Compiler-Based Approach for Exploiting Scratch-Pad in Presence of Irregular Array Access

M. J. Absar† and F. Catthoor†
†IMEC vzw., Katholieke Universiteit Leuven, Belgium.
{javed.absar, francky.catthoor}@imec.be

Abstract

Scratch-pad memory is becoming an important fixture in embedded multimedia systems. It is significantly more efficient than the cache, in performance and power, and has the added advantage of better timing-predictability. Current techniques for the management of the scratch-pad are quite mature in the case of arrays accessed in a regular fashion, i.e. inside nested-loop by index expressions which are affine functions of the loop-iterators. Many multimedia codes, however, also use arrays as subscripted variables in the index expression of other arrays, thereby making the access pattern irregular. Existing techniques fail in such cases, bringing down the performance. In this paper, we extend the framework that exists today, to the case of irregular access. We provide a clear and precise compiler-based technique for analyzing irregular array-access, and efficiently mapping such arrays to the scratch-pad. On the average, 20% reduction in energy consumption, for a set of realistic applications, was achieved using our methods [1].

1. Introduction

Multimedia and signal processing embedded systems access large amounts of data, abstracted as multi-dimensional arrays, inside multilevel nested-loop. Such software include audio, video, image processing, encryption-decryption and network software, among others. Optimization of the memory access, therefore, forms an important step in the power-reduction and performance-enhancement of such applications [3].

Scratch-pad memory (SPM), a relatively small (2-4 KB) on-chip data-memory, is becoming popular these days. Being a memory without any control hardware, it operates at a much lower cost compared to the cache.

In the cache, decisions such as which data-set should continue to reside in the cache, and which ones must be evicted to make room for new ones, are made by the hardware. On the other hand, data-movement to and from the SPM happens under program control. This implies that the compiler or the software developer must identify which arrays, and which section therein, should be mapped to the SPM at any point.

Today, in the industry, this optimization is largely done manually. This makes it difficult to keep track of global trade-offs, and requires significant effort for exploration and verification.

[1]This work was partly funded by STMicroelectronics, Asia Pacific Pte Ltd, under PhD. Fellowship Program.

In the past, investigation has been done [8] [6] of techniques for SPM-management of arrays indexed using affine functions of the loop-iterators. We identify this type of array-access as *direct-indexing*. Direct-indexing is also sometimes referred to as regular-access in the current literature. Many multimedia codes, however, also index arrays using other arrays. Current techniques map these *indirectly-indexed* arrays either, as a whole [12] onto the SPM or, reject it altogether if it does not fit completely in the SPM. Not incorporating the flexibility of moving only the relevant parts of the indirectly-indexed multidimensional-arrays onto the SPM, implies a sub-optimal solution. The space in the SPM is limited and one array occupies it at the expense of others.

In this paper, we identify, for indirectly-indexed arrays, the critical portions that need to be on the SPM at different instances of time. We propose a new framework for representing indirectly-indexed arrays. This framework builds upon the existing ones [7] [14] for directly-indexed arrays, by an important extension. Using a cost-model, we perform a trade-off exploration between the size of the array that must be moved to the SPM, and the resulting benefit from any increased reuse of that data. The right loop level, for a multilevel loop, at which the transfer-code should be inserted is analyzed to maximize reuse. To our best knowledge, this is the first work that investigates dynamic-management of SPM for arrays that are indirectly-indexed.

2. Motivating Examples

```
for( i = 0 ; i < N ; i++)
 for( j  = 0; j < M ; j++ )
  for( k = 0 ; k < 16 ; k++ )
   for( l = 0 ; l < 16 ; l++ )
 ...= image[ 16*i + x[i][j] + k ]
           [ 16*j + y[i][j] + l ];
```

The code above is from QSDPCM [13], an inter-frame image compression algorithm. In the code, three arrays - 'x', 'y', and 'image' - are referenced. These three, therefore, form potential candidates for mapping to the SPM. Arrays 'x' and 'y' (motion-vectors) are relatively small in size (N*M). So they can be migrated completely to the SPM. Also, since they are indexed in a regular manner (only via iterators), we know precisely, at compile-time, the row-number of each of them that would be accessed for a given value of 'i'. Therefore, it is also possible to map only a single row of 'x' and 'y', for

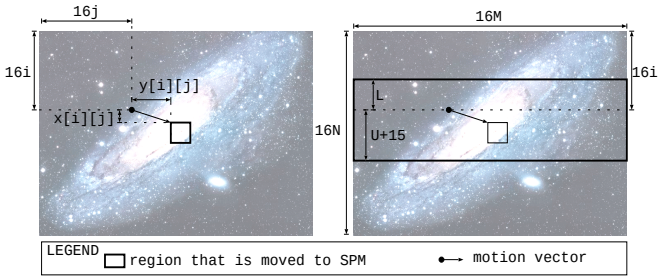

Fig. 1. Migrating the relevant section of an, indirectly-indexed, image-data onto SPM.

a given value of 'i'. As the computation moves to the next value of 'i', the next row of 'x' and 'y' can be moved to the SPM, overwriting the previous one. This is one way of mapping regularly accessed arrays using data-space tiling [8].

