# Data Space Oriented Scheduling in Embedded Systems

M. Kandemir, G. Chen, W. Zhang
CSE Department
Penn State University
University Park, PA, 16802

I. Kolcu
Computation Department
UMIST
Manchester M60 1QD, UK

## Abstract

*With the widespread use of embedded devices such as PDAs, printers, game machines, cellular telephones, achieving high performance demands an optimized operating system (OS) that can take full advantage of the underlying hardware components. This paper presents a locality conscious process scheduling strategy for embedded environments. The objective of our scheduling strategy is to maximize reuse in the data cache. It achieves this by restructuring the process codes based on data sharing patterns between processes.*

## 1. Introduction

As embedded designs become more complex, so does the process of embedded software development. In particular, with the added sophistication of Operating System (OS) based designs, developers require strong system support. A variety of sophisticated techniques maybe required to analyze and optimize embedded applications. One of the most striking differences between traditional process schedulers used in general purpose operating systems and those used in embedded operating systems is that it is possible to customize the scheduler in the latter [9]. In other words, by taking into account the specific characteristics of the application(s), we can have a scheduler tailored to the needs of the workload.

While previous research used compiler support (e.g., loop and data transformations for array-intensive applications) and OS optimizations (e.g., different process scheduling strategies) in an isolated manner, in an embedded system, we can achieve better results by considering the interaction between the OS and the compiler. For example, the compiler can analyze the application code statically (i.e., at compile time) and pass some useful information to the OS scheduler so that it can achieve a better performance at runtime. This paper is a step in this direction. We use compiler to analyze the process codes and determine the portions of each process that will be executed in each time quanta in a pre-emptive scheduler. This can help the scheduler in circumstances where the compiler can derive information from that code that would not be easily obtained during execution. While one might think of different types of information that can be passed from the compiler to the OS, in this work, we focus on information that helps improve data cache performance. Specifically, we use the compiler to analyze and restructure the process codes so that the data reuse in the cache is maximized.

Such a strategy is expected to be viable in the embedded environments where process codes are extracted from the same application and have significant data reuse between them. In many cases, it is more preferable to code an embedded application as a set of co-operating processes (instead of a large monolithic code). This is because in general such a coding style leads to better modularity and maintainability. Such light-weight processes are often called threads. In array-intensive embedded applications (e.g., such as those found in embedded image and video processing domains), we can expect a large degree of data sharing between processes extracted from the same application.

Previous work on process scheduling in the embedded systems area include works targeting instruction and data caches. A careful mapping of the process codes to memory can help reduce the conflict misses in instruction caches significantly. We refer the reader to [10] and [5] for elegant process mapping strategies that target instruction caches. Li and Wolfe [6] present a model for estimating the performance of multiple processes sharing a cache. More recently, Kadayif et al [2] have presented a locality-conscious process scheduling strategy where they first evaluate the potential data reuse between processes, and then, using the results of this evaluation, select an order for the process executions (i.e., a schedule). The approach in [2] is different from ours and is restricted in the sense that each process can contain only a single nest.

## 2. Our Approach

Our process scheduling algorithm takes a data space oriented approach. The basic idea is to restructure the process codes to improve data cache locality. Let us first focus on a single array; that is, let us assume that each process manipulates the same array. We will relax this constraint shortly. We first *logically* divide the array in question into tiles (these tiles are called *data tiles*). Then, these data tiles are visited one-by-one (in some order), and we determine for each process a set of iterations (called *iteration tile*) that will be executed in each of its quanta. This approach will obviously increase data reuse in the cache (as in their corresponding quanta the processes manipulate the same set

1530-1591/03 $17.00 © 2003 IEEE

of array elements). The important questions here are how to divide the array into data tiles, in which order the tiles should be visited, and how to determine the iterations to execute (for each process in each quanta). In the following discussion, after presenting a simple example illustrating the overall idea, we explain our approach to these three issues in detail.

