I/O-Conscious Tiling for Disk-Resident Data Sets

Mahmut Kandemir[1], Alok Choudhary[2], and J. Ramanujam[3]

[1] EECS Dept., Syracuse University, Syracuse, NY 13244, USA
(mtk@top.cis.syr.edu)
[2] ECE Dept., Northwestern University, Evanston, IL 60208, USA
(choudhar@ece.nwu.edu)
[3] ECE Dept., Louisiana State University, Baton Rouge, LA 70803, USA
(jxr@ee.lsu.edu)

Abstract. This paper describes a tiling technique that can be used by application programmers and optimizing compilers to obtain I/O-efficient versions of regular scientific loop nests. Due to the particular characteristics of I/O operations, straightforward extension of the traditional tiling method to I/O-intensive programs may result in poor I/O performance. Therefore, the technique proposed in this paper customizes iteration space tiling for high I/O performance. The generated code results in huge savings in the number of I/O calls as well as the data volume transferred between the disk subsystem and main memory.

1 Introduction

An important problem that scientific programmers face today is one of writing programs that perform I/O in an efficient manner. Unfortunately, a number of factors render this problem very difficult. First, programming I/O is highly architecture-dependent, i.e., low-level optimizations performed with a specific I/O model in mind may not work well in systems with different I/O models and/or architectures. Second, there is very little help from compilers and run-time systems to optimize I/O operations. While optimizing compiler technology [6] has made impressive progress in analyzing and exploiting regular array access patterns and loop structures, the main focus of almost all the work published so far is the so called *in-core* computations, i.e., computations that make frequent use of the cache–main memory hierarchy rather than *out-of-core* computations, where the main memory–disk subsystem hierarchy is heavily utilized. Third, the large quantity of data involved in I/O operations makes it difficult for the programmer to derive suitable data management and transfer strategies.

Nevertheless it is extremely important to perform I/O operations as efficiently as possible, especially on parallel architectures which are natural target platforms for grand-challenge I/O-intensive applications. It is widely acknowledged by now that the per-byte cost (time) of I/O is orders of magnitude higher than those of communication and computation. A direct consequence of this phenomenon is that no matter how well the communication and computation are optimized, poor I/O performance can lead to unacceptably low overall performance on parallel architectures.

A subproblem within this context is one of writing I/O-efficient versions of loop nests. This subproblem is very important as in scientific codes the bulk of the execution times is spent in loop nests. Thus, it is reasonable to assume that in codes that

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 430–439, 1999.
© Springer-Verlag Berlin Heidelberg 1999

perform frequent I/O, a majority of the execution time will be spent in loop nests that perform I/O in accessing disk-resident multidimensional arrays (i.e., I/O-intensive loop nests). Given this, it is important to assess the extent to which existing techniques developed for nested loops can be used for improving the behavior of the loops that perform I/O. Although at a first glance, it appears that current techniques easily extend to I/O-intensive loop nests, it is not necessarily the case as we show in this paper. In particular, tiling [3, 2, 6], a prominent locality enhancing technique, can result in suboptimal I/O performance if applied to I/O-performing nests without modification.

In this paper we propose a new tiling approach, called *I/O-conscious tiling*, that is customized for I/O performing loops. We believe that this approach will be useful for at least two reasons. First, we express it in a simple yet powerful framework so that in many cases it can be applied by application programmers without much difficulty. Second, we show that it is also possible to automate the approach so that it can be implemented within an optimizing compiler framework without increasing the complexity of the compiler and compilation time excessively. Our approach may achieve significant improvements in overall execution times in comparison with traditional tiling. This is largely due to the fact that while performing tiling it takes I/O specific factors (e.g., file layouts) into account, which leads to substantial reductions in the number of I/O calls as well as the number of bytes transferred between main memory and disk.

The remainder of this paper is organized as follows. Section 2 reviews the basic principles of tiling focusing in particular on the state-of-the-art. Section 3 discusses why the traditional tiling approach may not be very efficient in coding I/O-performing versions of loop nests and makes a case for our modified tiling approach. Section 4 summarizes our experience and briefly comments on future work.

