

Reducing the Dynamic Energy Consumption in the Multi-Layer Memory of Embedded Multimedia Processing Systems*

Ilie I. Luican* Hongwei Zhu† Florin Balasa‡ Dhiraj K. Pradhan§

*Dept. of Computer Science, University of Illinois at Chicago, Chicago, Illinois, U.S.A. E-mail: iluica2@uic.edu

†ARM, Inc., Sunnyvale, California, U.S.A. E-mail: davzhu01@arm.com

‡Dept. of Computer Science, Southern Utah University, Cedar City, Utah, U.S.A. E-mail: fbalasa@suu.edu

§Dept. of Computer Science, University of Bristol, Bristol, United Kingdom E-mail: pradhan@cs.bris.ac.uk

Abstract – The memories in data-intensive signal processing systems – including video and image processing, artificial vision, real-time 3-D rendering, advanced audio and speech coding, medical imaging applications – have an important impact on the overall energy budget. This paper focuses on the reduction of the dynamic energy consumption in the memory subsystem, starting from the high-level algorithmic specification of the application. The approach to address this problem uses elements of the theory of polyhedra and relies on a variety of algebraic techniques specific to the data-flow analysis used in modern compilers.

1 Introduction

A typical architecture of an embedded signal processing system includes programmable hardware (e.g., DSP core), custom hardware (application-specific accelerator datapaths and logic), controller, and a distributed memory organization [6]. The on-chip memory subsystem 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 system [6]. 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. The reduction of dynamic energy consumption is not only achieved by partitioning the data between the on-chip SPM and the off-chip memory, but

also by partitioning the SPM into multiple blocks. The idea relies on the fact that, for a memory block, (a) the memory accesses are not uniformly distributed, (b) dynamic energy is consumed only during memory accesses, and (c) its cost is proportional to the size of the block times the total number of accesses. It is thus intuitive to split the virtual address space into multiple, independently accessed memory blocks in such a way that most of the accesses will occur into the smaller blocks and only few ones in the larger blocks.

With the scaling of the technology below $0.1 \mu\text{m}$, the static energy due to leakage currents has become increasingly important. While leakage is a problem for any transistor, it is even more critical for memories: their high density of integration translates into a higher power density that increases temperature, which in turn increases leakage currents significantly. As technology scales, the importance of static energy consumption increases even when memories are idle (not accessed). To reduce the static energy, proper schemes to put a memory block into a dormant (*sleep*) state with negligible energy spendings are required. These schemes normally imply a timing overhead: transitioning a memory block into and, especially, out of the dormant state consumes energy and time. Putting a memory block into the dormant state should be done only if the cost in extra energy and decrease of performance can be amortized. For dealing with dynamic energy, we are interested only in the total number of accesses, and not to their distribution over time. Introducing the *time* dimension makes the problem of energy reduction much more complex.

The storage allocation in a hierarchical organization (on- and off-chip) and the energy-aware partitioning of the SPM into several blocks must be complemented with a comprehensive solution for mapping the multidimensional arrays from the code of the application into the physical memories. This operation aims (a) to map the arrays into an amount of data storage as small as possible, (b) to use mapping functions simple enough in order to ensure an address generation hardware of a reasonable complexity, and (c) to ascertain that any distinct scalar signals (array elements) *simultaneously alive* be mapped to distinct storage locations.

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

1.1 Previous Works

The problem on energy-efficient assignment of signals to the on- and off-chip memories has been studied since the late nineties. These previous works focused on partitioning the arrays into *copy candidates* and on the optimal selection and mapping of these into the memory hierarchy [24, 5, 12]. The general idea was to identify the data (arrays or parts of arrays) that are most frequently accessed in each loop nest. Copying these heavily accessed data from the large off-chip memory to a smaller on-chip memory can potentially save energy (since most accesses will take place on the smaller copy and not on the large, more energy consuming, original array) and also improve performance. Many different possibilities exist for deciding on which parts of the arrays should be copy candidates and, also, for selecting among the candidates those which will be instantiated as *copies* and their assignment to the different memory layers. For instance, Kandemir *et al.* analyze and exploit the temporal locality by inserting local copies [13]. Their layer assignment builds a separate hierarchy per loop nest and then combines them into a single hierarchy. However, the approach lacks a global view on the (part of) arrays lifetimes in applications having imperfect nested loops. Brockmeyer *et al.* use the steering heuristic of assigning the arrays having the highest access number over size ratio to the cheapest memory layer first, followed by incremental reassignments [5]. Hu *et al.* can use *parts* of arrays as copies, but they typically use cuts along the array dimensions [12] (like rows and columns of matrices).