Next, consider the mapping of 'image' to the SPM. Since it is indirectly-indexed, current techniques would try to map it as a whole. But that may not be always possible (e.g. a 256×256 image would need $64KB$ of SPM). Observe that for a given value of 'i' and 'j', the 16×16 block in the 'image' that is accessed is actually precisely known at (and only at) run-time. Therefore, one possible mapping of 'image' to SPM is by copying the 16×16 block inside the loop of 'i' and 'j', as shown in the code below:

```
for( i = 0 ; i < N ; i++)
 for( j  = 0; j < M ; j++ )
 { spm_image[0:15][0:15] <-
   image[ 16*i + x[i][j] + (0:15)]
        [ 16*j + y[i][j] + (0:15)];
   for( k = 0 ; k < 16 ; k++ )
    for( l = 0 ; l < 16 ; l++ )
     ...= spm_image[k][l];}
```

In the above code, the symbol *dest* <- *src* means that the specified set of elements are moved from the external memory (*src*) to the SPM (*dest*). This can be done efficiently using a DMA (Direct-Memory Access) controller. Fig. 1 (left-side) shows pictorially this selection process. The SPM-mapping of 'image', as described so far, exploits only spatial reuse. In practice, however, the values that the elements of 'x' hold, is much smaller than the dimensions of 'image'. This information enables our algorithm to do a much better mapping, with potentially high temporal reuse. Assume that the range of values in 'x' is limited to $[L, U]$. Fig. 1 (right-side) shows for a given 'i', the region of 'image' that should be mapped to the SPM. All access to 'image' for that value of 'i' would, therefore, always result in hit to the SPM. Temporal reuse resulting from overlapping blocks are then well exploited. Simplified code for the transfer is shown below:

```
for( i = 0 ; i < N ; i++)
{ spm_image[0:(U-L+15)][0:(16*M-1)] <-
   image[16*i+(L:(U+15))][0:(16*M-1)];
  for( j  = 0; j < M ; j++ )
   for( k = 0 ; k < 16 ; k++ )
    for( l = 0 ; l < 16 ; l++ )
```

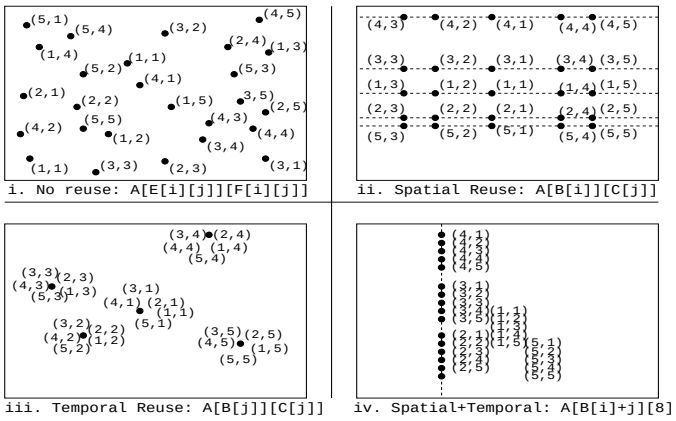

Fig. 2. Examples of compile-time discernible spatial and temporal reuse in indirectly-indexed arrays.

```
...= spm_image[  -L  + x[i][j] + k ]
               [ 16*j + y[i][j] + l ];}
```

We now show some general cases where spatial and temporal reuse can (or cannot) be identified at compile-time. We show four different access patterns of the same array $A[20][30]$. Array A is accessed inside a two-level nested-loop: $for(i = 1...5)for(j = 1...5)... = A[\cdots][\cdots]$. In Fig 2.i, A is referenced as $A[E[i][j]][F[i][j]]$. The rectangle in Fig. 2.i represents the 2-D data-space of A. The pair of numbers (i, j) beside each black dot (element of A) shows points in the iteration-space at which that particular element of A is accessed. Fig 2.i has low compile-time detectable reuse, because for each value of i and j, a potentially different row of A (via array E) and a different column of A (via array F) can be selected. In Fig. 2.ii, A is referenced as $A\,[B[i]]\,[C[j]]$. In this case, as long as i remains constant, the same row of A would be accessed, independent of the contents of C. Therefore, before the start of each j-loop, the entire row $A\,[B[i]]\,[0:29]$ can be copied to the SPM. Fig. 2.iii shows an instance of temporal reuse. With the reference as $A\,[B[j]]\,[C[j]]$, the same set of elements of A are reused over each value of i. Fig. 2.iv, with reference $A[B[i]+j][15]$, shows a case where both temporal and spatial reuse exist. Since the number of access (25) exceeds the distinct number of elements accessed (20), we clearly have reuse (pigeon-hole principle). Fig. 2.iv shows one possible reuse pattern.