Figure 1 shows a simple case (for illustrative purposes) where three processes access the same two-dimensional array (shown in Figure 1(a)). The array is divided into four data tiles. The first and the third processes have two nests while the second process has only one nest. In Figure 1(b), we show for each nest (of each process) how the array is accessed. Each outer square in Figure 1(b) corresponds to an iteration space and the shadings in each region of an iteration space indicate the array portion accessed by that region. Each different shading corresponds to an iteration tile, i.e., the set of iterations that will be executed when the corresponding data tile is being processed. That is, an iteration tile access the data tile(s) with the same type of shading. For example, we see that while the nest in process II accesses all portions of the array, the second nest in process III accesses only half of the array. Our approach visits each data tile in turn, and at each step executes (for each process) the iterations that access that tile (each step here corresponds to three quanta). This is depicted in Figure 1(c). On the other hand, a straightforward (pre-emptive) process scheduling strategy that does not pay attention to data locality might obtain the schedule illustrated in Figure 1(d), where at each time quanta each process executes the half of the iteration space of a nest (assuming all nests have the same number of iterations). Note that the iterations executed in a given quanta in this schedule do not reuse the data elements accessed by the iterations executed in the previous quanta. Consequently, we can expect that data references would be very costly due to frequent off-chip memory accesses. When we compare the access patterns in Figures 1(c) and (d), we clearly see that the one in Figure 1(c) is expected to result in a much better data locality. When we have multiple arrays accessed by the processes in the system, we need to be careful in choosing the array around which the computation is restructured. In this section, we present a possible array selection strategy as well.

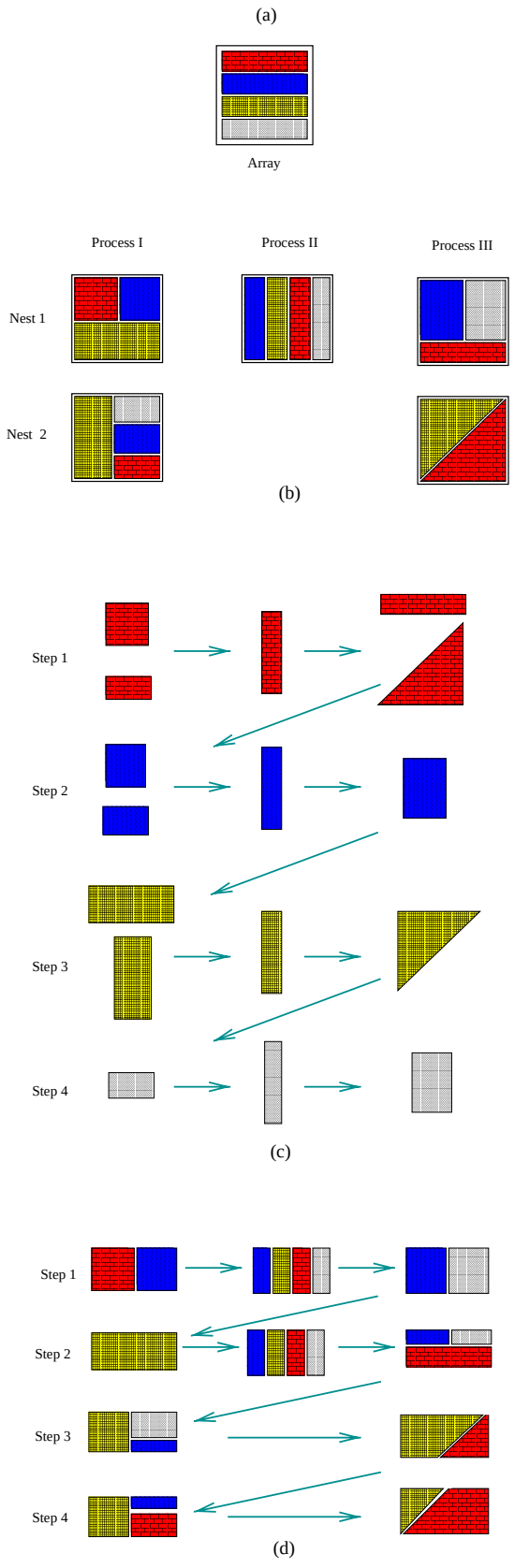


**Figure 1. (a) An array divided into four portions (data tiles). (b) Access pattern exhibited by three processes (iteration tiles). (c) Scheduling steps for our approach. (d) Steps of an alternative scheduling.**

