Blocked Array Layouts for Multilevel Memory Hierarchies

Evangelia Athanasaki and Nectarios Koziris

Computing Systems Laboratory
School of Electrical and Computer Engineering
National Technical University of Athens, 15773 Athens, Greece
Email: {valia,nkoziris}@cslab.ece.ntua.gr

Abstract. Minimizing cache misses is one of the most important factors to reduce average latency for memory accesses. Tiling is a widely used loop iteration reordering technique for improving locality of references. Tiled codes modify the instruction stream to exploit cache locality for array accesses. In this paper, we further reduce cache misses, restructuring the memory layout of multidimensional arrays, that are accessed by tiled instruction code. In our method, array elements are stored in a blocked way, exactly as they are swept by the tiled instruction stream. We present a straightforward way to easily translate multidimensional indexing of arrays into their blocked memory layout using simple binary-mask operations. Indices for such array layouts are easily calculated based on the algebra of dilated integers, similarly to morton-order indexing. Actual experimental results, using matrix multiplication and LU-decomposition on various size arrays, illustrate that execution time is greatly improved when combining tiled code with tiled array layouts and binary mask-based index translation functions. Simulations using the Simplescalar tool, verify that enhanced performance is due to the considerable reduction of total cache misses.

1 Introduction

Due to the constantly dilating gap between the performance of processors and DRAMs [11], most applications still waste much of their execution time, waiting for data to be fetched from main memory. To bridge this gap, multi-level memory hierarchies are used and compiler transformations are introduced to increase locality of references for iterative instruction streams.

In order to increase the percentage of memory accesses satisfied by caches, loop transformations are used to change the order of iterations in favor of locality of references. Loop interchange, reversal and skewing modify the data access order. Loop unrolling and software pipelining exploit registers and pipelined datapath [4]. Loop tiling and data shackling [8] both restructure the control flow of a program. These control transformations modify the order of the instruction stream, to increase possible reuse of same or nearby array elements. In their most cited work regarding unimodular control transformations, Wolf and Lam in [15] attempt to find the best combination of such transformations which ensure the correct computation order, while increasing locality of accesses to cache memory.

Using control transformations, we modify data access order but not the data storage order. Cierniak and Li in [3, 7] present a cache locality optimization algorithm which combines both loop (control) and linear array layout transformations. However, lessening its complexity extra constraints should be defined. Another unified, systematic algorithm, is the one presented by Kandemir et al in [5–7], which aims at utilizing spatial reuse in order to obtain good locality.

All previous approaches assumed linear array layouts. Nevertheless, the linear array memory layouts do not exactly correspond to the order, array elements are accessed by the tiled instruction stream. Arrays should be stored, exactly as they are accessed, so that instruction and data stream are aligned. Chatterjee et al [1, 2] explored the merit of nonlinear memory layouts and quantified their implementation cost. Although they claim for increasing execution-time performance, referring to four-dimensional arrays, as proposed, counterbalances any advantage data locality obtained due to blocked layouts. Furthermore, blocked layouts in combination with level-order, Ahnentafel and Morton indexing were used by Wise et al in [14, 13]. Their quad-tree based layouts can gain no locality advantage at non-recursive codes, although efficient element indexing is used. Especially, since they use recursion down to the level of a single array element, extra loss of performance is induced.

Nonlinear layouts were proved to favor cache locality in all levels of memory hierarchy, including L1, L2 caches and even avoiding TLB thrashing [10]. Notwithstanding, it does not suffice to identify the optimal blocked layout for a specific array. We also need to automatically generate the mapping from the multidimensional iteration indices to the correct location of the respective data element in linear memory. Thus, blocked layouts are very promising subject to an efficient address computation method.

In this paper, in order to facilitate the automatic generation of tiled code that accesses blocked array layouts, we propose an address calculation method of the array indices. We adopted some kind of L_{4D} layout and dilated integer indexing similar to Morton-order arrays. Thus, we combine data locality, due to blocked layouts, with efficient element access, due to our straightforward indexing. Our method is very effective at reducing cache misses, since the deployment of the array data in memory follows the order of accesses by the tiled instruction code, achieved at no extra runtime cost. Experimental results were conducted using the matrix multiplication and LU-decomposition codes. We ran two types of experiments, actual execution of real codes and simulation of memory and cache usage using SimpleScalar. We compared our method with the methods of Kandemir and Cierniak, and show reduction on overall misses and thus acceleration of the final code performance. The comparison with Chatterjee's implementation, which uses non-linear four-dimensional array layouts, proves that limiting cache misses do not suffice if address computation is not efficient.

The rest of the paper is organized as follows: Section 2 briefly discusses the problem of data locality using as example the typical matrix multiplication algorithm. Section 3 reviews definitions related to Morton ordering. Section 4 presents blocked data layouts, collating Chatterjee's et al implementation to ours for efficient array indexing. Section 5 illustrates execution and simulation comparisons with so far presented methods, with results showing that our algorithm reduces cache misses and improves overall performance. Finally, concluding remarks are presented in Section 6.

2 The Problem: Improving Cache Locality for Array Computations

In this section, we present, stepwise, all optimization phases to improve locality of references with the aid of the typical matrix multiplication kernel.