Exactly which elements are reused may not be known at compile-time. However, even without that precise information, in many cases, as in the above examples, it is often clear at compile-time, itself, that reuse exists. This is often sufficient to enable a good SPM-mapping.

3. Related Work

Techniques for improving the performance of hierarchal memories in the case of regular (directly-indexed) array access have been deeply studied. Over a decade ago, Utpal Banerjee [2] showed that unimodular matrices could be used to represent loop transformations, such as interchange, skewing and reversal, in a mathematically elegant form. Wolf and Lam

have shown that using unimodular matrices simplifies data-dependence checks [14] when a nested-loop undergoes a series of loop-transformations, each instrumented as a unimodular transformation. Lam and others use loop-transformations to take advantage of any data-reuse. That is, loop structure is changed so that all the consumption of the same data is moved closer in time. Though proposed with cache-based system in mind, temporal locality enhancement is a fundamental step in improving code for any hierarchical memory structure, including scratch-pad based systems.

In addition to changing the loop-structure, which affects all arrays inside the loop sometimes in a conflicting way, several researchers, in the recent past, have focused on applying data transformations to improve spatial locality for the cache [7] [11]. Kandemir proposed an efficient compiler-based technique for mapping applications to the scratch-pad [8]. His technique, however, applies only to arrays with regular access.

The Chaos Group has studied partitioning and reorganization of data at run-time [4], in order to balance computational load across processors. They introduce the concept of *inspector-executor*. The *inspector* first analyzes the pattern of access, and then the *executor* changes the loop-structure (computation) and layout to exploit the reuse. The analysis, grouping and loop transformations are all performed at run-time. Using this technique, problems such as ocean simulation, n-body problem and molecular dynamics [5] can be optimized. This technique, however, requires expensive run-time processing to isolate temporal and spatial locality. Run-time analysis is unavoidable in some cases. But based on our observation that multimedia codes, even those with irregular access, can be quite well analyzed at compile time for locality, we take a different approach in this paper. A more thorough analysis is possible in our case, because it is performed at compile-time and not at run-time.

4. Irregularity over an Iterator

In this section, we describe the concept of regularity and irregularity of an array over an iterator. We use this concept, later, to optimize the SPM-mapping of indirectly-indexed arrays. Our context is a nested-loop represented by an iteration vector, $\vec{I}$, which contains the loop iterators from the outermost position to the innermost. The arrays are accessed inside the body of the loop-nest. Each array may contain other arrays in its index expression, in addition to the iterators and constants. We use Example 3 below to explain our representation.

```
EXAMPLE 3
for(i = ...) for(j = ...) for(k = ...)
 ... = A[ B[ C[k][k+1] + D[j] ] + i ][i+j]
```

Definition (Indirectly-Indexed Array) 1: An array is defined as indirectly-indexed if its index expression contains one or more arrays. Otherwise, it is termed as directly-indexed. $\diamond$

In the reference $B\,[C[k][k+1] + D[j]]$, the arrays C and D are directly-indexed, while array B is indirectly-indexed. In the current literature [7], a directly-indexed array such as $C[k][k+1]$, is represented as:

$$\Re_C = R_C\vec{I} + \vec{o}_C = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

We extend this representation to incorporate indirect-indexing. We consider an m-dimensional array X, with reference $\Re_X$, as composed of three components:

- $\vec{S}_X$: an $m \times 1$ vector of array elements that indirectly-index X. Each element of $\vec{S}_X$ is an affine function of such arrays.
- $R_X\vec{I}$: represents the iterators involved in the direct-indexing of X. For an m-dimensional array in an n-level nested-loop, R will be an $m \times n$ matrix.
- $\vec{o}_X$: $m \times 1$ vector of constants.

The arrays in $\vec{S}_X$ have these three same components, as well, in their reference.