## 2.1. Selecting Data Tile Shape/Size

The size of a data tile (combined with the amount of computation that will be performed for each element of it) determines the amount of work that will be performed in a quanta. Therefore, selecting a small tile size tends to cause frequent process switchings (and incur the corresponding performance overhead), while working with large tile sizes can incur many cache misses (if the cache does not capture locality well). Consequently, our objective is to select the largest tile size that does not overflow the cache. The tile shape, on the other hand, is strongly dependent on two factors: data dependences and data reuse. Since we want to execute all iterations that manipulate the data tile elements in a single quanta, there should not be any circular dependence between the two iteration tiles belonging to the same

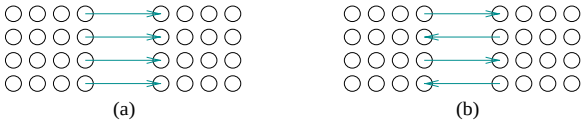


**Figure 2. (a) Legal iteration space tiling. (b) Illegal tiling. Each node denotes an iteration point and each group of nodes is an iteration tile. The arrows between iteration points indicate data dependences.**

process. In other words, all dependences between two iteration tiles should flow from one of them to the other. Figure 2 shows a legal (iteration space) tiling in (a) and an illegal tiling in (b). The second factor that influences the selection of a data tile shape is the degree of reuse between the elements that map on the same tile. More specifically, if, in manipulating the elements of a given data tile, we make frequent accesses to array elements outside the said tile, this means that we do not have good intra-tile locality. An ideal data tile shape must be such that the iterations in the corresponding iteration tile should access only the elements in the corresponding data tile.

Data tiles in $m$-dimensional data (array) space can be defined by $m$ families of parallel hyperplanes, each of which is an ($m$-1) dimensional hyperplane. The tiles so defined are parallelepipeds (except for the ones that reside on the boundaries of data space). Note that each data tile is an $m$-dimensional subset of the data space. Let $\mathcal{M}$ be an $m$-by-$m$ nonsingular matrix whose each row is a vector perpendicular to a tile boundary. $\mathcal{M}$ matrix is referred to as the *data tile matrix*. It is known from previous research on iteration space tiling [1] that $\mathcal{M}^{-1}$ is matrix, each column of which gives the direction vector for a tile boundary (i.e., its columns define the shape of the data tile).

Let $\mathcal{F}$ be a matrix that contains the vectors that define the relations between array elements. More specifically, if, during the same iteration, two array elements $\vec{a_1}$ and $\vec{a_2}$ are accessed together, then $\vec{a_2} - \vec{a_1}$ is a column in $\mathcal{F}$ (assuming that $\vec{a_2}$ is lexicographically greater than $\vec{a_1}$). It can be proven (but not done so here due to lack of space) that the non-zero elements in $\mathcal{MF}$ correspond to the tile-crossing edges in $\mathcal{F}$. So, a good $\mathcal{M}$ should maximize the number of zero elements in $\mathcal{MF}$. The entire iteration space can also be tiled in a similar way that the data space is tiled. An iteration space tile shape can be expressed using a square matrix $\mathcal{H}$ (called the *iteration tile matrix*), each row of which is perpendicular to one plane of the (iteration) tile. As in the data tile case, each column of $\mathcal{H}^{-1}$ gives a direction vector for a tile boundary. It is known from [1] that, in order for an iteration tile to be legal (i.e., dependence-preserving), $\mathcal{H}$ should be chosen such that *none* of the entries in $\mathcal{HD}$ is negative, where $\mathcal{D}$ is the dependence matrix.[1] For clarity, we write this constraint as $\mathcal{HD} \geq 0$.

