Dynamic Management of Scratch-Pad Memory Space

M. Kandemir, J. Ramanujam[*], M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh
Microsystems Design Lab
The Pennsylvania State University
University Park, PA 16802

Abstract

Optimizations aimed at improving the efficiency of on-chip memories are extremely important. We propose a compiler-controlled dynamic on-chip scratch-pad memory (SPM) management framework that uses both loop and data transformations. Experimental results obtained using a generic cost model indicate significant reductions in data transfer activity between SPM and off-chip memory.

1 Introduction

Embedded systems are often designed as a *system-on-chip* (SoC) architecture to cater to the demands of small form factors. An important characteristic of a SoC design process is the design of the memory configuration and the management of the flow of data. While it is very important to select a correct memory configuration, it might be equally important to choreograph the data flow between on-chip and off-chip memories in an optimal manner. Many SoC applications have significant data processing requirements. In particular, many codes from the video processing and signal processing domains manipulate large arrays of signals using multi-level nested loops. An important issue then is maintaining good data locality; that is, satisfying a majority of data accesses from fast on-chip memories instead of slow off-chip DRAMs.

Unfortunately, a simple on-chip cache hierarchy may not be very suitable for an embedded system where meeting hard real-time constraints is critical[7]. Such a constraint, in most cases, requires programmers to determine exactly how much processing time a given code segment will take. Existence of a cache memory makes it nearly impossible to predict execution time accurately. Not being able to predict the execution time of a given piece of code can also adversely affect the scope and effectiveness of software optimizations.

Consequently, systems that contain a *software managed scratch-pad memory* (called SPM henceforth) can be of great value as, in such systems, the software is in full control of the flow of data between on-chip and off-chip memory, so it is relatively easy to predict data access times. Previous work on SPM[10] investigates a *static* data management scheme in which program data structures are partitioned between the off-chip memory and the SPM, and this data partitioning is valid *throughout the execution*. In this scheme, the scalar variables are stored in the SPM, and the large data arrays that do not fit in the SPM are stored in the off-chip memory (and accessed through on-chip cache). Each of the remaining arrays is stored in either SPM or off-chip memory so as to minimize the conflict misses in the on-chip cache.

While several applications benefit from this static partitioning approach, in many codes, we may need to perform dynamic data transfers between off-chip memory and on-chip SPM during the course of execution. Important issues in such a dynamic scheme are determining memory layouts and best loop access patterns, partitioning the available SPM space between competing arrays, and restructuring the code for explicit data transfers. This paper addresses these issues in the context of SPM and array-based applications which are dominant in video and image processing domains. The proposed compilation framework has been tested using a suite of five applications. Experimental data and comparison results with previous work show that our approach is very effective in reducing the activity between on-chip SPM and off-chip memory. In general, such reductions can lead to large savings in energy consumption[8] and effective data access latency[14].

The remainder of this paper is organized as follows. Section 2 introduces our memory system architecture and explains the programming model. Section 3 presents our approach to the dynamic management of data transfers between the off-chip memory and the SPM. The performance numbers are given in Section 4. Section 5 discusses related work, and Section 6 concludes the paper.

2 Memory Architecture and Execution Model

Our *data memory* architecture consists of three components: a cache memory, a scratch-pad memory (SPM), and a main memory. The cache memory and the SPM are on-chip SRAMs (with the same access latency), and the main memory can be assumed to be an off-chip DRAM (with a higher access latency). As shown in Figure 1, the address space is divided between off-chip memory and on-chip SPM, the former of which is accessed through the on-chip cache.

Figure 1 also shows the necessary control signals to activate either SPM or cache depending on the issued memory address. The same data and address buses are fed into cache as well as the SPM. In this work, we are interested in managing the data transfers between the off-chip memory and the on-chip SPM in cases where total data size is larger than SPM capacity. We assume the existence of a higher level mechanism that decides which data accesses should bypass the SPM and be accessed through cache (if it exists). Therefore, in the rest of the paper, we drop the cache from consideration. It is assumed that the scalar variables are stored in registers, so they do not interfere with array references.

In order to generate optimized code for a memory architecture that contains an SPM, in addition to the conventional optimization steps, the compiler also has to *schedule explicit data transfers* between the off-chip memory and the SPM. To accomplish this, the compiler needs to take into account the data layout in the off-chip memory, the application access pattern, and the available memory space in the SPM. *The portions of arrays required by the current computation are fetched from the off-chip memory to the SPM.* Throughout this paper, these portions are called *data tiles* or simply *tiles*. The data tiles brought into the SPM should also have a high degree of reuse and should fit in the SPM. Note that the SPM space should be divided suitably among the data tiles of different arrays. Thus, during the course of execution, a number of data tiles belonging to a number of different arrays are brought into the SPM, the new values for these data tiles are computed, and the tiles are stored back into appropriate locations in off-chip memory, if necessary.