The reference $C[k][k+1]$ (directly-indexed) is represented as: $\Re_C = \underbrace{\begin{bmatrix} 0 \\ 0 \end{bmatrix}}_{\vec{S}_C} + \underbrace{\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix}}_{R_C\vec{I}} + \underbrace{\begin{bmatrix} 0 \\ 1 \end{bmatrix}}_{\vec{o}_C}$. Reference $B[C[k][k+1] + D[k]]$ (indirectly-indexed) is expressed as:

$$\Re_B = \underbrace{\begin{bmatrix} C\langle\Re_C\rangle + D\langle\Re_D\rangle \end{bmatrix}}_{\vec{S}_B} + \underbrace{\begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix}}_{R_B\vec{I}} + \underbrace{\begin{bmatrix} 0 \end{bmatrix}}_{\vec{o}_B}$$

Operator $\langle\cdot\rangle$ takes as input a reference (column) vector, and returns the corresponding element. For example, $C\langle \begin{bmatrix} k \\ k+1 \end{bmatrix} \rangle$ translates to element $C[k][k+1]$.

We now give the generic representation for any directly or indirectly-indexed access. Consider an m-dimensional array X, referenced as $\Re_X$, inside an n-level loop-nest. In the p^{th} dimension of X, the q^{th} indirectly-indexing array is denoted as $U_{p,q}$. For example: $X[U_{11}\langle\cdot\rangle + U_{12}\langle\cdot\rangle \ldots][U_{21}\langle\cdot\rangle \ldots]$. The reference of $U_{p,q}$ is denoted as $\Re_{U_{p,q}}$. A generic reference is, therefore, represented as:

$$\Re_X = \begin{bmatrix} \sum_q U_{1,q}\langle\Re_{U_{1,q}}\rangle \\ \sum_q U_{2,q}\langle\Re_{U_{2,q}}\rangle \\ \vdots \end{bmatrix} + R_X\vec{I} + \vec{o}_X$$

To keep it simple, we have omitted coefficients for $U_{p,q}$.

Definition (Irregular over an Iterator in a Dimension) 2: An m-dimensional array with reference $\Re_X$ is defined as "*irregular in the p^{th} dimension, over iterator i*", if at least one of the two hold:

1) for some q, i appears in the index expression of $U_{p,q}$
2) for some q, $U_{p,q}$ is irregular over i in at least one of its dimensions. $\diamond$

For example, array C, with reference $C[k][k+1]$, is regular over $\{k\}$ in both its dimension. Array B in $B\,[C[k][k+1] + D[j] + i]$, is irregular over $\{j,k\}$, but is

regular over $\{i\}$. Array Y, in $Y\,[Z[i][j]+j]\,[k]$, is irregular over $\{i,j\}$, and regular over $\{j\}$, in the first dimension. It is regular over $\{k\}$ in the second dimension. Technique for computing the regular and irregular sets is given in the Appendix.

5. Identifying and Exploiting Locality

Here, we describe an algorithm for SPM mapping of indirectly and directly indexed arrays, using the concept of regular and irregular sets developed in the previous section.

A. The Cost Model

Consider a loop-nest with $\vec{I} = [\ i_1 \quad i_2 \quad \cdots \quad i_n\ ]^T$. Suppose the iterator i_k assumes the values $1, 2 \cdots, I_k$. An array X referenced inside the body of such a loop-nest will be accessed $I_1 \times I_2 \times \ldots \times I_n$ times. We have two scenarios:

- *Array X not mapped to SPM*: In this case, the access is always directly to the external-memory. If m_E is the cost per access for a word in the external memory, then $Total\_Cost = I_1 \times I_2 \times \cdots \times I_n \times m_E$.
- *Array X mapped to SPM*: In this case, $Total\_Cost = Transfer\_Cost + Access\_Cost$. $Transfer\_Cost$ is the cost of all the transfers of X to the SPM. $Access\_Cost$ is the cost of all the access to X on the SPM.
  $Transfer\_Cost$: Suppose the transfers to SPM, of the relevant portion of array X, is done at loop-level k, i.e. inner to loop-nest $[i_1, i_2, \cdots, i_k]$ but outer to the loop-nest $[i_{k+1}, i_{k+2}, \cdots, i_n]$. Therefore, there will be in total $I_1 \times I_2 \times \cdots \times I_k$ number of transfers. We model each of those transfers as composed of N number of sub-transfers. For each sub-transfer, the DMA is programmed to move Q contiguous elements to the SPM from the external memory. A DMA-transfer incurs a fixed startup cost, C, and a cost that is proportional to number of elements, ℓ, that are transfered [8], i.e. $DMA\_transfer\_cost = C + \ell t$. Therefore, each sub-transfer has a cost $C + Qt$. And so, $Transfer\_Cost = (C + Qt) \times N \times I_1 \times I_2 \times \cdots \times I_k$.
  $Access\_Cost$: Let m_S be the cost per access for a word in the SPM. Thus: $Access\_Cost = I_1 \times I_2 \times \cdots \times I_n \times m_S$.