The energy-aware partitioning of an on-chip memory in multiple banks has been studied by several research groups, as well. Techniques of an explorative nature analyze possible partitions, matching them against the access patterns of the application [20], [7]. Other approaches exploit the properties of the dynamic energy cost and the resulting structure of the partitioning space to come up with algorithms able to derive the optimal partition for a given access pattern [4], [1]. Leakage-aware partitioning of memory structures is addressed at circuit-level – especially for caches. The cache-decay architecture turns off the cache lines during the time they are not used [15]. The drowsy cache architecture puts the cache lines into state-preserving low-power modes based on usage statistics [10]. These techniques, together with dynamic resizing, require the modification of the internal structure of caches, which are normally highly-optimized designs. Argyrides and Pradhan proposed techniques for soft error elimination, while keeping a low power consumption [2]. On a higher level of abstraction, Kandemir *et al.* exploit bank locality for maximizing the idleness, thus ensuring maximal amortization of the energy spent on memory re-activation [14]. Golubeva *et al.* proposed a trace-based architectural approach, considering both the dynamic and static energy consumption [11].

In order to efficiently map the multidimensional arrays into the

physical memory, De Greef *et al.* choose one of the canonical linearizations of the array, followed by a modulo operation that wraps the set of “virtual” memory locations into a smaller set of actual physical locations [9]. Instead of a 1-dimensional window in the linearized space of addresses as in [9], Tronçon *et al.* compute an m -dimensional bounding box for the simultaneously alive elements of an m -dimensional array [23] by computing maximal index differences in each dimension. Lefebvre and Feautrier, addressing parallelization of static control programs, developed an intra-array storage approach based on modular mapping as well [16]. Quilleré and Rajopadhye studied the problem of memory reuse for systems of recurrence equations [19]. In their model, the loop iterators first undergo an affine mapping before modulo operations are applied to the array indices. Darte *et al.* proposed a lattice-based mathematical framework for intra-array mapping, establishing a correspondence between valid linear storage allocations and integer lattices called *strictly admissible* relative to the set of differences of the conflicting indices [8].

1.2 Research Motivation

The motivation of this research is summarized as follows:

- *The lack of a general model for identifying those parts of arrays from a given application code that are more intensely accessed, in order to steer their assignment to different memory layers such the dynamic energy consumption be reduced, and also taking into account the relative lifetimes of signals in order to reduce the data storage on each hierarchical level of the memory subsystem.*

Such a model could be extended to steer also the partitioning of the memory on each hierarchical layer in order to further reduce the dynamic energy consumption. In a later phase, the static energy could be considered as well considering the distribution of the memory accesses over *time*. Note that the existent works use as input either a *memory trace*, which can be very inefficient for applications requiring a very long sequence of datapath instructions (as in the case of deep loop nests, where the ranges of the iterators are large), or overly constrained specifications in syntactical point of view (like specifications having only *perfect* nested loops).

1.3 Contributions

This paper presents a formal model based on lattices [21], which allows to identify with accuracy those parts of arrays from a given application code that are more intensely accessed for *read* or *write* operations. Storing on-chip these parts of arrays yields the highest reduction of the dynamic energy consumption in the hierarchical memory subsystem. The specifications are considered to be *procedural*, therefore the execution ordering is induced by the

loop structure and it is thus fixed.¹ Since the mathematical model is very general, the proposed approach is able to handle the entire class of *affine* specifications [6], the code being organized in sequences of loop nests having as boundaries linear functions of the outer loop iterators, conditional instructions where the conditions may be both data-dependent or data-independent (relational and/or logical operators of linear functions of loop iterators), and multi-dimensional signals whose array references have (possibly complex) linear indices. The model was tested for the time being assuming two memory layers (on-chip scratch-pad and off-chip memories), focusing on the reduction of the dynamic energy consumption due to memory accesses. Extensions of the model to the exploitation of memory banking, as well as taking also into account the leakage energy consumption, will be considered in the future.

In addition, this methodology solves in a consistent way the problem of mapping the multidimensional arrays to the data memory.² Similarly to [23], our approach computes bounding boxes for live elements in the index space of arrays, but, since this algorithm works not only for *entire* arrays, but also *parts* of arrays (like, for instance, array references or, more general, sets of array elements represented by lattices), this signal-to-memory mapping technique can be also applied in a multi-layer memory hierarchy [17].

