy = EnergySPM ; // the new solution is
better!
    crtBestSolution = top (SolutionStack) ∪ {addr_k};
      //  set of bank boundaries on top of the stack, plus
addr_k
}
  if ( m < M ) { // if max. number of banks not reached yet
//  then explore finer SPM partitions
    push (SolutionStack, top (SolutionStack) ∪ {addr_k});
    for (int j=k+1; j<n; j++) {
      e = EnergyConsumed +
SPM_energy (addr_j−addr_k, Σ_{i=k}^{j} rw_i);
      if (e ≥ crtMinEnergy) break;  // no chance of
finding
// a better finer partition for any j; then, backtrack!
      Multi_Bank (m+1, M, j, e);
// else explore solutions with new bank [addr_k, addr_j]
}
    pop (SolutionStack);
  }
}
```

Example 4: Let us consider again the illustrative example from Fig. 4, where the 64 central lattices of the array A (numbered 96–159 in Fig. 8) were stored in an SPM of 8 Kbyte, the external DRAM being of 56 Kbytes. As shown in Section 3, the energy consumption of this monolithic SPM is 3,063.6 μJ, assuming a 32 *nm* technology. Running the banking algorithm from [5], where the maximum number of banks M was set to 4, the 4-bank optimal banking solution (of 1,602.23 μJ) was found in 4433 s, after the exploration of 58.73

billion SPM partitions. (Setting M at a higher value than 4 proved to be computationally infeasible for [5]). In contrast, *Algorithm 3* (for M=8) found a 6-bank solution of a lower energy (1564.95 μJ) in only 30.38 s, exploring less than 1 % SPM partitions (363.6 million versus 58.73 billion). Figure 13 displays the graphs of the best energies of banking solutions (the values of `crtMinEnergy` from *Algorithm 3*) found by the two techniques after analyzing the first ten thousand banking configurations. The figure shows that our algorithm finds faster the energetically-better solutions. To be sure, *Algorithm 3* finds only sub-optimal solutions for values of M larger than 3; but these solutions are near-optimal and they are found very fast. For instance, setting M=4, *Algorithm 3* found a 4-bank solution of 1604.01 μJ, slightly worse than the optimal value 1,602.23 μJ; on the other hand, the run time was only 0.012 s, that is, several orders of magnitude faster than 4433 s!

5.2 Lattice-based dynamic programming algorithm

The inputs are identical as in the previous algorithm, except the last one:

Input 3: An array $\mathcal{E}$ =[0 ΔE_2 ΔE_3 ... ΔE_M] which elements ΔE_k ($1 < k \leq M$) are energy overheads resulting from moving from a monolithic SPM to one with k banks (obviously, ΔE_1=0). These energy overheads were estimated with the power calculator from Lattice Diamond [31].

The main data structures used by the algorithm are:

- 2-D "cost" array C : each element $C[i\,,\,j]$ ($0 \leq i < j \leq n$) is initialized to the energy consumed by a monolithic SPM having the address space $[\,addr_i\,,\,addr_j)$ and storing the linearly bounded lattices $L_{i+1}, \ldots, L_j$; in particular, $C[0\,,\,n]$ is, initially, the energy consumed by the whole monolithic SPM. At the end of the algorithm, each element $C[i\,,\,j]$ will contain the energy consumption *after* the address space $[\,addr_i\,,\,addr_j)$ was optimally partitioned—under the constraint that bank boundaries are only addresses from the input array $\mathcal{A}$. The additional exploration constraint—that no disjoint lattice assigned to the SPM can cross a bank boundary— ensures the effectiveness of the approach.
- 2-D array m : each element $m[i\,,\,j]$ ($0 \leq i < j \leq n$) is the number of banks in the address space $[\,addr_i\,,\,addr_j)$; their initial value is 1.
- 2-D array s : used for constructing an optimal partitioning solution.

