International Journal of Parallel Programming, Vol. 35, No. 1, February 2007 (© 2007)
DOI: 10.1007/s10766-005-0027-1

Reducing Off-Chip Memory Access via Stream-ConsciousTiling on Multimedia Applications

Chunhui Zhang,[1,2] Fadi Kurdahi[1]

Received March 15, 2006; accepted October 10, 2006

The iteration space of a loop nest is the set of all loop iterations bounded by the loop limits. Tiling the iteration space can effectively exploit the available parallelism, which is essential to multiprocessor compiling and pipelined architecture design. Another improvement brought by tiling is the better data locality that can dramatically reduce memory access and, consequently, the relevant memory access energy consumptions. However, previous studies on tiling were based on the data dependence, thus arrays without dependencies such as input arrays (data streams) were not considered. In this paper, we extend the tiling exploration to also accommodate those dependence-free arrays, and propose a stream-conscious tiling scheme for off-chip memory access optimization. We show that input arrays are as important, if not more, as the arrays with data dependencies when the focus is on memory access optimization instead of parallelism extraction. Our approach is verified on TI's low power C55X DSP with popular multimedia applications, exhibiting off-chip memory access reduction by 67% on average over the traditional iteration space tiling.

KEY WORDS: tiling; memory hierarchy; data locality; iteration space; DSP; low power.

1. INTRODUCTION

Tiling, or blocking, has been extensively studied in the context of *uniform loops* due to its uniformity — computations are identical over the entire

[1]Department of EECS, University of California, Irvine, CA, 92697, USA
[2]To whom correspondence should be addressed. E-mail: chunhuiz@uci.edu

loop index space, thereby the analysis can be simplified within a single iteration instance.(1–4) It is well-known that tiling is an effective method in defining more convenient memory access patterns, improving data localities and, consequently, saving energy consumptions.

1.1. Iteration Space and Data Space

The iteration space $\mathcal{I}$ (computation domain) is the set of all loop iterations bounded by the loop limits. Each iteration instance can be represented by a vector $\vec{i} = (i_1, \ldots, i_n)$ for a loop nest of depth n, where i_k is the value of the k-th loop index, counting from the outermost to the innermost loop. An iteration space is represented by the integer points contained within a convex polyhedron $\mathcal{I} = \{\vec{i} \mid \mathcal{B}\vec{i} \leq \vec{b}\}$.(5)

Constrained by array bounds, the data space $\mathcal{D}$ can also be viewed as a set of polytopes. For a given nested loop, $\mathcal{D}$ is formed only by the referenced array elements despite the original array declarations. Each statement in the loop may include one or several references to the arrays or scalars (the degenerations of arrays).

1.2. Iteration Space Tiling

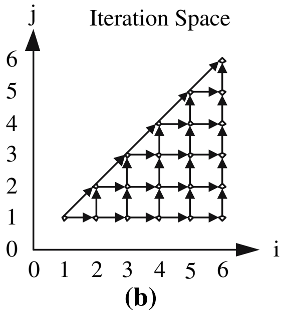
Traditional tiling schemes focused on iteration space, known as the iteration space tiling.(6) A tile in the iteration space is the collection of iterations to be executed as an atomic unit. Usually the nest depth is doubled after tiling, although there are exceptions, e.g., strip-mining (a special case of tiling). Fig. 1(a)–(d) illustrates a piece of example code together with its iteration space before and after tiling, where the original 2-D loop nest is mapped to a 4-deep loop nest by tiling. The tiled code in Fig. 1 (c) uses a tile size of 2×2. The two innermost loops execute the iteration instances within each tile, represented as 2×2 shadowed square in Fig. 1 (d). The two outer loops, represented by the two axes in the figure, execute the 6 tiles.

Tiling the iteration space may improve the available parallelism, hence many tiling techniques were tailored for parallel machines to exploit the intrinsic concurrencies.(1,2,6) Their major concerns were the execution legality and how to distribute tiles onto processors, which are similar to the time-space transformation in systolic processing. Likewise, tiling on iteration space introduces more parallelism to structural or functional pipelining in single processor systems.(7) Loop tiling may also improve data locality.

The essential issue in iteration space tiling, either for multiple or single processor systems, is the data dependence relation. Since ordering

```
for i = 1 to 6 do
  for j = 1 to i do
S1:  d1(i, j) = 5 * d3(i-1, j);
S2:  d2(i, j) = d3(i-1, j-1) + 4.7;
S3:  d3(i, j) = d1(i, j-1) + d2(i, j);
```

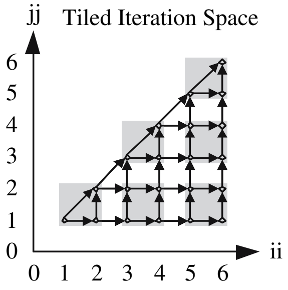
(a)



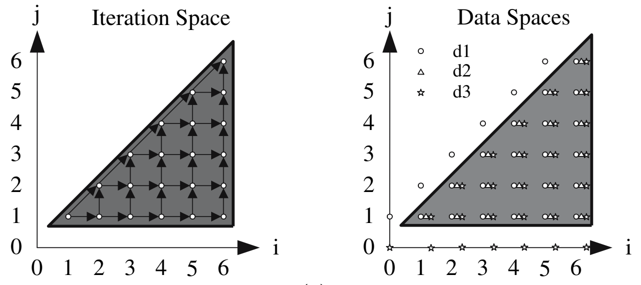
(b)

```
for ii = 1 to 6 by 2 do
  for jj = 1 to ii by 2 do
    for i = ii to min(6, ii+1) do
      for j = jj to min(ii, jj+1) do
S1:         d1(i, j) = 5 * d3(i-1, j);
S2:         d2(i, j) = d3(i-1, j-1) + 4.7;
S3:         d3(i, j) = d1(i, j-1) + d2(i, j);
```

(c)



(d)



(e)

Fig. 1. Iteration space, data space, and tiling: (a) example code of a nested loop; (b) the corresponding iteration space and dependencies; (c) the code after tiling and (d) the tiled iteration space; (e) data spaces match the iteration space (the solid area) except at the boundaries.

constraints determine the extractable parallelism, it is not surprising that tiling related models were branded with data dependence, that we call "dependence models", e.g., the *Multi-dimensional Data Flow Graph* (MDFG),(4) the data dependence graph,(8) the dependence array,(9) etc. It also justifies that why previous studies(1,2,6,7) favored certain uniformly recurrent equations (UREs)–all data arrays in the UREs involve data dependencies thus can be abstracted in the dependence models. However, data arrays not involving dependence, among which input array is the common case, were excluded from the consideration of the traditional iteration space tiling.

1.3. The Motivating Examples

A common characteristic in those UREs is the tightly matched iteration space and data space. Actually, the example code in Fig. 1(a) is a typical URE loop nest, in which the iteration space $\mathcal{I}$ and the data spaces of $\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_3$ (representing the data arrays d1, d2, d3, respectively) share the same triangle shape except at the boundaries, shown in Fig. 1(e). Therefore, the data spaces can be tiled in the same shape spontaneously and implicitly when the iteration space tiling executes. At the same time, it can show decent localities.

Unlike the URE example where the iteration space and data spaces are well matched, the $\mathcal{I}$–$\mathcal{D}$ relationship of our second example, matrix multiplication, is a little intricate as illustrated in Fig. 2. In the polyhedron-view, the iteration space is a cube while the data spaces are composed of three faces of the cube, each representing 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 the xy, xz and yz planes. When we tile $\mathcal{I}$ into an orthogonal polytope with the side sizes of t_1, t_2, t_3, the corresponding data needed will be from the projected rectangular areas on the faces of $\mathcal{D}$.

Motivated by the two examples, we come to a point that memory access optimization should be application-dependent, which can be justified by two pieces of evidence. First of all, an application may have well-matched loop and data array indices while another may exhibit complex relationships between its iteration space and data space. Secondly, the input for an application may be either a small amount of data serving as a prologue (the URE example) or large-sized data streams in another (the matrix multiply example).

1.4. Stream-Conscious Tiling

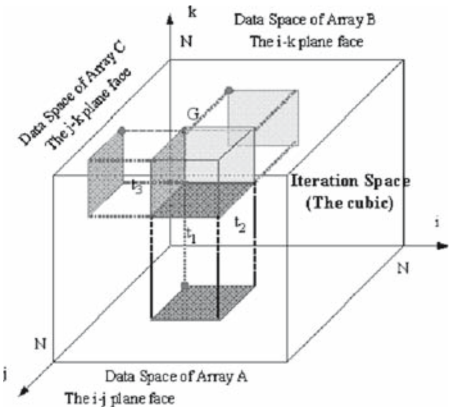
The disproportionate developments of processor and memory speeds have hindered the overall system performance. To circumvent this speed gap, memory hierarchy plays a dominant role with advantages in terms of area, performance and power. In fact, most multimedia applications have

```
for i = 0 to N do
  for j = 0 to N do
    for k = 0 to N do
      A(i, j) = A(i, j) + B(i, k) * C(k, j);
```

(a)

```
for ii = 0 to N by t1 do
  for jj = 0 to N by t2 do
    for kk = 0 to N by t3 do
      for i = ii to ii+t1-1 do
        for j = jj to jj+t2-1 do
          for k = kk to kk+t3-1 do
            A(i, j) = A(i, j) + B(i, k) * C(k, j);
```

(b)



(c)

Fig. 2. (a) The code of matrix multiply before tiling and (b) after tiling; (c) Connecting the $\mathcal{I}$ and the $\mathcal{D}$ of matrix multiply.

large input data streams buffered in the external main memory. Tiling on those streams is somehow mandatory and significantly impacts conventional iteration space tiling techniques in several ways:

(i) Tiling on input arrays has more significant effects on communication. Here we refer to "communication" as the off-chip memory access, and the off-chip memory access count is simplified as the "memory access". For UREs, communication volume is approximated by the number of dependence vectors across a tile boundary.(7) It is significantly smaller than the communication volume of the input arrays, which is proportional to the tile itself.

(ii) The data space of an input array usually exhibits its own shape rather than that of the iteration space, e.g., the matrix multiply example in Fig. 2. We will demonstrate later that it is exactly the mismatched iteration and data space that introduces a large exploration potential for memory access optimization.

(iii) Dependence models are incapable of abstracting any information from dependence-free input arrays.

Actually, dependence-oriented iteration space tiling was not contrived for demands other than parallelism extraction. Therefore, there is a clear need to propose a stream-conscious tiling technique with the goal of memory access optimization, in terms of access count, time and energy consumption.

