Formal Model of Data Reuse Analysis for Hierarchical Memory Organizations*

Ilie I. Luican Hongwei Zhu Florin Balasa

Dept. of Computer Science, University of Illinois at Chicago, Chicago, IL 60607, U.S.A.

Abstract – In real-time data-dominated communication and multimedia processing applications, due to the manipulation of large sets of data, a multi-layer memory hierarchy is used to enhance the system performance and also to reduce the energy consumption. Savings of dynamic energy can be obtained by accessing frequently used data from smaller memories rather than from large background memories. The optimization of the hierarchical memory architecture implies the addition of layers of smaller memories to which heavily used data can be copied. This paper presents a formal model for data reuse analysis which identifies those parts of arrays more intensely accessed, taking also into account the relative lifetimes of the signals. Tested on a two-layer memory hierarchy, this model led to savings in the dynamic energy from 40% to over 70% relative to the energy used in the case of a flat memory design.

1 Introduction

In advanced 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. This is due to the significant amount of data transfers to/from large and energy consuming off-chip data memories. The power cost can be reduced and the system performance enhanced by introducing an optimized custom memory hierarchy that exploits the temporal locality in the data accesses [2, 10].

Power savings can be obtained by accessing frequently used data from smaller memories rather than from large background memories. The optimization of the hierarchical memory architecture implies the addition of layers of smaller memories to which heavily used data can be copied. On one hand, this optimization must trade-off between the reduction of power consumption by accessing the data from smaller memories, and the increase of power consumption because of the additional transfers between memory layers; on the other hand, the optimization must trade-off between the reduction of power consumption due to memory fragmentation, and the increase in area (another significant component of the chip cost in data-dominated applications) and interconnect cost due to the additional memory necessary to store the copies, and also because of the additional area overhead (like addressing logic) due to the memory fragmentation.

The aim of the memory hierarchy design is to find the best solutions for these trade-offs at an early stage of the system design. Note that this problem is basically different from caching for performance [5, 9], where the question is to find how to fill the cache such that the data needed have been loaded in advance from the main memory.

Part of the research work focused on how to restructure the application code to make better use of the available memory hierarchy [8]. For instance, the importance of loop fusion (alone, or in combination with loop shifting) was revealed as the basic transformation for improving the data locality [6]. But since it was proven [4] that the search for optimal loop fusion for global array contraction is an NP-complete problem, heuristics based on data locality – which is a rather abstract measure – are used in the existing works.

A second major research direction of the previous works was the partitioning of the arrays into *copy candidates* and the optimal selection and mapping of these into the memory hierarchy [15, 1, 6]. The general idea is 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, [7] analyses and exploits the temporal locality by inserting local copies. 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. [1]

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

ICCAD'06, November 5-9, 2006, San Jose, CA

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. They take into account the relative lifetime differences between arrays ("inter in-place") and between the scalars covered by each array ("intra in-place"). However, it is not clear whether the copy candidates can be also *parts* of arrays instead of entire arrays (and if so, how they identify these parts) since the access patterns are, in general, not uniform. [6] can use *parts* of arrays as copies, using the index spaces of the array references and also their intersections.

The previous models of data reuse, even those claiming being *formal*, contain approximate determinations which may influence significantly the accuracy of the results. Typical approximations made in the name of "computation efficiency" [6] are done when, for instance, the sizes of copy candidates are computed (which are, actually, only *estimated*), or in the computation of the number of memory accesses, or in the computation of the number of misses (the number of data transfers directly from the farther memory layer). As a result, the sizes of the memory layers are *estimated* as well (rather than *computed*), even when the code of the application is procedural (that is, the loop organization is fixed and provides the execution order).

This paper introduces a formal model for data reuse analysis, by applying algebraic techniques specific to the data-flow analysis used in modern compilers. 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 [2], 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. This model identifies those parts of arrays which are more intensely accessed, taking into account the relative lifetimes of the signals both for each array and between distinct arrays (intra and inter in-place). The data reuse model was tested for the time being assuming two memory layers (scratch-pad[2] and off-chip memories), focusing on the reduction of the dynamic energy consumption due to memory accesses. Extensions of the model to an arbitrary number of memory layers, as well as improving performance in addition to energy savings will be considered in the future.

The rest of the paper is organized as follows. Section 2 presents the polyhedral data reuse model. Section 3 discusses the main ideas of the memory allocation and hierarchy layer assignment. Section 4 discusses implementation aspects and presents experimental results. Finally, Section 5 summarizes the main conclusions of this research.

