

On Combining Iteration Space Tiling with Data Space Tiling for Scratch-Pad Memory Systems

Chunhui Zhang and Fadi Kurdahi
 Department of EECS, University of California, Irvine
 Irvine, CA, 92697
 Tel: +1-949-824-5106
 {chunhuiz, kurdahi}@ece.uci.edu

Abstract— Most previous studies on tiling concentrate on iteration space only for cache-based memory systems. However, more and more real-time embedded systems are adopting Scratch-Pad Memories (SPMs) which emphasize on the management of data flow through data-oriented tiling. In this paper, we analyze the relationships between iteration space \mathcal{I} and data space \mathcal{D} , proposing a preliminary classification based on subscript functions. An important real-life application, matrix multiply, is selected to illustrate how we combine the mismatched iteration space tiling with data space tiling for optimal solutions.

I. INTRODUCTION

Traditional tiling concentrated on iteration space, known as iteration space tiling [9]. Many of them were tailored for multiprocessor systems [3]. The execution legality analysis [12] and how to distribute tiles into processors, similar to time-space transformation in systolic processing, were their major concern. The data space was either assumed to be overlapped perfectly with iteration space thus the tiling on data space was executed implicitly and simultaneously, or just ignored. The effective scope of those approaches are narrow and facing severe impacts in data dominated applications which are popular in embedded systems.

The IMEC group [5] investigated the construction problem of customized memory hierarchies and data reuse exploration for low power, they performed experiments on tile sizes with different cache topologies. In fact, most former tiling approaches [4] were developed for cache-based systems which are characterized by cache block size, associativity, (re)placement rules and so on. By contrast, more and more real-time embedded systems adopt Scratch-Pad Memories (SPMs) to solve the hard time-constraint issue because complete software-managed SPMs allow one to predict the exactly processing time. Due to the pure software-control characteristic, SPM-based systems emphasize more on memory configuration and the management of data flow [1, 2]. Therefore, data space analysis is indispensable and novel issues appear in SPM memory model.

Kandemir's work [6] investigated a dynamical management method for SPM data reuse exploration. However, only square tile or strip-mining rather than a full range of exploration on tiling were considered. Sha's group [8, 10] established a comprehensive analytical framework for tiling. They assumed software-managed memories which are actually SPMs. However, their approach was restricted to uniform loops in which data space matches iteration space almost perfectly (this will

be detailed in Section 2).

In this paper, we analyze the relationships between iteration space and data space for nested loops with regular data access patterns. Based on the degree of similarity between them, we propose a preliminary classification. For the hardest category where iteration space and data space mismatch the most, we use an important application, matrix multiply, to illustrate the efficiency of our approach—fully tiling exploration through combining iteration space with data space. The idea can be incorporated into a preprocessor or optimizing compiler.

The remainder of this paper is organized as follows: Section 2 reviews subscript function, followed by the proposed classification. In Section 3, we introduce the memory architecture and execution model. The iteration and data space combined tiling exploration is illustrated through matrix multiply in Section 4. Finally, the conclusion is drawn in Section 5.

II. ITERATION SPACE VS. DATA SPACE

A. Background

The iteration space, denoted \mathcal{I} , is the set of all loop iterations bounded by loop limits. Each iteration instance can be represented by vector $\vec{i} = (i_1, \dots, i_n)$ for a loop nest of depth n , where i_k is the value of the k -th loop index in the loop nest, counting from the outmost to the innermost loop. An iteration space \mathcal{I} can be viewed as a polytope constrained by loop bounds [12].

Constrained by array bounds, a data space \mathcal{D} can also be viewed as a set of polytopes. Each statement in the loop may include one or several references to arrays (or scalar, since the paper emphasizes on data dominated applications, scalars are ignored compared to relatively size-tremendous data arrays). The *subscript function* for a reference is a mapping from the iteration space \mathcal{I} to the data space \mathcal{D} , or more exactly, from the iteration vectors to the array elements.