1.5. Target Application Domain

Although it is a "hard" problem in general, the objective of memory access optimization is achieved in this work by focusing on a specific application domain — multimedia applications, including video processing, medical imaging, multimedia terminals, artificial vision, and so on. The main characteristics of those applications are: data-dominated, nested loops, manifest affine conditions and indices, static data, and so on. As a consequence, most example loops in this context are regular, that is to say, perfectly nested, with simple indexing, and free of conditional statements, thereby containing possible analytical solutions. Admittedly, more and more multimedia applications, such as MPEG-4, have data dependent conditions and indexing, and are quite irregularly nested. Nevertheless, appropriate pruning on such specifications can solve this issue partially, and the "complete" solution is a topic for future research.

In order to take advantage of data locality without violating execution legality, we propose the stream-conscious tiling exploration technique for memory access reductions considering both the iteration space and data arrays. Since memory access, especially external memory access, makes a prominent contribution towards system-level power consumption, this idea is further extended for the purpose of energy reduction. The approaches are verified on TI's low power C55X DSP. Experiments show that the reduction on off-chip memory access can reach 85% (about 67% on average) over traditional iteration space tiling and more than 38% (about 26% on average) over another data oriented scheme.

The rest of this paper is organized as follows: Section 2 reviews the related work and Section 3 describes the models. A novel loop representation, xMDFG (eXtended Multi-dimensional Data Flow Graph), is proposed in Section 4. Our tiling exploration scheme is presented in Section 5 with the case studies illustrated in Section 6. Section 7 verifies our approaches on a TI's DSP platform. Finally, the conclusion and future work are drawn in Section 8.

2. RELATED WORK

The work done by Andonov et al.(1) and Robert et al.(9) represents the typical studies on the effects of orthogonal tiling on multiple processor systems. In,(1) the authors modeled programs of uniform recurrence equations (UREs), and showed how the methods of systolic array synthesis can be profitably used for optimal tiling. Although the realistic model of the communication costs of general purpose multiprocessors yields a particular non-linear optimization problem, they used an analytic solution and gave a closed form formula for the optimal tile size and processor count. Robert et al.(9) further developed the problem to the mapping and scheduling of the tiles on to physical processors under limited computational resources. In addition, Carter used the Parallel Memory Hierarchy (PMH) model to improve superscalar performance(2) via tiling.

In any case, the focusses in those tiling works were mainly parallel computations on multiprocessors. They were interested in URE examples whereas arrays that did not show dependence were assumed to be ready on local memories for processing. Therefore, the demands on data communication given by their formulas were low and the communication models in those works were overlooked due to their insignificant effects. They usually assumed (sometimes implicitly) caches to be of the local memory type.

The emergence and popularity of real-time embedded systems pushed the demand of predicability as well as the improvement on the worst case execution time (WCET) for the applications running on them. It turned out that software-managed scratchpad memory was satisfactory. In,(10) an architecture containing both cache and scratch pad memory is used. Arrays that are too large to fit in the SPM are therefore kept in the main memory and accessed through the data cache. Scores of similar works to improve either localities or predicability using the SPM can be found.(11)

To this date, however, not many studies have looked into the problem of tiling on software-managed memories. Kandemir's work(12) investigated the dynamically partitioning issue on software-managed memory systems for data reuse exploration, but only square tile and strip-mining were considered. Sha's group(3) established an analytical framework addressing both tile shape and size for computation and communication overlapping. Although memory hierarchy was addressed, they only modeled UREs without communication budget for large input arrays.

As both the data stream sizes and the adoption of software-managed memories increase dramatically in embedded multimedia systems, the new emphasis lies on the data flow management. Hence, unlike previous tiling studies that worked mainly on the iteration space, there is a clear need for data-oriented tiling. In this work, we analyze the relationship between iteration space and data space based on subscript functions, and propose a stream-conscious tiling scheme. Consequently, data tiling is no longer simply the outcome of routine code generation. Instead, it is incorporated into the exploration stage for memory access optimization.

Kadayif et al.(13) present a data space-oriented tiling strategy, representing the latest progress in this field. Their strategy works on data arrays and employs the subscript function, which is very similar to our approach. However, there are still significant differences between these two optimization strategies. First, the dependence-free input data is ignored in Kadayif et al.[13]. We note that input data arrays often contribute the largest portion to total communication volume and is the main focus in our stream-conscious tiling approach. Therefore, our stream-conscious tiling approach outperforms Kadayif's strategy when input data arrays are present.

Second, the objective of their data tile selection strategy is to minimize the number of *nontile elements* (refer to Ref. 13 for the definition), and even they admit the inaccuracy of using *nontile elements* to estimate the communication volume. In contrast, our approach works by connecting tiling shapes directly with the memory access measures, and therefore the objective of tiling exploration are established accurately for measuring memory access related costs. Although they take advantage of data locality across loop nests while our strategy focuses on singleton loop nest, Darte's cl,[14] that "general algorithms do not really subsume simpler ones because the objective functions for simpler algorithms can be more sophisticated", works perfectly here.

Finally, their tiling exploration algorithm is more expensive than ours. In their strategy, the algorithm iterates on every data array in order to identify the best seed array, while inside each iteration, the code needs to be re-generated many times, which is very costly. Furthermore, when one calculates the *seed iteration* (refer to Ref. 13), it involves matrix inversions on the subscript functions, which are usually very expensive. By contraries, our approach provides algorithms to formalize the relationship between tiling and memory access parameters straightforwardly, which can be solved in an easier manner.

3. MODEL

3.1. Program Model

The program model we use is based on the one proposed by Banerjee.[5] The model is shown in Fig. 3, where L_1 and U_1 are integer constants; L_r and U_r are integer-valued affine (linear) functions of $i_1, i_2, \ldots, i_{r-1}$. The loop bodies $S_1, \ldots, S_k$ are a totally ordered set of assignment statements, composed of a finite number of affine functions of $i_1, i_2, \ldots, i_n$. It is a perfect loop nest in the sense that there are no statements between loops. For a general loop nest, pruning it with only the innermost statements left is adequate in most cases since the innermost loop body has the largest computation domain.

The ordered execution induces dependencies between iterations. There is a dependence between statements S_i and S_j, if an instance $S_i(I)$ of S_i, an instance $S_j(J)$ of S_j (I, J are loop index vectors), and a memory location $\mathcal{M}$, satisfy the following conditions:

```
for $i_1 = L_1$ to $U_1$ do
   for $i_2 = L_2$ to $U_2$ do
      ⋱
         for $i_n = L_n$ to $U_n$ do
                Statement $S_1$
                Statement $S_2$
                      ⋮
                Statement $S_k$
         endfor
      ⋰
   endfor
endfor
```

Fig. 3. The program model.

- Both $S_i(I)$ and $S_j(J)$ refer to $\mathcal{M}$, and at least one of the references is a write;
- $S_i(I)$ is to be executed before $S_j(J)$;
- The memory location $\mathcal{M}$ is not written in the time period from the end of execution of $S_i(I)$ to the beginning of the execution of $S_j(J)$.

The dependence vector between $S_i(I)$ and $S_j(J)$ is $J - I$. The loop nest is said to be uniform if the dependence vectors do not depend on either I or J (except at the boundaries). We can then represent the loop nest as a reduced graph with k nodes (the statements) linked by edges corresponding to the dependence vectors.

3.2. Architecture Model

We adopt a generic architecture model using software-managed memories, which is different from many tiling approaches(2,15) developed for cache-based systems. The reasons are, but not restricted to, that the dynamic nature of cache restrains compile-time optimization from the extensive use of static data scheduling, and the fine-grain (cache line) feature of cache lessens the advantages of medium-grain tiling. Nevertheless, it does not mean that cache-based systems may not benefit from our method.

Many DSPs(16,17) use RAMs/ROMs as the on-chip memories; real-time embedded systems adopt Scratch-Pad Memories (SPMs) increasingly to solve the hard time-constraint issue because complete software-managed SPMs allow one to exactly predict the processing time. Figure 4 shows

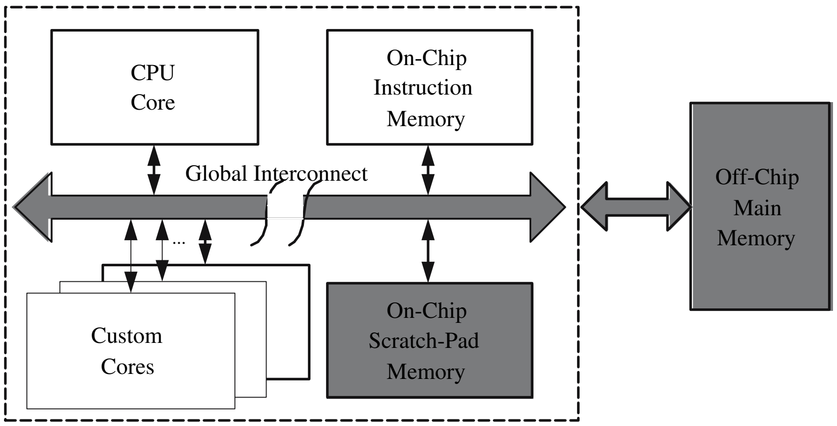


Fig. 4. The architecture model.

the architecture model block diagram. It consists of an instruction memory, an SPM for data storage, a main memory, a DSP or RISC CPU core and custom cores. 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. In the stream-conscious tiling algorithm, we are interested in the data flow between the on-chip SPM and the off-chip memory, that are shaded in color in Fig. 4.

3.3. Execution Model

The adopted execution model assumes a general situation where all data and instructions are located in the external memory initially. An instruction can operate only when it has been loaded on-chip and the required data is available in the SPM. The iteration space is divided into uniform tiles except at the boundaries to exploit better locality. Accordingly, tiling on iteration space leads to tiles on the involved arrays, called a data tile set in a whole.

Numerous achievements in pipeline synthesis since the late 1980s can be used for more compact intra-tile execution scheduling. Since the loop code is usually condensed with limited length and desirable locality, a nested loop can reside in on-chip instruction memory and run for a certain time with ignorable memory loading cost. Consequently, the major task is to manage the data transfers between the SPM and the off-chip memory. Additionally, the execution model assumes the following:

(1) SPM allocation for data tile set. An on-chip SPM has a limited size, while larger sized tiles usually provide better locality, only two data tile sets are in the SPM concurrently (one for prefetching).
(2) Communication cost. The communication cost of one transaction—to transfer n consecutive data items—is modeled as $C = C_s + n * C_t$, where C_s stands for the start-up penalty including access latency and software initialization overhead, and C_t is the cycles to transfer one data item in the stable status.
(3) Inter-tile scheduling. Tiles are atomic with synchronization happening only at the starting and ending points of tiles. Here, "atomic" means that tiles are the same in size and shape, and are applied with the same processing; "synchronization" refers to the choreographing of computation and communication. The execution sequence of tiles is lexicographic. Tiling may bring in extra communication costs to avoid a dependency violation.