[1] The 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 [6], for instance) in an earlier phase to increase it.

[2] Software-controlled SRAM or DRAM, more energy-efficient than caches.

2 The partitioning of the array space

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 space*. The iterator space signifies the set of all iterator vectors $\mathbf{i} = (i_1,\ldots,i_n) \in \mathbf{Z}^n$ in the scope of the array reference. The index (or array) space is the set of all index vectors $\mathbf{x} = (x_1,\ \ldots\ ,x_m) \in \mathbf{Z}^m$ of the array reference. When the indices of an array reference are linear expressions with integer coefficients of the loop iterators, the index space consists of one or several *linearly bounded lattices* (LBLs) [13]:

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

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

$$for\ (i=0;\ i\leq 2;\ i++)$$
$$\quad for\ (j=0;\ j\leq 3;\ j++)\quad \cdots A[3i+j][5i+2j]\cdots$$

the index space of the array reference can be represented as

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

For simplicity of presentation, it will be assumed along this paper that each array reference can be represented as one LBL (but an array reference in the scope of a condition $if\ (i \neq j)$ has two LBLs, one for $i \geq j+1$ and one for $i \leq j-1$).

The goal is to identify the parts of the arrays in the given algorithmic specification that are heavily accessed during the code execution. This can be accomplished (as it will be seen) by a partitioning of each index space into sets which are all LBLs.

2.1 The index space of an array reference

Let $\{\ \mathbf{x} = \mathbf{T}\cdot\mathbf{i}+\mathbf{u} \mid \mathbf{A}\cdot\mathbf{i}\geq\mathbf{b}\ \}$ be the linearly bounded lattice of a given array reference. This section will show how to model the index space of an array reference, that is, what are the relations satisfied by the coordinates $\mathbf{x}$ of the points in this set. After the theoretical part, illustrative examples will be provided.

For any matrix $\mathbf{T}\in\mathbf{Z}^{m\times n}$ having $rank\,\mathbf{T} = r$, and assuming the first r rows of $\mathbf{T}$ are linearly independent,[3] there exists a unimodular matrix $\mathbf{S}\in\mathbf{Z}^{n\times n}$ such that $\mathbf{T}\cdot\mathbf{S} = \begin{bmatrix} \mathbf{H}_{11} & \mathbf{0} \\ \mathbf{H}_{21} & \mathbf{0} \end{bmatrix}$, where $\mathbf{H}_{11}\in\mathbf{Z}^{r\times r}$ is a lower triangular matrix with positive diagonal elements, and $\mathbf{H}_{21}\in\mathbf{Z}^{(m-r)\times r}$ [12]. The block matrix is called the reduced Hermite form of matrix $\mathbf{T}$.

Let $\mathbf{S}^{-1}\mathbf{i} \stackrel{def}{=} \mathbf{j} \equiv \begin{bmatrix} \mathbf{j}_1 \\ \mathbf{j}_2 \end{bmatrix}$, where $\mathbf{j}_1,\ \mathbf{j}_2$ are r-, respectively $(n-r)$-, dimensional vectors. Then

$$\mathbf{x} = \mathbf{T}\mathbf{i}+\mathbf{u} = \mathbf{T}\mathbf{S}\mathbf{j}+\mathbf{u} = \begin{bmatrix} \mathbf{H}_{11} \\ \mathbf{H}_{21} \end{bmatrix}\mathbf{j}_1+\mathbf{u} \qquad (2)$$

[3] This assumption does not decrease the generality: it is done only to simplify the formulas, affected otherwise by a row permutation matrix.

Denoting $\mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix}$, and $\mathbf{u} = \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{bmatrix}$ (where $\mathbf{x}_1$, $\mathbf{u}_1$ are r-dimensional vectors), it follows that $\mathbf{x}_1 = \mathbf{H}_{11}\mathbf{j}_1 + \mathbf{u}_1$. As $\mathbf{H}_{11}$ is nonsingular (being lower triangular of rank r), $\mathbf{j}_1$ can be obtained explicitly:

$$\mathbf{j}_1 = \mathbf{H}_{11}^{-1}(\mathbf{x}_1 - \mathbf{u}_1) \tag{3}$$

The iterator vector $\mathbf{i}$ results with a simple substitution:

$$\mathbf{i} = \mathbf{S}\begin{bmatrix} \mathbf{j}_1 \\ \mathbf{j}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{S}_1 & \mathbf{S}_2 \end{bmatrix}\begin{bmatrix} \mathbf{H}_{11}^{-1}(\mathbf{x}_1 - \mathbf{u}_1) \\ \mathbf{j}_2 \end{bmatrix} = \mathbf{S}_1\mathbf{H}_{11}^{-1}(\mathbf{x}_1 - \mathbf{u}_1) + \mathbf{S}_2\mathbf{j}_2$$

where $\mathbf{S}_1$ and $\mathbf{S}_2$ are the submatrices of $\mathbf{S}$ containing the first r, respectively the last $n-r$, columns of $\mathbf{S}$. As the iterator vector must represent a point inside the iterator space $\mathbf{A} \cdot \mathbf{i} \geq \mathbf{b}$, it follows that:

$$\mathbf{A}\mathbf{S}_1\mathbf{H}_{11}^{-1}\mathbf{x}_1 + \mathbf{A}\mathbf{S}_2\mathbf{j}_2 \geq \mathbf{b} + \mathbf{A}\mathbf{S}_1\mathbf{H}_{11}^{-1}\mathbf{u}_1 \tag{4}$$

If $r < n$, the $n-r$ variables of $\mathbf{j}_2$ can be eliminated with the Fourier-Motzkin technique [3].

As the rows of matrix $\mathbf{H}_{11}$ are r linearly independent r-dimensional vectors, each row of $\mathbf{H}_{21}$ is a linear combination of the rows of $\mathbf{H}_{11}$. Then from (2), it results that there exists a matrix $\mathbf{C} \in \Re^{(m-r)\times r}$ such that[4]

$$\mathbf{x}_2 - \mathbf{u}_2 = \mathbf{C} \cdot (\mathbf{x}_1 - \mathbf{u}_1) \tag{5}$$

Taking into account that the elements of $\mathbf{j}_1$ must be integers, it follows (by multiplying and dividing the right member of (3) with $det\ \mathbf{H}_{11}$) that the points $\mathbf{x}$ inside the index space must supplementarily satisfy the divisibility constraints

$$det\ \mathbf{H}_{11} \mid \mathbf{h}_i^T(\mathbf{x}_1 - \mathbf{u}_1) \qquad \forall i = 1, \ldots, r \tag{6}$$

where $\mathbf{h}_i^T$ are the rows of the matrix with integer coefficients $det\ \mathbf{H}_{11} \cdot \mathbf{H}_{11}^{-1}$, and $a|b$ means "a divides b". According to (6), when $r = n$, the points $\mathbf{x}$ are uniformly spaced along the r linear independent coordinates, the size of gaps in these dimensions being equal to the diagonal elements of $\mathbf{H}_{11}$: if h_{ii} are the diagonal elements of matrix $\mathbf{H}_{11}$, it can be verified that the divisibility constraints (6) are not affected when $\mathbf{x}_1$ is subject to translations of vectors $\mathbf{v}_i = [0\ \cdots\ h_{ii}\ \cdots\ 0], \forall i = 1, \ldots, r$. Indeed, $\mathbf{h}_i^T(\mathbf{x}_1 - \mathbf{u}_1 + \mathbf{v}_i) = \mathbf{h}_i^T(\mathbf{x}_1 - \mathbf{u}_1) + \mathbf{h}_i^T\mathbf{v}_i = \mathbf{h}_i^T(\mathbf{x}_1 - \mathbf{u}_1) + det\ \mathbf{H}_{11}$.

The system of inequalities (4), the equations (5), and the divisibility conditions (6) characterize the index space of the given array reference. Several examples will illustrate the generality of this model.

Example 1: $for\ (i = 0;\ i \leq 2;\ i++)$
$for\ (j = 0;\ j \leq 3;\ j++)\quad \cdots A[3i][5i + 2j] \cdots$