As an example, let us consider the nest in Figure 3(i). A key aspect of the compilation process is the use of a tiling-like transformation. When applied to a loop, tiling replaces it (in the most general case) with two loops: a *tiling loop* and an *element loop*[14]. The loop nest in Figure 3(i) can be translated by the compiler into the code shown in Figure 3(ii); in this translated code, the outer loops `it`, `jt`, and `kt` are the tiling loops, and the inner loops `i'`, `j'`, `k'` are the element loops.

The explicit data transfer calls, **read_tile** and **write_tile**, are inserted at tile boundaries (outside the element loops). The transfer call **read_tile**

[*] Department of ECE, Louisiana State University, Baton Rouge, LA 70803.

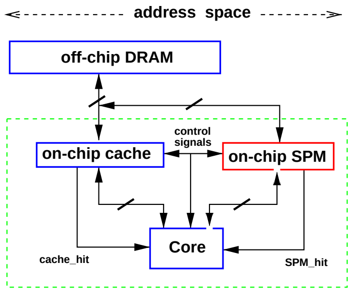

Figure 1: System architecture and address space partitioning.

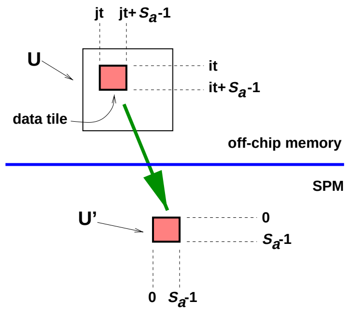

Figure 2: Reading an $S_a \times S_a$ data tile from off-chip memory to SPM

`U[it:it+`S_a`-1,jt:jt+`S_a`-1]` $\rightarrow$ `U'[0:`S_a`-1,0:`S_a`-1]` copies the elements of the array `U` that satisfy the constraint {(`it` $\leq$ `i` $\leq$ `it+T-1`) and (`jt` $\leq$ `j` $\leq$ `jt+T-1`)} to array `U'[0:`S_a`-1, 0:`S_a`-1]`. It should be noted that while array `U` resides in off-chip memory, array `U'` resides in the SPM. In other words, **read_tile** indicates an explicit copy operation from off-chip memory to SPM as depicted in Figure 2. The implementation of the **write_tile** call is similar except that the direction of the transfer is reversed. For the sake of clarity, we will write the compiler-translated version as shown in Figure 3(iii), where all element loops are omitted. Each reference is replaced by its corresponding submatrix version (in terms of the original array). For example, a reference such as `U'[i'][j']` is replaced by `U[it:it+`S_a`-1,jt:jt+`S_a`-1]`, where S_a is the size (in array elements) of a dimension of a data tile (called *tile size* or *blocking factor*).

3 Management of Data Transfers

3.1 Overview

Efficient use of an SPM depends critically on an important factor: *maximizing the reuse of data portions (tiles) brought from off-chip memory*. This is crucial because an item sitting in the SPM without reuse not only occupies space that could have been used for some other array but also wastes bandwidth (as it needs to be brought over the interconnect between SPM and off-chip memory). Our objective can be achieved by (i) maximizing the number of accesses to the data in the SPM, and (ii) minimizing the number of data transfers to and from off-chip memory. Our approach follows three complementary steps:

• *Memory Layout Detection and Loop Transformation:* Maximizing data reuse and minimizing the number of explicit data transfers between off-chip memory and the SPM requires a good combination of loop access pattern and off-chip memory layout.

• *Memory Space Partitioning:* Since the on-chip SPM space has a limited capacity, its management is very important. A crucial issue here is to determine how to partition the available SPM space dynamically between competing arrays. We show that a simple strategy that divides the available SPM space evenly between arrays may not work well in practice.

• *Code Modifications:* After deciding a suitable partitioning of the SPM space between arrays, the compiler needs to modify the code accordingly.

3.2 Cost Model

We assume a *generic cost model* which can be made to work with several performance and/or energy metrics. Each data transfer from off-chip memory to the SPM or vice versa is assumed to incur a fixed *startup cost* in addition to a cost proportional to the amount of data requested. The startup cost for a data access (read or write), C, is assumed to include all the costs due to bookkeeping/handshaking activity between the SPM and the off-chip memory and the software overhead involved (e.g., the runtime call activation to initiate the transfer). Let the cost of transferring a single data item (e.g., an array element) between the off-chip memory and the SPM be t. Thus, the cost of transferring ℓ *consecutive elements* between the off-chip memory and the SPM can roughly be modeled as $T = C + \ell t$. We refer to this cost as *memory access cost*, or simply *access cost*. Note that the per item transfer cost does not include the cost incurred in accessing the off-chip memory circuitry itself. While this model is highly simplified, it is useful for our purpose.