The rest of the paper is organized as follows. The technical core of the paper is Section 2 which presents the polyhedral model used to maximize the reduction of the dynamic energy consumption in the hierarchical memory subsystem. Section 3 discusses implementation aspects and presents experimental results. Finally, Section 4 summarizes the main conclusions of this research.

2 Framework for the Reduction of the Dynamic Energy Consumption in a Hierarchical Memory Subsystem

Each *array reference* $M[x_1(i_1, \dots, i_n)] \dots [x_m(i_1, \dots, i_n)]$ 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 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 (or array) 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 expressions with integer coefficients of the loop iterators, the index space consists of one

¹The search space becomes much larger still when also the available freedom in loop organization is incorporated. If the original loop ordering is not optimally suited to exploit data locality, code transformations should be applied (like in [12], for instance) in an earlier phase to increase it.

²Although our methodology for memory management takes into account the mapping problem as well, this topic is only marginally addressed in this paper.

or several *linearly bounded lattices* (LBLs) [22]:

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

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.

Example 1: for ($i = 0; i \leq 2; i++$)

$$\text{for } (j = 0; j \leq 3; j++) \dots A[3i][5i + 2j] \dots$$

The index space of the array reference can be represented as

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

For simplicity of presentation, it will be assumed along this paper that each array reference can be represented as one lattice, but there are situations when more than one lattice are necessary (for instance, an array reference in the scope of the condition *if* ($i \neq j$) has two disjoint lattices, one for $i \geq j + 1$ and one for $i \leq j - 1$).

As $\mathbf{T} = \begin{bmatrix} 3 & 0 \\ 5 & 2 \end{bmatrix}$, $\mathbf{T}^{-1} = \frac{1}{6} \begin{bmatrix} 2 & 0 \\ -5 & 3 \end{bmatrix}$, $\mathbf{u} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}$, the index space of the array reference can be represented using the indexes x and y as coordinates: from $\mathbf{A} \cdot \mathbf{i} \geq \mathbf{b}$, it follows $\mathbf{A} \cdot \mathbf{T}^{-1} \cdot (\mathbf{x} - \mathbf{u}) \geq \mathbf{b}$, that is, the inequalities $6 \geq x \geq 0$, $18 \geq -5x + 3y \geq 0$, representing the quadrilateral in Fig. 1(a). Not all the lattice points in the quadrilateral have coordinates the index values of the array reference. Only the lattice points satisfying the divisibility conditions: $6 \mid 2x$ (or $3 \mid x$) and $6 \mid (-5x + 3y)$ belong to the index space. These divisibility conditions result from the necessity that the elements of the iterator vector $\mathbf{i} = \mathbf{T}^{-1} \cdot (\mathbf{x} - \mathbf{u})$ be integer.

2.1 Basic Operations with Bounded Lattices

Two operations are relevant in our context: the *intersection* and the *difference* of two LBLs. While the intersection of two LBLs was addressed also by other works (in different contexts, though) as, for instance, [22], the difference operation is far more difficult.

(1) *The intersection of two LBLs.*

Let $Lbl_1 = \{ \mathbf{x} = \mathbf{T}_1 \mathbf{i}_1 + \mathbf{u}_1 \mid \mathbf{A}_1 \mathbf{i}_1 \geq \mathbf{b}_1 \}$, $Lbl_2 = \{ \mathbf{x} = \mathbf{T}_2 \mathbf{i}_2 + \mathbf{u}_2 \mid \mathbf{A}_2 \mathbf{i}_2 \geq \mathbf{b}_2 \}$ be two LBLs derived from the same indexed signal, where \mathbf{T}_1 and \mathbf{T}_2 have obviously the same number of rows – the signal dimension. Intersecting the two linearly bounded lattices means, first of all, solving a linear Diophantine system³ $\mathbf{T}_1 \mathbf{i}_1 - \mathbf{T}_2 \mathbf{i}_2 = \mathbf{u}_2 - \mathbf{u}_1$ having the elements of \mathbf{i}_1 and \mathbf{i}_2 as unknowns. If the system has no solution, the intersection is empty. Otherwise, let

$$\begin{bmatrix} \mathbf{i}_1 \\ \mathbf{i}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{V}_1 \\ \mathbf{V}_2 \end{bmatrix} \mathbf{t} + \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \end{bmatrix}$$

be the solution of the Diophantine system. If the set of coalesced

³Finding the integer solutions of the system. Solving a linear Diophantine system was proven to be of polynomial complexity, all the known methods being based on bringing the system matrix to the Hermite Normal Form [21].