Since $\mathbf{T}=\mathbf{H}_{11}=\begin{bmatrix} 3 & 0 \\ 5 & 2 \end{bmatrix}$, $\mathbf{H}_{11}^{-1}=\frac{1}{6}\begin{bmatrix} 2 & 0 \\ -5 & 3 \end{bmatrix}$, $\mathbf{u}=\mathbf{u}_1=\begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $\mathbf{S}=\mathbf{S}_1=\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ ($\mathbf{S}_2$, $\mathbf{j}_2$ do not exist since $n - r = 2 - 2 = 0$; $\mathbf{H}_{21}$, $\mathbf{x}_2$, $\mathbf{u}_2$ do not exist since $m - r = 2 - 2 = 0$), the inequalities (4) with $\mathbf{x}_1=\begin{bmatrix} x \\ y \end{bmatrix}$ are: $6 \geq x \geq 0$, $18 \geq -5x + 3y \geq 0$, representing the quadrilateral in Fig. 1. Not all the lattice points in the quadrilateral have coordinates the index values of the array reference. Only the lattice points satisfying also the divisibility conditions (6): $6 \mid 2x$ (or $3 \mid x$) and $6 \mid -5x + 3y$ belong to the index space. Note also that these lattice points, colored *black* in the figure, are uniformly spaced along the two axes Ox and Oy, the size of the gaps in these dimensions being 3 and 2, the diagonal elements of $\mathbf{H}_{11}$.

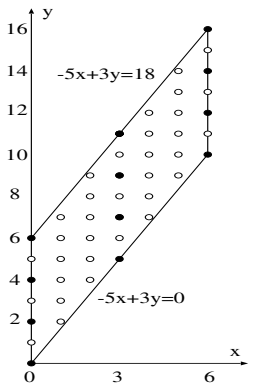

Figure 1: The index space of the array reference in *Example 1*.

Example 2: $for\ (i = 0;\ i \leq 2;\ i++)$
$for\ (j = 0;\ j \leq 3;\ j++)\quad \cdots A[3i+j][5i+2j] \cdots$

Since $\mathbf{T}=\begin{bmatrix} 3 & 1 \\ 5 & 2 \end{bmatrix}$, $\mathbf{H}_{11}=\mathbf{H}_{11}^{-1}=\begin{bmatrix} 1 & 0 \\ 2 & -1 \end{bmatrix}$, $\mathbf{S}=\mathbf{S}_1=\begin{bmatrix} 0 & 1 \\ 1 & -3 \end{bmatrix}$, the inequalities (4) are: $2 \geq 2x - y \geq 0$, $3 \geq -5x + 3y \geq 0$. Since $det\ \mathbf{H}_{11} = 1$, there are no divisibility conditions (6).

This representation model of the index space of an array reference is used in the decomposition algorithm (Section 2.2), specifically, in computing the difference between two LBLs.

2.2 The full decomposition of the array references into disjoint bounded lattices

The analytical decomposition of the array references of every signal into disjoint bounded lattices can be performed by a recursive intersection, starting from the array references in the code. 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, [13], the difference operation is far more difficult. Because of lack of space, this operation will be described elsewhere. Let S be a multi-dimensional signal in the algorithmic specification. A high-level pseudo-code of the LBL decomposition is as follows:

for all the array references of signal S
select an array reference and let Lbl_1 *be its representation;*

[4]The coefficients of matrix $\mathbf{C}$ are determined by backward substitutions from the equations: $\mathbf{H}_{21}.\text{row}(i) = \sum_{j=1}^{r} c_{ij} \cdot \mathbf{H}_{11}.\text{row}(j)$ for any $i = 1, \ldots, m-r$.