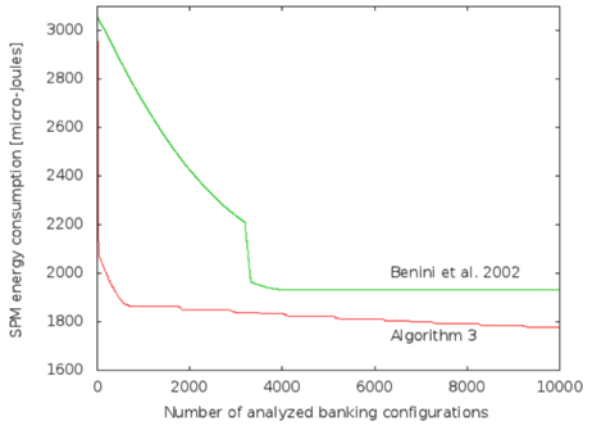

Fig. 13 Energy-aware SPM banking: the decrease of the SPM energy consumption as different banking solutions are analyzed by our algorithm and by the exhaustive partitioning in [5]

Algorithm 4 Energy-aware SPM banking by dynamic programming

```
1.  let C be a new array [0 .. n-1 , 1 .. n] ;
2.  let m be a new array [0 .. n-1 , 1 .. n] ;
3.  let s be a new array [0 .. n-1 , 1 .. n] ;
4.  for j = 1 to n do   // initialization of the arrays C, m, s
5.     for i = 0 to j − 1 do {
6.        C[i , j] = SPM_energy (addr_j−addr_i , Σ_{k=i+1}^{j} rw_k) ;
7.        m[i , j] = 1 ;  s[i , j] = i ;
       }
8.  for L = 2 to n do
9.     for i = 0 to n − L do {
10.       j = i + L ;
11.       q = C[i , j] ;
12.       for k = i + 1 to j − 1 do {
13.          if ( m[i , k] + m[k , j] > M ) continue ;
14.          Δ = ΔE_{m[i,k]+m[k,j]} − ΔE_{m[i,k]} − ΔE_{m[k,j]} ;
15.          if ( q > C[i , k] + C[k , j] + Δ ) {
16.          q = C[i , k] + C[k , j] + Δ ;
17.          s[i , j] = k ; }
          }
18.       m[i , j] = m[i , s[i,j]] + m[s[i,j] , j] ;
19.       C[i , j] = q ;
       } // C[0 , n] = optimal energy consumption
         //            after the SPM partitioning
```

The first loop nest (instructions 4–7) initializes the three arrays: C – see Fig. 14a, m, and s. The function SPM_{energy} uses information provided by CACTI 6.5 [28]—the dynamic energy per access and the static power—to compute the energy consumption of a monolithic SPM of $(addr_j - addr_i)$ bytes, which is accessed $\sum_{k=i+1}^{j} rw_k$ times. A 32-*nm* technology is assumed by default. For the computation of the static energy consumption, the number of clock cycles for the execution of the given application is obtained by simulation; a frequency of 400 *MHz* is used by default (but this value can be modified by the user).