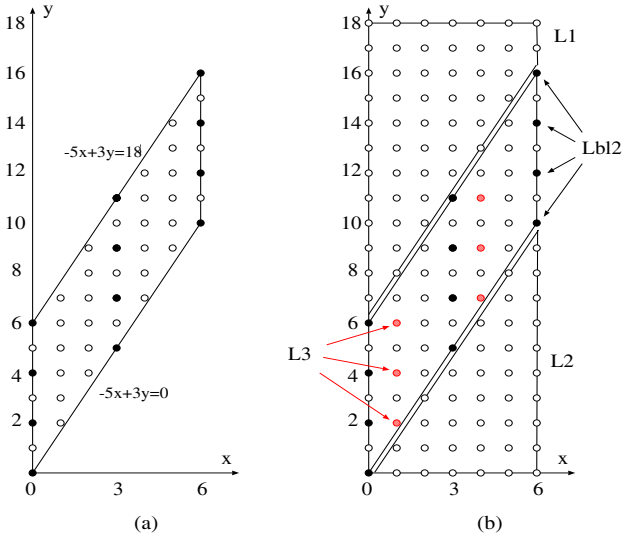


Figure 1: (a) The index space of the array reference in *Example 1*. (b) The difference $Lbl_1 - Lbl_2$, where the two lattices correspond to the array references in *Example 2*. Lbl_1 contains all the lattice points in the rectangle and Lbl_2 contains the 12 black points. The difference has (at least) 7 LBL components. Two components are $L1$ (all the lattice points in the upper quadrilateral) and $L2$ (all the lattice points in the lower triangle). Another LBL component $L3$ covers the 6 red points in the middle area.

constraints of the two LBLs

$\mathbf{A}_1 \mathbf{V}_1 \cdot \mathbf{t} \geq \mathbf{b}_1 - \mathbf{A}_1 \mathbf{v}_1$, $\mathbf{A}_2 \mathbf{V}_2 \cdot \mathbf{t} \geq \mathbf{b}_2 - \mathbf{A}_2 \mathbf{v}_2$ (2)
has integer solutions, then the intersection is a new LBL:

$$L_{bl_1} \cap L_{bl_2} = \{ \mathbf{x} = \mathbf{T}_1 \mathbf{V}_1 \cdot \mathbf{t} + (\mathbf{T}_1 \mathbf{v}_1 + \mathbf{u}_1) \mid \text{s.t. constraints (2)} \}$$

(2) *The difference of two LBLs.*

The difference is not always a single LBL (as an illustrative example will show below), so the goal is to determine an *LBL cover* of the difference, therefore a set of LBLs which union be equal to $L_{bl_1} - L_{bl_2}$. Since this operation is more difficult, it will be explained using an example:

Example 2: for ($k = 0; k \leq 6; k++$)
for ($l = 0; l \leq 18; l++$) $\dots A[k][l] \dots$
for ($i = 0; i \leq 2; i++$)
for ($j = 0; j \leq 3; j++$) $\dots A[3i][5i+2j] \dots$

$$L_{bl_1} = \{ x = k, y = l \mid 6 \geq k \geq 0, 18 \geq l \geq 0 \}$$

$$L_{bl_2} = \{ x = 3i, y = 5i + 2j \mid 2 \geq i \geq 0, 3 \geq j \geq 0 \}$$

$$= \{ 6 \geq x \geq 0, 18 \geq -5x + 3y \geq 0, 3 \mid x, 6 \mid -5x + 3y \}$$

The index space of L_{bl_2} (the 2nd array reference) was computed at *Example 1* and it is shown in Fig. 1(a).

Taking one of the 4 inequalities of L_{bl_2} and adding its negated inequality to the minuend L_{bl_1} will create an LBL which (if not empty) is disjoint from L_{bl_2} and is included in L_{bl_1} . For instance, negating the inequality $18 \geq -5x + 3y$ from L_{bl_2} , we obtain $19 \leq -5x + 3y$, or $19 \leq -5k + 3l$ with the iterators of L_{bl_1} . Adding it to the iterator space of L_{bl_1} , we obtain

$L_1 = \{ x = k, y = l \mid 6 \geq k \geq 0, 18 \geq l, 3l \geq 5k + 19 \}$ shown in Fig. 1(b).