Without loss of generality, we assume that the *subscript functions* are affine since many array references in embedded multimedia and communication codes are affine functions. Under this assumption, we can write a reference to an array A as $f(\vec{i}) = F^A \vec{i} + \vec{a}^A$, where F^A is a linear transformation matrix called access matrix and \vec{a}^A is the offset vector [11]. For instance, the reference to array B in Fig. 1 (b) can be written as:

$$f(\vec{i}) = F^{B\vec{i}} + \vec{a}^B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad (1)$$

```

for (i=1; i<=m; i++)
  for (j=1; j<=n; j++)
    { v1[i][j]=5*v4[i-1][j];
      v2[i][j]=v4[i-1][j-1]+4.7;
      v3[i][j]=v1[i][j-1]+v2[i][j];
      v4[i][j]=2*v3[i][j]+0.5;
    }
  (a)

for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      A[i][j]=A[i][j]+B[i][k]*C[k][j];
  (b)

for all i, j, k
  A'[i][j][k]=A[i][j];
  B'[i][k][j]=B[i][k];
  C'[k][j][i]=B[k][j];
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      A'[i][j][k]=A'[i][j][k-1]+B'[i][k][j]*C'[k][j][i];
A[i][j]=A'[i][j][N];
  (c)

for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    A[i][j]= $\sum_{k=0}^{N-1} B[i][k]*C[k][j]$ 
  (d)

```

Fig. 1. Different code fragments

B. Proposed Classification on \mathcal{I} - \mathcal{D} Similarity

Iteration space tiling usually aims at *uniform loops* only [8, 10]. *Uniform loops* are those that present the characteristic of constant dependency vectors. Thus, the whole iteration space \mathcal{I} can be represented by one iteration instance (including the dependencies with its neighbors) with boundary descriptions. An *uniform loop* example is given in Fig. 1 (a).

For *uniform loops*, the iteration space \mathcal{I} and data space \mathcal{D} can be easily linked and merged together to apply tiling transform. Since the \mathcal{I} and the \mathcal{D} have the same shape (when viewed as polytopes), tiling on one will automatically and simultaneously apply to the other. However, this rule is not true for general loops.

Based on subscript function, we propose a preliminary classification on \mathcal{I} - \mathcal{D} similarity for general loops, shown in Table I. Case 1 and 2 are for *uniform loops*, where the access matrix is an identity matrix. Distinguished by whether offset vector is zero or not, case 1 is “*perfect match*” and case 2, “*perfect match with offset*”. The third case is a superset of the first two cases, where the access matrix is a nonsingular square matrix. Since diagonalization of a matrix is always possible, the access matrix can be reduced to a diagonal matrix (could be Smith normal form specifically [12]), after the corresponding and respective data array and iteration space transforms. The transformed \mathcal{I} and \mathcal{D} are linearly related, thus called “*linear match*”. Otherwise, \mathcal{I} and \mathcal{D} are classified as “*mismatch*” and hard to be combined together through conventional approaches.

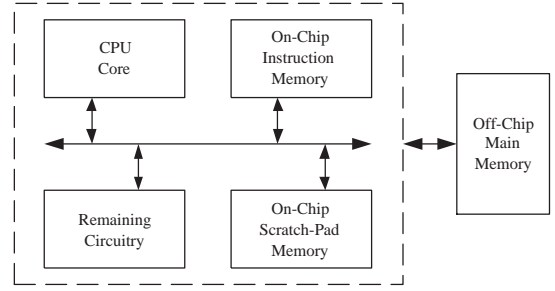


Fig. 2. A generic SPM-based system architecture

TABLE I
RELATIONSHIP BETWEEN ITERATION AND DATA SPACE

Case	Access matrix	Offset vector	Note
1	Identity matrix ^a	Zero	Perfect match
2	Identity matrix ^a	Non-Zero	Perfect match with offset
3	Square & nonsingular	-	Linear match
4	others	-	Mismatch

^aApplicable to matrix which could be reduced to identity matrix

III. ARCHITECTURE AND EXECUTION MODELS

A. Architecture Model

Unlike cache which is hardware-controlled, SPM is managed by software. For the applications with regular data access pattern, an SPM can outperform traditional cache as software can lead to better data flow management. Furthermore, the software-controlled data flow is predictable and could happen at any level of granularity. Compared to cache’s fine-grain (cache line) granularity, SPM is a better choice for coarse-grain tiling transform.

Fig. 2 shows the architecture block diagram of a typical single processor embedded core system. It consists an instruction memory, an SPM for data storage, a CPU core and a main memory. The main memory is assumed to be off-chip and usually realized by DRAM with high access latency. The rest of the components are fabricated on-chip. The instruction memory can be either implemented by cache, which is popular in RISC embedded architectures, or SRAM as in most DSP platforms. The architecture is highly abstracted that it can adapt to most current embedded processors, e.g., the LSI Logic CW3300 RISC, the TI C5X DSP core processor.

B. Execution Model

The adopted execution model assumes a common situation where all data and instructions are located in off-chip memory initially. An instruction can only operate when it has been loaded in on-chip memory and the needed data is available in the SPM.