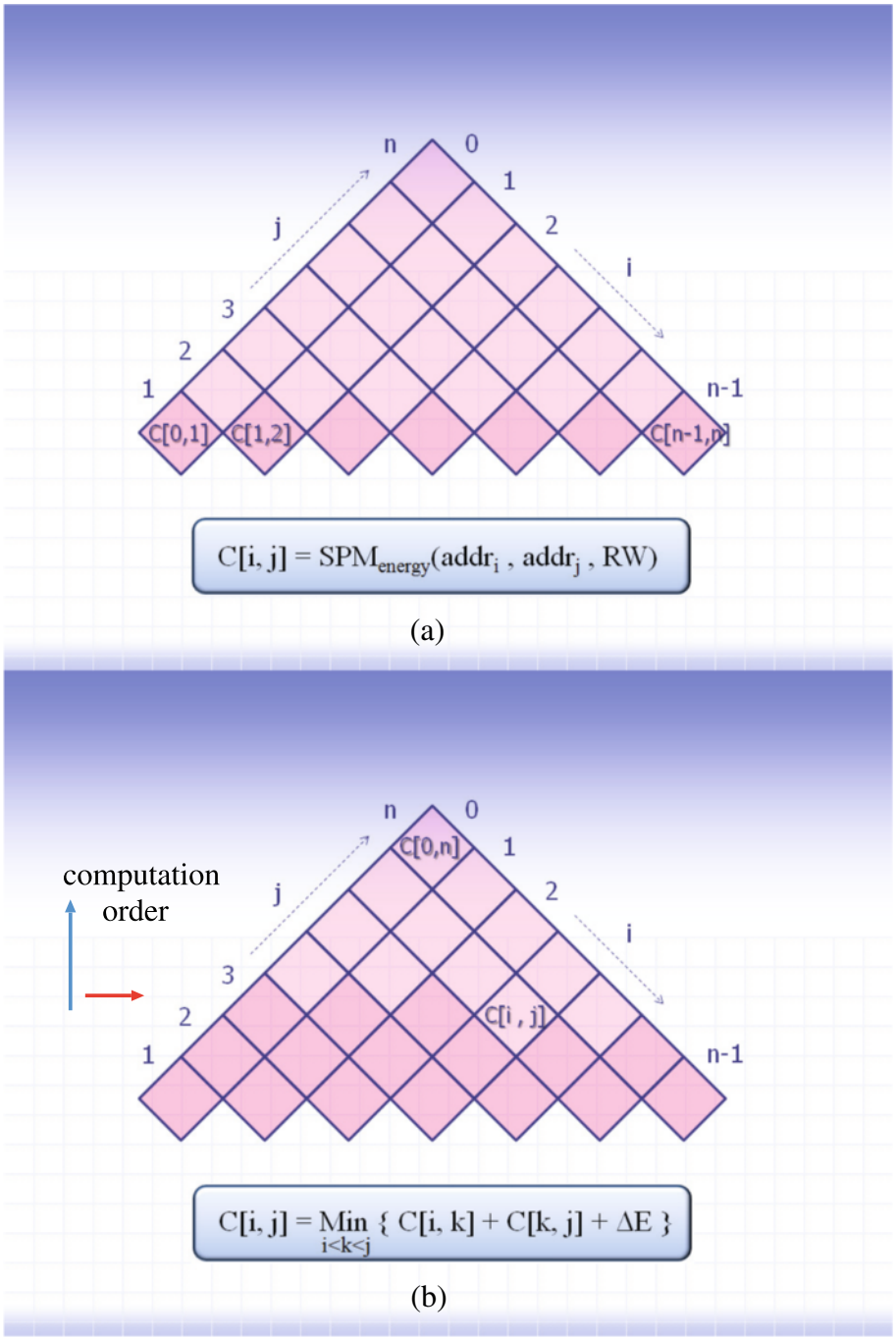

Fig. 14 a The initialization of the cost array *C* (see instruction 6 in *Algorithm* 4). **b** The update of the cost array *C*: the element *C*[0, *n*] is the energy consumption corresponding to the optimal SPM partitioning

Afterwards, the next loop nest (starting at instruction 8) computes with a bottom-up approach (see Fig. 14b) the energetically-optimal banking in the address space $[addr_i\ ,\ addr_j)$, where $j - i = L$ increases gradually from $L = 2$ to $L = n$. The last value of L corresponds to the entire address space of the SPM: $[addr_0\ ,\ addr_n)$. For each pair (i, j), the optimal energy cost $C[\,i\ ,\ j]$ is computed (instructions 11–19) as the minimum between the energy consumed in the monolithic case, and

$$min_{i<k<j}\{\ C[i\ ,\ k]\ +\ C[k\ ,\ j]\ +\ EnergyOverhead\ \}$$

that is, when a bank boundary is introduced at $addr_k$ (in between $addr_i$ and $addr_j$). Since the component address spaces $[addr_i\ ,\ addr_k)$ and $[addr_k\ ,\ addr_j)$ are already partitioned into $m[i,k]$ and, respectively, $m[k,j]$ banks, the number of banks in the address space $[addr_i\ ,\ addr_j)$ would increase to $m[i,k]\ +\ m[k,j]$, entailing an energy overhead of $\Delta E_{m[i,k]+m[k,j]}-$ which must be added to the energy cost. At the same time, the energy overheads $\Delta E_{m[i,k]}$ and $\Delta E_{m[k,j]}$—corresponding to the $m[i,k]$ and $m[k,j]$ banks—must be subtracted from the cost (see instructions 14 and 16).