As an example, let us consider the matrix multiply code given in Figure 3(i). We assume that the SPM has a memory of size M allocated for a given computation. Let us also assume that (for simplicity) each array is of $n \times n$, where n is also assumed to be the number of iterations of all the loops. We assume that $d \times n \leq M \ll n^2$ for an integer d. In cases where $n > M$, we can apply the technique in this paper recursively (which, in a sense, corresponds to multi-level tiling).

Suppose for now that the compiler works on square tiles of size $S_a \times S_a$ as those shown Figure 4(i), and the default array layout is *row-major.* The access cost of a tile of size $S_a \times S_a$ is $S_a C + S_a^2 t$, and n^2/S_a^2 of these data tiles are read (for array `U`, assuming that S_a divides n evenly). Consequently, the total *read access cost* for array `U` is $T_{\mathtt{U}} = (n^2/S_a^2)[S_a C + S_a^2 t]$. The access costs for the other arrays can be computed similarly. Therefore, the overall access cost for the nest shown in Figure 3(iii) (T_{ov}^a) considering *all three arrays* and the *read activity alone* can be calculated as

$$T_{ov}^a = T_{\mathsf{U}} + T_{\mathsf{V}} + T_{\mathsf{W}} \quad = \quad \frac{3n^3}{S_a^2} C + \frac{3n^3}{S_a} t$$

under the memory constraint $3S_a^2 \leq M$. Here, T_{U}, T_{V}, and T_{W} denote the read access costs for arrays `U`, `V`, and `W`, respectively. Assuming that the entire available SPM capacity (M) will be used for this computation (i.e., $3S_a^2 = M$), this last formulation above can be re-written in terms of M as

$$T_{ov}^a = \frac{3n^3}{M/3} C + \frac{3n^3}{\sqrt{M/3}} t.$$

This straightforward translation can be improved substantially by being more careful when choosing memory layouts, loop ordering, and data tile allocations. First, we observe from Figure 3(iii) that it is not necessary to perform all **tile_read** and **tile_write** activities inside the `kt` loop. In particular, data tile of array `U` can be read (and written) between the tiling loops `jt` and `kt`. Second, reading square tiles does not allow the compiler to take advantage of the layout of data in off-chip memory. For example, the array `V` is stored in row-major in off-chip memory and instead of reading a square-chunk, reading a row-chunk should result in fewer transfer calls in the code. Figure 3(iv) shows the code corresponding to this scenario. Note that since the compiler reads data tiles of sizes $S_b \times n$ and $n \times S_b$, for arrays `V` and `W`, respectively, the tiling loop `kt` does not appear in Figure 3(iv). The corresponding tile allocation is shown in Figure 4(ii). During the execution, tiles of sizes $S_b \times S_b$, $S_b \times n$, and $n \times S_b$ are accessed for arrays `U`, `V`, and `W`, respectively. In addition to determining the tile allocation and loop order, our approach assigns memory layouts to arrays. In the case of Figure 4(ii), it assigns

```
        (i)
for(i=0;i<$n$;i++)
  for(j=0;j<$n$;j++)
    for(k=0;k<$n$;k++)
      U[i][j] = U[i][j] + V[i][k] * W[k][j]

        (ii)
for(it=0;it<$n$;it=it+$S_a$)
  for(jt=0;jt<$n$;jt=jt+$S_a$)
    for(kt=0;kt<$n$;kt=kt+$S_a$)
      {
        read_tile U[it:it+$S_a$-1,jt:jt+$S_a$-1] → U'[0:$S_a$-1,0:$S_a$-1]
        read_tile V[it:it+$S_a$-1,kt:kt+$S_a$-1] → V'[0:$S_a$-1,0:$S_a$-1]
        read_tile W[kt:kt+$S_a$-1,jt:jt+$S_a$-1] → W'[0:$S_a$-1,0:$S_a$-1]
        for(i'=0;i'<$S_a$;i'++)
          for(j'=0;j'<$S_a$;j'++)
            for(k'=0;k'<$S_a$;k'++)
              U'[i'][j'] = U'[i'][j'] + V'[i'][k'] * W'[k'][j']
        write_tile U'[0:$S_a$-1,0:$S_a$-1] → U[it:it+$S_a$-1,jt:jt+$S_a$-1]
      }

        (iii)
for(it=0;it<$n$;it=it+$S_a$)
  for(jt=0;jt<$n$;jt=jt+$S_a$)
    for(kt=0;kt<$n$;kt=kt+$S_a$)
      {
        read_tile U[it:it+$S_a$-1,jt:jt+$S_a$-1]
        read_tile V[it:it+$S_a$-1,kt:kt+$S_a$-1]
        read_tile W[kt:kt+$S_a$-1,jt:jt+$S_a$-1]
        U[it:it+$S_a$-1,jt:jt+$S_a$-1] = U[it:it+$S_a$-1,jt:jt+$S_a$-1]
          + V[it:it+$S_a$-1,kt:kt+$S_a$-1] * W[kt:kt+$S_a$-1,jt:jt+$S_a$-1]
        write_tile U[it:it+$S_a$-1,jt:jt+$S_a$-1]
      }

        (iv)
for(it=0;it<$n$;it=it+$S_b$)
  {
    read_tile V[it:it+$S_b$-1,1:$n$]
    for(jt=0;jt<$n$;jt=jt+$S_b$)
      {
        read_tile U[it:it+$S_b$-1,jt:jt+$S_b$-1]
        read_tile W[1:n,jt:jt+$S_b$-1]
        U[it:it+$S_b$-1,jt:jt+$S_b$-1] = U[it:it+$S_b$-1,jt:jt+$S_b$-1]
          + V[it:it+$S_b$-1,1:$n$] * W[1:n,jt:jt+$S_b$-1]
        write_tile U[it:it+$S_b$-1,jt:jt+$S_b$-1]
      }
  }

        (v)
for(jt=0;jt<$n$;jt=jt+$S_c$)
  {
    read_tile U[1:$n$,jt:jt+$S_c$-1]
    for(kt=0;kt<$n$;kt=kt+$S_c$)
      {
        read_tile V[1:$n$,kt:kt+$S_c$-1]
        read_tile W[kt:kt+$S_c$-1,jt:jt+$S_c$-1]
        U[1:$n$,jt:jt+$S_c$-1] = U[1:$n$,jt:jt+$S_c$-1]
          + V[1:$n$,kt:kt+$S_c$-1] * W[kt:kt+$S_c$-1,jt:jt+$S_c$-1]
      }
   write_tile U[1:$n$,jt:jt+$S_c$-1]
  }
```