B. Search-Space Exploration

1) Illustrating Example: We explain our algorithm, firstly, with an example. Fig. 3 (left-column) shows a four-level nested-loop with $\vec{I} = [\ i_1 \quad i_2 \quad i_3 \quad i_4\ ]^T$. In this loop, array $A[100][200]$ is referenced as $A[\quad B[i_2]+i_3 \quad][\quad C[i_1]+i_4 \quad]$.

We can choose to do transfers of A, to the SPM, at five possible points. For instance, we could make a copy just before the start of loop i_4. That point is labeled as *migration-point* m_4 in the figure. At the point m_4, i_1, i_2 and i_3 each have a fixed value, temporary albeit. Starting at row $B[i_2] + i_3$ and column $C[i_1] + 2$, the next four locations in that row can be copied to the SPM. These five locations, therefore, form a copy-candidate. The SPM-Mapping (third) column in Fig. 3 shows how much of the SPM gets filled with that copy. The number of times this transfer would happen equals the number

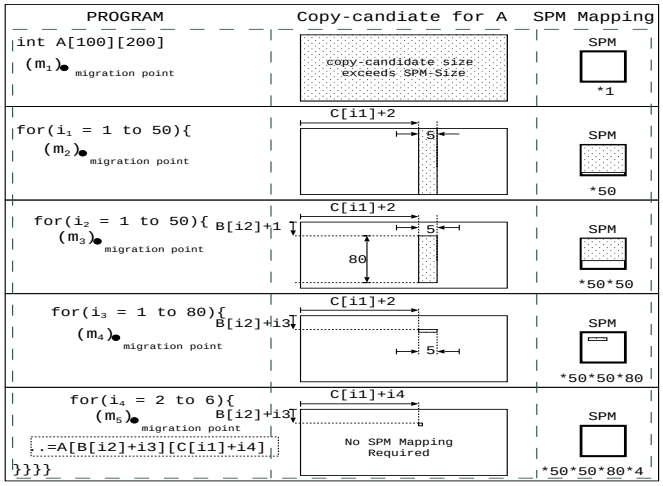

Fig. 3. Trade-off between number of transfers and the size of each transfer.

of times the program comes to *migration-point* m_4. That is, $50 \times 50 \times 80 = 200000$ times. The $Total\_Cost$ at m_4 is:

$$Total\_Cost = \underbrace{(C + 5t) \times 200000}_{Transfer\_Cost} + \underbrace{50 \times 50 \times 80 \times \times 5 \times m_S}_{Access\_Cost}$$

Rather than m_4, we could choose *migration-point* m_3. In that case we have to transfer 5×80 elements of A, starting at row: $B[i_2] + 1$ and column: $C[i_1] + 2$. But then, we have to do it only 50×50 times.

If we next try to do the copying at *migration-point* m_2, we must copy the entire 100 rows for the column $C[i_1]+2$ to column $C[i_1]+6$ (inclusive). The transfer will have to be done only 50 times. As pictorially illustrated in the Fig. 3, with a slight increase in copy size we get an enormous reduction in the number of transfers (from 250, down to 50).

That leaves us with two additional migration points: m_1, where we attempt to copy the entire array, but it is not possible because of limited SPM size; and m_5, where it is cheaper to access the external memory directly.

For each of the *migration-points* above, we can compute the $Transfer\_Cost$. We chose the point that has the least cost. Note that $Access\_Cost$ is independent of the *migration-point*.

2) Search-Space Exploration Algorithm: Consider a nested-loop with $\vec{I} = [\ i_1 \quad i_2 \quad \cdots \quad i_n\ ]^T$. In our algorithm, we will need to describe subsets of the iterators. The set of all iterators is denoted as $\mathcal{I} = \{i_1, i_2, \cdots, i_n\}$. We define $\mathcal{I}_k$, subset of $\mathcal{I}$, as $\{i_k, i_{k+1}, \cdots, i_n\}$; e.g. $\mathcal{I}_2 = \{i_2, i_3, \cdots, i_n\}$.