Note that, if any of the component address spaces is not partitioned, say $m[i,k]$=1, then $\Delta E_{m[i,k]} = \Delta E_1 = 0$. All these energy overheads are elements of the array $\mathcal{E}$ (*Input 3*).

The conditional instruction 13 eliminates the solutions exceeding M number of banks. Arbitrarily fine partitioning is prevented since an excessively large number of small banks is area inefficient, imposing a severe wiring overhead, which also tends to increase communication power and decrease performance.

The time complexity of the algorithm is $\Theta(n^3)$ due to the three `for` loops (instructions 8, 9, and 12). The main parameter n is the number of disjoint linearly bounded lattices that are assigned to the SPM; these n lattices represent only a subset of the disjoint LBLs that result from the partitioning of the multidimensional signals from the behavioral specification (part of the LBLs being stored off-chip in a DRAM).

The space complexity is $\Theta(n^2 + M)$. The latter term is entailed by the input matrix $\mathcal{E}$. Note that the maximum number of banks M has, typically, a small value, so it is negligible in comparison to the former term n^2.

The banking solution can be determined calling a recursive function `PrintOptimalPartition(`s`, 0,` n`,` $\mathcal{A}$`)`.

```
void PrintOptimalPartition (s, i, j, A) {
 if ( s[i , j] == i ) print addr[i] ;
 else
    PrintOptimalPartition (s, i, s[i , j], A) ;
    PrintOptimalPartition (s, s[i , j], j, A) ;
}
```

6 Experimental results

An EDA framework for memory management has been implemented in C++, incorporating the three memory management tasks described in this paper. In addition to these algorithms, the software system contains a tool for the computation of the minimum data storage of the given application [22]. The main input of the software tool is an algorithmic specification of the signal processing application, as described at the end of Section 1. An interface to CACTI 6.5 [28] has been implemented in order to obtain memory data concerning power consumption. (CACTI 6.5 supports 32, 45, 68, and 90 *nm* technologies.) Tables 1, 2, and 3 summarize the results of our experiments, carried out on a PC with an Intel Core 2 Quad 2.83 GHz processor.

Table 1 shows several experiments considering as input application a motion detection algorithm—used in the transmission of real-time video signals on data networks. It displays the energy consumption in the memory subsystem for different data assignments to the memory layers. Column 1 shows the values of the parameters of the motion detection algorithm, columns 2 and 3 display the numbers of array elements and scalar signals, and the total numbers of *read/write* accesses. Column 4 displays the storage requirements of the application, computed with the algorithm from [22]—which is embedded in our framework. For the motion detection, our mapping algorithm (Section 4) finds optimal mapping solutions in terms of storage (column 5). Actually, two multidimensional signals from the application code will be stored in two registers: their footprint is only 1 byte each, since our tool correctly detected that their elements have disjoint lifetimes. Then, columns 6–10 present different scenarios for data assignment between the on-chip SPM and the off-chip DRAM, together with the energy consumption (both static and dynamic) in these memories.[6] Column 11 displays the energy consumption in the memory subsystem, e.g., for the first set of parameters, the total energy increases from 2.24 μJ—when all the data is stored into the SPM—to 56.03 μJ—when all the data is stored off-chip. The computation times (column 12) are very similar for each data assignment, so only the ballpark values are given.

The benchmarks used in the next tables are algebraic kernels—Durbin's algorithm for solving Toeplitz systems; a singular value decomposition (SVD) updating algorithm [34] used in spatial division multiplex access (SDMA) modulation in mobile communication, in beamforming, and Kalman filtering— and a few multimedia applications: the kernel of an MPEG4 motion estimation algorithm for moving objects; a 2-D Gaussian blur filter algorithm from a medical image processing application which extracts contours from tomograms in order to detect brain tumors; the kernel of a voice coding application—an essential component of a mobile radio terminal.