Our model depicts a realistic scenario: to execute a specific loop, the input arrays need to be prefetched from the lower to the upper level of memory hierarchy. In other words, the input data is no longer assumed to be local, instead, their access cost is taken into account. This idea is also applied to the processed data that could or should not be kept in local memory and need to be output.

3.4. Activity-Based Power Consumption Model

It is well-known that a design decision made at an earlier stage has more significant impact on the final implementation. Low-energy design obeys this rule. Although there are many factors determining the power dissipation, e.g., process technology, supply voltage, operating frequency, wire length and capacitance, we are more interested in system level analysis on the memory subsystem in this work. Therefore, the dominant parameters are the count of memory access and the total access cycles, while the other factors are orthogonal to the discussion in this work.

Consequently, we accept the idea of an activity-based power analysis—the system is divided into modules that contribute their power independently based on their own properties and activities. Depending on its functionality, an individual module may have its specific power-related configurations, such as the switching ratio, the percentage of write, synchronous or asynchronous configurations, and so on. However, they share the one and often most important configuration—utilization ratio, which is popular in practical DSP energy models.(17,18)

Definition 1. Utilization ratio is a quantitative measurement of performance. The utilization ratio of a module is the percentage of its current performance with respect to its maximum achievable performance.

Utilization is an abstract concept and needs to be specified individually for different modules.

Definition 2. Activation Baseline Power, P_{base}, is the power consumption of a specific module when it is powered on but the utilization ratio is zero. Utilization Power, $P_{\text{utilization}}$, is the extra power consumption when the utilization ratio becomes non-zero.

Most of P_{base} comes from the clock tree, especially when the latter is not gated. Overall, there are two major components in the proposed energy model (shown in Eq. 1): the "baseline" energy, which is calculated by the product of P_{base} and the total run time; and the "activity" energy caused by "utilization".

$$E_{\text{total}} = E_{\text{static}} + E_{\text{activity}} = (P_{\text{base}} + P_{\text{utilization}}) \times T_{\text{run\_time}} \tag{1}$$

For memory subsystem, the "activity" energy can eventually be computed by per-access energy $E_{\text{per\_access}}$. Since the computational energy (activity energy of CPU) is relatively small, it is excluded from the discussion. Therefore, the energy model is improved in Eq. (2).

$$E_{\text{total}} = E_{\text{static}} + E_{\text{access}} = P_{\text{base}} \times T_{\text{run\_time}} + E_{\text{per\_access}} \times \text{Access\_count} \tag{2}$$

The activity-based model aids in system design for greater efficiency and is conformed to many real life DSPs.(17,18)

4. STREAM-CONSCIOUS LOOP RPRESENTATION

Most multimedia applications have very structured computations expressed in perfectly nested loops. In order to facilitate the analysis, mathematic methods such as graph theory and linear algebra have been applied to nested loops for modeling. However, classic loop representation models, e.g., the MDFG and the PRDG, are tailored for parallelism extraction and cannot abstract the input and output data flow across memory hierarchies. In this section, we propose a loop representation form, the xMDFG (eXtended Multi-dimensional Data Flow Graph), which is based on the MDFG using data alignment technique, to overcome the shortcoming. In the xMDFG, input and output data arrays are aligned and connected with the computation domain, thus the comprehensive analysis on both computation scheduling and bulk data flow becomes feasible.

4.1. Background

4.1.1. Subscript Function

The *subscript function* for a reference is a mapping from the iteration space $\mathcal{I}$ to the data space $\mathcal{D}$, specifically, from the iteration vectors to the array elements. We assume that the *subscript functions* are affine since many array references in practical codes are affine functions. Under this assumption, a reference to an array d can be written as $f(\vec{i}) = F_d\vec{i} + \vec{a}_d$, where F_d is a linear transformation matrix called access matrix and $\vec{a}_d$ is the offset (constant) vector.[8] For instance, the reference to array d in Fig. 5(a) S_1 can be written as:

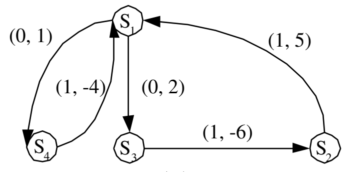
$$f(\vec{i}) = F_d\vec{i} + \vec{a}_d = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ 3 \end{pmatrix} \tag{3}$$

4.1.2. MDFG

A *Multi-dimensional Data Flow Graph* (MDFG)[4] $G = (V, E, d)$ 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 Z^n representing the multi-dimensional dependence vector between two nodes, where n is the number of dimensions. Other models, such as dependence array,[9] can be viewed as variations of the MDFG. An edge with zero-vector delay represents an intra-iteration dependence, and non-zero delay

```
for i = 0 to N do
  for j = 0 to N do
S1: a(i, j) = b(i, j - 6) + d(i - 1, j + 3);
S2: b(i + 1, j - 1) = c(i + 2, j + 5);
S3: c(i + 3, j - 1) = a(i, j - 2);
S4: d(i, j - 1) = a(i, j - 1) ;
```

(a)



(b)

Fig. 5. (a) Example code of a nested loop; (b) the corresponding MDFG.

represents an inter-iteration dependence (Loop Carried Dependence). A legal MDFG must have no zero-delay cycles. Figure 5 shows a well-known example taken from,[(19)] together with the corresponding MDFG.

4.2. $\mathcal{I}$-$\mathcal{D}$ "match degree"

Data arrays may have different shapes or even dimensions without breaking the "uniform dependence" constraint. We describe the relationships between iteration space and data space in the sense of *$\mathcal{I}$-$\mathcal{D}$ match degree*, which falls into one of the three categories: the *"perfect match"*, the *"dimensional match"*, or the *"mismatch"*. A *"perfect match"* refers to the case where the access matrix F is an identity matrix or an elementary matrix derived with only row interchange operations. The examples in Fig. 6(a) belong to this case. If the access matrix has the same rank as the loop nest depth n, we call it a *"dimensional match"*. Otherwise, the *$\mathcal{I}$-$\mathcal{D}$ match degree* falls into the *"mismatch"* category. For instance, the three arrays in the loop of Fig. 2(a) are "mismatched" to the computation domain.

4.3. xMDFG — Extended MDFG for Stream Modeling

The essential element in an MDFG is the data dependence, which constrains the scheduling of statements, while input array does not change the original MDFG at all. However, it may take more than data dependence to constrain execution: the availability of input arrays is also a constraint as long as hierarchical memories, rather than flat memories, are considered. That is, a statement can only be executed after its input data is located at the top of the memory hierarchy.

For the loops with large input data arrays, data availability dominates the data flow cost. Consequently, we refine the MDFG model, abstracting input and output arrays (streaming data) into loop representations. It belongs to a data alignment problem, where subscript function plays a key role in bridging data arrays with the computation domain.

Specifically, we associate the input array (if it exists) with the iteration space, and represent them in the MDFG using an extra node. Likewise, an extra "sink node" is assigned to identify the bulk communication for each output array. For the simple example code of Fig. 6(a), its xMDFG is given in Fig. 7(a). Those extra nodes for array input and output are actually external memory load and store operations as illustrated in 6(b).

In practice, such load and store instructions are usually grouped into independent loops executed (e.g., as DMA processing on DSP platforms) before and after the computation loop. Here, we use loop fusing to merge them into one loop for the convenience of modeling. The edge between

```
for x = 1 to M do
    for y = 1 to N do
        out(x, y) = a * in(x, y) + b;
```

(a)

```
for x = 1 to M do
    for y = 1 to N do
S1:  load in(x, y);
S2:  out(x, y) = a * in(x, y) + b;
S3:  store out(x, y);
```

(b)

```
for i = 0 to N do
  for j = 0 to N do
        d1(i, j) = d2(i, j-1) + d3(2*i, j-3);
        d2(i, j) = d1(i, j-1) + d3(i+j-2, i);
```

(c)

```
for m = 0 to IMAGE_WIDTH-MASK_SIZE do
    for n = 0 to IMAGE_HEIGHT-MASK_SIZE do
        sum = 0;
        for i = 0 to MASK_SIZE do
            for j = 0 to MASK_SIZE do
                if (mask(i, j) != 0)
                    sum += image(m+i, n+j);
        result(m, n) = sum;
```

(d)

Fig. 6. Code fragments (a) a loop belongs to a *"perfect match"* with (b) a variant of it; (c) an illustrative code for UGR; (d) code for algorithm of automatic target recognition.

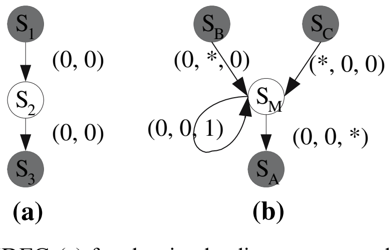


Fig. 7. Building the xMDFG (a) for the simple alignment example; (b) for matrix multiply.

the extra nodes and the original MDFG nodes is called *access dependence*. Unlike input dependence, which deals with two reads to the same memory location, *access dependence* handles data flow between different memory levels. Not only extra edges, but also extra nodes are added to the original MDFG. We name such modified MDFG as xMDFG.

4.4. Building the xMDFG

The relationship between iteration space and data arrays may be very complicated. The *access dependence* may violate the rule of "lexicographically non-negative".[20] Nevertheless, the legality analysis for the original MDFG part still stands. For each *$\mathcal{I}$-$\mathcal{D}$ match degree* category, we briefly demonstrate the methods in aligning input data with MDFG, from simple to complex. Throughout those methods, the principal idea is to re-declare (transform) the data array as compactly as possible. In the following text, A stands for the original array while B, the transformed one, and F_A, F_B are the access matrices:

(1) *The "perfect match"*. The alignment process is straightforward as in the example illustrated in Subsection 3.3. If F_A is an identity matrix, no transformation is needed and the *access dependence* is the offset vector, which can further be re-timed to a zero vector if it is not. Otherwise, the indices of the data array are interchanged so that the resulted access matrix F_B becomes an identity matrix.

(2) *The "dimensional match"*. There are two cases in a "dimensional match": F_A is non-singular square, or F_A has more rows than columns (n columns) but the rank is still n. In both cases, B is declared as $B(\vec{i}) = A(F_A\vec{i} + \vec{a}_A))$. For instance, $f_A(\vec{i}) = \left(\begin{smallmatrix} 5 & 1 \\ 3 & 2 \end{smallmatrix}\right)\left(\begin{smallmatrix} i \\ j \end{smallmatrix}\right) + \left(\begin{smallmatrix} 4 \\ 1 \end{smallmatrix}\right)$, then B is defined as $B(\vec{i}) = A(5i + j + 4, 3i + 2j + 1)$. This is the same as applying a shifting $\mathcal{M}_1 = -\vec{a}_A$ plus a transformation $\mathcal{M}_2 = F_A^{-1}$ on array A. Some elements in A may not be represented in B if F_A is not unimodular. However, all the referenced array elements in A are well mapped to array B (indices are integers). The resulting *access dependence* is a zero-vector.