2 Tiling for Data Locality

We say that an array element has *temporal reuse* when it gets accessed more than once during the execution of a loop nest. *Spatial reuse*, on the other hand, occurs when nearby items are accessed [6]. Tiling [3, 2, 7] is a well-known optimization technique for enhancing memory performance and parallelism in nested loops. Instead of operating on entire columns or rows of a given array, tiling enables operations on multidimensional sections of arrays at a time. The aim is to keep the active sections of the arrays in faster levels of the memory hierarchy as long as possible so that when an array item (data element) is reused it can be accessed from the faster (higher level) memory instead of the slower (lower level) memory. In the context of I/O-performing loop nests, the faster level is the *main memory* and the slower level is the *disk subsystem* (or *secondary storage*). Therefore, we want to use tiling to enable the reuse of the array sections in memory as much as possible while minimizing disk activity. For an illustration of tiling, consider the matrix-multiply code given in Figure 1(a). Let us assume that the layouts of all arrays are *row-major*. It is easy to see that from the locality perspective, this loop nest does not exhibit good performance; this is because, although array `Z` has temporal reuse in the innermost loop (the `k` loop) and the successive iterations of this loop access consecutive elements from array `X` (i.e., array `X` has spatial reuse in the innermost loop), the successive accesses to array `Y` touch different rows of this array. Fortunately,

```
for i = 1, N
 for j = 1, N
  for k = 1, N
   Z(i,j) += X(i,k)*Y(k,j)
  endfor
 endfor
endfor
```
(a)

```
for i = 1, N
 for k = 1, N
  for j = 1, N
   Z(i,j) += X(i,k)*Y(k,j)
  endfor
 endfor
endfor
```
(b)

```
for kk = 1, N, B
 kkbb = min(N,kk+B-1)
 for jj = 1, N, B
  jjbb = min(N,jj+B-1)
  for i = 1, N
   for k = kk, kkbb
    for j = jj, jjbb
     Z(i,j) += X(i,k)*Y(k,j)
    endfor
   endfor
  endfor
 endfor
endfor
```
(c)

```
for kk = 1, N, B
 kkbb = min(N,kk+B-1)
 ioread X[1:N,kk:kkbb]
 for jj = 1, N, B
  jjbb = min(N,jj+B-1)
  ioread Z[1:N,jj:jjbb]
  ioread Y[kk:kkbb,jj:jjbb]
  for i = 1, N
   for k = kk, kkbb
    for j = jj, jjbb
     Z(i,j) += X(i,k)*Y(k,j)
    endfor
   endfor
  endfor
  iowrite Z[1:N,jj:jjbb]
 endfor
endfor
```
(d)

```
for ii = 1, N, B
 iibb = min(N,ii+B-1)
 for kk = 1, N, B
  kkbb = min(N,kk+B-1)
  for i = ii, iibb
   for k = kk, kkbb
    for j = 1, N
     Z(i,j) += X(i,k)*Y(k,j)
    endfor
   endfor
  endfor
 endfor
endfor
```
(e)

```
for ii = 1, N, B
 iibb = min(N,ii+B-1)
 ioread Z[ii:iibb,1:N]
 for kk = 1, N, B
  kkbb = min(N,kk+B-1)
  ioread X[ii:iibb,kk:kkbb]
  ioread Y[kk:kkbb,1:N]
  for i = ii, iibb
   for k = kk, kkbb
    for j = 1, N
     Z(i,j) += X(i,k)*Y(k,j)
    endfor
   endfor
  endfor
 endfor
 iowrite Z[ii:iibb,1:N]
endfor
```
(f)

Fig. 1. (a) Matrix-multiply nest (b) Locality-optimized version (c) Tiled version (d) Tiled version with I/O calls (e) I/O-conscious tiled version (f) I/O-conscious tiled version with I/O calls.

state-of-the-art optimizing compiler technology [6] is able to derive the code shown in Figure 1(b) given the one in Figure 1(a). In this so called optimized code array X has temporal reuse in the innermost loop (the j loop now) and arrays Z and Y have spatial reuse meaning that the successive iterations of the innermost loop touch consecutive elements from both the arrays.