Table 2 displays in columns 2–3 information on the behavioral specification of the given application (column 1): the amounts of scalar signals (array elements) and the numbers of memory accesses. Column 4 shows the amount of data storage computed by the mapping algorithm. Then, column 7 displays the (static and dynamic) energy consumption in the memory subsystem when the sizes in bytes of the SPM and DRAM are the ones shown in columns 5–6. For a better evaluation of our energy-aware data assignment model, we implemented

Table 1 Experimental results for a motion detection algorithm

Parameters	Scalars (array elements)	Memory accesses	Minimum data storage [22]	Data storage (after mapping) [bytes]	SPM size [bytes]	SPM banks	SPM energy [μJ]	DRAM size [bytes]	DRAM energy [μJ]	Total energy [μJ]	CPU [sec]
$M = N = 32$	185,239	361,250	3364	3364	3362	2	2.24	0	0	2.24	1.8
$m = n = 8$					2306	2	2.14	1056	15.02	17.16	
					1650	2	1.86	1712	18.09	19.95	
					1250	1	1.64	2112	20.62	22.26	
					0	–	0	3362	56.03	56.03	
$M = N = 64$	2,632,615	5,229,378	13,124	13,124	13122	2	97.47	0	0	97.47	24.7
$m = n = 16$					6370	3	37.23	6752	263.59	300.82	
					4802	2	32.52	8320	302.29	334.81	
					0	–	0	13122	854.38	854.38	

another signal assignment strategy—similar to the one used in [23], where the steering mechanism is based on the intensely-accessed cuts within the array space. The savings of energy consumption (column 8) were, typically, between 18 and 28 % relative to this model. The CPU times when executing the entire memory management flow are shown in column 9. The tests have been done for a 32 *nm* technology and assuming a clock frequency of 400 *MHz*.

Table 3 shows the savings of energy consumption after SPM banking (32 *nm* technology) for various benchmarks. Column 2 displays the number of addresses in the on-chip memory. Column 3 reports the computation times for a full exploration with backtracking—implemented as the one presented in [4, 5] —targeting energy reduction, but using CACTI 6.5 [28] for power estimation (the maximum number of banks was set to $M = 4$, since for larger values of M the times were unknown—as the exploration had to be stopped after several hours). Our own energy results for $M = 4$ for all the benchmark tests were no more than 0.4 % higher than the optimal ones; but they all were obtained in only a fraction of a second, in contrast to the significant running times from column 3. Column 4 reports the computation times in seconds for our recursive banking algorithm that explored the search space for up to $M = 8$ banks—a value that proved impossible for [5] if the word-length is 1 byte.

Column 5 shows the computation times obtained running an implementation of the dynamic programming approach of Angiolini et al. [6]. The main input of this algorithm is the graph of memory accesses during the execution of the application code. The main data structure is an array having the numbers of rows and columns equal to the size (in words) of the graph of memory accesses and, respectively, the size of the SPM. The array elements are profit values targeting energy (or, alternatively, performance) optimization. The silicon area is indirectly taken into account by increasing heuristically the indexes of the profits computed during the dynamic programming by amounts depending on ratios of SPM areas. The time complexity of the algorithm is polynomial, depending on the product SPM size squared times the size of the graph. The practical running times can be significant, though, for benchmarks with a large memory address space and/or a large SPM (while typically faster than the full exploration with backtracking [5], we also encountered examples where this technique was slower, due in part to the fact that the number of banks is unconstrained).