Figure 3: (i) The matrix multiply code. (ii-iii) A straightforward translation (for SPM) using square data tiles for all arrays. (iv-v) Optimized codes.

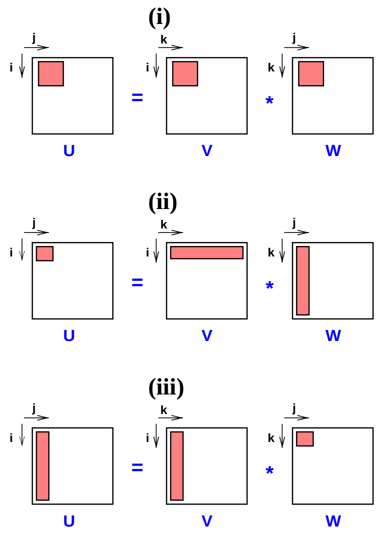

Figure 4: Different tile allocations in the SPM for the matrix multiply code.

a row-major layout for V and a column-major layout for W. The layout of array U can be either row-major or column-major (as we read square tiles for this array). The overall *read access cost* of this loop order,[1] layout assignment, and tile allocation scheme is:

$$T^b_{ov} = \underbrace{\frac{n^2}{S_b^2}\left(S_bC + S_b^2 t\right)}_{T_{\mathsf{U}}} + \underbrace{\frac{n}{S_b}\left(S_bC + nS_bt\right)}_{T_{\mathsf{V}}} + \underbrace{\frac{n^2}{S_b^2}\left(S_bC + nS_bt\right)}_{T_{\mathsf{W}}}$$

$$= \left(\frac{2n^2}{\sqrt{n^2+M}-n} + n\right)C + \left(2n^2 + \frac{n^3}{\sqrt{n^2+M}-n}\right)t$$

under the assumption of $2nS_b + S_b^2 = M$, and at most n array elements can be read with a single **read_tile** call from within the code. This assumption is just for making the presentation clear. Actually, if we can read/write more than n elements in a single call, the number of transfer operations in the code can be reduced further. It should also be noted that a single (data transfer) call in the code can correspond to multiple transfer activities at the hardware level (depending on the bandwidth between the off-chip memory and the SPM). A reduction in the number of transfer calls in the code also leads, in general, to a reduction in hardware-level transfer operations. Note that although S_b is *different* from S_a, the total memory (SPM) space (M) used in both cases is the *same*. When $n = 128$ and $M = 8,192$ (for illustrative purposes only), it can be shown that, as compared to T^a_{ov}, T^b_{ov} improves the coefficient of C by 45%, and the coefficient of t by 12.2%. The memory access cost of the code given in Figure 3(v) (and its corresponding tile allocation depicted in Figure 4(iii)) is very similar to that in Figure 3(iv). In this case however, the compiler selects a different loop order, and assigns column-major memory layouts for both U and V.