Negating the inequality $-5x + 3y \geq 0$, we obtain

$L_2 = \{ x = k, y = l \mid 6 \geq k \geq 0, l \geq 0, 5k - 1 \geq 3l \}$. Negating the other inequalities ($6 \geq x \geq 0$) yields empty LBLs.

The other LBLs in the difference must violate at least one of the divisibility conditions $3 \mid x$ and $6 \mid (-5x + 3y)$. To obtain them, we simply replace the vector \mathbf{u}_2 in L_{bl_2} , keeping the same periodicity of the index space from L_{bl_2} (which here is 3 and 2 along the two axes), but doing a translation along the axes. Therefore, taking

$\mathbf{u}_2 = \begin{bmatrix} k_1 \\ k_2 \end{bmatrix} \neq \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, where $k_1 = 0, 1, 2$ and $k_2 = 0, 1$, five new LBLs are obtained. For instance, choosing $(k_1, k_2) = (1, 0)$, replacing $x = 3i + k_1$ and $y = 5i + 2j + k_2$ in the inequalities $6 \geq x \geq 0, 18 \geq -5x + 3y \geq 0$ of L_{bl_2} , we get $1 \geq i \geq 0, 3 \geq j \geq 1$. Therefore,

$L_3 = \{ x = 3i + 1, y = 5i + 2j \mid 1 \geq i \geq 0, 3 \geq j \geq 1 \}$ is included in $L_{bl_1} - L_{bl_2}$, and it is shown in Fig. 1(b) with 6 red points. The other four LBLs are:

$$L_4 = \{ x = 3i + 2, y = 5i + 2j \mid 1 \geq i \geq 0, 4 \geq j \geq 2 \}$$

$$L_5 = \{ x = 3i, y = 5i + 2j + 1 \mid 2 \geq i \geq 0, 2 \geq j \geq 0 \}$$

$$L_6 = \{ x = 3i + 1, y = 5i + 2j + 1 \mid 1 \geq i \geq 0, 3 \geq j \geq 1 \}$$

$$L_7 = \{ x = 3i + 2, y = 5i + 2j + 1 \mid 1 \geq i \geq 0, 4 \geq j \geq 2 \}$$

Note that the decomposition is minimal (although not unique): it is not possible to obtain a decomposition of $L_{bl_1} - L_{bl_2}$ with less than 7 LBLs for this example – which is a “difficult” one! In most of the practical cases encountered, the difference can be represented as only one LBL and, if this is the case, the algorithm (informally explained above) will find it. Otherwise, in the general case, the minimality cannot be guaranteed, unless all the combinations of inequalities can be negated, instead of selecting only one at a time. (This would increase the computation time, without practical benefits.)

2.2 Identifying Intensely Accessed Parts of Arrays

The array references of every signal can be analytically decomposed into *disjoint* lattices using the two operations described in the previous section – the *intersection* and the *difference* of lattices. The disjoint lattices of the 2-D signal A from the code displayed in Fig. 2(a) are the following (in non-matrix format to save space):