However, unless the faster memory in question is large enough to hold the $N \times N$ array Y, many elements of this array will most probably be replaced from the faster memory before they are reused in successive iterations of the outermost i loop. Instead of operating on individual array elements, tiling achieves reuse of array sections by performing the calculations on array sections (in our case sub-matrices). Figure 1(c) shows the tiled version of Figure 1(b). In this tiled code the loops kk and jj are called the *tile loops* whereas the loops i, k, and j are the *element loops*. Note that the tiled version of the matrix-multiply code operates on $N \times B$ sub-matrices of arrays Z and X, and a $B \times B$ sub-matrix of array Y at a time. Here it is important to choose the *blocking factor* B such that all the $B^2 + 2NB$ array items accessed by the element loops i, k, j fit in the faster memory. Assuming that the matrices in this example are in-core (i.e., they fit in main memory) ensuring that the $B^2 + 2NB$ array elements can be kept in cache will be help in obtaining high levels of performance. However, in practice, cache conflicts

occur depending on the cache size, cache associativity, and absolute array addresses in memory. Consequently, the choice of the blocking factor B is very important but difficult [3].

One important issue that needs to be clarified is *how* to select the loops that are to be tiled. A straightforward approach of tiling every loop that can be tiled (subject to dependences [6]) may actually degrade the performance. More sophisticated techniques attempt to tile only the loops that carry some form of reuse. For example, the *reuse-driven tiling* approach proposed by Xue and Huang [7] attempts to tile a given loop nest such that the outer untiled loops will not carry any reuse and the inner tiled loops will carry all the reuse and will consist of as few loops as possible. We stress that since after the linear locality optimizations (a step preceding tiling) most of the inherent reuse in the nest will be carried by the *innermost* loop, almost all tiling approaches tile the *innermost* loop, provided it is legal to do so. To sum up, tiling improves locality in scientific loop nests by capturing data reuse in the inner *element* loops. However, a straightforward application of traditional tiling to I/O-intensive codes may not be very effective as shown in the following section.

3 Tiling for Disk-Resident Data

3.1 The Problem

At first glance the traditional tiling method summarized above seems to be readily applicable to computations on disk-resident arrays as well. Returning to our matrix-multiply code, assuming that we have a total in-core memory of size H that can be allocated to this computation and that all the arrays are disk-resident (e.g., out-of-core), we only need to ensure that $\mathtt{B}^2 + 2\mathtt{NB} \leq \mathtt{H}$. This tiling scheme is shown in Figure 1(d). The call `ioread` is an I/O routine that reads a section of a disk-resident array from file on disk into main memory; `iowrite` performs a similar transfer in the reverse direction. `U[a:b,c:d]` denotes a section of $(\mathtt{b}-\mathtt{a}+1)\times(\mathtt{d}-\mathtt{c}+1)$ elements in a two dimensional array U; the sections for higher-dimensional arrays are defined similarly. It should also be emphasized that since the sections are brought into main memory by explicit I/O commands, the conflict problem mentioned above for cache memories does not happen here.

While such a straightforward adaptation of the state-of-the-art (in-core) tiling leads to *data reuse* for the array sections brought into memory, it can cause *poor* I/O performance. The reason for this becomes obvious from Figure 2(a) which illustrates the layout of the arrays and representative sections from arrays (bounded by thick lines) that are active in memory at the same time. The lines with arrows within the arrays indicate the storage order—assumed to be *row-major* in our case—and each circle corresponds to an array element. Let us envision now how a section of array Z will be read from file into main memory. Since the array is row-major, in order to read the 16 elements shown in the section, it requires 8 I/O calls to the file system, each for only 2 consecutive array elements. Note that even though a state-of-the-art parallel I/O library allows us to read this rectangular array section using only a single high-level library call, since the elements in the section to be read are not fully consecutive in the file, it will still require 8 internal I/O calls for the said library to read it. It should also be noted that

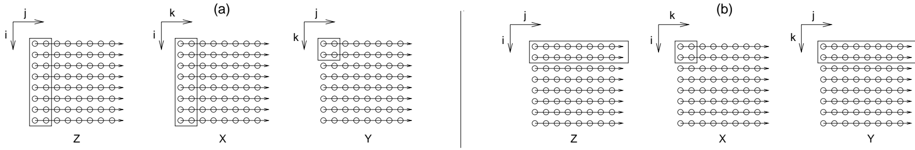

Fig. 2. (a) Unoptimized and (b) Optimized tile access patterns for matrix-multiply nest.