3.3 Desired Array Reference Forms

We want each reference to an m-dimensional *row-major* array U to be in either of the following two forms, where f_i and g_j are subscript functions:

• U$[f_1][f_2]...[f_m]$: In this form, f_m is an affine function of all loop indices with a coefficient of 1 for the innermost loop index whereas f_1

[1] Similar calculations can be done for write costs as well.

```
INPUT: A nested loop and the access matrices for the references in the nest
OUTPUT: A loop transformation matrix $T$ and a data transformation matrix $M_i$ for each
array $i$

1: determine the access matrix $L_i$ for each array i ($1 \le i \le s$)
2: for each of the $2^s$ alternatives do
        2.1: determine target $L'_1, L'_2, \ldots, L'_s$
        2.2: using $L_i T^{-1} = L'_i$ determine a $T$
        2.3: for each array $j$ with the spatial reuse at some loop level do
                2.3.1: let $\gamma_k$ be the row (if any) containing the only non-zero element in
                the last column for $L'_j$
                2.3.2: find an $M_j$ such that $L''_j = M_j L'_i$ will be in the desired form (i.e.,
                $\gamma_k$ will be the last row)
        2.4: endfor
        2.5: record the current alternative with the computed coefficients of $C$ and $t$
3: endfor
4: select the most suitable alternative (see the explanation in the text)
5: apply tiling (see the explanation in Sections 3.5 & 3.6)
```

Figure 5: Optimization algorithm (single nest). 2^s alternatives are evaluated as each array can have spatial or temporal reuse.

through $f_{(m-1)}$ are affine functions of all loop indices except the innermost one.

- U$[g_1][g_2]...[g_m]$: In this form, all g_1 through g_m are affine functions of all loop indices except the innermost one.

In the first case, we have *spatial reuse* for the reference in the *innermost loop*, and in the second case, we have *temporal reuse* in the *innermost loop*. Note that the second case helps to reduce the coefficient of both C and t whereas the first case helps more to reduce the coefficient of C. The compiler's task, then, is to bring each array (reference) to one of the forms given above.

3.4 Algorithm for Determining Memory Layouts and Loop Order

We assume that the memory layout for an m-dimensional array can be in one of $m!$ forms, each corresponding to the linear layout of data in off-chip memory by a nested traversal of the array axes in some predetermined order. The innermost axis is called the *fastest-changing dimension*. As an example, for row-major memory layout of a two-dimensional array, the second dimension is the fastest changing dimension.

In our framework, each execution of an n-level nested loop is represented using an *iteration vector* $\bar{\mathtt{I}} = (\mathtt{i}_1, \mathtt{i}_2, ..., \mathtt{i}_l)$, where $\mathtt{i}_j$ corresponds to j^{th} loop from the outermost position. We assume that the array subscript expressions and loop bounds are *affine functions* of enclosing loop indices and loop-independent variables. Each reference to an m-dimensional array U is represented by an *access* (*reference*) *matrix* $\mathcal{L}_u$ and an *offset vector* $\bar{l}_u$ such that $\mathcal{L}_u \bar{\mathtt{I}} + \bar{l}_u$ is the element accessed by a specific iteration $\bar{\mathtt{I}}$[14].

The class of loop and data transformations we are interested in can be represented using non-singular square *transformation matrices*. It can be shown that, when both a loop transformation (represented by $\mathcal{T}$) and a data transformation (represented by $\mathcal{M}_u$) are applied together, a reference $\mathcal{L}_u \bar{\mathtt{I}} + \bar{l}_u$, becomes $\mathcal{M}_u \mathcal{L}_u \mathcal{T}^{-1} \bar{\mathtt{I}}' + \mathcal{M} \bar{l}_u$. Our algorithm given in Figure 5 tries to select $\mathcal{T}$ and $\mathcal{M}_u$ such that the resulting reference fits in one of our desired forms. For now, we assume that there is a single *uniformly generated reference set* (UGR)[6] in the nest for a given array. A UGR is a set of array references (to the same array) whose subscript functions differ in a constant term only. Therefore, in the following, we use the terms 'array' and 'reference' interchangeably. Simply, the algorithm in Figure 5 enumerates all possible combinations of spatial/temporal locality alternatives for all arrays; for each combination, it computes the coefficients of C and t, and then selects the alternative with the minimum cost (taking into account the actual values for C and t). Note that if the reference is to be optimized for spatial locality, the algorithm first uses loop transformation, and then data transformation.

For an example application of the algorithm of Figure 5, consider once more the matrix multiply code in Figure 3(i). The access matrices for arrays U, V, and W are

$$\mathcal{L}_u = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \mathcal{L}_v = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}; \mathcal{L}_w = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

In this nest, temporal locality can be exploited for only one of the three arrays. The remaining two arrays can be optimized for spatial locality. This gives us a total of three alternatives. To keep the presentation simple, we focus on only one of them. In this alternative, we attempt to optimize array W for temporal locality and arrays U and V for spatial locality. Consequently, the desired access matrices are of the form[2]

$$\mathcal{L}'_u = \begin{bmatrix} \times & \times & 1 \\ \times & \times & 0 \end{bmatrix}, \mathcal{L}'_v = \begin{bmatrix} \times & \times & 1 \\ \times & \times & 0 \end{bmatrix}; \mathcal{L}'_w = \begin{bmatrix} \times & \times & 0 \\ \times & \times & 0 \end{bmatrix}.$$

Here, $\times$ denotes 'don't care'. Now, using $\mathcal{L}_u$, $\mathcal{L}_v$, $\mathcal{L}_w$, $\mathcal{L}'_u$, $\mathcal{L}'_v$, and $\mathcal{L}'_w$, we can determine $\mathcal{T}^{-1} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$. If we transform this nest using the inverse of this matrix, the new loop order is j, k, i from outermost to innermost. Note that this new access pattern exploits temporal locality for W in the innermost loop, and imposes a column-major memory layout for U and V. If the default memory layout is row-major, these two arrays should be data-transformed using matrices $\mathcal{M}_u = \mathcal{M}_v = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. Note that this version eventually transforms to the tiled code given in Figure 3(v).