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

(a) unoptimized version

```
for (jj=0; jj<N; jj+=step)
  for (kk=0; kk<N; kk+=step)
    for (i=0; i<N; i++)
      for (j=jj; (j<N && j<jj+step); j++)
        for (k=kk; (k<N && k<kk+step); k++)
          C[i,j]+=A[i,k]*B[k,j];
```

(b) tiled code

```
for (kk=0; kk<N; kk+=step)
  for (jj=0; jj<N; jj+=step)
    for (i=0; i<N; i++)
      for (k=kk; (k<N && k<kk+step); k++)
        for (j=jj; (j<N && j<jj+step); j++)
          Cr[i,j]+=Ar[i,k]*Br[k,j];
```

(c) loop and data transformation

```
for (ii=0; ii<N; ii+=step)
  for (kk=0; kk<N; kk+=step)
    for (jj=0; jj<N; jj+=step)
      for (i=ii; (i<N && i<ii+step); i++)
        for (k=kk; (k<N && k<kk+step); k++)
          for (j=jj; (j<N && j<jj+step); j++)
            Czz[i_m+j_m]+=Azz[i_m+k_m]*Bzz[k_m+j_m];
```

(d) blocked array layout

Fig. 1. Matrix multiplication

Loop Transformations: Figure 1a shows the typical, unoptimized version of the matrix multiplication code. Tiling (figure 1b) restructures the execution order, such that the number of intermediate iterations and, thus, data fetched between reuses, are reduced. So, useful data are not evicted from the cache, before being reused.

Loop and Data Transformations: Since, loop transformation alone can not result in the best possible data locality, the utilization of both control and data transformations becomes necessary. In figure 1b, loop k scans different rows of B. Given a row-order array layout, spatial reuse can not be exploited for B along the innermost loop k. Focusing on self-spatial reuse [7] (since self-temporal reuse can be considered as a subcase of self-spatial, while group spatial are rare), the transformed code takes the form of figure 1c, which has proved to give the best performance, so far. Firstly, we fixed the layout of array C (Left Hand Side array: LHS). Choosing j to be the innermost loop, the fastest changing dimension of array $C[i,j]$ should be controlled by this index, namely C should be stored by rows (Cr). Similarly, array $B[k,j]$ should also be stored by rows (Br). Finally, placing loops in ikj order is preferable, because we can also exploit self-temporal reuse in the second innermost loop for array C. Thus, $A[i,k]$ should also be stored by rows (Ar).

Loop and non-linear Data Transformations: In order to evaluate the merit of non-linear data transformations, we used the code of figure 1d. We assume that the array element storage of all three arrays follows the scanning order of the program execution (we call this layout ZZ-order, as

extensively presented in the following section). The loop ordering remains the same as in figure 1c, except that, tiling is also applied in loop i, so as to have homomorphic

shaped tiles in all three arrays and simplify the computations needed to find the array element location.

3 Morton Order Matrices

In this section we present the basic elements of the dilated integer algebra that are needed to explain the notion of Morton order [12], a recursive storage order of an array. Morton defined the indexing of a two-dimensional array as in figure **??**, and pointed out the conversion to and from cartesian indexing available through bit interleaving.

The hexadecimal constant $evenBits = 0x55555555$ has all the even bits set and all odd bits cleared. Likewise, $oddBits = evenBits << 1$ has the value $0xaaaaaaaa$. Let us consider a cartesian row index i with its significant bits "dilated" so that they occupy the digits set in $oddBits$ and a cartesian index j been dilated so its significant bits occupy those set in $evenBits$. If array A is stored in Morton order, then element $[i, j]$ can be accessed as $A[i + j]$ regardless of the size of array. The following definition is only for two-dimensional arrays.

Definition 1 *The even-dilated representation of* $j = \sum_{k=0}^{w-1} j_k\, 2^k$ *is* $\sum_{k=0}^{w-1} j_k 4^k$*, denoted* $\overrightarrow{j}$*. The odd-dilated representation of* $i = \sum_{k=0}^{w-1} i_k\, 2^k$ *is* $2\overrightarrow{i}$ *and is denoted* $\overleftarrow{i}$*.*

Theorem 1. *The Morton index for the* $< i, j >^{th}$ *element of a matrix is* $\overleftarrow{i} \vee \overrightarrow{j}$*, or* $\overleftarrow{i} + \overrightarrow{j}$*.*

If an index is only used as a column index, then its dilation should be odd. Otherwise it is even-dilated. If used in both roles, then doubling gives the odd-dilation as needed. So, after i and j are translated to their images, $\overleftarrow{i}$ and $\overrightarrow{j}$, the code of matrix multiplication will be as follows:

```
#define evenIncrement(i)(i = ((i - evenBits)&evenBits))
#define oddIncrement(i)(i = ((i - oddBits)&oddBits))
    for (i=0; i < colsOdd; oddIncrement(i))
        for (j=0; j < rowsEven; evenIncrement(j))
            for (k=0; k < rowsAEven; evenIncrement(k))
                C[i + j]+ = A[i + k] * B[2 * k + j];
```

where $rowsEven$,$colsOdd$ and $rowsAEven$ are the bounds of arrays when transformed in dilated integers. Index k is used

both as column and as row index. Therefore, it is translated to $\overrightarrow{k}$ and for the column indexing of array B, $2 * k$ represents $\overleftarrow{k}$.

4 Blocked Array Layouts

Since the performance of so far presented methods [3, 7, 15] is better when tiling is applied, we would achieve even better locality, if array data are stored neither column-wise nor row-wise, but in a blocked layout. Such layouts were used by Chatterjee et al in [1, 2], in order to obtain better interaction with cache memories. According to [1], a 2-dimensional $m \times n$ array can be viewed as a $\lceil \frac{m}{t_R} \rceil \times \lceil \frac{n}{t_C} \rceil$ array of $t_R \times t_C$ tiles. Equivalently, the original 2-dimensional array space (i, j) is mapped into a 4-dimensional

space (t_i, t_j, f_i, f_j) as seen in figure 3. The proposed layout for the space $L_F : (f_i, f_j)$ of the tile offsets is a canonical one, namely column-major or row-major, according to the access order of array elements by the program code. The transformation function for the space $L_T : (t_i, t_j)$ of the tile co-ordinates can be either canonical or follow the Morton ordering.