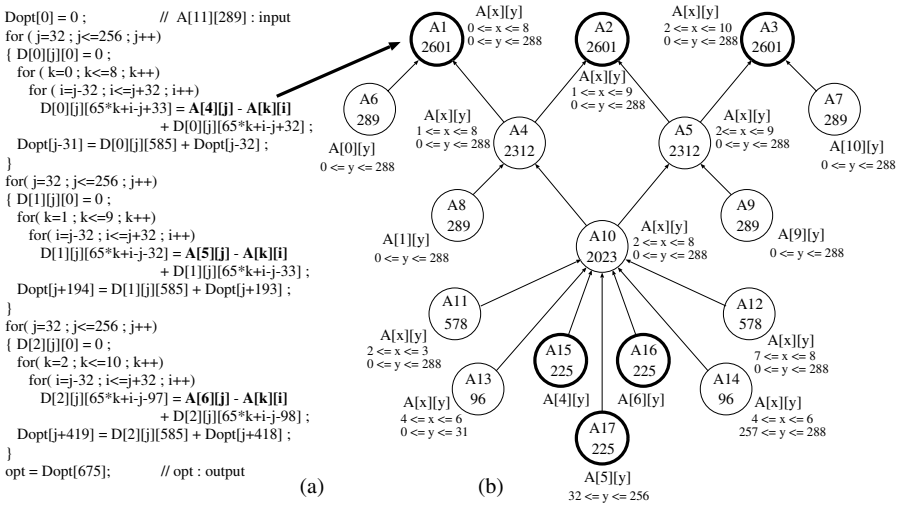

Figure 2: (a) Illustrative example. (b) Inclusion graph resulted from the partitioning of the index (array) space of signal A; the arcs in the graph show the inclusion relations between the sets; the weights of the nodes are the number of covered scalars. The index space of each node is represented, as explained in Section 2.1, by inequalities (4) (equality and divisibility conditions are not necessary here).

for all the current disjoint LBLs of the signal S
select an LBL, let it be called Lbl_2;
compute $Lbl_1 \cap Lbl_2$;
if the intersection is not empty
then compute $Lbl_1 - (Lbl_1 \cap Lbl_2)$ and $Lbl_2 - (Lbl_1 \cap Lbl_2)$;
update the LBL collection of S and their inclusion graph;
repeat the above operations till no new LBL is created;
end for;
end for;

The inclusion graph is a directed acyclic graph whose nodes are LBLs, and the arcs denote inclusion relations of the respective sets. This graph is used on one hand to speed up the decomposition (for instance, if the intersection $Lbl_1 \cap Lbl_2$ results to be empty, there is no sense of trying to intersect Lbl_1 with the LBLs included in Lbl_2 since those intersections will be empty as well), and on the other hand, to determine the structure of each array reference in terms of disjoint LBLs.

Figure 2(b) shows the result of the decomposition for the 2-dimensional signal A from the illustrative example in Fig. 2(a). The bold circles represent the 6 array references of A from the code. The 11 leaves of the inclusion graph represent the disjoint LBLs that partition the index space of signal A. Each array reference in the code is either a disjoint LBL itself (like $A15$, $A16$, and $A17$), or it can be written as a union of disjoint LBLs (e.g., $A1 = A6 \cup A8 \cup A11 \cup \ldots \cup A17$).

3 Hierarchical memory layer assignment

The decomposition of the index space of each multi-dimensional signal allows to compute the number of memory accesses when addressing the different parts of the arrays. If, for instance, the number of *read* accesses of a certain partition (*leaf* in the inclusion graph) is desired, the following computation scheme is used:

#accesses = 0;
for all the array references (operands) including the partition
select an array reference and find the expressions of
the iterators mapping the array reference to the partition;
#accesses += size of this set;
end for;

Example: Compute the number of memory (*read*) accesses to the partition $A17$ (see Fig. 2(b)).

Since $A17$ is included in the array references $A[k][i]$ from all the three loop nests (i.e., $A1$, $A2$, $A3$ in Fig. 2(b)), and also it coincides with the operand $A[5][j]$ in the second loop nest, the contributions of the 4 array references must be computed. Since the LBL of the partition is $\{x = 5,\ y = t \mid 256 \geq t \geq 32\}$ and the LBL of the first array reference is $\{x = k,\ y = i \mid 256 \geq j \geq 32,\ 8 \geq k \geq 0,\ j + 32 \geq i \geq j - 32\}$, the expressions of the iterators mapping the array into $A17$ are $\{j = t_1,\ k = 5,\ i = t_2 \mid 256 \geq t_1,\ t_2 \geq 32,\ t_1 + 32 \geq t_2 \geq t_1 - 32\}$. The size of this set is 13,569 [16].

The contributions of the other two array references $A[k][i]$ are also 13,569 accesses each. Since the contribution of $A[5][j]$ is 131,625 accesses, the total number of accesses is 172,332.

The potential benefit of loading a copy of a partition into the scratch-pad memory (SPM) is quantified by a *gain factor* $GF = (\#accesses - \#misses)/\#partition\_size$, similar as in [6]. The number of misses refers to the partitions that are also written. In order to keep the copy partition in the SPM consistent, when a *write* occurs, that data is copied from the off-chip memory to the SPM, and this is called a *miss*.