3.5 Memory Space Partitioning

Once a suitable loop order and memory layouts for the arrays are determined, the compiler needs to partition the available SPM storage between the arrays accessed in the nest. Let us assume first that there is a single reference per array. Assuming that the loops are tiled using a blocking factor (tile size) of S_a, if the reference in question fits in the first form desirable discussed in Section 3.3, the compiler accesses a data tile of size $S_a \times S_a \times S_a \times ... \times S_a \times n$ (assuming that n is the number of iterations of the innermost loop, and the memory layout is row-major as in C). Note that accessing such a data tile tends to minimize the number of transfer calls per tile. As for the arrays that fit in the second form, we can access data tiles of $S_a \times S_a \times S_a \times ... \times S_a \times S_a$ as the loops whose indices appear in some subscript function of this array are tiled. After that, the data tiles can be hoisted into upper loop positions depending on the loop indices used by their subscript functions.

If, however, we consider the entire UGR set (which might contain multiple uniformly generated references to the same array), we use the concept of *spread* of a UGR set[1]. Informally, it consists of all of the elements that are accessed by the references in the set. If we have more than one UGR set for a given array, our approach treats each UGR set as if it belongs to a different array. The compiler performs this memory space partitioning for each alternative (considered by the algorithm in Figure 5); however, it does *not* generate code until it determines the most suitable alternative.

3.6 Code Modifications

Another important issue is the placement of explicit data transfer calls in a given tiled nest. In fact, there are two subproblems here: (i) the insertion of the transfer calls in the code, and (ii) determination of the parameters to be passed to them. In determining the insertion points, the compiler looks at the indices used in the subscripts of the reference in question, and inserts the transfer call associated with the reference in between appropriate tiling loops. For handling the second subproblem, we use the method of *extreme values of affine functions*. Given an affine function of a number of variables (in our case, a subscript expression)

[2]Other alternatives are also possible.

and inequalities that represent the bounds for the variables, the extreme values method determines the maximum and minimum values of the affine function in the bounded region.

3.7 Multiple Nests and Inter-Procedural Problem

Unlike loop transformations, the impact of a data transformation is *global* in the sense that it affects locality properties of all references to the same array in every loop nest and in every procedure. Here, we briefly discuss our approach to this global optimization problem. First, we focus on the *intra-procedural* locality optimization problem and present an approach to optimize a series of loop nests collectively. Given a series of nested loops that access (possibly different) subsets of arrays declared in the procedure, our algorithm first ranks the nests, then optimizes one nest at a time determining memory layouts for its arrays, and propagates these newly-found memory layouts to remaining nests to be optimized.

We have also implemented an inter-procedural optimization framework on top of this. The current implementation performs two traversals on the *call graph* representation of the program. At the end of the first traversal (bottom-up, from callees to callers), it collects all layout and loop transformation constraints at the root procedure (main program), and during the second traversal (top-down, from callers to callees), it propagates down these constraints on the call graph allowing each procedure to determine suitable memory layouts for its local arrays.

4 Experiments

In this section, we present experimental results and compare our dynamic technique with four other approaches. Our experimental suite consists of five benchmarks: `int_mxm`, an integer matrix multiply program (that contains one initialization and one multiplication nest); `full_search` and `parallel_hier`, two different motion estimation codes; `rasta_fft`, a discrete Fourier analysis code; and `rasta_flt`, a filtering routine. The data set sizes (input sizes) for `int_mxm`, `full_search`, `parallel_hier`, `rasta_fft`, and `rasta_flt` are 196K, 71K, 71K, 224K, and 128K, respectively.

We use five different versions of each code:

• **tiled** is the version in which all arrays involved in the computation are accessed using square data tiles, and all array layouts are fixed at row-major. At a given time, the SPM space is divided between the involved arrays *evenly*.

• **static** is the version that allocates the entire SPM space for one chunk of data throughout the execution. In order for this scheme to be beneficial, we place the most frequently used data chunk in the SPM.

• **c_opt** is the dynamic SPM management strategy proposed in this paper.

• **hand** is an *hand-optimized* version. In selecting the tile shapes, it considers not only the loop nest in question, but it takes into account the opportunities for tile reuse between multiple nests.