4.1 The 4D Layout

In L_{4D} both L_T and L_F are canonical layouts (row-major in our example). In figure 2 there is the original array with 27×27 elements, divided to 4×4 tiles. On the border of the array, padding is added, so that whole tiles are formed. The storage order of tiles is also marked.

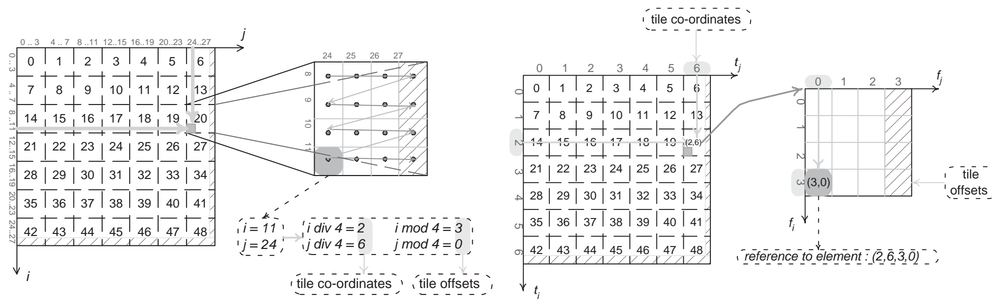

Fig. 2. The tiled array according to L4d

Fig. 3. The 4-dimensional array in 4D layout

The final 4-dimensional array is drawn in figure 3, presented as a 2-dimensional space of tile co-ordinates which contain pointers on 2-dimensional arrays of tile offsets. The value of each dimension is calculated as follows:

tile co-ordinates : $t_i = i \; div \; step$ and $t_j = j \; div \; step$

tile offsets : $f_i = i \; mod \; step$ and $f_j = j \; mod \; step$

4.2 The Morton Layout

In L_{MO}, L_T follows Morton ordering while L_F has a canonical layout (in our example row-major layout, in order to keep step with access order). The original array is in figure 4, where the storage order of tiles is also marked, The padding elements are more than in L_{4D}, because the array dimension should be a power of two.

If we draw the 4-dimensional array (similarly to figure 3), the padding elements will no more be on the border of the array, but they will be mixed with the original elements. The value of each dimension is calculated as follows:

tile co-ordinates : $tile_{num} = \overleftarrow{(i \; div \; step)} | \overrightarrow{(j \; div \; step)}$

$t_i = tile_{num} \; div \; step$ and $t_j = tile_{num} \; mod \; step$

tile offsets : $f_i = i \; mod \; step$ and $f_j = j \; mod \; step$

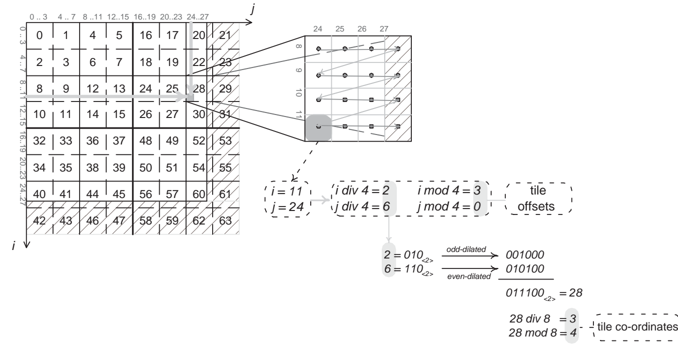

Fig. 4. The tiled array according to Morton layout

where $\overleftarrow{x}$ and $\overrightarrow{x}$ is, respectively, the odd and even dilated representation of x.

When applying the Morton layout, the resulting code does not contain any time-consuming calculations when locating the right array elements, since boolean operations are considered, which add the minimum possible overhead. However, referring to 4-dimensional arrays produces repetitive load and add instructions, which, as seen in experimental results, are too time consuming and degrade the total performance.

4.3 Our Approach

We expand the blocked layouts, storing array elements in the same order as they are accessed when operations are executed. This storage layout is presented in figure 5 for an 8×8 array which is split into tiles of size 4×4. The grey line shows the order by which the program sweeps the array data, while numbering illustrates the order, data are stored. We split arrays in tiles of the same size as those used by the program that scans the elements of the array. We denote the transformation of figure 5 as "ZZ", because the inner of the tiles is scanned row-wise (in Z-like order) and the shift from tile to tile is Z-like as well. The first letter (Z) of transformation denotes the shift from tile to tile while the second indicates the sweeping within a tile. The the code shown below does a "ZZ" sweeping in array A.

Analogously, the other three types of transformation are shown in figures 6 to 8. For example, in "NZ" transformation, the sweeping of the array data within a tile is done row-wise (Z-like order), while the shift from tile to tile is column-wise (N-like order).