In Fig. 3 the gain factors are indicated on the index space of signal A, the darker areas being those parts more heavily accessed. Note that the memory accesses are not uniformly distributed inside the partitions. (For instance, $A[0][48]$ in the partition $A6$

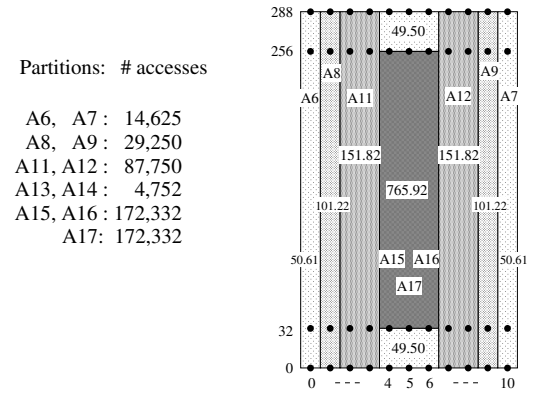

Figure 3: (a) The partitions of the index space of the signal A and their number of memory accesses, and (b) the gain factors of the different parts of the index (array) space of A. The darker partitions are more heavily accessed.

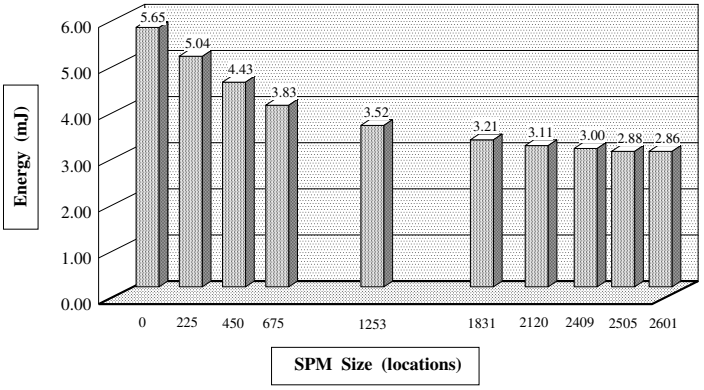

Figure 4: Variation of the dynamic energy consumption with the SPM size for the illustrative example in Fig. 2(a).

is accessed 49 times, whereas $A[0][148]$ is accessed 65 times.) However, this approach allows to identify those parts of arrays more accessed than others. What is different from other previous works, 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 partitions of the array space exhibiting high-value gain factors. It can be seen from Fig. 3(b) that the array reference $A[k][i]$ in the first loop nest, covering the columns 0 to 8 of the array space, has zones accessed with very different intensities. Even cutting along the dimensions x and y, the cut lines would intersect areas of very different gain factors. The exploration of the partitions having high gain factors leads to a better reduction of the dynamic energy, as shown in Section 4.

Note that every element of the arrays D and $Dopt$ is accessed exactly once for *reading* and once for *writing*. Then, the gain factors of the partitions of D and $Dopt$ are all zero since the number of *read* accesses equals the number of misses. Therefore, these partitions will not be chosen as copy candidates: loading (parts of) D and/or $Dopt$ in the SPM would not help reduce energy consumption, but would actually increase it.

The mapping of the signal partitions to the off-chip and scratch-pad memories is done using the model from [14]. According to it, for each n-dimensional array A a window $(w_1, \ldots, w_n)$ is computed; any access to an element of the array $A[exp_1] \ldots [exp_n]$ is redirected to $A[exp_1 \bmod w_1] \ldots [exp_n \bmod w_n]$ (relative to a base address). The window is computed such that two distinct array elements alive should not be mapped by the modulo operations to the same location. What is different in our case is that mapping windows are computed not only for the whole arrays, but also for the copy partitions to be loaded in the SPM (their windows being typically "smaller" than the corresponding array window).

4 Experimental results

A hierarchical memory allocation tool has been implemented in C++, incorporating the the data reuse analysis 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. In computing the dynamic energy consumptions for SPM and main memory, the CACTI power model is used [11]. 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). The dynamic energy values for *write* accesses are slightly higher.

Note that, for the time being, the leakage energy was not taken into account in the current computations, in part because of the lack of accurate models and in part because the optimization should be extended to take into account performance as well, besides energy consumption (leakage energy being spent as long memory is powered on, whereas dynamic energy is expanded only when an access occurs). Since leakage becomes the dominant part of energy consumption for 0.10 μm (and finer) technologies, one of the future developments we are considering is to take leakage into account as well.