• **cache** is the version that uses the available SPM space as a conventional cache; that is, the hardware controls the data transfers between the on-chip memory and the off-chip memory.

Out of the five codes in our experimental suite, two codes (`rasta_fft` and `rasta_flt`) benefited from the inter-procedural layout optimization part of our framework. We also collect statistics (by instrumenting the code) during execution on how many data transfer calls are executed (i.e., coefficient of C), and how many data items are transferred between the off-chip memory and the SPM (i.e., coefficient of t). For the experiments that involve the **cache** version, we employ a trace-driven cache simulator (DineroIV) [5]. It should be noted that the **c_opt** and **cache** versions use *exactly the same* memory layouts and loop optimizations to isolate the benefits that are solely due to the management of the on-chip memory space.

Figure 6 gives the *total data access costs* for four different versions. The total data access cost has *four components*: transfer initiation cost (C), per item transfer cost (t), off-chip memory access cost (K_{off}) which does *not* include the per item transfer cost, and on-chip memory access cost (K_{on}). We present results for nine combinations of the ratio $C : t : K_{on} : K_{off}$. For example, a ratio such as 5:5:1:10 indicates that the data transfer initiation cost and per item transfer cost are the same (5). At the same time, the cost of off-chip memory (circuitry) access, K_{off}, is twice that of C, and the cost ratio between the on-chip and the off-chip memory is 1:10. Each combination differs from others in relative costs of C, t, K_{on}, and K_{off}. The performance of the version **static** is assumed to be independent of C and t as their contribution to the overall cost (for that version) is negligible. From these results, we can make two main observations. First, for almost all experiments, the performance of **static** is very poor, indicating the need for dynamic management of the SPM space. We also performed another set of experiments (with **static**) where the non-SPM data is accessed through an on-chip cache. The experiments performed with 1K, 2K, and 4K direct-mapped and set-associative caches showed that the performance of the **static** version is improved by 8.1-10.2% (over **static** without cache support) which is still much lower than the performance of other versions. Moreover, note that, in this second set of experiments, the **static** version uses more on-chip storage space (i.e., cache+SPM) than others. Another observation from Figure 6 is that **c_opt** significantly improves over **tiled**, and its performance is close to that of **hand**. For example, when $C : t : K_{on} : K_{off}$ = 5:5:1:10, our approach reduces the total data access cost over **tiled** by 26.3% on average. With the same parameters, **hand** improves over **tiled** by 32.8%.

We also compared the performance of a dynamically managed SPM with a *traditional cache memory of the same* size using the `int_mxm` code. We fix[3] $C : t : K_{on} : K_{off}$ = 5:5:1:10, and assume that the block size used in transfers between the cache/SPM and off-chip memory is 32 bytes. Our results (not presented in detail here) show that using a conventional cache instead of dynamically-managed SPM increases the total cost by 41.6% and 22.6% for write-through (WT) and write-back (WB) cache, respectively, on the average. Note that similar cache results have also been reported by Benini et al[3]. We also observed that increasing the associativity from 1 (direct-mapped case) to 2 improves cache performance whereas going from 2 to 4 (except for one case) degrades the performance of the **cache** version (due to the overhead factor we used and the lack of a significant drop in the number of conflict misses as a result of increased associativity).

5 Related Work

Several strategies have been proposed to improve cache performance (e.g.,[14]). While these techniques reduce the number of cache misses, they do not completely eliminate them, and they do not solve the problem of unpredictable data access latency associated with cache memories. Memory optimizations for embedded systems are addressed, among others, by Panda et al[11], Catthoor et al[4], and Shiue and Chakrabarti [12]. Kolson et al[9] present a technique for memory access scheduling in high-level synthesis. Wang et al[13] propose a framework for analyzing the flow of values and data reuse for on-chip memories. They perform no inter-procedural analysis, and assume that the loops are perfectly-nested. Panda et al[10] present an elegant static data partitioning scheme for efficient utilization of scratch-pad memory. Their approach is oriented toward eliminating the potential conflict misses due to limited associativity of on-chip cache. Benini et al[3] discuss a memory management scheme that is based on keeping the most frequently used data items in a software-managed memory. This is a static management technique as it does not adapt the contents of on-chip memory to dynamically changing working set.

[3]This is assuming a direct-mapped cache. For a two-way (resp. four-way) associative cache, we assume $K_{on} = 1.1$ (resp. $K_{on} = 1.2$) to take into account the additional overhead due to associativity.