"ZZ"
```
for (ii=0; ii < N; ii+=step)
  for (jj=0; jj < N; jj+=step)
    for (i=ii; (i < ii + step && i < N); i++)
      for (j = jj; (j < jj+step && j < N); j++)
        A[i,j]=...;
```

"NZ"
```
for (jj=0; jj < N; jj+=step)
  for (ii=0; ii < N; ii+=step)
    for (i = ii; (i < ii + step && i < N); i++)
      for (j = jj; (j < jj + step && j < N); j++)
        A[i,j]=...;
```

4.4 Mask Theory

Since compilers support only linear layouts (column-order or row-order) and not blocked array layouts, sweeping the array data of our approach, can be done using one-dimensional

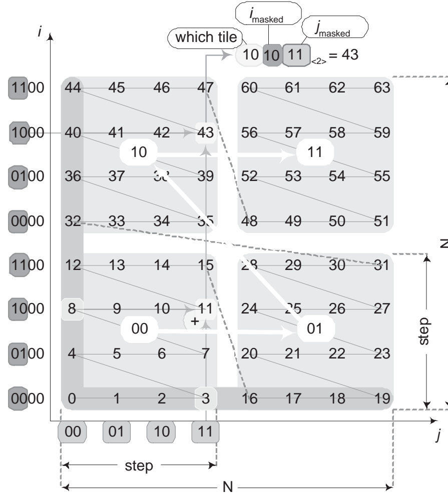

Fig. 5. ZZ-transformation

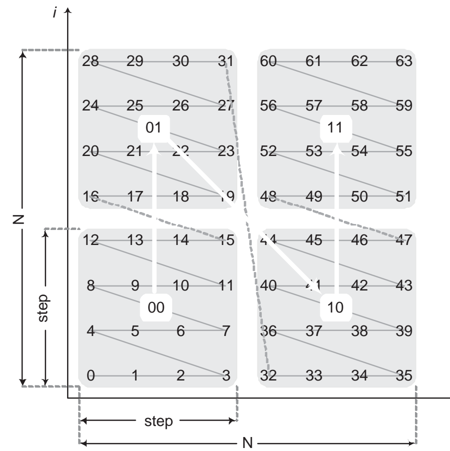

Fig. 6. NZ-transformation

arrays and indexing them through dilated integers. Morton indexing [14] cannot be applied, since it implies a recursive tiling scheme, whereas, in our case, tiling is applied only once (1-level tiling). Thus, instead of *oddBits* and *evenBits*, binary masks are used.

ZZ transformation

row index ↦	00..0	11..1	00..0	11..1
column index ↦	11..1	00..0	11..1	00..0
	$\leftarrow t_i \rightarrow$	$\leftarrow t_j \rightarrow$	$\leftarrow m_i \rightarrow$	$\leftarrow m_j \rightarrow$

NZ transformation

11..1	00..0	00..0	11..1
00..0	11..1	11..1	00..0
$\leftarrow t_j \rightarrow$	$\leftarrow t_i \rightarrow$	$\leftarrow m_i \rightarrow$	$\leftarrow m_j \rightarrow$

NN transformation

row index ↦	11.1	00.0	11..1	00..0
column index ↦	00..0	11..1	00..0	11..1
	$\leftarrow t_j \rightarrow$	$\leftarrow t_i \rightarrow$	$\leftarrow m_j \rightarrow$	$\leftarrow m_i \rightarrow$

ZN transformation

00..0	11..1	11..1	00..0
11..1	00..0	00..0	11..1
$\leftarrow t_i \rightarrow$	$\leftarrow t_j \rightarrow$	$\leftarrow m_j \rightarrow$	$\leftarrow m_i \rightarrow$

most significant set(mss) ↩ ↪ least significant set(lss) mss↩ ↪lss
depends on the shift ↩ ↪ depends on
from tile to tile inner of tiles

We consider an array $A[i, j]$ of size $N_i \times N_j$ and tile size $step_i \times step_j$. The number of subsequent 0 and 1 that consist every part of the masks is defined by the functions $m_x = log(step_x)$ and $t_x = log\left(\frac{N_x}{step_x}\right)$, where $step_x$ is selected to be a power of 2. If N is not a power of 2, we round up by allocating the just larger array with $N = 2^n$ and padding the empty elements. The padding does not aggravate the execution time, since padding elements are never scanned. Row-wise scanning (Z-like order), requires

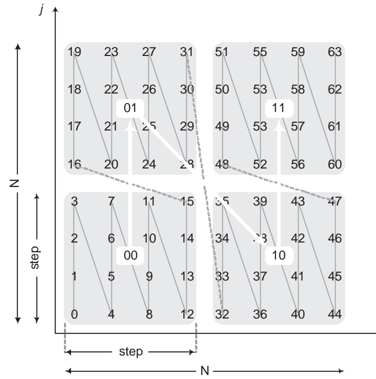

Fig. 7. NN-transformation

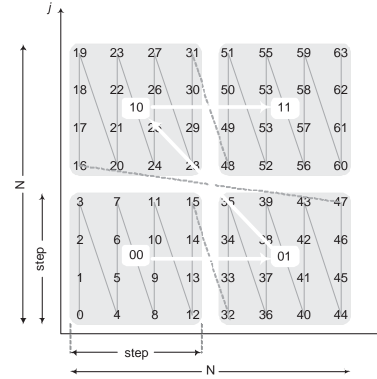

Fig. 8. ZN-transformation

for a mask of the form 0..01..1 for row indices, while the one for columns is 1..10..0. The opposite stands for the case of column-wise scanning (N-like order).

Likewise in quad-tree indexing, every array element $A[i,j]$ can be found in the one-dimensional array in position $[i_m + j_m] = [i_m | j_m]$, where i_m, j_m are the masked values of i, j. In our example, with indices i and j to control columns and rows respectively (the array size is 8×8 and $step = 4$), the masked values are shown in the table below. These values are equal to the storage position of the first column and the first row elements (figure 5).

column/row :	0	1	2	3	4	5	6	7
i :	0	4	8	12	32	36	40	44
j :	0	1	2	3	16	17	18	19

Requesting the element of the 2nd row and 3rd column, namely $A[2,3]$, the index values in the one-dimensional array would be i=8, j=3. The desired element is: $A[i + j] = A[8+3] = A[11]$.