(3) *The "mismatch"*. Although the method for a "dimensional match" case is still feasible, A can be re-declared in a more compact way. We use the *reduced form*, a special echelon form that each leading non-zero entry is a one and all entries above and below such leading ones are zeros. For example, $F_A(\vec{i}) = \begin{pmatrix} 1 & 2 & 0 \\ 3 & 0 & 2 \\ 0 & 1 & 1 \end{pmatrix}$ has the *reduced form* $R = \begin{pmatrix} 1 & 2 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$. The complete reduction procedure uses row interchange and multiplication operations (refer to Ref. 21 for details). Via the *reduced form*, array B is declared in almost the most compact way, $F_B = R$ and $B(f_B(\vec{i})) = A(f_A(\vec{i}))$.

However, the *reduced form* of a matrix may contain fractional numbers. In this case, row multiplication operation is applied to ensure that all entries are integers. Moreover, the *access dependence* is no longer a distance. Figure 7(b) gives the xMDFG for the matrix multiply code in Fig. 2 (a). The notation $*$ stands for $[-\infty, +\infty]$ as defined in.[20] Such dependence is known as direction dependence. Although direction dependence is imprecise, it is accurate enough for our subsequent analysis using the xMDFG. Furthermore, the quantitative calculation in our proposed algorithm uses the subscript function directly.

(4) *Multiple references and Uniformly Generated Reference set (UGR).* An array can be referenced multiple times in the loop body. A UGR is a set of references to the same array whose subscript functions differ only in the offset vector, e.g., $d_1(i, j)$ and $d_1(i, j-1)$. In those cases where there are multiple UGRs for one array, each UGR should be treated as an independent single reference. Even a single UGR may be handled as different arrays, e.g., $d(2i+1, j)$ and $d(2i-4, j+3)$ are a UGR, but should be considered as two arrays.

In summary, aligning input data arrays with iteration space is nontrivial, but is generally feasible after proper array transformations and re-declarations. To align output data arrays may be even more complicated because of the involvement with data dependence. However, many multimedia applications are coded in a way that the access patterns are quite simple and regular, thus building their xMDFGs is rather straightforward, as shown later in the case studies.

5. THE STREAM-CONSCIOUS TILING TECHNIQUE

5.1. Preprocessing for Orthogonal Tiling

Despite the fact that tiles can be of any size and shape, we only consider orthogonal tiling in this context [1](squares and rectangles in 2-D). Although intuitively it may seem restrictive, orthogonal tiling is tacitly taken by most researchers, even in mature compilers such as Paradigm and Fortran 90.(23,24) In addition, parallelepiped tiling brings undesirable complexities in handling the tile boundaries during the code generation step, and finally increases the code size. On the other hand, parallelepiped tiles can usually be transformed into orthogonal tiles by applying loop skewing techniques.(20) For a given n-Dimensional iteration space $\mathcal{I}$, orthogonal tiling can be represented as $\vec{T} = (T_1, \ldots, T_n)$ with T_i standing for the tile size of the ith dimension of $\mathcal{I}$.

A tile on the iteration space corresponds to one or more tiles on the data space set. Relying on the subscript functions, the data tile set can have the shapes different from the iteration space tile shape. In order to guarantee a "uniform" and "simple" relationship between iteration and data space, an extra preprocessing procedure is raised. As a result, the properties of "atomic" and "orthogonal" of iteration space tiling remain

[1]Note that directly applying orthogonal tiling to some loops may result in illegal scheduling,(3) however, re-timing techniques(22) can increase the legal range of orthogonal tiling largely.

applicable for data space tiling. Roughly speaking, *this preprocessing procedure is the procedure of building the xMDFG*. As demonstrated in Section 3.4, data arrays are all changed into the following forms after preprocessing: the access matrix is an (1) identity matrix; (2) elementary matrix derived with only row interchange operations; (3) (general) integer-only *reduced form*. The consequent loop nest has following properties:

Property 1. For all three forms of the preprocessed access matrix, there is at most one non-zero number in any column.

The property is obvious through the definitions of the *reduced form* and the identity matrix.

Property 2. After building the xMDFG, orthogonal tiling on the iteration space results in orthogonal data space tiles.

Proof: The iteration space $\mathcal{I}$ can be viewed as a linear space Z^n spanned by *standard basis* of the iteration indices, where $\vec{i}_1 = (1, 0, \dots, 0)$; $\vec{i}_2 = (0, 1, \dots, 0)$; $\vec{i}_n = (0, \dots, 0, 1)$. The orthogonal tiling on the iteration space, including both tile directions and sizes, are represented by tiling vectors, $T_1\vec{i}_1, T_2\vec{i}_2, \dots, T_n\vec{i}_n$, which are mutually perpendicular. A data space Z^m can be viewed as a linear transformation from the iteration space, while the access matrix F is the transformation matrix. If the transformed data tiling vectors, $\vec{dT}_1 = \sum_{k=1}^{n}(F_{1k}T_k\vec{i}_k), \dots, \vec{dT}_m = \sum_{k=1}^{n}(F_{mk}T_k\vec{i}_k)$ (F_{rc} is the r-th row and c-th column entry of F) are also mutually perpendicular, then it is proven. Suppose there are two data tile vectors $\vec{dT}_a$ and $\vec{dT}_b$ that are not perpendicular, so their dot product $(\vec{dT}_a) \cdot (\vec{dT}_b) = \sum_{k=1}^{n}(F_{ak}F_{bk}T_k^2)$ is non-zero. Therefore, at least one of the sum items is not zero. Let $(F_{ac}F_{bc}T_c^2) \neq 0$, then both F_{ac} and F_{bc} are not equal to zero, which violates Property 1. Thus, the data tiles are also orthogonal[2].

Property 3. After building an xMDFG, uniform tiling on the iteration space will result in uniform data tile sets except at the boundaries.

Proof: The data tile size can be calculated via the formula $(f(\vec{i}_o+\vec{T})-f(\vec{i}_o))'$, where $\vec{i}_o$ is the tile offset vector. Since f is a linear transformation composed of an affine matrix and an offset vector, $f(\vec{i}_o + \vec{T})$ equals $f(\vec{i}_o) + f(\vec{T})$ according to the linearity laws. Consequently, the data tile size is $f\vec{T}$, independent of the tile offset $\vec{i}_o$, thus is uniform.

[2]Since the offset vectors are constant, they do not affect the data space tiling vectors at all and are excluded from the proof.

```
CONNECTID($\mathcal{I}$, $\mathcal{D}$, $\mathcal{W}$, $S$, $S_r(\vec{T})$)

Input: Iteration space $\mathcal{I}$, indices $\vec{i} = (i_1, \cdots, i_n)$; Data space set $\mathcal{D}$;
  Data format weights $\mathcal{W} = \{w_1, \cdots, w_p\}$; on-chip memory size $S$.
Output: The on-chip data memory size budget for a given iteration space tiling configuration $S_r(\vec{T})$.

1. Build the xMDFG (refer to Section 4)
  Data space set $\mathcal{D} = \{\mathcal{D}_1, \cdots, \mathcal{D}_p\}$ with dimensions $d_1, \cdots, d_p$;
  For each $\mathcal{D}_j$, access matrix $F_j(\vec{i})$, offset matrix set $\{\vec{a}_{j1}, \cdots, \vec{a}_{jl}\}$
  while $\vec{a}_{jk} = (a^1_{jk}, \cdots, a^{d_j}_{jk})$;
2. Raise an orthogonal iteration space tiling $\vec{T} = (T_1, \cdots, T_n)$
3. Do data space tiling under $\vec{T}$
  $for\ j = 1, p$
    $for\ k = 1, d_j$
      $ra^k_j = max\{a^k_{j1}, \cdots, a^k_{jl}\} - min\{a^k_{j1}, \cdots, a^k_{jl}\}$
    $endfor$
    $\mathcal{D}^{\vec{T}}_j = (F_j\vec{T})' + (ra^1_j, \cdots, ra^{d_j}_j)$
  $endfor$
  Resulted data space tile of $\mathcal{D}_j$:
    $\mathcal{D}^{\vec{T}}_j = (dT^1_j, \cdots, dT^{d_j}_j)$
4. Calculate the memory size budget for one tile
  $S_r(\vec{T}) += \sum_{j=1}^{p}(w_j \times \prod_{k=1}^{d_j}(dT^k_j))$
```

Fig. 8. The algorithm CONNECTID extracts the relationship between data space tiling and iteration space tiling, and is used as memory size constraint.

Because each data tile is also orthogonal, it can be represented by $\vec{dT} = (dT_1, \ldots, dT_m)$, where $dT_j = \sum_{k=1}^{n}(F_{jk}T_k)$. There may be "hole"s (un-referenced data) in the data tile when the access matrix is in a *reduced form*.

5.2. Building the Objective Functions

Due to the tiling properties established between the iteration space and data space via the preprocessing, we can then propose an $\mathcal{I}$-$\mathcal{D}$ combined and stream-conscious tiling scheme for off-chip memory access optimization. The scheme is composed of two algorithms ordered sequentially: algorithm CONNECTID is used for the preprocessing as well as the derivation of the memory size constraint (Fig. 8), and algorithm BUILDOBJFUNC is used to build the objective functions (Fig. 9).