In this work, we are interested in loop-level applications and \mathcal{I} - \mathcal{D} combined tiling for efficient data flow management. Since the loop code is usually condense with limited length and desirable locality, a nested loop can reside in on-chip instruction memory and run for certain time with ignorable loading costs. Consequently, the major task is to manage the data transfers between SPM and off-chip memory, specially when the total data size far exceeds the SPM capacity. Additionally, the execution model assumes the following:

- **Memory hierarchy and cost model.** On-chip SPM access is fast and even costless in most cases when memory allocation and access scheduling techniques are adopted [7], but the size is limited thus only restricted number of tiles can be accommodated at one time. Access to the off-chip DRAM takes considerable time. The cost to transfer n consecutive data items between off-chip DRAM and the SPM can be model as $T = C_s + n * C_t$. C_s stands for the start-up penalty including access latency to off-chip DRAM and software initialization overheads. C_t is the cycles to transfer one data item in the stable status.
- **Tile scheduling.** Tiles are atomic with synchronization happening only at the starting and ending points of tiles. The execution sequence of tiles is lexicographic. Tiling would bring in extra communication costs to avoid dependency-violation. The communication cost also depends on tile shape and size.

In our execution model, larger size tiles can always reach better performance.

IV. COMBINED TILING EXPLORATION WITH CASE STUDY ON MATRIX MULTIPLY

A. Matrix Multiply—Obscuring Solely Iteration Space Tiling

Matrix multiply has always been an important function which is widely used in communication, image and video processing, as well as many other areas in science and engineering. Its extension covers a broad application scope. QR, LU and Cholesky decompositions can all be viewed as the subsets of matrix multiply.

Fig. 1 (b) shows a matrix multiply code segment. The corresponding subscript functions show that matrix multiply belongs to case 4 in the proposed classification of Table I, where iteration space and data space are mismatched. In order to merge data space \mathcal{D} into iteration space \mathcal{I} to unify their dimensional mismatch, there are two solutions—either (1) expand \mathcal{D} to 3D or (2) collapse \mathcal{I} to 2D. Fig. 1 (c) and (d) show the corresponding code modified for solution (1) and (2) respectively. However, the first solution incurs too much overheads for data copying. The second solution is further considered.

If a *Multi-dimensional Data Flow Graph* (MDFG) $G = (V, E, d, t)$ ¹ [8] is applied for pre-tiling analysis, the *Loop Carried Dependency* (LCD) is along k index, $(0, 0, 1)$. Once index k is collapsed and the 3D iteration space is changed to 2D, no LCD exists at all. Therefore, the MDFG gives almost no helpful information.

Since MDFG only exposes true data dependencies, an extended version of MDFG which adds the value dependencies for better data flow analysis can be employed. In this extended MDFG model, LCD is a direct vector ranging between $(-N + 1, N - 1)$. In other words, the dependencies are non-uniform which complicates tiling exploration. Generally, non-uniform dependencies are circumvented by loop interchange. Loop interchange restricts all non-uniform dependencies in the

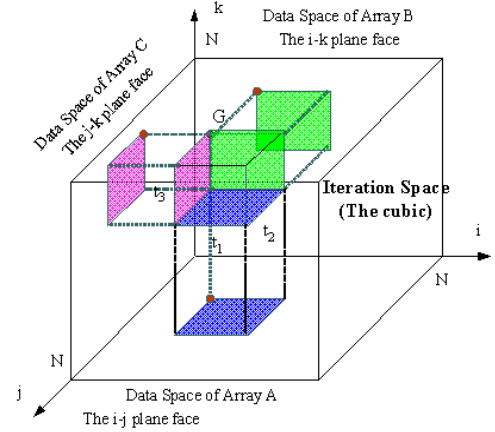


Fig. 3. Connect \mathcal{I} and \mathcal{D} of matrix multiply

innermost loops in order to expose possible coarse-grain parallelism in the outmost loops. Unfortunately, loop interchange is incapable of extracting such coarse-grain parallelism from matrix multiply.

B. Connecting \mathcal{I} and \mathcal{D} of Matrix Multiply

The failures of those conventional approaches expose a paradox for \mathcal{I} - \mathcal{D} “mismatch” problems. On one side, \mathcal{I} and \mathcal{D} show great dissimilarities, e.g., tile shapes for \mathcal{I} and \mathcal{D} usually differ from each other. On the other side, tiling on \mathcal{I} and tiling on \mathcal{D} are somehow related, a tiling configuration on \mathcal{I} will naturally result in a particular tile shape for \mathcal{D} despite their “dimensional mismatch”.