Table 2 Experimental results for energy-aware assignment of signals to the on- and off-chip memory layers

Application	Scalars (array elem.)	Memory accesses	Data memory after mapping	SPM size	DRAM size	Energy [μJ]	Energy savings	CPU [sec]
Motion detection	2,632,615	5,229,378	13,124	4802	8320	334.81	28.42 %	24.7
Motion estimation	265,633	1,053,089	4513	256	4257	138.31	22.12 %	2.8
Gaussian blur filter	53,615	77,619	14803	5003	9800	6.13	26.66 %	3.6
Durbin algorithm	252,499	1,005,993	1,998	500	1498	123.62	20.40 %	39.1
SVD updating algorithm	3,045,447	29,500,000	34,950	4096	30854	1601.52	24.81 %	47.5
Voice coding kernel	33,835	47,416	14,634	2032	12602	2.56	18.76 %	4.8

Table 3 Experimental results for energy-aware SPM banking

Application	Address space	$CPU^{full}_{expl.}$ [5] M=4 [s]	CPU Alg.3 M=8 [s]	$CPU^{dyn}_{prog.}$ [6] [s]	CPU Alg.4 M=8 [s]	Energy savings vs. [5] (M=4)	[6] [s]	monolithic
Motion detection	6370	3163	2.0	1592	1.4	7.2 %	2.9 %	41.2 %
Motion estimation	1024	736	0.8	1.8	0.6	5.1 %	2.3 %	36.5 %
Durbin's algorithm	500	247	2.8	3.2	1.7	5.0 %	3.9 %	26.5 %
SVD updating alg.	4096	2405	10.7	3048	8.2	6.4 %	3.6 %	28.4 %
Voice coding kernel	2032	1297	1.5	336	1.1	7.8 %	5.4 %	31.4 %

Column 6 reports the computation times for our banking algorithm using dynamic programming: this technique proves to be faster than *Algorithm 3*, which was expected due to the polynomial complexity of *Algorithm 4*. The additional exploration constraint—that no disjoint lattice assigned to the SPM can cross a bank boundary—ensures the effectiveness of our basic approach when $M \geq 4$: this constraint significantly reduces the search space, typically yielding near-optimal results.

The data structures of our our dynamic programming approach (see Section 5.B) are significantly smaller in size and the computation of energy costs (the elements of array C) allows portability from the back-end tool CACTI to other memory models. In contrast, the dynamic programming approach from [6] uses a heuristic index increase (based on ratios of SPM areas) in the array of the energy profits, which is dependent on the memory model employed.[7] Our dynamic programming technique can optimize die area instead of energy consumption (or a weighted combination of the two) by redesigning the function *SPM_energy* from *Algorithm 4*.

Not only the computation times of our tool were far better, but our tool found partitions of more than 4 banks which were superior in terms of energy consumption than the four-bank solutions found by the previous technique [5]: column 7 reports the energy savings versus the full exploration for $M = 4$. Column 8 shows the savings of energy consumption of our algorithm versus the dynamic programming approach similar as [6]. Note that this dynamic programming technique yielded better results than [5] since it found energetically-better solutions that had more than four banks. On the other hand, our algorithm found even better solutions since it could exploit the idleness intervals of the memory banks (which [6] does not do). Column 9 displays the energy savings obtained by our tool, with respect to the case of a monolithic SPM.

We also tested the algorithms from this EDA framework on a larger code of about 900 lines (mentioned also in [22]), containing 113 loop nests three-level deep and 906 array references—many having complex indexes. *Algorithm 1* ran in about 2.4 minutes, building the DAG of inclusions (like the one illustrated in Fig. 6) with 3159 nodes (LBLs), and preparing the polyhedral data structures required by the memory management tasks. *Algorithm 2* was fast, running in less than 10 s. (Note that there was a preliminary step, not taken here into account, when our CACTI interface obtained data on power and access times by running CACTI 6.5 for a range of DRAM and SPM sizes: afterwards, these data can be used in other benchmarks as well). The signal-to-memory mapping step was more computationally-expensive (almost 4 min) since many LBLs from the specification code were produced and consumed in the same loop nests, and the number of canonical linearizations of 3-D arrays is 48. *Algorithm 3*—the recursive algorithm with backtracking, ran in 3.7 min for a maximum number of SPM banks of $M = 5$, while *Algorithm 4* was even faster: 2.3 min.

7 Conclusions

This paper has presented an EDA framework for the high-level design of hierarchical memory architectures, targeting embedded data-intensive signal processing applications. The methodology presented in this paper is focused on the reduction of the energy consumption is the memory subsystem. The data assignment to the storage layers, the signal-to-memory mapping, as well as the on-chip memory banking, are all efficiently addressed within a common polyhedral framework. The steering assignment mechanism is based on the identification of the intensely-accessed regions within the array space of the multidimensional signals. The added flexibility of this assignment model led to superior energy savings in comparison to earlier approaches.

Endnotes

[1] That is, the execution ordering is induced by the loop structure and, hence, it is fixed. The research on code transformation is orthogonal to our methodology, but it could be used as a preliminary step.