Filtering values 0-7 (the values given to indices when array size is 8×8, according to the way compilers handle arrays so far), through the appropriate binary masks, the desired values will arise (figure 9). In our example mask 010011 should be used for the row index j and 101100 for the column index i.

4.5 Example: Matrix Multiplication

According to Kandemir et al in [7], the best data locality is achieved when the code has the form of figure 1. We use a modified version of [7] with a nested loop of depth 6, instead of depth 5, since the implementation of the mask theory is simpler in this case (explained in section 2). All three arrays A, B, C are scanned according to ZZ-transformation.

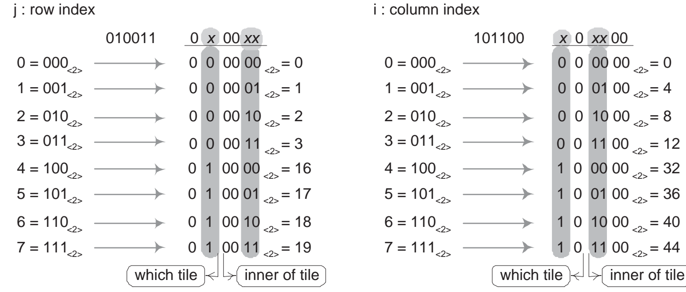

Fig. 9. Conversion of the linear values of row and column indices to dilated ones through the use of masks

In the nested code of our example, the three inner loops (i, k, j) control the sweeping within the tiles. The 4 least significant digits of the masks are adequate to sweep all iterations within these loops (figure 9). The three outer loops (ii, kk, jj) control the shifting from tile to tile, and the 2 most significant digits of the masks can define the moving from a tile to its neighboring one. Thus, i takes values in the range of $xx00$ where $x = 0$ or 1 and ii takes values in the range of $x00000$, so $i|ii \in x0xx00$ gives the desired values for the columns of matrix A and C. Indices j, jj are handled similarly. Index k, does not control the same kind of dimension in the arrays in which it is involved. For array A, the proper mask is a row one, namely $kA \in 00xx$ and $kkA \in 0x0000$. On the other hand, for array B the mask is a column one. Thus, $kB \in xx00$ and $kkB \in x00000$. A useful notice is that $kB = kA << logstep$ and $kkB = kkA << log\left(\frac{N}{step}\right)$.

For our example,

$$ibound{=}(7 = 111_{<2>} \text{ masked through } 101100){=}101100_{<2>} = 44,$$
$$iiincrement = 100000_{<2>} = 32 = 4 \times 4 << \frac{N}{step},$$
$$iincrement = 100_{<2>} = 4 = step$$

and $ireturn = min\{ibound, ii|1100_{<2>}\}$.

```
for(ii=0; ii < ibound; ii+=iiincrement) {
     itilebound=(ii|imask)+1;
     ireturn=(ibound < itilebound?ibound : itilebound);
     for(kk=0; kk < jkbound; kk+=jjkkincrement) {
          ktilebound=(kk|jkmask)+1;
          kreturn=(jkbound < ktilebound?jkbound : ktilebound);
          kkB=kk << logNxy;
          for(jj=0; jj < jkbound; jj+=jjkkincrement) {
               jtilebound=(jj|jkmask)+1;
               jreturn=(jkbound < jtilebound?jkbound : jtilebound);
               for(i=ii; i < ireturn; i+=iincrement)
                    for(k=kk; k < kreturn; k+ = jkincrement) {
                         kB=(k & (step_1))<< logstep;
                         ktB=kkB|kB;
                         for(j=jj; j < jreturn; j+=jkincrement)
                              C[i|j]+=A[i|k] * B[ktB|j];
                    }
          }
     }
}
```

4.6 The Algorithm

In order to succeed in finding the best possible transformation which maximizes performance, we propose the following steps, based on the algorithm presented in [7], adjusted to allow for blocked array layouts:

- Create a matrix R of size $r \times l$, where $r = number\ of\ different\ array\ references$ and $l = nested\ loop\ depth$ (before tiling is applied). If there are two identical references to an array, no extra line is needed. We just make an indication to this array for double optimization priority. The LHS array (in our example this is array C) is also important, because in every access the referred element is both read and written. For the matrix multiplication example:

$$R = \begin{array}{c} \begin{matrix} i & j & k \end{matrix} \\ \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \end{array} \begin{matrix} \text{C**} \\ \text{A} \\ \text{B} \end{matrix}$$