Fig. 3 illustrates such a \mathcal{I} - \mathcal{D} relationship in matrix multiply. In the polyhedron view, the iteration space is a cube while the data space is actually composed of three faces of the cube, each represents one of the three data arrays.

Every iteration instance G , represented by iteration vector $G = (i_G, j_G, k_G)$, needs the corresponding data from the three data arrays which are just the projections from G onto xy , xz and yz planes. When we tile \mathcal{I} into orthogonal polytope with side sizes t_1, t_2, t_3 , the corresponding data needed will be from the projected rectangular areas on the faces of \mathcal{D} , referred to Fig. 3. Thus, Fig. 3 explains how \mathcal{D} and \mathcal{I} are connected, what and how many data items are necessary to be transferred to SPM for executing the statements within one tile.

C. Formalizing the Ideal Data Reuse Efficiency

According to the \mathcal{I} - \mathcal{D} relationship shown in Fig. 3, we formalize the extent of data reuse as *data reuse efficiency ratio*, called **computation-communication ratio** R_{cc} . The ratio denotes how many iterations can be executed per data item transfer. For a tiling configuration $\vec{T} = (t_1, t_2, t_3)$, R'_{cc} in Equation (2) gives a partial efficiency ratio without considering temporal reuse. Since tiles are scheduled lexicographically, one of the three data arrays shows good temporal locality. Matrix multiply is also permutable, so we can further re-arrange the scheduling sequence among indices for the best temporal locality exploration without violating the legality.

$$R'_{cc} = \frac{\prod_{i=1}^3 t_i}{t_1 t_2 + t_1 t_3 + t_2 t_3} \quad (2)$$

$$R_{cc} = \frac{\prod_{i=1}^3 t_i}{t_1 t_2 + t_1 t_3 + t_2 t_3 - \max\{t_1 t_2, t_1 t_3, t_2 t_3\}} \quad (3)$$

¹MDFG $G = (V, E, d, t)$ is a node-weighted and edge-weighted directed graph used to represent a nested loop of computation. V is the set of computation nodes, $E \subseteq V \times V$ is the set of data dependence edges, d is a function from E to \mathbb{Z}^n representing the multi-dimensional delay vector between two nodes, where n is the number of dimensions, and t is the computation time of each node.

Consequently, Equation (3) gives the ideal data reuse efficiency R_{cc} with optimal temporal locality. Here “ideal” means that the matrices are assumed to have infinite sizes thus the boundary effects can be ignored, and only input is considered. “Optimal” means that the scheduling keeps the largest data array tile to be reused throughout the innermost loop.

Since larger tile always results in better data locality, the tile size is enlarged to possible sizes constrained only by SPM capacity. Thus, tiling exploration is pursued by adjusting the lengths of any two tile sides since the third side is then fixed. As a result, our tiling scheme offers an $O(N^2)$ exploration space as compared with the approach in [6] which only discussed three particular situations.

Through Equation (3) we find that the optimal candidates share the same tiling configuration: one of the three arrays is tiled as a square while the other two arrays are tiled as a truncated line (partial row or column) with the length being the same as the square side. In short, one of t_1 - t_3 is one and the rest two are equal. Which means the optimal solution is a specific position in the $O(N^2)$ exploration space. This optimal solution can be used directly in real applications without searching in the $O(N^2)$ exploration space again.

D. Tiling for Communication

As abstracted in the execution model, the communication cost is composed of two parts, start-up penalty and stable cost. We assume that the data layout in off-chip DRAM is either row or column-major. Each data array is determined independently for smaller start-up penalty optimization. The SPM is allocated with size S which is divided among all the arrays, and S is far less than N^2 where N is the array size. Because of the symmetry, we assume the relation $t_1 \geq t_2 \geq t_3$ for simplification. Equation (4) gives the overall cost for the loop shown in Fig. 1 (b) after tiled as (t_1, t_2, t_3) with read activity alone.

$$\begin{aligned} T &= T_A + T_B + T_C \\ &= \frac{N^2}{t_1} \left(1 + \frac{2N}{t_2}\right) C_s + N^2 \left(1 + \frac{N}{t_2} + \frac{N}{t_1}\right) C_t \quad (4) \end{aligned}$$

Several conclusions can be drawn from the above equation. First, larger t_1 and t_2 values will result in smaller communication cost, therefore, the best t_3 value is 1. Second, the costs of start-up and stable add reverse requirements on the tile shape—“strip” versus “square”. Third, the optimal communication cost should lie between “strip” and “square”, and affected by the relative values of C_s, C_t .