The m-dimensional, indirectly-indexed, array X is accessed inside the loop-nest $\vec{I}$ described above. The size of X is $D_1 \times D_2 \times \cdots \times D_m$. In C-Language, X could be declared as $int \quad X[D_1][D_2] \cdots [D_m]$. X is referenced as $X[E_1][E_2] \cdots [E_m]$, where each E_p is a linear expression of iterators and indirectly-indexing arrays ($U_{p,q}$). That is, $\Re_X = \left[\ \sum_q U_{1,q}\langle \Re_{U_{1,q}} \rangle \quad \sum_q U_{2,q}\langle \Re_{U_{2,q}} \rangle \quad \cdots\ \right]^T + R_X \vec{I} + \vec{o}_X$. We refer to $U_{p,q}$'s as the indirectly-indexing arrays at the *first-level*. The iterators over which X is regular (irregular) in the p^{th} dimension is represented as α_p (β_p).

The n-level input loop-nest has $n!$ possible permutations. In practice, n is usually small (less than 7). For each permutation, σ, we do the following. Starting from the innermost loop (i_n), we traverse to the outermost loop (i_1), computing the $Transfer\_Cost$ for each *migration-point*. The *migration point* m_k is located inner to the loop of iterator i_{k-1} but outer to loop of i_k (e.g. see Fig. 3). Note that, during the execution, each time the program reaches the point m_k, the iterators in the set $\mathcal{I} - \mathcal{I}_k$ have a fixed valid value.

Our main data-structure for the algorithm is *config*. It has three components: 1. An array called $range$, 2. A number k which specifies the *migration-point* m_k, and 3. The $Transfer\_Cost$ for the *migration-point* m_k. The element $range[p]$ contains the lower and upper limits of the p^{th} dimension that must be transferred. As seen in *Algorithm 1*, the search-space is traversed via two loops. The outer loop, $k = n \ldots 1$, is over the *migration-points*, starting from m_n (inner-most) and moving up to m_1. The second loop, $p = 1 \ldots m$, is over the dimensions of the array X. For a given m_k, and a particular dimension p, our focus is to: find the range of the p^{th} dimension of X that must be transferred to the SPM, assuming that the *migration-point* has been fixed as m_k.

We, first of all, check if any of the iterators in $\mathcal{I}_k$ are present in the irregular set (β_p). Recall that iterators in β_p are those used to indirectly index X in the p^{th} dimension. Also, iterators in the set $\mathcal{I}_k$ are those not having a definite fixed value at m_k. They cover a range. Therefore, if any iterator in $\mathcal{I}_k$ is also present in β_p, we cannot know the exact locations in p^{th} dimension of X that would be accessed. Therefore, if $\mathcal{I}_k \cap \beta_p \neq \phi$ (ϕ is the null-set), we copy the entire range $(0 \ldots D_p - 1)$ to the SPM (see *Algo. 1*).

If $\mathcal{I}_k \cap \beta_p = \phi$, then we check the regular set, α_p, which represents iterators generating regular access in the index-expression E_p. Iterators in $\mathcal{I} - \mathcal{I}_k$ have a fixed value at m_k. Therefore, the free-iterators in E_p - those representing not one particular value at m_k, but a range - is the set $\alpha_p \cap \mathcal{I}_k$. These iterators in E_p are analyzed to determine the range of the p^{th} dimension of X that should be transferred to the SPM. We use the Omega Library to do symbolic range computation [9]. If $\mathcal{I}_k \cap \alpha_p = \phi$, then only a single point in this dimension is to be transferred.

For each m_k, a $config_k$ is generated. It contains the $range$ information and associated $Transfer\_Cost$. If the sizes in $range$ amount to more than the allocated SPM-size, the cost is set to infinity (not shown in *Algo. 1*). We choose the optimal config as one with the minimum cost. This is compared with the *No SPM-Mapping*, i.e. direct access from external memory.

The data-layout of X should be changed so that the dimension that has the largest range is the *fastest changing dimension* [7]. This data-layout will result in the least number of DMA-transfers. In other words, based on the dimensions of the hyper-cuboid that must be transferred, the layout must be such that the longest edge is stored contiguous in memory. Now we consider the case when the range of values assumed by the elements of (at least some) indirectly-indexing arrays are provided to the algorithm. In real embedded application, this is usually possible because, even for data-dependent parameters, the designers typically apply some bounds. With that information, in *Algo. 1*, for the case $\mathcal{I}_k \cap \beta_p \neq \phi$, we do not automatically extend the range to full-length $(0 \ldots D_P - 1)$. Instead, arrays, such as $U_{p,1}, U_{p,2}, \ldots$, which indirectly index X at the *first-level* in the p^{th} dimension, are treated as variables that can assume any value in the specified range. Also, iterators in E_p belonging to the set $\mathcal{I}_k \cap \alpha_p$, i.e. those iterators whose value is not fixed at m_k, are also treated as variables. Treating the rest of the iterators in E_p as constants, the total range is computed. Note: only those indirectly-indexing arrays that are regular or irregular, in any dimension, over the set $\mathcal{I}_k$, are treated as variables with a range. The rest of the *first-level* arrays are to be treated as constants, because they will compute to a fixed value at the point m_k.