  Each column of R represents one loop index. So, in each row, array elements are set when the corresponding loop controls one of its dimensions. Otherwise array elements are reset.
- Optimize first the layout of the array with the greatest priority. (In our example this is array C). The applied loop transformations should bring one of the indices that control an array dimension in the innermost position of the nested loop. Thus, the C row of R should take the form $(x, \ldots, x, 1)$, where $x = 0$ or 1. To achieve this, swapping of columns can be involved. The chosen index has to be the only element of the stated dimension and should not appear in any other dimension of C. Thus, the reference to array C be: $C[*, \ldots, *, i_{in}, *, \ldots, *]$, where i_{in} is the innermost index of the transformed code and controls the x-th dimension of C, while * indicates terms independent of i_{in}.
  After that, in order to exploit spatial locality for this reference, array C should be stored in memory such that the x-th dimension is the fastest changing dimension. Notice that all possible values for x should be checked.
- Then, fix the remaining references to arrays by priority order. The target is to bring as many of the R-rows as possible in the form $(x, \ldots, x, 1)$. So, if an array index, lets consider this is the one that controls the y-th dimension of array A, is identical to i_{in}, namely the form of the reference to array A is $A[*, \ldots, *, i_{in}, *, \ldots, *]$, then store A such that its fastest changing dimension is y-th.
  If there is no such dimension for A, then we should try to transform the R-row of A to the form $(x, \ldots, x, 1, 0)$. So, the spatial locality along i_{in-1} is exploited. If no such transformation is possible, the transformed loop index i_{in-2} is tried and so on. If all loop indices are tried unsuccessfully, then the order of loop indices is set arbitrarily, taking into account the data dependencies.
- When the nested loop has been completely reordered, we apply tiling. In this stage, the complete layout type is defined. For all array references, the dimension which was defined to be the fastest changing should remain the same for the tiled version of the program code as far as the storage within each tile is concerned. This means

that, for two-dimensional arrays, if the reference is $C[*, i_{in}]$ (so, the storing order inside the tile is row-major), the xZ-order should be used, namely ZZ- or NZ-order. If the reference is $C[i_{in}, *]$ (so, the storing order inside the tile should be column major), the xN-order should be used, namely ZN- or NN-order. The shifting from tile to tile, and therefore the first letter of the applied transformation, is defined by the kind of tiling we will choose. For the two dimensional example, when no tiling is applied to the dimension marked as * then we will have NZ or ZN, respectively, transformation. Otherwise, if for a nested loop of depth n: $(i_1, i_2, \ldots, i_n)$ the tiled form is: $(ii_1, ii_2, \ldots, ii_n, i_1, i_2, \ldots, i_n)$ (where ii_x is the index that controls the shifting from one tile to the other of dimension i_x), then the layout should be ZZ or NN respectively. In most cases, applying tiling in all dimensions brings uniformity in the tile shapes and sizes that arrays are split and as a result, fewer computations for finding the position of desired elements are needed. The size of tiles depends on the capacity of the cache level we want to exploit.

5 Experimental Results

5.1 Execution Environment

In this section we present experimental results using matrix multiplication and LU-decomposition as benchmarks. There are two kinds of experiments: actual execution times of optimized codes using non-linear layouts and simulations using the SimpleScalar toolkit [9]. Firstly, we execute the programs on a 4-processor UltraSPARC II 450 machine, with CPUs at 400MHz. Each processor has a direct mapped 16Kbyte L1 cache with 32bytes cache line size and 8 cycles L1 miss latency, and a 4-way set associative L2 external cache of 4Mbyte with 64bytes cache line size and 84 cycles L2 miss latency, and a 32-entry data TLB with 8Kbyte pagesize. We used the cc compiler, first without any optimization flags (cc -xO0) and then using the highest optimization level (-fast -xtarget=ultra2). The -fast optimization level includes memory alignment, loop unrolling, software pipeline and other floating point optimizations. The experiments were executed for various array dimensions (N) ranging from 16 to 2048 elements, and tile sizes (*step*) ranging from 16 to N elements.

Measured times concern only nested loop-code execution, since we assume that a single layout for all instances of each specific array has already been selected, that optimizes all other possible references for this array, that is, runtime layout transformation from linear to blocked ones is not required.

5.2 Time Measurements

For matrix multiplication benchmark we implemented 5 different codes: **B**locked **a**rray **L**ayou**t**s using **M**asks for address computation (MBaLt), L_{MO} (Lmo), L_{4D} (L4d) and the Kandemir's method [7] using both 2-dimensional arrays (kand2D) and 1-dimensional arrays (kand1D). Using -xO0 optimization level, the best performance was obtained when *step*=512. This size allows data to fit in L2 cache size: 1 tile of each array contains 512×512 elements and takes up $512 \times 512 \times 4$ bytes. Since matrix multiplication uses 3 arrays (A,B,C), the sum is $3 \times 512 \times 512 \times 4$ bytes = 3,145,728bytes

$<$ 4Mbytes. Although the minimum execution time would be expected for a tile size 32×32, corresponding to the size of elements that fit in L1 cache, this is not the case, since the miss penalty for the L1 cache is relatively small (8 cycles) and can usually be outweighed by the increase on the L2 cache hits.

Consequently, for tile size 512×512 we compared our method (MBaLt) with the ones proposed in the literature [7]. In figure 10 we notice that the execution time of our method is almost 25% less than the one achieved using the optimal code from Kandemir et al for 2-dimensional arrays (kand2D). In fact, since current compilers do not support non-linear (blocked) array layouts and, as a result, in the implementation of our methods we use one-dimensional arrays, the comparison should be done with one-dimensional arrays (kand1D). In this case, MBaLt over-exceeds by 60%. Chatterjee's implementation performs worse than kand2D, even though the $L4D$ array layout is the same as MBaLt, because four-dimensional arrays have a much more complicated memory location scheme.

With the -fast optimization (figure 11) MBaLt stil gives the best performance and additionally its curve increases smoothly when array sizes increase. In this case the best tile size is either 32×32 or 16×16, that is L1 cache locality is exploited.