$$A_0 = \{ x = 32, y = j \mid 32 \leq j \leq 128 \}$$

$$A_1 = \{ x = 33, y = j \mid 32 \leq j \leq 128 \}$$

$$A_2 = \{ x = 34, y = j \mid 32 \leq j \leq 128 \}$$

$$A_3 = \{ x = 0, y = j \mid 0 \leq j \leq 160 \}$$

$$A_4 = \{ x = 66, y = j \mid 0 \leq j \leq 160 \}$$

$$A_5 = \{ x = 1, y = j \mid 0 \leq j \leq 160 \}$$

$$A_6 = \{ x = 65, y = j \mid 0 \leq j \leq 160 \}$$

$$A_7 = \{ x = i, y = j \mid 2 \leq i \leq 31, 0 \leq j \leq 160 \}$$

$$A_8 = \{ x = i, y = j \mid 35 \leq i \leq 64, 0 \leq j \leq 160 \}$$

$$A_9 = \{ x = i, y = j \mid 32 \leq i \leq 34, 0 \leq j \leq 31 \}$$

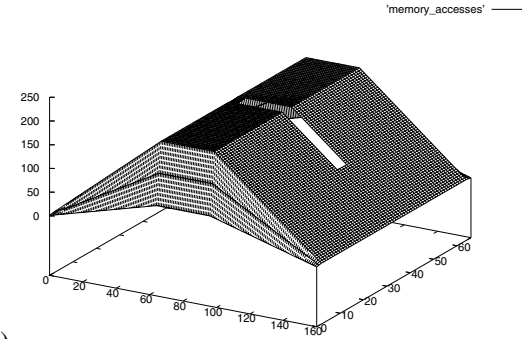
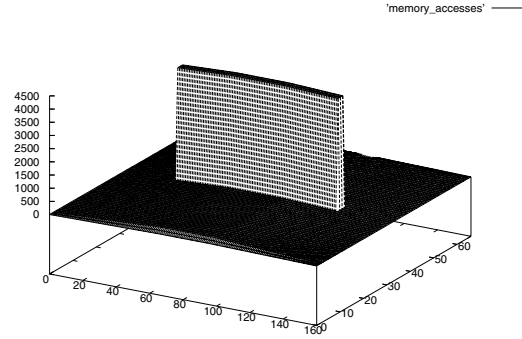
$$A_{10} = \{ x = i, y = j \mid 32 \leq i \leq 34, 129 \leq j \leq 160 \}$$

```

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

```

(a)



(b)

Figure 2: (a) An affine algorithmic specification derived from a motion detection kernel. (b) The distribution of memory accesses in the array (index) space of the 2-D signal $A[x][y]$ from the code.

Since A_0 is included in the array references $A[k][i]$ from all the three loop nests and it also coincides with the operand $A[32][j]$ in the first loop nest, the contributions of memory accesses for all these 4 array references must be computed. Since the lattice of the first array reference is $\{x = k, y = i \mid 128 \geq j \geq 32, 64 \geq k \geq 0, j + 32 \geq i \geq j - 32\}$, the expressions of the iterators mapping the array into A_0 are $\{j = t_1, k = 32, i = t_2 \mid 128 \geq t_1, t_2 \geq 32, t_1 + 32 \geq t_2 \geq t_1 - 32\}$. The size of this set is 5,249 [3], representing the number of memory accesses to A_0 due to the first array reference $A[k][i]$. Performing similar computations, the distribution of accesses in the whole array space is computed. For instance, the lattices $A_0, A_1,$ and A_2 – covering roughly the center of the array space of signal A – are the most intensely accessed array parts: 425,572 memory accesses (the total number of accesses being 2,452,654) for each of these three lattices (therefore, an average of 4,387.34 accesses for each A -element covered by them). Fig. 2(b) displays in 3-D the distribution of memory accesses for the whole array space $A[67][161]$ of the 2-D signal A . The area covered by the lattices $A_0, A_1,$ and A_2 is intensely accessed: it appears like a steep mountain crest in the first 3-D graph of the figure. The second graph displays the same memory access distribution, but with a scale limit of only 250 accesses for a better visibility of the intensity distribution for the other lattices: it can be seen better than in the first graph that

the area around the three lattices A_0, A_1, A_2 is not flat (that is, the other array elements are not uniformly accessed), but still the access intensity is much lower.

2.3 Estimating the Dynamic Energy Consumption in the Memory Subsystem

This evaluation model assumes a two-level memory hierarchy, where an SPM is used between the main memory and the processor core. The dynamic energy is computed based on the number of accesses to each memory layer. In computing the dynamic energy consumptions for the SPM and the off-chip memory, the CACTI 4.2 power model is used [25]. In general, the ratio between the energy consumed by an SPM access and the main memory varies between one and two orders of magnitude. The energy per access for an SPM is not a constant, but a size-dependent function – the energy per access tends to increase as the SPM size grows; however, for small SPM sizes up to a few KBytes the energy per access is relatively constant. Typical SPM and main memory energy values for *read* accesses are 0.048 nJ and 3.57 nJ, respectively (assuming memory sizes used in the illustrative example from the previous section). The dynamic energy values for *write* accesses are slightly higher.