CONNECTID connects the iteration space and the data space in a specific manner (building the xMDFG), and formulates the relationship between the data tile sizes and the iteration space tiling configuration. The

```
BuildObjFunc($\mathcal{I}$, $\mathcal{D}$, $\mathcal{W}$, $S$, $R_{cc}(\vec{T})$, $Cost(\vec{T})$)

Input: Iteration space $\mathcal{I}$, indices $\vec{i} = (i_1, \cdots, i_n)$; Data space set $\mathcal{D}$;
  Data format weights $\mathcal{W} = \{w_1, \cdots, w_p\}$; on-chip memory size $S$.
Output: The objective functions $R_{cc}(\vec{T})$, $Cost(\vec{T})$.

1. Call ConnectID($\mathcal{I}$, $\mathcal{D}$, $\mathcal{W}$, $S$, $S_r(\vec{T})$)
2. Extract $keep(\vec{T})$ for inter-tile temporal reuse
  $keep(\vec{T}) = \sum_{j=1}^{p}(\mathcal{D}_j^{\vec{T}^k} \cap \mathcal{D}_j^{\vec{T}^{k+1}})$; $\forall\ neighboring\ full\ \vec{T}^k, \vec{T}^{k+1}$
  $transfer\_per\_tile(\vec{T}) = S_r(\vec{T}) - keep(\vec{T})$
3. Optimize memory layout in order to minimize $transfer\_per\_tile$
  $transaction\_per\_tile(\vec{T}) = \sum_{j=1}^{p}[(\prod_{k=1}^{d_j} dT_j^k)/max\{dT_j^k\}]$
4. Calculate intermediate communication budgets [3]
  $transfer\_per\_tile(\vec{T})$ += $intermediate\_comm\_transfer\_per\_tile(\vec{T})$
  $transaction\_per\_tile(\vec{T})$ += $intermediate\_comm\_transaction\_per\_tile(\vec{T})$
5. Build the objective functions of data reuse and communication cost
  $R_{cc}(\vec{T}) = (\prod_{j=1}^{n} T_j)/transfer\_per\_tile(\vec{T})$;
  $Cost(\vec{T}) \approx (transfer\_per\_tile(\vec{T}) \times C_t + transaction\_per\_tile(\vec{T}) \times C_s) \times \#tile(\vec{T})$;
```

Fig. 9. The algorithm BuildObjFunc builds the objective functions for data reuse and communication cost on tiling.

input of the algorithm includes the iteration/data space information of the studied loop, together with the data format weights that are used to capture and normalize data-width (e.g., if a data array is 16-bit wide and the memory is byte-wide, then $w = 2$). The algorithm is quite straightforward: it first builds the xMDFG as a preprocessing step, then it raises the iteration space tiling configuration (as a vector variable) and derives the corresponding data tiles. Finally, it extracts the relationship between the data space tiling and the iteration space tiling as $S_r(\vec{T})$, that is, how much data memory is required for a tiling configuration $\vec{T}$.

The output of ConnectID is fed to algorithm BuildObjFunc, which is used to build the objective functions of the data reuse and the communication cost under various tiling configuration $\vec{T}$. The extent of data reuse is formalized as the computation-communication ratio R_{cc}. The ratio denotes how many iterations can be executed per (normalized) data transfer. For a given loop, improving data reuse is equivalent to reducing the memory access count.

The first step of the algorithm BuildObjFunc is to derive the inter-tile temporal reuse (data that can be kept in the local memory for the next tile execution) thereby giving a closed form of data transfer amount per tile. In subsequence, the memory layout optimization is carried out in step 2. The intermediate communication cost, if exists, is calculated using the formulas in.[3] Finally, the objective functions for data reuse and communication cost are formulated in step 4. In addition, several important concerns about the two algorithms are discussed in detail in the following:

(1) *Data space*. Each data space $\mathcal{D}_j$ in the algorithm is assumed to be a single UGR, with one access matrix $F_j(\vec{i})$ and an offset matrix set. For instance, the reference set for the array d_1 in Fig. 6(c) can be represented by: access matrix $F_1(\vec{i}) = \left(\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right)$; offset matrix set $\vec{a}_{11} = (0\ 0)'$ and $\vec{a}_{12} = (0\ -1)'$, thus $l = 2$; for $\vec{a}_{12}$, $a^1_{12} = 0$ and $a^2_{12} = -1$. Note that A' is the transpose of matrix A.

(2) *Data tile shape computation*. The data tile of the jth array under tiling $\vec{T}$, $\mathcal{D}^{\vec{T}}_j = (dT^1_j, \ldots, dT^{d_j}_j)$, is computed by subscript function. It is composed of two parts, the bulk data tile using access matrix, and the part for boundary data with size $(ra^1_j, \ldots, ra^{d_j}_j)$. The full tile computation expression is given in step 3 of ConnectID. By changing the tile sizes accordingly, the expression is also applicable to partial tiles that may present at the boundaries.

(3) *Inter-tile temporal locality*. A data array or part of a data array can be kept in the on-chip memory for more than one tile execution if it has inter-tile temporal locality. Since the data tile shape is known and any tile offset can also be calculated using the access matrix, the amount of inter-tile reuse data, $keep(\vec{T})$, can be approximated by any overlapping of two neighboring full tiles, shown in step 2 of BuildObjFunc.

(4) *Memory layout optimization*. The memory layout affects communication cost. In order to reduce the communication cost for a data array tile $\mathcal{D}^{\vec{T}}_j = (dT^1_j, \ldots, dT^{d_j}_j)'$, one solution is to store the largest side $max\{dT^1_j, \ldots, dT^{d_a}_j\}$ consecutively in memory, thus reducing the transactions.

The formulated results in the experiments are based on these algorithms with refinement when considering partial tiles.

5.3. Tiling Exploration

The algorithms of ConnectID and BuildObjFunc provide us with three formulas, the memory size constraint $S_r(\vec{T}) \leq S/2$ (*assume prefetching*), and the objective functions $R_{cc}(\vec{T})$ and $Cost(\vec{T})$. Since the tiling sizes are integer numbers only, brute-force can be a feasible way of searching for the optimal solution. Assuming that the loop level is k and each loop index has the same range of $(1, N)$, the problem complexity using brute-force is $O(N^{k-1})$. Although N may be large in image and video applications (e.g., image or video frame size), fortunately, most nested loops have a nesting level less than 5, which means a very small number of k, thereby indicating an acceptable exploration space. In fact, due to the limited number of loop levels, the running time of exhaustive tiling exploration for any of the benchmarks that we ran (details refer to Section 7) is less than half an hour. In short, tiling exploration can be tackled in a brute-force manner practically.

For particular subset problems, brute-force searching can even be replaced by an analytical method. Indeed, our target application domain, multimedia applications, is DSP intensive with very structured computations and loop structures. Although loop tiling is a hard discrete non-linear optimization problem in general,(1) the properties of our target applications bring us the opportunity to analytically find the the optimal (or sometimes approximately optimal) solutions to the objective functions. We will demonstrate it in the following case studies.

6. CASE STUDIES

Despite the fact that the algorithms of motion estimation and matrix multiply have been frequently used before, we use them as illustrative examples for three reasons: (1) they are simple and easy to understand; (2) they are common and there are similar algorithms throughout the signal and image processing applications; (3) they have been well studied but we still show what has been neglected and how it is improved by us. In addition, more applications will be demonstrated in the experimental section.

6.1. Matrix Multiply

Using the code in Fig. 2(a), the iteration space of matrix multiply is 3-D as $\vec{i} = (i, j, k)$ while the three data arrays are all 2-D, therefore the nested loop has a mismatched iteration and data space. However, the access matrices for the arrays, $F_A(\vec{i}) = \begin{pmatrix} 1\,0\,0 \\ 0\,1\,0 \end{pmatrix}$, $F_B(\vec{i}) = \begin{pmatrix} 1\,0\,0 \\ 0\,0\,1 \end{pmatrix}$, and $F_C(\vec{i}) = \begin{pmatrix} 0\,1\,0 \\ 0\,0\,1 \end{pmatrix}$, are all already in reduced form thus no transformation is needed for the pre-processing. Applying the remaining steps in the ConnectID algorithm (Fig. 8), the required memory size is acquired as $S_r(\vec{T}) = T_i T_j + T_i T_k + T_j T_k$ for tiling configuration $\vec{T} = (T_i, T_j, T_k)$.

Subsequently, algorithm BuildObjFunc (Fig. 9) is applied. For any neighboring full tiles (note that the loop execution order by index is k, j, i), the data tile of array A with size of T_i by T_j can be reused. Therefore, $transfer\_per\_tile(\vec{T})$, which equals to $S_r(\vec{T}) - keep(\vec{T})$ is derived as $T_i T_k + T_j T_k$. The refinement of including the boundary effect of loading and storing array A is made in the communication cost calculation.

According to the legality criteria,[20] any orthogonal tiling on the code is legal. The objective functions are finally built as below:

$$\begin{cases} \text{Size Constraint}: & S_r(\vec{T}) = T_iT_j + T_iT_k + T_jT_k \le S/2 \\ \text{Data reuse}: & R_{cc} = \frac{T_iT_jT_k}{T_iT_k+T_jT_k} = \frac{T_iT_j}{T_i+T_j} \\ \text{Communication Cost}: & \text{Cost} = \frac{N^2}{T_i}(1+\frac{2N}{T_j})C_s + N^2(1+\frac{N}{T_j}+\frac{N}{T_i})C_t \end{cases} \tag{4}$$

The data reuse R_{cc} function is given with the assumption that the matrix size is infinite thus the boundary effects can be ignored. Our tiling exploration is pursued by adjusting the lengths of any two tile sides since the third side is then fixed by the size constraint. As a consequence, the scheme offers an $O(N^2)$ (N is the matrix size, where $N_i = N_j = N_k = N$) exploration space, in contrast to the approach in,[12] which discussed only three particular situations.

In fact, regarding the relationship $S_r(\vec{T}) = T_i \cdot T_j + T_i \cdot T_k + T_j \cdot T_k \le S/2$, the optimal tiling for data reuse $\vec{T}_{Opt\_Rcc}$ can be obtained analytically and directly from the equation: T_k is 1, while T_i and T_j are equal and are as large as possible — $(T_i, T_j, T_k) \approx (\lfloor\sqrt{1+S/2}\rfloor - 1, \lfloor\sqrt{1+S/2}\rfloor - 1, 1)$. *We use this optimal configuration directly in our experiments without searching the $O(N^2)$ space in the brute-force manner anymore.* We note that matrix multiply represents a subset of many practical problems, e.g., LU decomposition, Cholesky factorization, etc., whose objective functions are almost the same. Therefore, this analytical optimization solution is effective for a group of applications, instead of the matrix multiply alone.

The objective function of communication cost is more complex, whereas it still provides several useful clues: First, like the analysis for $\vec{T}_{Opt\_Rcc}$, larger T_i and T_j values result in a smaller communication cost and T_k should equal 1. Second, the communication costs in the start-up and the stable status add reverse requirements on the tile shape — "strip-mining" versus "square". Therefore, the optimal tiling for communication cost lies between "strip-mining" and "square", affected by the relative values of C_s, C_t.

The tiling exploration potential is represented by the dynamic range of the communication cost, $\Re = max\{Cost(\vec{T})\}/min\{Cost(\vec{T})\}$. Figure 10 displays how the matrix size N and on-chip memory size S affect $\Re$. Most common configurations exhibit a dynamic range of 10–40X with the larger S, the larger $\Re$. It is interesting here to note the analogy between the behavior of R and the MOS I-V characteristics, the dynamic range of tiling shows "Tri-state": *cutoff*, *linear* and *saturation*.