Based on the discussion above, our tile selection problem can be formulated as follows. Given a loop nest, select an $\mathcal{M}$ such that the corresponding $\mathcal{H}$ does not result in circular data dependences and that $\mathcal{MF}$ has the minimum number of non-zero elements. More technically, assuming that $\mathcal{G}$ is the access (reference) matrix,[2] we have $\mathcal{M} = \mathcal{GH}$. Assuming that $\mathcal{G}$ is invertible (if not, pseudo-inverses can be used), we can obtain $\mathcal{H} = \mathcal{G}^{-1}\mathcal{M}$. Therefore, the condition that needs to be satisfied is $\mathcal{G}^{-1}\mathcal{M} \geq 0$. That is, we need to select an $\mathcal{M}$ matrix such that the number of non-zero entries in $\mathcal{MF}$ is minimum and $\mathcal{G}^{-1}\mathcal{M} \geq 0$.

As an example, let us consider the nest in Figure 3(a). Since there are three references to array $A$, we have three columns in $\mathcal{F}$ (one between each pair of distinct references). Obtaining the differences between the subscript expressions, we have

$$\mathcal{F} = \begin{pmatrix} 1 & 1 & 2 \\ 0 & 1 & 1 \end{pmatrix}.$$

Figure 5(a) shows the columns of this $\mathcal{F}$ on a data space fragment. Figure 5(b), on the other hand, illustrates these vectors on the entire data space. Since there are no data dependences in the code, we can select any data tile matrix $\mathcal{M}$ that minimizes the number of non-zero entries in $\mathcal{MF}$. Assuming

$$\mathcal{M} = \begin{pmatrix} a & b \\ c & d \end{pmatrix},$$

we have

$$\begin{aligned} \mathcal{MF} &= \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} 1 & 1 & 2 \\ 0 & 1 & 1 \end{pmatrix} \\ &= \begin{pmatrix} a & a+b & 2a+b \\ c & c+d & 2c+d \end{pmatrix}. \end{aligned}$$

Recall that our objective is to minimize the number of non-zero entries in $\mathcal{MF}$. One possible choice is $a = 0$, $b = 1$, $c = 1$, and $d = -1$, which gives

$$\mathcal{M} = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}.$$

[1] In this work, we consider three types of data dependences. Consider two iteration points, $\vec{I_1}$ and $\vec{I_2}$ in a given nest. A flow-dependence exists from $\vec{I_1}$ to $\vec{I_2}$ if a variable is computed during execution of $\vec{I_1}$ and subsequently used during execution of $\vec{I_2}$. An anti-dependence exists between $\vec{I_1}$ and $\vec{I_2}$ if a variable is read by $\vec{I_1}$ and subsequently modified by $\vec{I_2}$. An output-dependence occurs when a variable is written by both $\vec{I_1}$ and $\vec{I_2}$. Many dependences that occur in practice have constant distance in each loop level. If there is such a dependence between $\vec{I_1}$ and $\vec{I_2}$, the vector $\vec{d} = \vec{I_2} - \vec{I_1}$ is called the distance vector. All dependence vectors in a given nest can be written collectively as a dependence matrix $\mathcal{D} = [\vec{d_1}, \vec{d_2}, ..., \vec{d_s}]$.

[2] A reference to an array can be represented by $\mathcal{G}\vec{I} + \vec{g}$, where $\mathcal{G}$ is a linear transformation matrix called the array reference (access) matrix, $\vec{g}$ is the offset (constant) vector; $\vec{I}$ is a column vector, called iteration vector, whose elements written left to right represent the loop indices $i_1$, $i_2$, $\cdots$, $i_n$, starting from the outermost loop to the innermost in the loop nest.

```
for i = 3...N
  for j = 2...N
    B[i][j] = A[i-2][j-1] + A[i-1][j-1] + A[i][j]
```
(a)

```
for i = 3...N
  for j = 2...N
    A[i][j] = A[i-2][j-1] + A[i-1][j-1] + B[i][j]
```
(b)

**Figure 3. Two example nests. Note that while both the nests have similar data access patterns as far as accesses to array $A$ are concerned, the second nest also exhibits data dependences. Consequently, selecting a suitable data tile matrix/iteration tile matrix for the second case is more difficult.**

Figure 5(b) also shows how this $\mathcal{M}$ divides the array space into (data) tiles (assuming that a data tile can accommodate maximum nine array elements). Another alternative is $a = -1$, $b = 1$, $c = 0$, and $d = -1$, which results in

$$\mathcal{M} = \begin{pmatrix} -1 & 1 \\ 0 & -1 \end{pmatrix}.$$

Note that in both the cases we zero out two entries in $\mathcal{MF}$.