Fig. 4 displays the dependence of the dynamic energy consumption as a function of the SPM size. The first bar is the reference and corresponds to a "flat" memory design, in which all operands have to be retrieved from the main memory. The second bar shows the energy used when the partition $A15$ (of size 225 and gain factor 765.92) is copied from the main memory to the SPM and it is accessed afterwards from there. Note that placing $A15$, $A16$, and $A17$ in the SPM leads to 30% energy savings for an SPM size of 675 locations (the 4th bar). The energy overhead due to the copy operations between the memory layers is taken into account, but its value is negligible since the gain factors have high values. Furthermore, this approach allows savings in the SPM size, too: since some of the copy candidates have different lifetimes, different such

Application	#Array refs.	#Scalars	#Mem. accesses	Mem. size	Energy [μJ]	SPM size	Energy saved	CPU [s]
Motion est.	13	265,633	864,900	2,465	3,088	1,416	50.73 %	23
Durbin alg.	21	252,499	1,004,993	1,249	3,588	764	73.25 %	28
SVD updating	85	3,045,447	29,500,000	34,950	105,315	12,672	46.51 %	37
Vocoder	236	33,619	200,000	11,890	714	3,879	39.44 %	8
Dyn. prog.	3,992	21,082,751	83,834,000	124,751	299,287	27,316	56.14 %	47

Table 1: Experimental results.

copies can share the same memory locations. For instance, the partitions $A6$ and $A9$ have disjoint lifetimes, hence $A9$ can replace $A6$ in the SPM without any increase of the SPM size, this being also the case for $A8$ and $A7$. Hence, the difference between the size of array A (3179 locations) in the main memory and the SPM size of only 2601 locations. Due to the signal-to-storage mapping model based on lifetime analysis, the main memory needs only 3181 locations, although the number of scalars in the illustrative example is 399,405. In addition, using also an SPM of size 2601 locations, the saving in dynamic energy consumption is over 45% (Fig. 4).

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) and algorithms used in multimedia applications (like, for instance, an MPEG4 motion estimation algorithm). The table displays the numbers of array references, scalar signals, and memory accesses; the main memory size (in storage locations) and the dynamic energy consumption if there is only one memory layer; the SPM size and the savings in dynamic energy versus the single-layer memory; the CPU 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. For a better evaluation of our data reuse model, another strategy of selecting the copy candidates – based on array references and cuts along their dimensions – has been tested as well. Although that latter model produced significant energy savings as well, our model let to better savings by 20%-30% (e.g., 24.65% for the motion estimation benchmark).

5 Conclusions

This paper has presented a formal model data reuse analysis which allows to partition the index space of arrays in 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 let to energy savings of, typically, 40-70% and proved to be better than a more classic model based on array references and their cuts along the main dimensions.

References

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

[2] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, A. Vandecapelle, *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design*, Kluwer Academic Publishers, Boston, 1998.

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

[4] A. Darte, "On the complexity of loop fusion," *Parallel Computing*, vol. 26, no. 9, pp. 1175-1193, 2000.

[5] J.Z. Fang, M. Lu, "An iteration partition approach for cache or local memory thrashing on parallel processing," IEEE Trans. on Computers, vol. C-42, no. 5, pp. 529-546, May 1993.

[6] Q. Hu, A. Vandecapelle, M. Palkovic, P.G. Kjeldsberg, E. Brockmeyer, F. Catthoor, "Hierarchical memory size estimation for loop fusion and loop shifting in data-dominated applications," *Proc. Asia-South Pacific Design Automation Conf.*, pp. 606-611, Yokohama, Japan, Jan. 2006.

[7] M. Kandemir, A. Choudhary, "Compiler-directed scratch-pad memory hierarchy design and management," *Proc. 39th ACM/IEEE Design Automation Conf.* pp. 690-695, Las Vegas, June 2002.

[8] M. Kandemir, G. Chen, F. Li, "Maximizing data reuse for minimizing space requirements and executive cycles," *Proc. Asia-South Pacific Design Aut. Conf.*, pp. 808-813, Yokohama, Japan, Jan. 2006.

[9] N. Manjiakian, T. Abdelrahman, "Reduction of cache conflicts in loop nests," Tech. Report CSRI-318, Univ. Toronto, Canada, 1995.

[10] P.R. Panda, N. Dutt, A. Nicolau, "On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems," *ACM Trans. on Design Automation of Electronic Systems*, Vol. 5, No. 3, pp. 682-704, July 2000.

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

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

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

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

[15] S. Wuytack, J.-P. Diguet, F. Catthoor, 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.

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