The *cutoff* state exists in the region where N is very small compared with S. Since almost the entire data arrays can be stored in an on-chip

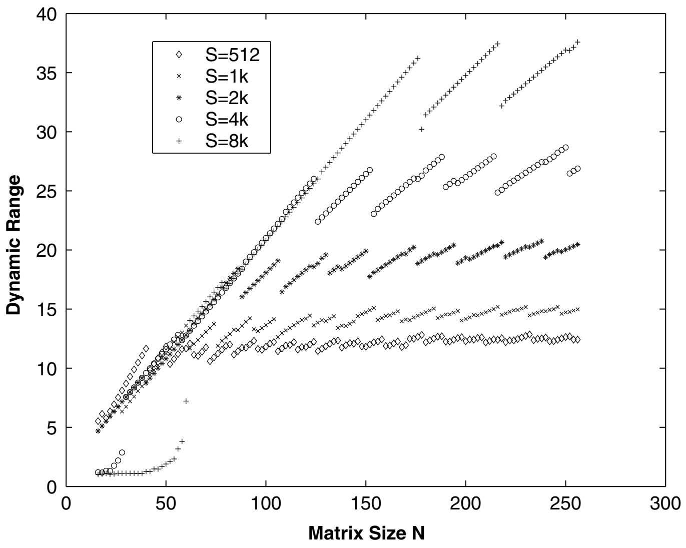


Fig. 10. The exploration potential of tiling.

memory, tiling gives little profit. Illustrated in Fig. 10, S = 8K and S = 4K present such *cutoff* state when N is less than about 55 (30 for S = 4K). As N increases, the potential benefit of tiling also expands since on-chip memory can not contain all the data at any specific time and thus tiling becomes mandatory. The dynamic range increases quite linearly in this region, therefore denoted as the *linear* state. However, the *linear* state ceases to hold when N exceeds a certain value. The larger S, the larger value the dynamic range saturates in. It substantiates that a larger storage size provides better data reuse opportunities. The *saturation* region exhibits heavy discrete effects.

6.2. Automatic Target Recognition — ATR

This application performs the matching between a given template and the windows of a gray scale image. The code of the computation kernel is shown in Fig. 6(d), consisting of image correlations between the template matrix and shifted windows over the input image. The iteration space $\vec{i} = (m, n, i, j)$ is 4-D while the data spaces *image* and *result* are both two dimensional. The subscript functions are:

$$f_{\text{image}}(\vec{i}) = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} m & n & i & j \end{pmatrix}' + \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{5}$$

$$f_{\text{result}}(\vec{i}) = \begin{pmatrix} 1\,0\,0\,0 \\ 0\,1\,0\,0 \end{pmatrix} \begin{pmatrix} m\ n\ i\ j \end{pmatrix}' + \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{6}$$

When the iteration space is tiled as $\vec{T} = (T_m, T_n, T_i, T_j)$, the corresponding data tiles are $\mathcal{D}^{\vec{T}}_{\text{image}} = (T_m + T_i, T_n + T_j)$ and $\mathcal{D}^{\vec{T}}_{\text{result}} = (T_m, T_n)$. Therefore, to fulfil the execution of one tile, $(T_m+T_i)\times(T_n+T_j)$ size of the image needs to be loaded from the main memory and $T_m \times T_n$ of the correlated results should be stored back to the main memory. After the inter-tile temporal locality optimization, the number of memory access for one tile is reduced to $T_n \times (T_m - 1 + T_i) + T_m \times T_n$. The size constraint and objective functions are built as:

$$\begin{cases} \text{Size Constraint :} & S_r(\vec{T}) = (T_m + T_i)(T_n + T_j) + T_m T_n \le S/2 \\ \text{Data reuse :} & R_{cc} = \frac{T_m T_n T_i T_j}{T_n(T_m+T_i-1)+T_m T_n} = \frac{T_m T_i T_j}{2T_m+T_i-1} \\ \text{Communication Cost :} & Cost = \frac{2N_m N_n N_i N_j}{T_m T_i T_j} C_s + \frac{N_m N_n N_i N_j}{T_m T_i T_j}(2T_m+T_i-1)C_t \end{cases} \tag{7}$$

Likewise, the (approximately) optimal solution for the objective function of data reuse in Eq. 7 can be solved analytically. According to the formula, it is obvious that a larger T_j results in better data reuse. Since N_i and N_j are usually small (8 or 16 in most cases), T_j is equal to N_j. T_n is excluded from the formula, counting in the size constraint, $T_n = 1$ (for possibly larger T_m and T_i thus larger R_{cc}). Since N_i is also small, the optimal tile configuration for best data reuse is $(Tm, 1, N_i, N_j)$ in most cases.

Table I compares our stream-conscious tiling method with two other tiling approaches under the common situation that image size is 512 by 512 and mask is 8 by 8. The scheme *square* keeps the tile sides as equally as possible, and the *kernel* preserves the two innermost loops completely in one tile. We test the memory access counts under various on-chip memory size (S constraint). Compared with *square* and *kernel*, our proposal reduces memory access cost ranging from 22.6% to 70.0%. Unlike the traditional *kernel* approach, which takes the mask as a granule, our method justifies that the partial calculation of the "mask kernel" introduces better data reuse when S is small.

6.3. Motion Estimation

Motion estimation is a fundamental technique to obtain data compression in video coding by exploiting inter-frame prediction of temporal redundancies in video sequences.(25) Among the existing block matching algorithms, the full-search block-matching (FSBM) algorithm exhaustively

Table I. Different Tiling Schemes on ATR

	Shape $\vec{T}$			#Access (*millions*)			Ratio	
S	*our*	*square*	*kernel*	*our*	*square*	*kernel*	*square*	*kernel*
64	[1, 2, 2, 8]	[2,2,2,2]	–	3.15	10.49	–	70.0%	–
96	[3, 1, 4, 8]	[3,3,3,3]	–	1.80	4.97	–	63.8%	–
128	[4, 1, 6, 8]	[4,4,4,4]	[1,1,8,8]	1.14	2.88	2.36	60.6%	51.9%
196	[5, 1, 8, 8]	[5,5,5,5]	[2,2,8,8]	0.891	1.879	1.442	52.6%	38.2%
256	[12, 1, 8, 8]	[6,6,6,6]	[5,5,8,8]	0.677	1.320	0.891	48.7%	24.0%
384	[18, 1, 8, 8]	[7,7,7,7]	[6,6,8,8]	0.626	0.978	0.830	36.0%	24.6%
512	[31, 1, 8, 8]	[8,8,8,8]	[8,8,8,8]	0.583	0.754	0.754	22.6%	22.6%
1024	[44, 1, 8, 8]	[11,11,8,8]	–	0.572	–	0.763	25.0%	–
2048	[95, 1, 8, 8]	[18,18,8,8]	–	0.544	–	0.772	29.6%	–

compares each $N \times N$ block of the current frame with all candidate blocks of the $(N + 2p)^2$ search window defined within the previously processed frame.

The FSBM algorithm using the sum of absolute differences (SAD) distortion measure can be described by the four nested loops presented in Fig. 11. The nested loop does not conform to the program model we adopt (shown in Fig. 3). Instead, it is not perfectly nested as statements ($S1$, $S3$–$S5$) located beyond the innermost loop. In addition, conditional statement also exists. Nevertheless, the profiling of the code shows that the innermost statement $S2$ consumes most of the execution time, and the data arrays are all present as operands in $S2$. Therefore, the code can easily be pruned to be a regular and perfect loop.

The scalars, x, y and the $SAD(x, y)$, only occupy 3 words, thus their effect on the data locality can be ignored (usually they are allocated with the local memory). The memory access exploration is largely determined by the three data arrays, SAD, R and S. Likewise, we can use the proposed algorithms to obtain the objective functions, shown in Eq. (8). The formulas are very close to those of ATR in Eq. (7), and the analytical solution can be obtained in a similar way.

$$\begin{cases} \text{Size Constraint :} & S_r(\vec{T}) = (T_i + T_u)(T_j + T_v) + T_iT_j + T_uT_v \leq S/2 \\ \text{Data reuse :} & R_{cc} = \frac{T_iT_jT_u}{2T_u+T_i} \\ \text{Communication Cost :} & \text{Cost} = \frac{2N_iN_jN_uN_v}{T_iT_jT_u}C_s + \frac{N_iN_jN_uN_v}{T_iT_jT_u}(2T_u + T_i)C_t \end{cases} \tag{8}$$

```
(x, y) = (0, 0)                        // motion vector initialization
SAD(x, y) = +inf.
for i = -p to p do                     // (2p + 1) by (2p + 1) search area
  for j = -p to p do
S1:  SAD(i, j) = 0;                    // SAD measure initialization
     for u = 0 to (N - 1) do           // N by N reference macroblock
       for v = 0 to (N - 1) do
S2:        SAD(i, j) += abs(R(u, v) - S(i + u, j + v));
       end for
     end for
S3:  if SAD(i, j) < SAD(x, y) then
S4:      (x, y) = (i, j);
S5:      SAD(x, y) = SAD (i, j);
     end if
  end for
end for
return (x, y)                          // motion vector
```

Fig. 11. The pseudo code of FSBM algorithm using SAD distortion measure.

7. EXPERIMENTS

7.1. Experimental Setup and Tiling Overhead

The effectiveness of our combined tiling method is verified through the experiments on a software-managed memory system, TI's C55X DSP,(16) with the help of a cycle-accurate (including peripherals) simulator CCStudio v2.2.(26) C55X's DMA-EMIF (Extended Memory InterFace) engine is highly reconfigurable and the data memory can be flexibly mapped as a number of banks in any sizes, thereby we can test our proposal in a large configuration range.

TI provides an optimized DSP function library for C programmers on the TMS320C55X devices, DSPLIB. Figure 12 gives the reference assembly code of the matrix multiply routine in the DSPLIB and the corresponding function calling to it. The assembly code size is 65 16-bit words. Tiling doubles the nest depth from 3 to 6. Therefore, the code size after tiling is roughly doubled too. Nevertheless, the code size increase caused by tiling is small (less than 100 words) in comparison with the on-chip program memory of C55X device, which is usually in tens of kilo-words. In addition, the cost in transferring program code is negligible again due to the regular computation that compacts code size.