the alternative of reading the whole array and sieving out the unwanted array items is, in most cases, unacceptable due to the huge array sizes in I/O-intensive codes. Similar situations also occur with arrays X and Y. The source of the problem here is that the traditional (in-core) tiling attempts to optimize *what* to read into the faster memory, not *how* to read it. While this does not cause a major problem for the cache–main memory interface, the high disk access times render *"how to read"* a real issue. A simple rule is to always read array sections in a *layout conformant* manner. For example, if the file layout is row-major we should try to read as many rows of the array as possible in a single I/O call. In our matrix-multiply example, we have failed to do so due to the tiling of the innermost j loop, which reduces the number of elements that can be consecutively read in a single I/O call. Since this loop carries spatial reuse for both Z and Y, we should use this loop to read as many consecutive elements as possible from the said arrays. For example, instead of reading an $N \times B$ section from array Z, we should try to read a $B \times N$ section as shown in Figure 2(b) if it is possible to do so.

3.2 The Solution

Our solution to the tiling problem is as follows. As a first step, prior to tiling, the loops in the nest in question are reordered (or transformed) for maximum data reuse in the innermost loop. Such an order can be obtained either by an optimizing compiler or a sophisticated user. In the second step, we tile all the loops by a blocking factor B except the *innermost* loop which is *not tiled*. Since, after ordering the loops for locality, the innermost loop will, hopefully, carry the highest amount of spatial reuse in the nest, by not tiling it our approach ensures that the number of array elements read by individual I/O calls will be maximized.

As an application of this approach, we consider once more the matrix-multiply code given in Figure 1(a). After the first step, we obtain the code shown in Figure 1(b). After that, tiling the i and k loops and leaving the innermost loop (j) untiled, we reach the code shown in Figure 1(e). Finally, by inserting the I/O read and write calls in appropriate places between the tile loops, we have the final code given in Figure 1(f). The tile access pattern for this code is shown in Figure 2(b). We note that the section-reads for arrays Z and Y, and the section-writes for array Z are very efficient since the file layouts are row-major. It should be stressed that the amount of memory used by the sections shown in Figures 2(a) and (b) is exactly the *same*. It should also be noted that as compared to arrays Z and Y, the I/O access pattern of array X is not as efficient. This is due to the nature of the matrix-multiply nest and may not be as important. Since the sections of array X are read much less frequently compared to those of the other

two arrays (because, it has temporal reuse in the innermost loop), the impact of the I/O behavior of array `X` on the overall I/O performance of this loop nest will be less than that of the other two.

An important issue is the placement of I/O calls in a given tiled nest. There are two subproblems here: (1) *where are the I/O calls placed?* and (2) *what are the parameters to the I/O calls?* All we need to do is to look at the indices used in the subscripts of the reference in question, and insert the I/O call associated with the reference in between appropriate tile loops. For example, in Figure 1(e), the subscript functions of array `Z` use only loop indices `i` and `j`. Since there are only two tile loops, namely `ii` and `kk`, and only `ii` controls the loop index `i`, we insert the I/O read routine for this reference just after the `ii` loop as shown in Figure 1(f). But, the other two references use the element loop index `k` which varies with the tile loop index `kk`; therefore, we need to place the read routines for these references inside the `kk` loop (just before the element loops). The write routines are placed using a similar reasoning. For handling the second subproblem, namely, determining the sections to read and write, we employ the method of *extreme values of affine functions*; this method computes the maximum and minimum values of an affine function using Fourier-Motzkin elimination [6]. The details of this are omitted for lack of space.

3.3 Automating the Approach

In the matrix-multiply example it is straightforward to tile the loop nest. The reason for this is the presence of the innermost loop index in *at most* one subscript position of each reference. In cases where the innermost loop index appears in *more than one* subscript positions we have a problem of determining the section shape. Consider a reference such as `Y(j+k,k)` where `k` is the innermost loop index. Since, in our approach, we do not tile the innermost loop, when we try to read the bounding box which contains all the elements accessed by the innermost loop, we may end up having to read almost the whole array. Of course, this is not acceptable in general as in I/O-intensive routines we may not have a luxury of reading the whole array into memory at one stroke. This situation occurs in the example shown in Figure 3(a). Assuming *row-major* layouts, if we tile this nest as it is using the I/O-conscious approach, there will be no problem with array `Z` as we can read a $\mathtt{B}\times\mathtt{N}$ section of it using the fewest number of I/O calls. However, for array `Y` it is not trivial to identify the section to be read because the innermost loop index accesses the array in a diagonal fashion. The source of the problem is that this array reference does not fit our criterion which assumes at most a single occurrence of the innermost loop index. In our example reference, the innermost loop index `k` appears in both the subscript positions. In general we want each reference to an `m`-dimensional *row-major* array `Z` to be in either of the following two forms:

$\mathtt{Z}(f_1, f_2, ..., f_m)$: In this form f_m is an affine function of all loop indices with a coefficient of 1 for the innermost loop index whereas f_1 through $f_{(m-1)}$ are affine functions of all loop indices except the innermost one.

$\mathtt{Z}(g_1, g_2, ..., g_m)$: In this form all g_1 through g_m are affine functions of all loop indices except the innermost one.

In the first case we have *spatial reuse* for the reference in the innermost loop and in the second case we have *temporal reuse* in the innermost loop. Since, according to our I/O-conscious tiling, all loops except the innermost one will be tiled using a blocking factor $\mathtt{B}$, for the arrays which fit in the first form we can access $\mathtt{B} \times ... \times \mathtt{B} \times \mathtt{N}$ sections (assuming $\mathtt{N}$ is the number of iterations of the innermost loop), which minimizes the number of I/O calls per array section. As for the arrays that fit into the second form, we can access sections of $\mathtt{B} \times ... \times \mathtt{B} \times \mathtt{B}$ as all the loops that determine the section are tiled. Our task, then, is to bring each array (reference) to one of the forms given above.

Algorithm In the following we propose a *compiler algorithm* to transform a loop nest such that the resultant nest will have references of the forms mentioned above. We assume that the compiler is to determine the most appropriate file layouts for each array as well as a suitable loop (iteration space) transformation; that is, we assume that the layouts are *not* fixed as a result of a previous computation phase. Our algorithm, however, can be extended to accommodate those cases as well.

In our framework each execution of an $\mathtt{n}$-*deep* loop nest is represented using an *iteration vector* $\bar{\mathtt{I}} = (\mathtt{i}_1, \mathtt{i}_2, ..., \mathtt{i}_\mathtt{n})$ where $\mathtt{i}_\mathtt{j}$ corresponds to $\mathtt{j}^{th}$ loop from outermost. Also each reference to an $\mathtt{m}$-*dimensional* array $\mathtt{Z}$ is represented by an *access* (*reference*) *matrix* $\mathtt{L}_z$ and an *offset vector* $\bar{\mathtt{o}}_z$ such that $\mathtt{L}_z\bar{\mathtt{I}} + \bar{\mathtt{o}}_z$ addresses the element accessed by a specific $\bar{\mathtt{I}}$ [6]. As an example, consider a reference `Z(i+j,j-1)` to a two-dimensional array in a two-deep loop nest with `i` is the inner and `j` is the outer. We have, $\mathtt{L}_z = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ and $\bar{\mathtt{o}}_z = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$. Also we know from previous research in optimizing compilers [6, 4, 5] that (omitting the offset vector) assuming $\mathtt{L}$ is an access matrix, an $\mathtt{n} \times \mathtt{n}$ *loop* transformation matrix $\mathtt{T}$ transforms it to $\mathtt{LT}^{-1}$ and an $\mathtt{m} \times \mathtt{m}$ *data* transformation matrix $\mathtt{M}$ transforms it to $\mathtt{ML}$. Given an $\mathtt{L}$ matrix, our aim is to find matrices $\mathtt{T}$ and $\mathtt{M}$ such that the transformed access matrix will fit in one of our two desired forms mentioned above. Notice also that while $\mathtt{T}$ is unique to a loop nest, we need to find an $\mathtt{M}$ for each disk-resident array.