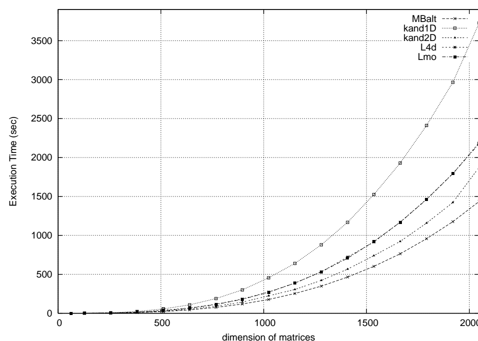

Fig. 10. Total execution results in matrix multiplication with -xO0 option

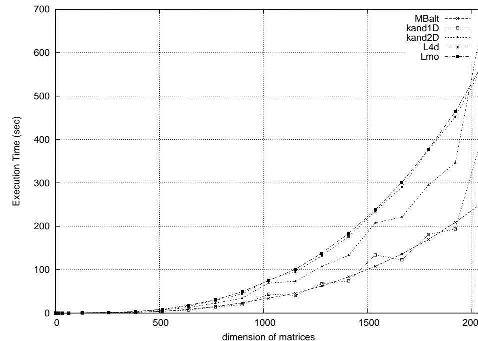

Fig. 11. Total execution results in matrix multiplication with -fast optimization

Although the MBalt code is larger in terms of instruction lines, it takes less to execute, since boolean operations are used for address computation. Furthermore, array padding does not affect

the performance, since execution time increase regularly analogously to the actual array sizes. The reader can verify that no sharp peaks occur in the execution time plots. Delving into the assembly code, one-dimensional arrays are more efficiently implemented, than greater dimension ones, since they require for fewer memory references to find the array elements. On the other hand, finding the storage location of an array element in non-linear layouts needs a lot of computation. Consequently, what we achieved by the proposed implementation is handling 1-dimensional arrays without posing any additional burden due to address computation, namely 1-dimensional arrays with low address computation cost.

The above results are also verified by the LU-decomposition

benchmark. The use of blocked array layouts in combination with our efficient indexing not only provides an average of 15% reduction in time measurements (when no optimization is applied), but also smooths the sharp peaks that come up due to conflict

misses in the simple tiled code. The reduction is even better (can reach even 30%) with -fast flag. Figures 12 and 13 illustrate the execution times of the simple non-tiled code, the tiled one (when tiling is applied in all 3 dimensions, that is the nested code has depth 6), the tiled code with 5-depth nested (tiled2D), the blocked layout with use of masked indexing (6-depth nested loop - masked) and the blocked layout with use of masked indexing (5-depth nested loop, which is 2-dimensional tiling - masked2D). The best tile size is either 256×256 or 16×16 which include data that fit in L2 and L1 cache respectively.

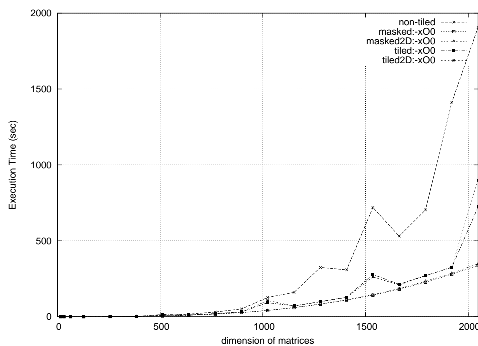

Fig. 12. Total execution results in LU-decomposition with -xO0 option

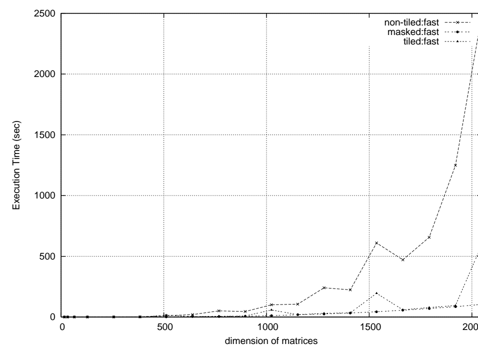

Fig. 13. Total execution results in LU-decomposition with -fast optimization

5.3 Simulation Results

In order to validate the results of time measurements, we applied the three program codes of matrix-multiplication (MBaLt, kand2D, kand1D) and the ones of LU-decomposition (non-tiled, tiled, masked) to the SimpleScalar 2.0 toolkit for various values of N ranging from 16 to 1024 and *step* from 8 to N. We set the size of L1 and L2 caches in accordance to the corresponding sizes of the machine used for our real experiments and measured the data L1 (dl1), unified L2 (ul2) cache and data TLB (dtlb) misses and accesses. The results show that for tile size 32×32 dl1 misses are reduced because data fit in the L1 cache. For the same reason, for $step = 1024$ misses in ul2 cache increase sharply, compared to ul2 misses for $step = 512$. The minimum value fluctuates with the problem size, because it depends on factors such as conflict misses which can not be easily foreseen. In figures 14 and 15 we see the number of misses for different array sizes. The scale in vertical axis, in all plots, is a logarithmic one.

Finally, we observed that among all the tile sizes used in matrix multiplication benchmark, in MBaLt the minimum TLB misses occur for *step* near to 256. This is in accordance with the minimization of L1 and L2 cache misses. Thus, the reduction of the total time performance is reinforced. On the contrary, in kand1D and kand2D the best *step* size is 128 and 32 respectively while for other values of *step*, misses increase rapidly.