The lattice-based framework presented so far allows to iden-

Application	# Array references	# Array elements	# Memory accesses	Dynamic energy reduction	CPU [sec]
Motion estimation	13	265,633	864,900	50.73 %	23
Durbin algorithm	21	252,499	1,004,993	73.25 %	28
SVD updating	85	3,045,447	29,500,000	46.51 %	37
Vocoder kernel	236	33,619	200,000	39.44 %	8

Table 1: Experimental results.

tify those parts of arrays more accessed than others. Different from other previous works (e.g., [5], [12]), the analysis of the copy candidates to be loaded in the SPM is not based on the arrays references and their cuts along the coordinates, but on the lattices that partition the array space exhibiting high-value memory accesses. It can be seen from Fig. 2(b) that the array reference $A[k][i]$ in the first loop nest (see Fig. 2(a)), covering the rows 0 to 64 of the array space, has zones accessed with very different intensities. Even cutting the array space of A along the dimensions x and y , the cut lines would intersect areas of very different intensities of accesses. The exploration of the partitions having high-value memory accesses clearly leads to a better reduction of the dynamic energy consumption. Assuming that storing each A -element requires 1 byte, with an SPM of only 291 bytes (the data memory necessary to store the lattices A_0 , A_1 , and A_2), the dynamic energy consumption caused by accesses to the array A would decrease from 8.75 mJ to 4.25 mJ according to the CACTI power model.

3 Experimental Results

A hierarchical memory allocation tool has been implemented in C++, incorporating the formal model described in this paper. For the time being, the tool supports only a two-level memory hierarchy, where an SPM is used between the main memory and the processor core. The dynamic energy is computed based on the number of accesses to each memory layer. As already mentioned, for computing the dynamic energy consumption in the two-layer memory subsystem, the CACTI 4.2 power model is used [25].

Table 1 summarizes the results of our experiments, carried out on a PC with a 1.85 GHz Athlon XP processor and 512 MB memory. The benchmarks used are algebraic kernels – like Durbin’s algorithm for solving Toeplitz systems, a singular value decomposition (SVD) updating algorithm [18] used in beamforming and Kalman filtering – and algorithms used in multimedia applications – like, for instance, an MPEG4 motion estimation algorithm, the kernel of a voice coding application – essential component of a mobile radio terminal. The table displays for each application the numbers of array references, scalar signals (array elements), and memory accesses; in addition, it shows the savings in dynamic energy versus the single-layer off-chip memory and the computation times. Our experiments show that the savings in dynamic energy consumptions are from 40% to over 70% relative to

energy used in the case of a flat memory design.

Our future work will focus on the reduction of the leakage energy consumption since this becomes dominant for 100 nm (and finer) technologies. The reduction of the static energy is based on the implementation of a low-leakage *sleep* state for the memory blocks. Since the intention is not to change the internal memory structure, the assumption is that the *sleep* state will be achieved by lowering the supply voltage (V_{dd}) to a value V_{dd}^{low} for which the memory cannot reliably be read or written, but there is a significantly lower leakage energy consumption. Transitions between the *sleep* and *active* states have an energy and time cost. Energy is actually spent only during the transitions from *sleep* to *active*, which causes the loading of the internal capacitances from V_{dd}^{low} to V_{dd} . Since the simulated data on caches shows that restoring the initial supply voltage from a “drowsy” state takes less than one cycle [10], the transition time penalty can be safely considered of 1 cycle. An optimization of the static energy can then be done within our framework, where the cost will comprise the leakage energies spent in the *sleep* and *active* states, and the energy spent per transition times the number of transitions.

4 Conclusions

This paper has presented a framework based on lattices which allows to partition the index space of arrays from data-dominated applications such that those array parts heavily accessed are identified and used as redundant data in scratch-pad memories in order to diminish the dynamic energy consumption due to memory accesses. This model can yield better savings of the dynamic energy consumption in a hierarchical memory subsystem than previous works that consider as potential copy candidates array references or cuts in the array space.

References

- [1] F. Angiolini, L. Benini, and A. Caprara, “An efficient profile-based algorithm for scratchpad memory partitioning,” *IEEE Trans. on CAD*, vol. 24, no. 11, pp. 1660-1676, Nov. 2005.
- [2] C. Argyrides and D.K. Pradhan, “Multiple upsets tolerance in SRAM memory,” to be published in *Proc. IEEE Int. Symp. on Circuits and Systems*, New Orleans LA, May 2007.