Since for a given $\mathtt{L}$ determining both $\mathtt{T}$ and $\mathtt{M}$ simultaneously such that $\mathtt{MLT}^{-1}$ will be in a desired form involves solving a system of non-linear equations, we solve it using a *two-step* approach. In the first step we find a matrix $\mathtt{T}$ such that $\mathtt{L}' = \mathtt{LT}^{-1}$ will have a zero last column except the element in an rth row which is 1 (for spatial locality in the innermost loop) or we find a $\mathtt{T}$ such that the last column of $\mathtt{L}' = \mathtt{LT}^{-1}$ will be zero column (for temporal locality in the innermost loop). If the reference is optimized for spatial locality in the first step, in the second step we find a matrix $\mathtt{M}$ such that this r^{th} row in $\mathtt{L}'$ (mentioned above) will be the *last* row in $\mathtt{L}'' = \mathtt{ML}'$.

The overall algorithm is given in Figure 4. In Step 1, we select a *representative reference* for each array accessed in the nest. Using profile information might be helpful in determining run-time access frequencies of individual references. For each array, we select a reference that will be accessed maximum number of times in a typical execution. Since for each array we have two desired candidate forms (one corresponding to temporal locality in the innermost loop and one corresponding to spatial locality in the innermost loop), in Step 2, we exhaustively try all 2^s possible loop transformations, each corresponding to a specific combination of localities (spatial or temporal) for the

```
for i =              for u =              for u =              for u =              for u =
 for j =              for v =              for v =              for v =              for v =
  for k =              for w =              for w =              for w =              for w =
   Z(i,k+j)=            Z(u,v)=              Z(u,v)=              Z(u+w,v-w)=          Z(u,w)=
     Y(k,i+j+k)           Y(u+v,v-w)           Y(u+v,v+w)           Y(v,u+v)             Y(u,u+v+w)
  endfor               endfor               endfor               endfor               endfor
 endfor               endfor               endfor               endfor               endfor
endfor               endfor               endfor               endfor               endfor
   (a)                  (b)                  (c)                  (d)                  (e)
```

Fig. 3. (a) An example loop nest; (b–e) Transformed versions of (a).

arrays. In Step 2.1, we set the desired access matrix $\mathtt{L}'_i$ for each array i and in the next step we determine a loop transformation matrix T which obtains as many desired forms as possible. A typical optimization scenario is as follows. Suppose that in an alternative v where $1 \le v \le 2^s$ we want to optimize references 1 through b for temporal locality and references $b+1$ through s for spatial locality. After Step 2.2, we typically have c references that can be optimized for temporal locality and d references that can be optimized for spatial locality where $0 \le c \le b$ and $0 \le d \le (s-b)$. This means that a total of $s-(c+d)$ references (arrays) will have *no locality* in the *innermost* loop. We do not apply any data transformations for the c arrays that have temporal locality (as they are accessed infrequently). For each array j (of maximum d arrays) that can be optimized for spatial locality, within the loop starting at Step 2.3, we find a data transformation matrix such that the resulting access matrix will be in our desired form. In Step 2.5 we record c, d, and $s-(c+d)$ for this alternative and move to the next alternative. After all the alternatives are processed we select the *most suitable* one (i.e., the one with the largest $c+d$ value). There are three important points to note here. First, in completing the partially-filled loop transformation matrix T, we use the approach proposed by Bik and Wijshoff [1] to ensure that the resulting matrix *preserves* all *data dependences* [6] in the original nest. Second, we also need a mechanism (when necessary) to favor some arrays over others. The reason is that it may not always be possible to find a T such that all $\mathtt{L}'_i$ arrays targeted in a specific alternative are realized. In those cases, we need to omit some references from consideration, and attempt to satisfy (optimize) the remaining. Again, profile information can be used for this purpose. Third, even an alternative does not specifically target the optimization of a reference for a specific locality, it may happen that the resultant T matrix generates such a locality for the said reference. In deciding the most suitable alternative we need also take such (unintentionally optimized) references into account.