Now, let us consider the nested loop in Figure 3(b). As far as array $A$ is concerned, while the access pattern exhibited by this nest is similar to the one above, this nest also contains data dependences (as array $A$ is both updated and read). That is,

$$\mathcal{F} = \begin{pmatrix} 1 & 1 & 2 \\ 0 & 1 & 1 \end{pmatrix} \quad \text{and} \quad \mathcal{D} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix},$$

where $\mathcal{D}$ is the data dependence matrix. Consequently, we need to select an $\mathcal{M}$ such that

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \geq 0.$$

In other words, we need to satisfy $b \geq 0$, $d \geq 0$, $a + b \geq 0$, and $c + d \geq 0$. Considering the two possible solutions given above, we see that while $a = 0$, $b = 1$, $c = 1$, and $d = -1$ satisfy these inequalities $a = -1$, $b = 1$, $c = 0$, and $d = -1$ do not satisfy them. That is, the data dependences restrict our flexibility in choosing the entries of the data tile matrix $\mathcal{M}$.

*Multiple references to the same array:* If there are multiple references to a given array, we proceed as follows. We first group the references such that if two references are uniformly references, they are placed into the same group. Two references $\mathcal{G}_1\vec{I} + \vec{g_1}$ and $\mathcal{G}_2\vec{I} + \vec{g_2}$ are said to be uniformly gererated if and only if $\mathcal{G}_1 = \mathcal{G}_2$. For example, $A[i+1][j-1]$ and $A[i][j]$ are uniformly generated, whereas $A[i][j]$ and $A[j][i]$ are not. Then, we count the number of references in each uniformly generated reference group, and the access matrix of the group with the highest count is considered as the representative reference of this array in the nest in question. This is a viable approach because of the following reason. In many array-intensive benchmarks, most of the references (of a given array) in a given nest are uniformly generated. This is particularly true for array-intensive image and video applications where most computations are of stencil type. Note that in our example nests in Figure 3, each array has a single uniformly generated reference set.

*Multiple arrays in a nest:* If we have more than one array in the code, the process of selecting suitable data tile shapes becomes more complex. It should be noted that our approach explained above is a data space centric one; that is, we reorder the computation according to the data tile access pattern. Consequently, if we have two arrays in the code, we can end up with two different execution orders. Clearly, one of these orders might be preferable over the other. Let us assume, for the clarity of presentation, that we have two arrays, $A$ and $B$, in a given nest. Our objective is to determine two data tile matrices $\mathcal{M}_A$ (for array $A$) and $\mathcal{M}_B$ (for array $B$) such that the total number of non-zero elements in $\mathcal{M}_A\mathcal{F}_A$ and $\mathcal{M}_B\mathcal{F}_B$ is minimized. Obviously, data dependences in the code need also be satisfied (i.e., $\mathcal{HD} \geq 0$, where $\mathcal{H}$ is the iteration tile matrix). We can approach this problem in two ways. Let us first focus on $\mathcal{M}_A$. If we select a suitable $\mathcal{M}_A$ so that the number of non-zero elements in $\mathcal{M}_A\mathcal{F}_A$ is minimized, we can determine a $\mathcal{H}$ for the nest in question using this $\mathcal{M}_A$ and the access matrix $\mathcal{G}_A$. More specifically, $\mathcal{H} = {\mathcal{G}_A}^{-1}\mathcal{M}_A$ (assuming that $\mathcal{G}_A$ is invertible). After that, using this $\mathcal{H}$, we can find an $\mathcal{M}_B$ from $\mathcal{M}_B = \mathcal{G}_B\mathcal{H}$. An alternative way would start with $\mathcal{M}_B$, then determine $\mathcal{H}$, and after that, determine $\mathcal{M}_A$. One way of deciding which of these strategies is better than the other is to look at the number of zero (or non-zero) entries in the resulting $\mathcal{M}_A\mathcal{F}_A$ and $\mathcal{M}_B\mathcal{F}_B$ matrices. Obviously, in both the cases, if there are data dependences, the condition $\mathcal{HD} \geq 0$ needs also be satisfied. These two strategies are depicted in Figures 4(a) and (b).