Algorithm 1 Scratch-Pad Mapping Technique.

function Find $config_{opt}$ for a given permutation σ
  for $k = n \ldots 1$ **do**
    for $p = 1 \ldots m$ **do**
      if $\mathcal{I}_k \cap \beta_p \neq \phi$ **then**
        $range[p] \leftarrow (0 \ldots D_p - 1)$
      else
        if $\mathcal{I}_k \cap \alpha_p \neq \phi$ **then**
          $range[p] \leftarrow$ compute range based on iterator-variables $\mathcal{I}_k \cap \alpha_p$ in E_p, treating rest as constants
        else
          $range[p] \leftarrow (0 \ldots 0)$
        end if
      end if
    end for
    $s \leftarrow size(range), \quad c \leftarrow transfer\_cost(range)$
    $config_k \leftarrow \langle range, k, c \rangle$
  end for
  $config_{opt} \leftarrow config_k$ with the minimum cost
end function

6. Experiments and Results

We have implemented our algorithm on top of an in-house optimizing compiler framework. For these experiments we ran our algorithm with the following parameters: C (constant factor in DMA-Transfer)$= 15$, t (cost per element, for the transfer) $= 1$, m_E (cost for data in external memory) $= 5$, and m_S (cost for data in SPM) $= 1$. These are typical values also used by other researchers [8].

We evaluated the resulting codes on an ARM (Advanced Risc Machine) Simulator. The simulator contains an ARM processor with a 4KB on-chip scratch-pad and a 4KB two-way associative data-cache. A DMA is available to transfer data between SPM and external memory. All components have an incorporated timing and power models. We present performance results on four realistic applications kernels, with relatively large code and many data arrays:

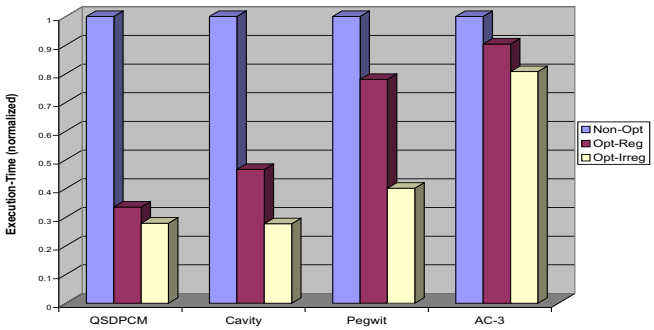

Fig. 4. Total execution-time for- 1. Non-Opt, 2. Opt-Reg, 3. Opt-Irreg.

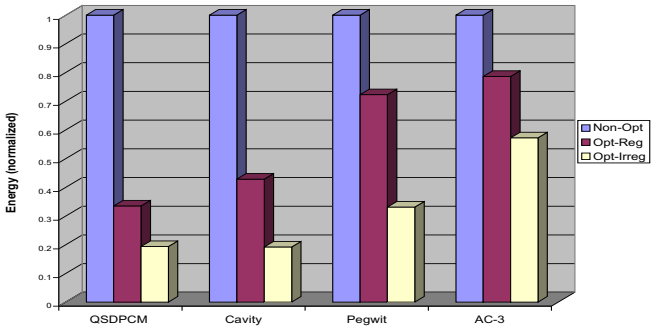

Fig. 5. Total Energy consumed by each application and each version.

- PEGWIT: A Public-Key Encryption algorithm from Media-Bench [10].
- QSDPCM (Quad-Tree Structured Difference Pulse Code Modulation) which is an inter-frame compression technique for video images [13].
- Cavity-Detector: A medical imaging algorithm.
- AC-3 Encoder: 5.1 channel audio compression algorithm (also known as Dolby-Digital) [1]. The bit-allocation part has several two level array-indexing.

For each application, Fig. 4 has a group of three columns. The first col. shows the execution time for the original program. No SPM mapping (*Non-Opt*) was done but the arrays were accessed through the cache. The second col. shows the execution-time when all the arrays with only regular access (directly-indexed) were mapped to the SPM, in the best possible way (*Opt-Reg*). Arrays not mapped to the SPM were put in the cacheable-section of the external memory. The third col. shows the execution time when both regularly and irregularly accessed arrays were allowed to be mapped to the SPM, using our technique (*Opt-Irreg*). Fig. 5, similarly, shows the energy values. Clearly, banishing large, indirectly-indexed, arrays from the SPM can result in significant penalties.