[2] Solving a linear Diophantine system was proven to be of polynomial complexity, the various methods being typically based on bringing the system matrix to Hermite Normal Form [26].

[3]CACTI 6.5 is an analytical tool that takes a set of SPM, cache, or DRAM parameters as inputs and calculates memory data – like access time, static power, dynamic energy spent per access, and area [28].

[4]The computation method employed by De Greef et al. consists of a sequence of integer linear programming (ILP) optimizations for each canonical linearization [10].

[5]This is based on the computation of the lexicographically- minimum and maximum iterator vectors of the lattice elements in normalized loops, operation described in [22].

[6]Memory generators do not allow all possible values for memory sizes or for bank boundaries: for instance, a memory generator may yield storage blocks with only a multiple of 16 bytes. Although our framework can take into account such kind of constraints, these tests aim to illustrate the algorithms, so no such constraint is imposed.

[7]This is a key reason why a comparison with the results on the benchmarks in [6] is difficult to achieve without insider knowledge.

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Department of Computer Science and Engineering, American University in Cairo, Cairo, Egypt. [2]Fermilab, Batavia, IL, USA. [3]Microsoft, Inc., Redmond, WA, USA. [4]ARM, Inc., San Jose, CA, USA.

Received: 22 September 2015 Accepted: 9 July 2016
Published online: 25 July 2016

References

1. F Catthoor, K Danckaert, C Kulkarni, E Brockmeyer, PG Kjeldsberg, TV Achteren, T Omnes, *Data Access and Storage Management for Embedded Programmable Processors*. (Springer, 2010)
2. PR Panda, N Dutt, A Nicolau, F Catthoor, A Vandecapelle, E Brockmeyer, C Kulkarni, E De Greef, Data memory organization and optimizations in application-specific systems. IEEE Design & Test of Computers, 56–68 (2001)
3. M Verma, P Marwedel, *Advanced Memory Optimization Techniques for Low-Power Embedded Processors*. (Springer, 2007)
4. A Macii, L Benini, M Poncino, *Memory Design Techniques for Low Energy Embedded Systems*. (Kluwer Academic Publ., Boston, 2002)
5. L Benini, L Macchiarulo, A Macii, M Poncino, Layout-driven memory synthesis for embedded Systems-on-Chip. IEEE Trans. VLSI Syst. **10**(2), 96–105 (2002)
6. F Angiolini, L Benini, A Caprara, An efficient profile-based algorithm for scratchpad memory partitioning. IEEE Trans. Computer-Aided Design IC's Syst. **24**(11), 1660–1676 (2005)
7. PR Panda, N Dutt, A Nicolau, On-chip vs.off-chip memory: the data partitioning problem in embedded processor-based systems. ACM Trans. Design Automation Electronic Syst. **5**(3), 682–704 (2000)
8. M Kandemir, G Chen, F Li, in *Proc. Asia-South Pacific Design Aut. Conf.* Maximizing data reuse for minimizing space requirements and executive cycles, (Yokohama, Japan, 2006), pp. 808–813
9. E Brockmeyer, M Miranda, H Corporaal, F Catthoor, in *Proc. 6th ACM/IEEE Design and Test in Europe Conf.* Layer assignment techniques for low energy in multi-layered memory organisations, (Munich, Germany, 2003), pp. 1070–1075
10. E De Greef, F Catthoor, H De Man, in *Parallel Computing*. Memory size reduction through storage order optimization for embedded parallel multimedia applications", special issue on Parallel Processing and Multimedia" (ed. A. Krikelis), vol. 23 (Elsevier, 1997), pp. 1811–1837
11. R Tronçon, M Bruynooghe, G Janssens, F Catthoor, Storage size reduction by in-place mapping of arrays. Verification, Model Checking and Abstract Interpretation, 167–181 (2002)