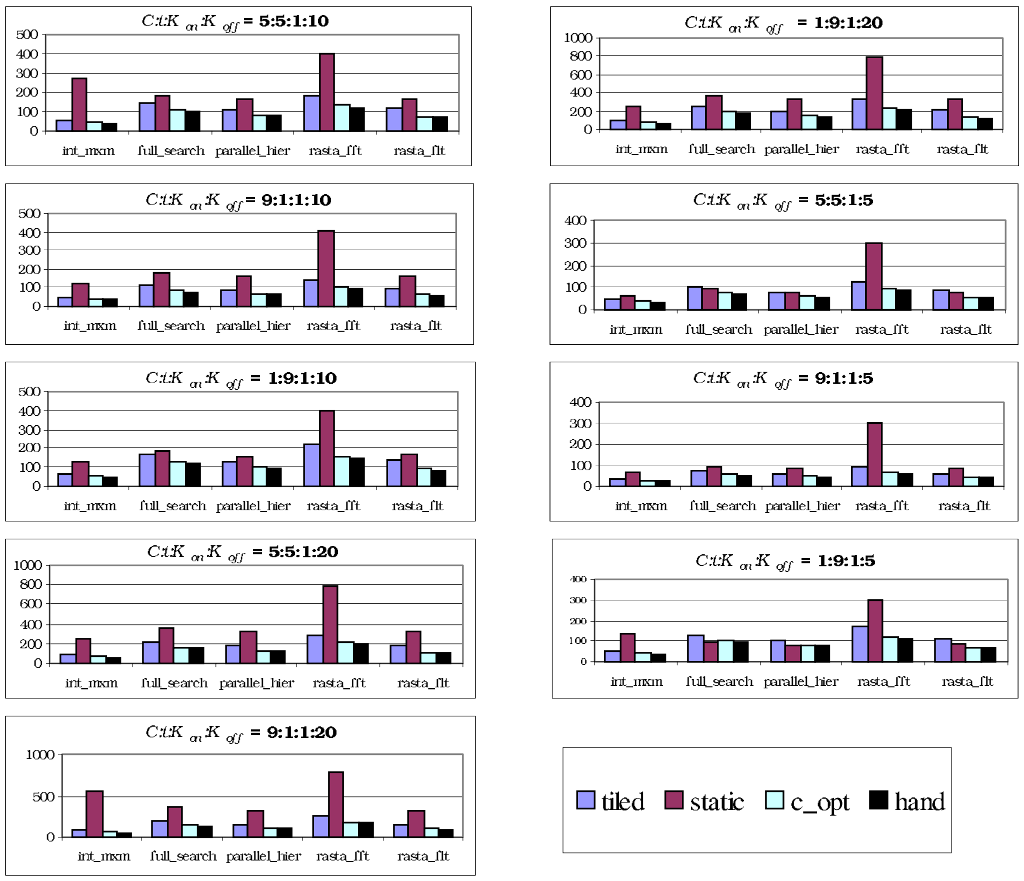

Figure 6: Total data access costs (in millions) for different versions (M=4K).

6 Summary

This paper presents a compiler-directed approach for dynamic management of SPM for array-based applications found in image and video processing domains. Our approach uses a set of compiler optimizations and an on-chip memory space partitioning strategy that aim at utilizing the on-chip memory space as efficiently as possible.

References

[1] A. Agarwal, D. Kranz, and V. Natarajan. Automatic partitioning of parallel loops and data arrays for distributed shared memory multiprocessors. In Proc. *International Conference on Parallel Processing*, 1993.

[2] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng. The SUIF compiler for scalable parallel machines. In Proc. *the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, February, 1995.

[3] L. Benini, A. Macii, E. Macii, and M. Poncino. Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation. *IEEE Design & Test of Computers*, pages 74–85, April-June, 2000.

[4] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. Custom memory management methodology – exploration of memory organization for embedded multimedia system design. *Kluwer Academic Publishers*, June, 1998.

[5] Dinero IV Trace-Driven Uniprocessor Cache Simulator. URL: http://www.cs.wisc.edu/∼markhill/DineroIV/

[6] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations, *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[7] J. Eyre and J. Bier. DSP processors hit the mainstream. *IEEE Computer Magazine*, pp. 51–59, August 1998.

[8] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In Proc. *the 37th Design Automation Conference (DAC'00)*, Los Angeles, California USA, June 5–9, 2000.

[9] D. J. Kolson, A. Nicolau, and N. Dutt. Minimization of memory traffic in high-level synthesis. In Proc. *the 30th Design Automation Conference (DAC*, June 1994.

[10] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient utilization of scratch-pad-memory in embedded processor applications. In Proc. *European Design and Test Conference (ED&TC'97)*, Paris, March 1997.

[11] P. R. Panda, N. D. Dutt, and A. Nicolau. Architectural exploration and optimization of local memory in embedded systems. In Proc. *ISSS'97*, Antwerp, Sept 1997.

[12] W-T. Shiue and C. Chakrabarti. Memory exploration for low power, embedded systems. In Proc. *Design Automation Conference (DAC'99)*, New Orleans, Louisiana, 1999.

[13] L. Wang, W. Tembe, and S. Pande. Optimizing on-chip memory usage through loop restructuring for embedded processors. In Proc. *9th International Conference on Compiler Construction*, March 30–31 2000, pp.141–156, Berlin, Germany.

[14] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, CA, 1996.