*Multiple nests in a process code:* Each nest can have a different iteration tile shape for the same data tile, and a legal iteration tile (for a given data tile) should be chosen for each process separately. As an example, let us focus on a process code that contains two nests (that access the same array). Our approach proceeds as follows. If there are no dependences, we first find a $\mathcal{M}$ such that the number of non-zero entries in $\mathcal{MF}_1$ and $\mathcal{MF}_2$ is minimized. Here, $\mathcal{F}_1$ and $\mathcal{F}_2$ are the matrices that capture the relations between array elements in the first nest and the second nest, respectively. Then, using this $\mathcal{M}$, we can determine $\mathcal{H}_1$ and $\mathcal{H}_2$ from $\mathcal{H}_1 = {\mathcal{G}_1}^{-1}\mathcal{M}$ and $\mathcal{H}_2 = {\mathcal{G}_2}^{-1}\mathcal{M}$, respectively. In these expressions, $\mathcal{G}_1$ and $\mathcal{G}_2$ are the access matrices in the first and the second nest, respectively. It should be noted, however, that the nests in a given process do not need to use the same data tile matrix $\mathcal{M}$ matrix for the same array. In other words, it is possible (and in some cases actually beneficial) for each nest to select a different $\mathcal{M}$ depending on its $\mathcal{F}$ matrix. This is possible because our data space tiling is a logical concept; that is, we do not physically divide a given array into tiles or change the storage order of its elements in memory. However, in our context, it is more meaningful to work with a single $\mathcal{M}$ matrix for all nests in a process code.

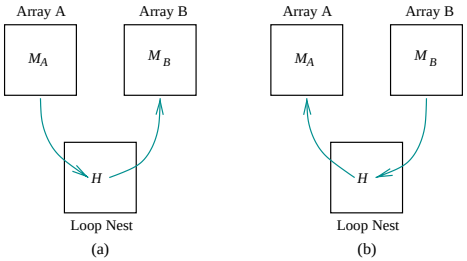


**Figure 4. Two different strategies for determining the data tile matrices and the iteration tile matrix for a given nest that accesses two arrays. In either case, the objective is to minimize the number of non-zero entries in $\mathcal{F}_A\mathcal{M}_A$ and $\mathcal{F}_B\mathcal{M}_B$; however, the case in (a) starts with array $A$ whereas the case in (b)starts with array $B$. Note that this can be extended to the cases where we have more than two arrays in the loop nest.**

This is because when a data tile is visited (during scheduling), we would like to execute *all* iteration tiles from *all* nests (that access the said tile). This can be achieved more easily if we work with a single data tile shape (thus, single $\mathcal{M}$) for the array throughout the computation.

There is an important point that we need to clarify before proceeding further. As mentioned earlier, our approach uses the $\mathcal{F}$ matrix to determine the data tile shape to use. If we have only a single nest, we can build this $\mathcal{F}$ matrix considering each pair of data references to the array. If we consider multiple nests simultaneously, on the other hand, i.e., try to select a single data tile matrix $\mathcal{M}$ for the array for the entire program, we need to have an $\mathcal{F}$ matrix that reflects the combined affect of all data accesses. While it might be possible to develop sophisticated heuristics to obtain such a global $\mathcal{F}$ matrix, in this work, we obtain this matrix by simply combining the columns of individual $\mathcal{F}$ matrices (coming from different nests). We believe that this is a reasonable strategy given the fact that most data reuse occurs within the nests rather than between the nests.