For data communication between the off-chip and on-chip memories, C55X uses its EMIF port and DMA engine. There is initialization overhead because the EMIF and DMA engines contain about 40 registers to be assigned with proper values in totality. However, after the initialization at the very beginning, only several registers need to be re-assigned with

```
#include <math.h>
#include <tms320.h>
#include <dsplib.h>

#include "test.h"

short i;
short eflag= PASS;

void main()
{
  // clear output buffer (optional)
  for (i=0;i<row1*col2;i++) r[i] =0;

  // compute
  mmul(x1, row1, col1, x2, row2, col2, r);

  // test
  eflag = test(r, rtest, row1*col2, MAXERROR);

  if(eflag != PASS)
  {
    exit(-1);
  }

  return;
}
```

(a)

```
.mmregs
        .if __far_mode
OFFSET          .set 1
        .else
OFFSET          .set 0
        .endif
         .text
         .def _mmul
_mmul:
        PSHM   AR1
        PSHM   AR6
        PSHM   AR7
        PSHM    ST0                     ; 1 cycle
        PSHM    ST1                     ; 1 cycle

; Matrix multiplication using 3 nested loops
mul1
        LD      *SP(11+OFFSET), A
        STLM    A, AR6                          ; AR6 -> SP(5+OFFSET)
mul2    MVMM AR2, AR4
        MVMM AR3, AR5
        LD      *SP(8+OFFSET), A
        STLM    A, AR7                          ; AR7 -> SP(2+OFFSET)
        LD      #0, A
mul3    MAC     *AR4+,*AR5,A,A
        MAR     *AR5+0
        BANZ    mul3,*AR7-
mul3end STH    A,*AR1+                  ; store output element
        MAR     *AR3+
        BANZ    mul2, *AR6-
mul2end
        LDM     AR2, A
        ADD     B, A
        STLM    A, AR2
        MVDK  *SP(9+OFFSET), AR3                ; AR3 -> input 2
mul1end NOP
```

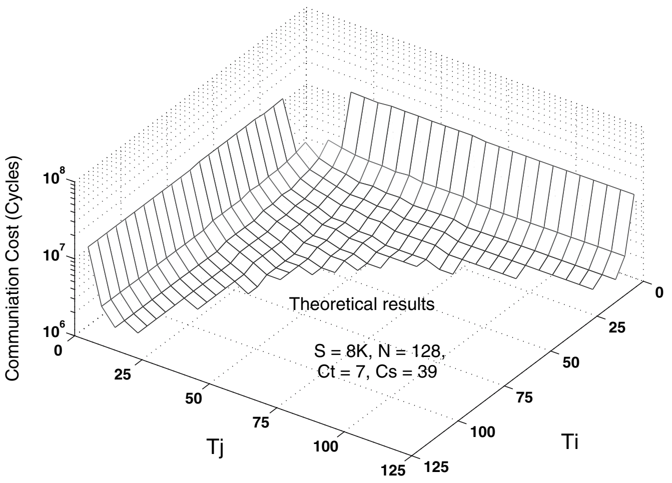
(b)

Fig. 12. (a) The C code to call the matrix multiply routine; (b) The DSPLIB assembly matrix multiply code for TMS320C55X implementation. © Texas instruments Inc, 1998.

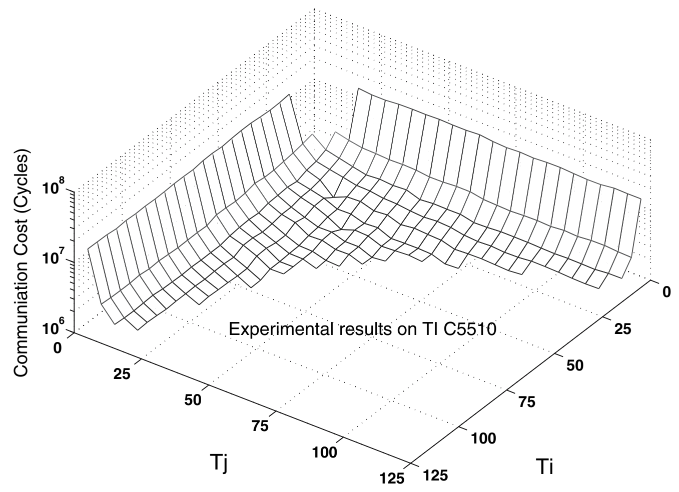
new values to trigger the next data transaction — a very low cost. Furthermore, data transfer control is always needed in code generation (either by tools or by designers) for software-managed memory systems in spite of tiling. Therefore, it should not be viewed as an extra burden brought by tiling. The experiments conducted by us show that the compound transaction start-up penalty C_s, which includes access latency and software initialization overhead, is about 40 clock cycles.

7.2. Experimental Results

Figure 13(a) and (b) compare the theoretical and the experimental tiling exploration results on matrix multiply in a typical configuration. They show large exploration potential (over 15 times) on tiling. The theoretical mesh is based on Eq. (4). The values of C_s, C_t are extracted from C55X's DMA-EMIF engine. The theoretical formula accurately estimates the experimental results with the average error of less than 6%. It validates the effectiveness of our stream-conscious tiling scheme in modeling the communication metrics.



(a)



(b)

Fig. 13. Tiling exploration on matrix multiply ($N = 128$, $S = 8\,\mathrm{KB}$) (a) the theoretical vs. (b) the experimental results on communication costs; the theoretical formula accurately estimates the experimental results with the average error less than 6% (closely matched profiles).

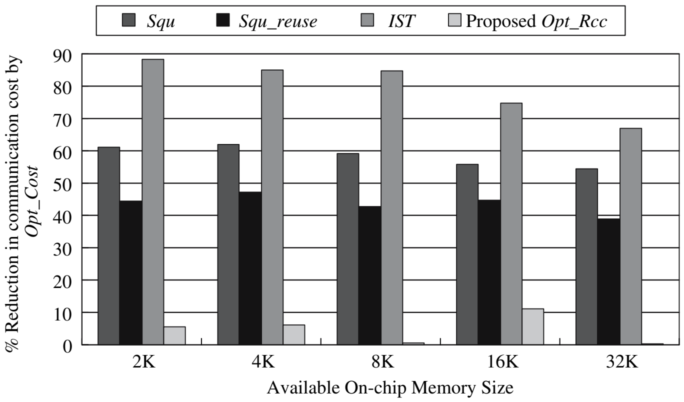


Fig. 14. Percentage reduction in communication cost over conventional tiling schemes (the first three columns in one group) when our approach *Opt_Cost* is used.

Inside the exploration space, we further compare our communication optimization schemes with previous approaches. *Squ*(12) partitions all three arrays to equal squares without inter-tile reuse. *Squ_reuse* further takes advantage of inter-tile temporal locality. *IST* (Iteration Space Tiling)(3) represents the traditional iteration space tiling approach. In order to reduce the number of dependencies across tile boundaries, *IST* partitions the iteration space along the direction of the dependence vector $(0, 0, 1)$, resulting in tiling shape as (a, a, N) where the value of a is constrained by the SPM size.[3] *Opt_Rcc* is our stream-conscious tiling scheme for data reuse optimization and *Opt_Cost*, our stream-conscious tiling for communication minimization.

The tiling configurations for *Squ*/*Squ_reuse*, *IST*, *Opt_Rcc* and *Opt_Cost* are (26 26 26), (128 7 7), (44 44 1) and (43 43 2), respectively in Fig. 13(b). Compared with *Squ*, *Squ_reuse* and *IST*, the proposed *Opt_Rcc* reduces the total memory access by 52.8%, 35.9% and 65.6%.

Figure 14 demonstrates the communication costs of different schemes when the available on-chip memory size S ranges from 2KB to 32KB. Although some commercial DSPs have larger on-chip data RAM than what we used in the experiments, the data RAM is usually divided and reserved for different purposes in the whole program. Therefore, tiling with limited available memory is still attractive and practical.

Even though all those schemes show good data reuse, our communication-minimal solution *Opt_Cost* still brings the total cycle reductions

[3]When applying the data space-oriented tiling from,(13) similar tile shape as *IST* is expected since the loop nest is singleton.

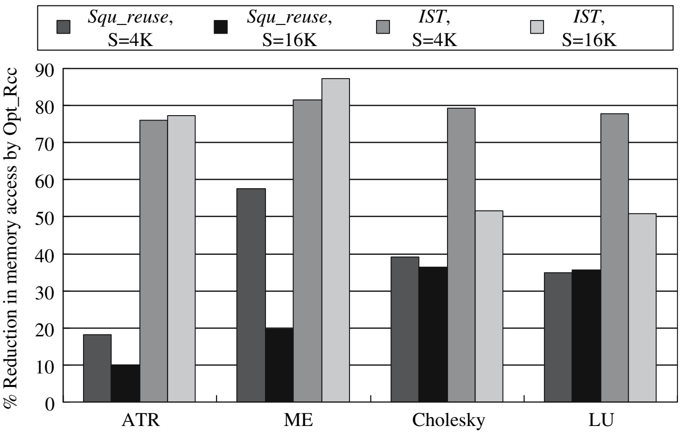


Fig. 15. Percentage reduction in off-chip memory access number over conventional tiling schemes when our approach *Opt_Rcc* is used.

to over 38.8% compared with the best previous schemes. The proposed *Opt_Rcc* scheme for data reuse, which is easier to obtain analytically, shows the communication costs close to the optimal ones (the worst case is 11.0% in Fig. 13(c)). Therefore, it can be used to approximate the expensive *Opt_Cost* scheme, which obtains the optimal solutions via brute-force or ILP ways.

Besides matrix multiply, several array intensive applications are evaluated. They are: *ATR*, *ME*, *Cholesky* and *LU*. In *ATR*, we assume a common situation that the image size is 512 by 512 and the mask is 8 by 8. *ME* is a full search motion estimation algorithm with a 16 by 16 searching window in a 512 by 512 image. *Cholesky*,(27) and *LU* are for Cholesky factorization and LU decomposition for image restoration with the size of 128 and 192, respectively. Note that one of the two arrays in *Cholesky* has 3 UGRs, thus there are always 8 data tiles in the SPM concurrently (4 for prefetching).

Applying the proposed stream-conscious tiling algorithm, we can obtain the objective function for optimal data reuse and derive the corresponding tile shape *Opt_Rcc* analytically (refer to the previous case study section). The benchmarks and experimental results are summarized in Table II. Figure 15 displays the reductions in off-chip memory access number over conventional *Squ_reuse*, and *IST* tiling schemes when our approach *Opt_Rcc* is used. For the four benchmarks, our approach reduces the memory access by about 37.4% when $S = 4$KB, and 25.4% when $S = 16$KB on average compared with the best of the previous schemes.

Table II. Summary of the Tested Applications

					Off–chip memory access count (in 10^4)					
					SPM size $S = 4K$			SPM size $S = 16K$		
Bench mark	Loop depth*	No. of data arrays	Arrays dimension	Array size(s)**	*Squ_reuse*	*IST*	*Opt_Rcc*	*Squ_reuse*	*IST*	*Opt_Rcc*
ATR	4	2	2-D	512; 8	69.1	235.9	56.6	59.2	235.9	53.4
ME	4	2	2-D	512; 16	194.6	445.6	82.7	70.3	445.6	56.4
Cholesky	3	4***	2-D, 1-D	128	8.29	24.39	5.06	4.34	5.72	2.77
LU	3	5***	2-D, 1-D	192	18.63	54.70	12.13	9.81	12.85	6.30

* The number of levels of the nested loop.
** All 2-D arrays are square here, therefore only one side size is given. ATR and ME have two different sizes of arrays.
*** One of the two arrays has 3 UGRs where each UGR should be treated as an independent single reference.