12. S Verdoolaege, JC Juega, A Cohen, JI Gomez, C Tenllado, F Catthoor, Polyhedral Parallel Code Generation for CUDA. ACM Trans. Arch. Code Optimization. **9**(4), 54–77 (2013)
13. A Darte, R Schreiber, G Villard, Lattice-based memory allocation. IEEE Trans. Comput. **54**, 1242–1257 (2005)
14. W Shiue, C Chakrabarti, in *Proc. 36th ACM/IEEE Design Aut. Conf.* Memory exploration for low power embedded systems, (New Orleans, 1999), pp. 140–145
15. O Golubeva, M Loghi, M Poncino, E Macii, in *Proc. ACM/IEEE Design Automation and Test in Europe*. Architectural leakage-aware management of partitioned scratchpad memories, (Nice, France, 2007), pp. 1665–1670
16. M Loghi, O Golubeva, E Macii, M Poncino, Architectural leakage power minimization of scratchpad memories by application-driven subbanking. IEEE Trans. Comput. **59**(7), 891–904 (2010)
17. W Kelly, V Maslov, W Pugh, E Rosser, T Shpeisman, D Wonnacott, The Omega Library interface guide (1995). Technical Report CS-TR-3445, Univ. of Maryland, College Park
18. R Wilson, R French, C Wilson, S Amarasinghe, J Anderson, S Tjiang, S-W Liao, C-W Tseng, M Hall, M Lam, J Hennessy, SUIF: An infrastructure for research on parallelizing and optimizing compilers. ACM SIGPLAN Notices. **29**(12), 31–37 (1994)
19. J Ramanujam, J Hong, M Kandemir, A Narayan, A Agarwal, Estimating and reducing the memory requirements of signal processing codes for embedded systems. IEEE Trans. Signal Process. **54**(1), 286–294 (2006)
20. V De La Luz, I Kadayif, M Kandemir, U Sezer, Access pattern restructuring for memory energy. IEEE Trans. Parallel Distributed Syst. **15**(4) (2004)
21. R Allen, K Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. (Morgan Kaufmann Publ., 2001)
22. F Balasa, H Zhu, II Luican Computation of storage requirements for multi-dimensional signal processing applications. IEEE Trans. VLSI Syst. **14**(4), 447–460 (2007)
23. Q Hu, A Vandecapelle, M Palkovic, PG Kjeldsberg, E Brockmeyer, F Catthoor, in *Proc. Asia-South Pacific Design Autom. Conf.* Hierarchical memory size estimation for loop fusion and loop shifting in data-dominated applications, (Yokohama, Japan, 2006), pp. 606–611
24. G Talavera, M Jayapala, J Carrabina, F Catthoor, Address generation optimization for embedded high-performance processors: A survey. J. Signal Process. Syst., Springer. **53**(3), 271–284 (2008)
25. L Thiele, in *State-of-the-art in Computer Science*, ed. by P Dewilde. Compiler techniques for massive parallel architectures (Kluwer Acad. Publ., 1992)
26. A Schrijver, *Theory of Linear and Integer Programming*. (John Wiley, New York, 1986)
27. PH Clauss, in *Proc. European Conf. on Parallel Processing*. Handling memory cache policy with integer point counting, (Passau, Germany, 1997), pp. 285–293
28. CACTI 6.5. [Online]. Available: http://www.cs.utah.edu/~rajeev/cacti6/
29. S Verdoolaege, K Beyls, M Bruynooghe, F Catthoor, in *Compiler Construction: 14th Int. Conf*, ed. by R Bodik. Experiences with enumeration of integer projections of parametric polytopes, vol. 3443 (Springer, 2005), pp. 91–105
30. A Helal, F Balasa, in *Proc. 20th IEEE Int. Conf. on Control Systems and Computer Science*. Multithreaded signal-to-memory mapping algorithm for embedded multidimensional signal processing, (Bucharest, Romania, 2015), pp. 255–260
31. [Online]. Available: www.latticesemi.com
32. K Flautner, N Kim, S Martin, D Blaauw, T Mudge, in *Proc. Symp. Computer Architecture*. Drowsy caches: simple techniques for reducing leakage power, (2002), pp. 148–157
33. M De Berg, O Cheong, M van Krefeld, M Overmars, *Computational Geometry: Algorithms and Applications*. (Springer, 2010)
34. M Moonen, PV Dooren, J Vandewalle, An SVD updating algorithm for subspace tracking. SIAM J. Matrix Anal. Appl. **13**(4), 1015–1038 (1992)