## 2.2. Tile Traversal Order

Data tiles should be visited in an order that is acceptable from the perspective of data dependences [4]. Since in selecting the iteration tiles (based on the data tiles selected) above we eliminate the possibility of circular dependences between iteration tiles, we know for sure that there exists at least one way of traversing the data tiles that lead to legal code. In finding such an order, we use a strategy similar to classical list scheduling that is frequently used in compilers from industry and academia. Specifically, given a set of data tiles (and an ordering between them), we iteratively select a data tile at each step. We start with a tile whose corresponding iteration tile does not have any incoming data dependence edges. After scheduling it, we look the remaining

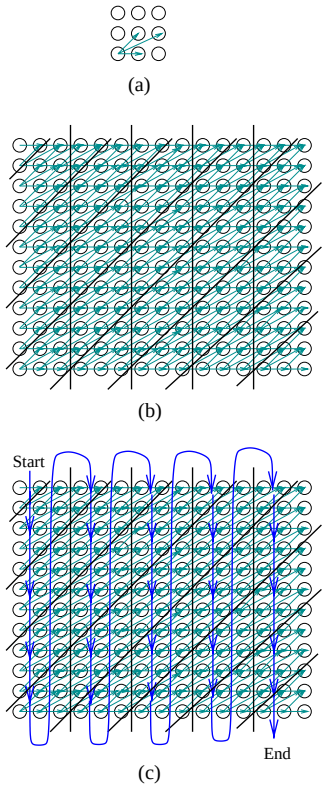


**Figure 5. (a) Columns of $\mathcal{F}$ on a data space fragment. (b) Entire data space with the relations between array elements and tile selection (i.e., how $\mathcal{M}$ divides data space into tiles). (c) Tile traversal order. Note that this is not the only order.**

tiles and select a new one. Note that scheduling a data tile might make some other data tiles scheduleable. This process continues until all data tiles are visited. This is a viable approach as in general the number of data tiles for a given array is small (especially, when the tiles are large). Returning to the example in Figure 5, one possible tile traversal order is illustrated in Figure 5(c).

## 2.3. Restructuring Process Codes and Code Generation

It should be noted that the discusion in the last two subsections is a bit simplified. The reason is that we assumed that the array access matrices are invertible (i.e., nonsingular). In reality, most nests have different access matrices and some access matrices are not invertible. Consequently, we need to find a different mechanism for generation iteration tiles from data tiles. To achieve this, we employ a polyhedral tool called the Omega Library [3]. The Omega library consists of a set of routines for manipulating linear constraints over Omega sets which can include integer variables, Presburger formulas, and integer tuple relations and

sets. We can represent iteration spaces, data spaces, access matrices, data tiles, and iteration tiles using Omega sets.

Let us consider the nest in Figure 3(b) again. We can represent the iteration space of this nest as

$$IS = \{(i,j) : 3 \leq i \leq N \text{ and } 2 \leq j \leq N\}.$$

The set of array elements accessed through reference $A[i-2][j-1]$ can be expressed as

$$DS = \{(a,b) : a = i-2 \text{ and } b = j-1 \text{ and } (i,j) \in IS\}.$$

Similarly, a data tile of array $A$ starting with coordinates $(t_1, t_2)$ is written as

$$\begin{aligned} DT_{t_1,t_2} \quad = \quad & \{(a,b) : \exists c_1 \exists c_2 \text{ s.t. } a = t_1 + c_1 \text{ and } b = t_2 + c_2 \text{ and} \\ & 0 \leq c_1 < C_1 \text{ and } 0 \leq c_2 < C_2 \text{ and } (a,b) \in DS\}. \end{aligned}$$

Here, $C_1$ and $C_2$ are the extents of the tile and $C_1C_2$ is the tile size (i.e., the total number of elements in the tile). Given this data tile, we can easily identify the set of iterations (iteration tile) that access the elements in the data tile. That is,

$$\begin{aligned} IT_{t_1,t_2} \quad = \quad & \{(i,j) : \exists a \exists b \text{ s.t. } a = i-2 \text{ and } b = j-1 \\ & \text{and } (a,b) \in DT_{t_1,t_2} \text{ and } (i,j) \in IS\}. \end{aligned}$$

Another important feature of the Omega Library is its capability of generating for-loops that enumerate the elements that satisfy an Omega set. For example, once $IT_{t_1,t_2}$ above is built, we can ask the library to generate for-loops that enumerate iterations $(i,j)$ that belong to this set.