7. Conclusion

Arrays indexed through both iterators and other arrays, occur commonly in many programs. In this paper, we showed, using several examples, that the access pattern of many indirectly-indexed arrays can be quite well analyzed at compile-time for spatial and temporal locality. The analysis was used to efficiently map such arrays to the SPM. We see our main contribution as extending the current framework, and providing an unambigious compiler-based technique for handling irregular arrays access.

References

[1] M. J. Absar and S. George. Development of ac-3 digital audio encoder. *105th Proceedings of the Audio Engineering Society (AES)*, 1998.

[2] U. Banerjee. *Data Dependenies*. Kluwer Aacdemic Publishers, 1988.

[3] F. Catthoor, F. Balasa, E. D. Greef, and L. Nachtergaele. *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design*. Kluwer Academic Publisher, 1998.

[4] R. Das, D. Mavriplis, J. Saltz, and S. Gupta. Communication optimizations for irregular scientific computation on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 1994.

[5] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. *ACM SIGPLAN Conference on Programming Language, Design and Implementation (PLDI)*, 1999.

[6] I. Issenin, E. Brockmeyer, M. Miranda, and N. Dutt. Data reuse analysis technique for software-controlled memory hierarchie. *Design Automation and Test in Europe*, pages 202–207, 2004.

[7] M. T. Kandemir, J. Ramanujan, and A. Chowdhury. Improving cache locality by a combination of loop and data transformation. *IEEE Transaction on Computers*, 48(2), 1999.

[8] M. T. Kandemir, J. Ramanujan, M. J. Irwin, N. Vijayakrishnan, I. Kadayif, and A. Parikh. A compiler-based approach for dynamically managing scratch-pad memories in embedded systems. *IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems*, 23(2), 2004.

[9] W. Kelly, V. Maslov, and W. Pugh. The omega library: Framework and algorithm for analysis and transformation of scientific programs. *http://www.cs.umd.edu/projects/omega*, 1996.

[10] C. Lee, M. Potkonjak, and M. Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communication systems. *International Symposium on Microarchitecture*, 1997.

[11] M. O'Boyle and P. Knijnenburg. Non-singular data transformations: Definition, validity and applications. *International Conference on Parallel Architectures and Compilation Techniques*.

[12] P. R. Panda, N. Dutt, and A. Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. *Design Automation and Testing in Europe*, 1997.

[13] P. Stobach. A new technique in scene adpative coding. *European Signal Processing Conference (EUSIPCO)*, 1998.

[14] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *ACM SIGPLAN Conference on Programming Language, Design and Implementation (PLDI)*, 1988.

8. Generating Irregular and Regular Sets

For a reference $\Re_X = \vec{S}_X + R_X\vec{I} + \vec{o}_X$, in a loop-nest with iterators $\mathcal{I} = \{i_1, i_2, \cdots, i_n\}$, we define some operators:

Operator θ_1, when applied to a $m \times n$ matrix, R_X, returns a $m \times 1$ column vector $\theta_1(R_X) = [d_1 \quad d_2 \quad \ldots \quad d_m]^T$, such that: $d_p = \{i_k \mid i_k \in \mathcal{I} \wedge r_{p,k} \neq 0 \wedge r_{p,k} \in R_X\}$ Therefore, each component of $\theta_1(R_X)$ is a subset of $\mathcal{I}$. For a vector $\vec{a} = [a_1 \quad a_2 \quad \ldots \quad a_m]^T$, where each element a_i is a subset of $\mathcal{I}$, we define $\biguplus \vec{a} = \cup_k a_k$. That is, $\biguplus \vec{a}$ forms a set union of all the iterators present in $\vec{a}$.

$$\theta_2(\vec{S}_X) = \theta_2\left(\begin{bmatrix} \sum_q U_{1,q}\langle\Re_{U_{1,q}}\rangle \\ \sum_q U_{2,q}\langle\Re_{U_{2,q}}\rangle \\ \vdots \end{bmatrix}\right) = \begin{bmatrix} \bigcup_q \Theta\left(\Re_{U_{1,q}}\right) \\ \bigcup_q \Theta\left(\Re_{U_{2,q}}\right) \\ \vdots \end{bmatrix}$$

Finally: $\Theta(\Re_X) = \biguplus\theta_2(\vec{S}_X) \quad \cup \quad \biguplus\theta_1(R_X)$. The reference $\Re_X$, of m-dimensional array X, is with respect to its p^{th} dimension:

- Regular over the set of iterators in p^{th} row of $\theta_1(R_X)$.
- Irregular over the set of iterators in the p^{th} row of the column-vector $\theta_2(\vec{S}_X)$.
- Independent over the iterators which do not appear in the above two.