It has been shown that 50–75% of the power consumption in embedded multimedia systems is the consequence of memory accesses.(28,29) Since the largest portion of memory access activities come from data array communications, we extend the stream-conscious tiling algorithm (Fig. 9) for energy reduction with moderate modifications. Our approach achieves this goal by reducing the off-chip memory access activities as well as shortening the overall run time. We note that the tile shapes for optimal memory access count and access time may not be the same, thus the objective function (minimizing Eq. 2) for energy reduction provides a tradeoff between those two metrics.

The power model for C5510 is activity-based and provided by TI, where each active component can be isolated and accurately modeled to determine its contribution to the overall power consumption.(18) The EMIF utilization is related to the maximum bandwidth and the one hundred percent utilization corresponds to the maximum transfer rate for a given frequency. This number will be scaled down by both slower and less frequent transfers. The same rule is applicable to the rest of memory subsystems.

Table III lists the activation baseline power of each individual module. At the frequency of 200 MHz with the temperature being 25°C, the total activation baseline power is 187.94 mW. Note that there is no memory access and not a single instruction is executed by the CPU. Such high cost on P_{base} underlines the importance of shortening the run time.

Table III. Activation Baseline Power of C5510 DSP

Module	DPLL	CPU	SPM+ EMIF	DMA	I-Cache	Timer	Other Circuitry	Total
Core Current (mA)	46.13	2.06	11.27	5.77	6.01	2.53	4.50	78.26
I/O Current (mA)	0	0	12.89	0	0	0	6.12	19.01
Power (mW)	73.81	3.30	60.55	9.23	9.61	4.05	27.39	187.94

(Frequency = 200 MHz; Temperature = 25°C; Core voltage = 1.6 V; IO voltage = 3.3 V)

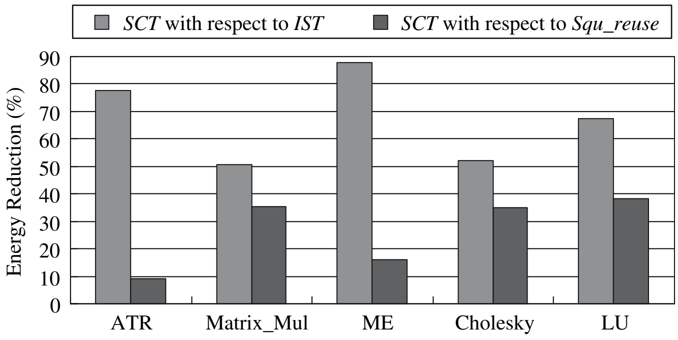


Fig. 16. The energy reductions by the proposed *SCT* scheme.

By running tests on memory access, the per-access-energy to the off-chip memory is obtained as 7.101 nJ.

We compare our tiling exploration scheme, Stream-Conscious Tiling, (*SCT*) with previous approaches Iteration Space Tiling (*IST*)(3) and *Squ_reuse*.(12) The access count, run time, together with the corresponding access energy and activation baseline energy are summarized in Table IV. Our tiling exploration brings a considerable reductions in memory access count. Although there is no rigidly linear relationship between the memory access count and time, reducing the access count usually shortens the total access time. On average, the reductions in access energy by *SCT* are 67.8% with respect to *IST* and 27.7% with respect to *Squ_reuse*. Similar reductions are achieved in the measure of run time, 66.6% compared with *IST* and 25.7% compared with *Squ_reuse* on average. Figure 16 displays the total energy reductions when our approach *SCT* is used, ranging from 50.8 to 88.0% with respect to *IST*, and from 9.2% to 38.2% with respect to *Squ_reuse*.

Table IV. Summary of the Energy Consumptions

Benchmark		ATR	Matrix_ Mul	ME	Cholesky	LU	Average
	IST	16753.7	1396.1	31645.5	405.2	1846.3	–
Access energy (uJ)	*Squ_ reuse*	4040.6	1047.4	4498.6	309.3	1004.1	–
	SCT	3755.8	698.0	3854.5	198.9	619.2	–
Access energy reduction	*IST*	77.6%	50.0%	87.8%	50.9%	66.5%	67.8%
by SCT with respect to (%)	*Squ_ reuse*	7.01%	33.4%	14.3%	35.7%	38.3%	27.7%
	IST	16947	1566	32787	504.2	2147	–
run time (K cycles)	*Squ_ reuse*	4233	1212	4719	359.8	1106	–
	SCT	3753	758	3896	236.0	685.2	–
	IST	15925.3	1471.6	30810.3	473.8	2017.6	–
Activation Baseline Energy (uJ)	*Squ_ reuse*	3977.5	1138.8	4434.1	338.1	1039.3	–
	SCT	3526.5	712.3	3661.4	221.8	643.9	–
Activation Baseline Energy	*IST*	77.9%	51.6%	88.1%	53.2%	68.1%	66.6%
run time (K cycles)	*Squ_ reuse*	11.3%	37.5%	17.4%	34.4%	38.0%	25.7%
respect to (%)							

8. SUMMARY AND FUTURE WORK

In earlier researches on iteration space tiling, data tiling was simply the outcome of code generation. In contrast, the proposed technique combines both iteration space tiling and data space tiling into the exploration stage, therefore the objective functions for memory access optimization can be derived explicitly and accurately. The experiments show significant improvement in hiding memory latency, reducing memory access, and hence reducing energy consumptions.

For loops with iteration space and data space well matched, the exploration potential is usually limited. Instead, we would like to extend our approach to non-uniform loops (e.g., FFT). We note that the uniform dependence is still a common restriction in this area. Furthermore, the extension to multiple loop nests is interesting and challenging.

REFERENCES

1. R. Andonov, H. Bourzoufi, and S. Rajopadhye, Two-dimensional Orthogonal Tiling: From Theory to Practice. *in Proceedings of HPC '96*, pp. 225–231 (1996).
2. L. Carter, J. Ferrante, and S. F. Hummel, Hierarchical Tiling for Improved Superscalar Performance. *in Proceedings of IPPS '95*, pp. 239–245 (1995).
3. Fei Chen and E. Sha. Loop Scheduling and Partitions for Hiding Memory Latencies. *in Proceedings of ISSS '99*, pp. 64–70 (1999).
4. R. M. Karp, R.E. Miller, and S. Winograd, The Organization of Computations for Uniform recurrence equations. *J. ACM*, **14**(3):563–590 (July 1967).
5. U. Banerjee, *Loop Transformations for Restructuring Compilers*. Kluwer Academic Publishers (1993).
6. J. Ramanujam and P. Sadayappan, Tiling Multidimensional Iteration Spaces for Nonshared Memory Machines. *in Proceedings Supercomputing '91*, pp. 111–120 (1991).
7. Q. Wang, E. Sha, and N. L Passos Optimal Data Scheduling for Uniform Multi-Dimensional Applications, *IEEE Trans. Computers*, **45**(12):1439–1444 (Dec. 1996).
8. M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison Wesley Publishing Company (1996).
9. P. -Y. Calland, J. Dongarra, and Y. Robert, Tiling with Limited Resources. *in Proceedings ASSAP '97*, pp. 229–238 (1997).
10. P. R. Panda, N. D. Dutt, and A. Nicolau, Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications, *in Proceedings of EDTC '97*, pp. 7–11 (1997).
11. P. Marwedel, L. Wehmeyer, M. Verma, Stefan Steinke, and Urs Helmig, Fast, Predictable and Low Energy Memory References Through Architecture-Aware Compilation. *in Proceedings of ASP-DAC'04*, pp. 4–11 (2004).
12. M. Kandemir, J. Ramanujam, M. Irwin, V. Narayanan, I. Kadayif, and A. Parikh, A Compiler-Based Approach for Dynamically Managing Scratch-Pad Memories in Embedded Systems. *IEEE Trans. CAD*, **23**(2):243–260 (Feb. 2004).
13. I. Kadayif and M. Kandemir, Data Space-Oriented Tiling for Enhancing Locality. *ACM Trans on Embedded Comput Sys*, **4**(2):388–414 (May 2005).
14. A. Darte and G. Huard. Complexity of Multi-Dimensional Loop Alignment. *in Proceedings of STACS'02*, pp. 179–191 (2002).
15. J. J. Navarro, E. G. Diego, and J. R. Herrero. Data Prefetching and Multilevel Blocking for Linear Algebra Operations. *in Proceedings of Supercomputing '96*, pp. 109–116 (1996).
16. *TMS320C55x DSP Functional Overview*, Texas Instruments Inc., http://focus.ti.com/lit/ug/spru307a/spru307a.pdf.
17. *ADSP-21xx Processor*, Analog Devices Inc., http://www.analog.com/processors/processors/ADSP/.
18. Texas Instruments, Inc. TMS320VC5510 Power Consumption Summary (SPRA972) (2003).
19. J. -K. Peir and R. Cytron, Minimum Distance: A Method for Partitioning Recurrences for Multiprocessors. *IEEE Trans. on Comp.*, **38**(8):1203–1211 (1989).
20. J. Xue, *Loop Tiling for Parallelism*. Kluwer Academic Publishers (2000).
21. P. C. Shields. *Elementary Linear Algebra*. Worth Publishers, Inc. (1980).
22. A. Darte, G. -A. Silber, and F. Vivien, Combining Retiming and Scheduling Techniques for Loop Parallelization and Loop Tiling. *Parallel Process. Lett.*, **7**(4):379–392 (1997).
23. M. W. Hall, S. Hiranandani, K. Kennedy, and C. W. Tseng. Inter-Procedural Compilation of Fortran D for MIMD Distributed-Memory Machines. in *Proceedings of Supercomputing '92*, pp. 522–534 (1992).

24. D. J. Palermo, E. Su, J. A. Chandy, and P. Banerjee, Communication Optimizations Used in the PARADIGM Compiler for Distributed Memory Multicomputers. *in Proceedings of Supercomputing '94*, pp. 1–10 (1994).
25. V. Bhaskaran and K. Konstantinides, *Image and Video Compression Standards: Algorithms and Architectures, 2nd edn.*, Kluwer Academic (1997).
26. *Code Composer Studio Product*, Texas Instruments Inc., http://www.go-dsp.com/mm-help/swfs/profiler.htm.
27. W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press (1992).
28. M. R. Stan and W. P. Burleson, Bus-invert coding for low-power i/o. *IEEE Trans. VLSI*, **3**(1):49–58 (Mar. 1995).
29. S. Wuytack, F. Catthoor, L. Nachtergaele, and H. De Man, Power Exploration for Data Dominated Video Applications. *in Proceedings of ISLPED'96*, pp. 359–364 (1996).