- [3] F. Balasa, H. Zhu, and I.I. Luican, "Computation of storage requirements for multi-dimensional signal processing applications," *IEEE Trans. on VLSI Systems*, vol. 15, no. 4, April 2007.
- [4] L. Benini, L. Macchiarulo, A. Macii, E. Macii, and M. Poncino, "Layout-driven memory synthesis for embedded systems-on-chip," *IEEE Trans. on VLSI Systems*, vol. 10, no. 2, pp. 96-105, April 2002.
- [5] E. Brockmeyer, M. Miranda, H. Corporaal, and F. Catthoor, "Layer assignment techniques for low energy in multi-layered memory organisations," *Proc. ACM/IEEE Design and Test in Europe*, pp. 1070-1075, Munich, Germany, March 2003.
- [6] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P. G. Kjeldsberg, T. Van Achteren, and T. Omnes, *Data Access and Storage Management for Embedded Programmable Processors*, Boston: Kluwer Acad. Publ., 2002.
- [7] S. Coumeri and D.E. Thomas, "Memory modeling for system synthesis," *IEEE Trans. on VLSI Systems*, vol. 8, no. 3, pp. 327-334, June 2000.
- [8] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," *IEEE Trans. Computers*, vol. 54, pp. 1242-1257, Oct. 2005.
- [9] E. De Greef, F. Catthoor, and H. De Man, "Memory size reduction through storage order optimization for embedded parallel multimedia applications", special issue on "Parallel Processing and Multimedia" (ed. A. Krikelis), in *Parallel Computing*, Elsevier, vol. 23, no. 12, pp. 1811-1837, Dec. 1997.
- [10] K. Flautner, N. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: Simple techniques for reducing leakage power," in *Proc. Symp. Computer Architecture*, pp. 148-157, May 2002.
- [11] O. Golubeva, M. Loghi, M. Poncino, and E. Macii, "Architectural leakage-aware management of partitioned scratch-pad memories," in *Proc. ACM/IEEE Design Automation and Test in Europe*, pp. 1665-1670, Nice, France, April 2007.
- [12] Q. Hu, A. Vandecapelle, M. Palkovic, P.G. Kjeldsberg, E. Brockmeyer, and F. Catthoor, "Hierarchical memory size estimation for loop fusion and loop shifting in data-dominated applications," in *Proc. Asia-South Pacific Design Automation Conf.*, pp. 606-611, Yokohama, Japan, Jan. 2006.
- [13] M. Kandemir, A. Choudhary, "Compiler-directed scratch-pad memory hierarchy design and management," in *Proc. 39th ACM/IEEE Design Automation Conf.*, pp. 690-695, Las Vegas, June 2002.
- [14] M. Kandemir, M.J. Irwin, G. Chen, and I. Kolcu, "Compiler-guided leakage optimization for banked scratch-pad memories," *IEEE Trans. on VLSI Systems*, vol. 13, no. 10, pp. 1136-1146, Oct. 2005.
- [15] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: Exploiting generational behavior to reduce cache leakage power," in *Proc. Symp. Computer Architecture*, pp. 240-251, June 2001.
- [16] V. Lefebvre and P. Feautrier, "Automatic storage management for parallel programs," *Parallel Computing*, vol. 24, pp. 649-671, 1998.
- [17] I.I. Luican, H. Zhu, and F. Balasa, "Signal-to-memory mapping analysis for multimedia signal processing," in *Proc. Asia & South-Pacific Design Automation Conf.*, pp. 486-491, Yokohama, Japan, Jan. 2007.
- [18] M. Moonen, P. V. Dooren, and J. Vandewalle, "An SVD updating algorithm for subspace tracking," *SIAM J. Matrix Anal. Appl.*, vol. 13, no. 4, pp. 1015-1038, 1992.
- [19] F. Quilleré and S. Rajopadhye, "Optimizing memory usage in the polyhedral model," *ACM Trans. Programming Languages and Syst.*, vol. 22, no. 5, pp. 773-815, 2000.
- [20] W. Shiue and C. Chakrabarti, "Memory exploration for low-power embedded systems," in *Proc. 35th ACM/IEEE Design Automation Conf.* pp. 140-145, June 1998.
- [21] A. Schrijver, *Theory of Linear and Integer Programming*, New York: John Wiley, 1986.
- [22] L. Thiele, "Compiler techniques for massive parallel architectures," in *State-of-the-art in Computer Science*, P. Dewilde (ed.), Kluwer Acad. Publ., 1992.
- [23] 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.), pp. 167-181, 2002.
- [24] S. Wuytack, J.-P. Diguët, F. Catthoor, and H. De Man, "Formalized methodology for data reuse exploration for low-power hierarchical memory mappings," *IEEE Trans. VLSI Syst.*, vol. 6, no. 4, pp. 529-537, Dec. 1998.
- [25] * * * , *CACTI 4.2* [Online]. Available: <http://quid.hpl.hp.com:9081/cacti/>