### 2.4. Overall Algorithm

Based on the discussion in the previous subsections, we present the sketch of our overall algorithm in Figure 6. In this algorithm, in the for-loop between lines 2 and 14, we determine the array around which we need to restructure the process codes. Informally, we need to select an array such that the number of non-zero entries in $\mathcal{M}_s\mathcal{F}_s$ should be minimum, where $1 \leq s \leq L$, $L$ being the total number of arrays in the application. To achieve this, we try each array in turn. Note that this portion of our algorithm works on the entire application code. Next, in line 15, we determine a schedule order for the processes. In this work, we do not propose a specific algorithm to achieve this; however, several heuristics can be used. In line 16, we restructure the process codes (i.e., tile them).

The functionality of the for-loop between the lines 17 and 20 is rather subtle, and deserves some discussion. Note that, up to this point, we have structured a given code considering only a single array. The remaining arrays are affected from this restructuring as well; but, this is indirectly. However, there might be some portions of the code where the array in question ($A_k$) is not used at all. Consequently, the restructuring performed so far cannot touch those portions. In order to optimize these portions too, we need to select another array (different from $A_k$) and repeat the process performed so far. This should continue until all iterations are restructured.

1. max-count $\leftarrow$ 0
2. for each array $A_i$ in the application do
3. determine $\mathcal{F}_i$
4. determine $\mathcal{M}_i$
5. determine $\mathcal{H}$ for each nest
6. for each array $A_j$ where $j \neq i$ do
7. determine $\mathcal{M}_j$
8. count $\leftarrow$ the number of non-zero elements in $\sum_l^L \mathcal{M}_l\mathcal{F}_l$
9. if count $>$ max-count do
10. $k \leftarrow i$
11. max-count = count
12. endif
13. endfor
14. endfor
15. determine a schedule order for the processes
16. restructure (tile) each process code considering $\mathcal{M}_k$ (using the Omega Library)
17. if there are leftover iterations do
18. select another array $A_{k'}$ such that $k' \neq k$
19. drop $A_k$ from consideration and repeat the process
20. endif

**Figure 6. Overall scheduling algorithm.**

## 3. Concluding Remarks

Process scheduling is a key issue in any multiprogrammed system. In this paper, we present a locality conscious scheduling strategy whose aim is to exploit data cache locality as much as possible. It achieves this by restructuring the process codes based on data sharing between processes.

## References


[1] F. Irigoin and R. Triolet. Supernode partitioning. In *Proc. 15th POPL*, pages 319–328, San Diego, CA, January 1988.

[2] I. Kadayif, M. Kandemir, I. Kolcu, and G. Chen. Locality-conscious process scheduling in embedded systems. In *Proc. the Tenth International Symposium on Hardware/Software Codesign*, Colorado, USA May 6-8, 2002.

[3] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and David Wonnacott. The Omega Library interface guide. *Technical Report CS–TR–3445*, CS Dept., University of Maryland, College Park, MD, March 1995.

[4] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1997.

[5] C-G. Lee et al. Analysis of cache related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6), June 1998.

[6] Y. Li and W. Wolfe. A task-level hierarchical memory model for system synthesis of multiprocessors. *IEEE Transactions on CAD*, 18(10), October 1999, pp. 1405–1417.

[7] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proc. the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[8] WARTS: Wisconsin Architectural Research Tool Set. http://www.cs.wisc.edu/∼larus/warts.html

[9] W. Wolfe. *Computers as Components: Principles of Embedded Computing System Design*,Morgan Kaufmann Publishers, 2001.

[10] A. Wolfe. Software-based cache partitioning for real-time applications. In *Proc. the Third International Workshop on Responsive Computer Systems*, September 1993.