E. Experiments

The effectiveness of our combined tiling scheme is verified through testings on a customized simulator.

The proposed schemes are compared with conventional approaches. Scheme *3squ* [6] partitions all the three arrays to equal squares and share the SPM evenly. We further optimize *3squ* for temporal locality, namely *Opt_3squ*. *Chunk* [6] partitions two arrays into shape $l \times N$ with the third array, $l \times l$. *Opt_Rcc* is our scheme for data reuse optimization. The distinct feature of *Opt_Rcc* is: the SPM is almost occupied by one data array tiled as square and reused throughout the innermost loop. Our second scheme, *Opt_cost*, compromises between the two items in Equation (4) for smallest T , resulting in communication-minimal solution.

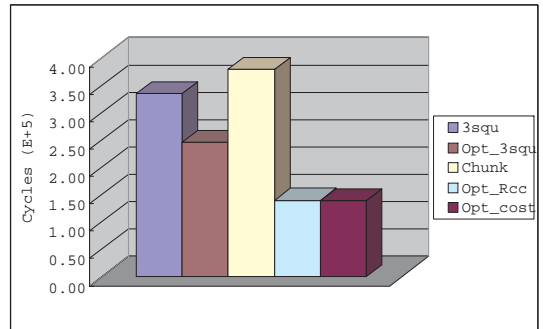


Fig. 4. Comparison of different schemes on total transfer cycles

Due to space concerns, we only present the figures in a typical configuration, where $S = 2K$, $N = 128$, $C_s/C_t = 10$. Fig. 4 compare the different schemes on the communication costs. In our approach *Opt_Rcc*, data array A is tiled into four 44 by 44 blocks, four 44 by 40 blocks and one 40 by 40 block.

The two proposed schemes show close results. Compared to *3squ*, *Opt_3squ* and *Chunk*, our scheme *Opt_Rcc* reduces the coefficient of C_s by 72.3%, 62.1% and 46.2% respectively. The improvements on the coefficient of C_t are 52.8%, 35.9%, 65.6%, and finally result in total cycles reductions as 58.2%, 43.2% and 63.2%.

V. CONCLUSION

This paper made a preliminary step towards combining iteration space tiling with data space tiling for scratch-pad memory systems. The idea presented in this paper can be adopted as an automatic pre-processing or compiler optimization step.

REFERENCES

- [1] *TMS320C54x DSP Functional Overview*. Texas Instruments Inc., <http://focus.ti.com/lit/ug/spru307a/spru307a.pdf>.
- [2] *ADSP-21xx Processor*. Analog Devices Inc., <http://www.analog.com/processors/processors/ADSP/>.
- [3] R. Andonov, H. Bourzoufi, and S. Rajopadhye. Two-dimensional orthogonal tiling: from theory to practice. In *Proc. HPC '96*, pages 225–231.
- [4] A. Badawy, A. Aggarwal, D. Yeung, and C. Tseng. Evaluating the impact of memory system performance on software prefetching and locality optimizations. In *Proc. Supercomputing '01*, pages 481–500.
- [5] F. C. et al. *Custom Memory Management Methodology/Exploration of Memory Organization for Embedded Multimedia System Design*. Kluwer Academic Publishers, 1998.
- [6] M. K. et al. A compiler-based approach for dynamically managing scratch-pad memories in embedded systems. *IEEE Trans. on CAD*, 23(2):243–260, Feb. 2004.
- [7] R. Leupers, A. Basu, and P. Marwedel. Optimized array index computation in dsp programs. In *Proc. ASP-DAC '98*.
- [8] N. L. Passos and E. H.-M. Sha. Achieving full parallelism using multi-dimensional retiming. *IEEE Trans. Parallel and Distributed Systems*, 7(11):1150–1163, Nov. 1996.
- [9] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for nonshared memory machines. In *Proc. Supercomputing '91*, pages 111–120.
- [10] Q. Wang, E. H.-M. Sha, and N. L. Passos. Optimal data scheduling for uniform multi-dimensional applications. *IEEE Trans. Computers*, 45(12):1439–1444, Dec. 1996.
- [11] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley Publishing Company, 1996.
- [12] J. Xue. *Loop tiling for parallelism*. Kluwer Academic Publishers, 2000.