As an example application of the algorithm, consider the loop nest shown in Figure 3(a) assuming that optimizing array Z is more important than optimizing array Y. The access matrices are $\mathtt{L}_z = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}$ and $\mathtt{L}_y = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$. We have four possible optimization alternatives here: (1) temporal locality for both arrays; (2) temporal locality for Z and spatial locality for Y; (3) temporal locality for Y and spatial locality for Z; and (4) spatial locality for both arrays. These alternatives result in the following transformations.

$$\textit{Alternative (i)}: \mathtt{T}^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & -1 \end{bmatrix}, \mathtt{M}_y = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \Rightarrow \mathtt{L}''_z = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \text{ and } \mathtt{L}''_y = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & -1 \end{bmatrix}.$$

INPUT: access matrices for the references in the nest
OUTPUT: loop transformation matrix $\mathtt{T}$ and data transformation matrix $\mathtt{M}_i$ for each disk-resident array i

(1) using profiling determine a representative access matrix for each array i $(1 \leq i \leq s)$
(2) *for* each of the 2^s alternatives *do*
(2.1) determine target $\mathtt{L}'_1$, $\mathtt{L}'_2$,...,$\mathtt{L}'_s$
(2.2) using $\mathtt{L}_i\mathtt{T}^{-1} = \mathtt{L}'_i$ determine a $\mathtt{T}$
(2.3) *for* each array j with the spatial locality *do*
(2.3.1) let r_k be the row (if any) containing the only non-zero element in the last column for $\mathtt{L}'_i$
(2.3.2) find an $\mathtt{M}_j$ such that $\mathtt{L}''_j = \mathtt{M}_j\mathtt{L}'_j$ will be in the desired form (i.e., r_k will be the last row)
(2.4) *endfor*
(2.5) record for the current alternative the number of references with temporal locality, spatial locality, and no locality in the innermost loop
(3) *endfor*
(4) select the most suitable alternative (see the explanation in the text)
(5) *I/O-conscious tile* the loop nest and insert the I/O read/write routines

Fig. 4. I/O-conscious tiling algorithm.

$$\textit{Alternative (ii)} : \mathtt{T}^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 1 \end{bmatrix}, \mathtt{M}_y = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \Rightarrow \mathtt{L}''_z = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \text{ and } \mathtt{L}''_y = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}.$$

$$\textit{Alternative (iii)} : \mathtt{T}^{-1} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \Rightarrow \mathtt{L}''_z = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & -1 \end{bmatrix} \text{ and } \mathtt{L}''_y = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}.$$

$$\textit{Alternative (iv)} : \mathtt{T}^{-1} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \mathtt{M}_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \Rightarrow \mathtt{L}''_z = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ and } \mathtt{L}''_y = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}.$$

The resulting programs are shown in Figures 3(b)– 3(e) for alternatives (i), (ii), (iii), and (iv), respectively (the transformed loop bounds are omitted for clarity). It is easy to see that the alternatives (i) and (ii) are superior to the other two. Alternative (iii) cannot optimize array Z and alternative (iv) optimizes both arrays for spatial locality. Our algorithm chooses alternative (i) or (ii) as both ensure temporal locality for the LHS array and spatial locality for the RHS array in the innermost loop.

4 Conclusions and Future Work

In this paper we have presented an I/O-conscious tiling strategy that can be used by programmers or can be automated in an optimizing compiler for the I/O-intensive loop nests. We have shown that a straightforward extension of the traditional tiling strategy to I/O-performing loop nests may lead to poor performance and demonstrated the necessary modifications to obtain an I/O-conscious tiling strategy. Our current interest is

in implementing this approach fully in a compilation framework and in testing it using large programs.

References

[1] A. Bik and H. Wijshoff. On a completion method for unimodular matrices. Technical Report 94–14, Dept. of Computer Science, Leiden University, 1994.

[2] F. Irigoin and R. Triolet. Super-node partitioning. In *Proc. 15th Annual ACM Symp. Principles of Programming Languages*, pp. 319–329, January 1988.

[3] M. Lam, E. Rothberg, and M. Wolf. The cache performance of blocked algorithms. In *Proc. 4th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, April 1991.

[4] W. Li. *Compiling for NUMA parallel machines*. Ph.D. Dissertation, Cornell University, Ithaca, NY, 1993.

[5] M. O'Boyle and P. Knijnenburg. Non-singular data transformations: Definition, validity, applications. In *Proc. 6th Workshop on Compilers for Parallel Computers*, pp. 287–297, 1996.

[6] M. Wolfe. *High Performance Compilers for Parallel Computing*, Addison-Wesley, 1996.

[7] J. Xue and C.-H. Huang. Reuse-driven tiling for data locality. In *Languages and Compilers for Parallel Computing*, Z. Li et al., Eds., Lecture Notes in Computer Science, Volume 1366, Springer-Verlag, 1998.