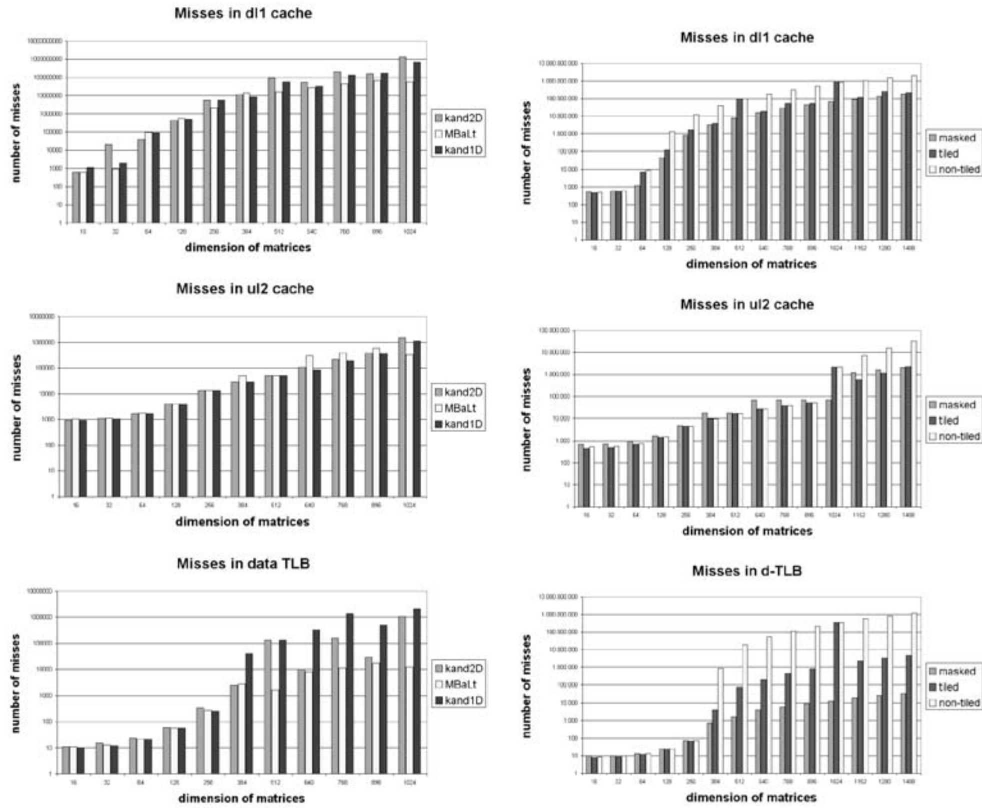

Fig. 14. Number of misses in Data L1 , unified L2 cache and data TLB for matrix multiplication

Fig. 15. Number of misses in Data L1 , unified L2 cache and data TLB for LU-decomposition

6 Conclusion

Low locality of references, thus poor performance in algorithms which contain multidimensional arrays, is due to incompatibility of canonical array layouts with the pattern of memory accesses from tiled codes. In this paper, we have examined the effectiveness of blocked array layouts in two tiled codes and have provided with an addressing

scheme that uses simple binary masks, which are based on the algebra of dilated integers, accomplished at low cost. Both experimental and simulation results illustrate the efficiency of the proposed address computation methods.

References

1. Siddhartha Chatterjee, Vibhor V. Jain, Alvin R. Lebeck, Shyam Mundhra, and Mithuna Thottethodi. Nonlinear Array Layouts for Hierarchical Memory Systems. In *Proceedings 13th ACM International Conference on Supercomputing (ICS)*, Rhodes, Greece, 1999.
2. Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala, and Mithuna Thottethodi. Recursive Array Layouts and Fast Parallel Matrix Multiplication. In *Proceedings 11th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Saint Malo, France, 1999.
3. Michael Cierniak and Wei Li. Unifying Data and Control Transformations for Distributed Shared-Memory Machines. In *Proceedings ACM SIGPLAN Conference*, La Jolla, CA, 1995.
4. Marta Jimenez. *Multilevel Tiling for Non-Rectangular Iteration Spaces.* PhD thesis, Universitat Politecnica de Catalunya, 1999.
5. M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A Linear Algebra Framework for Automatic Determination of Optimal Data Layouts. *IEEE Transactions on Parallel and Distributed Systems*, 10(2), 1999.
6. M. Kandemir, J. Ramanujam, A. Choudhary, and P. Banerjee. A Layout-conscious Iteration Space Transformation Technique. *IEEE Transactions on Computers*, 50(12):1321–1336, 2001.
7. M. Kandemir, J. Ramanujam, and Alok Choudhary. Improving Cache Locality by a Combinaion of Loop and Data Transformations. *IEEE Transactions on Computers*, 48(2):159–167, 1999.
8. Induprakas Kodukula, Keshav Pingali, Robert Cox, and Dror Maydan. An Experimental Evaluation of Tiling and Shackling for Memory Hierarchy Management. In *Proceedings 13th ACM International Conference on Supercomputing (ICS)*, Rhodes, Greece, 1999.
9. Alvin R. Lebeck and David A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE Computer*, 27(10):15–26, 1994.
10. Neungsoo Park, Bo Hong, and Viktor Prasanna. Analysis of Memory Hierarchy Performance of Block Data Layout. In *Proceedings International Conference on Parallel Processing (ICPP)*, Vancouver, Canada, 2002.
11. D. Patterson and J.Hennessy. *Computer Architecture. A Quantitative Approach*, pages 373–504. Morgan Kaufmann, San Francisco, CA, 3rd edition, 2002.
12. David S. Wise. Ahnentafel Indexing into Morton-ordered Arrays, or Matrix Locality for Free. In *Proceedings 1st Euro-Par International Workshop on Cluster Computing*, pages 774–783, Munich, Germany, 2001.
13. David S. Wise, Gregory A. Alexander, Jeremy D. Frens, and Yuhong Gu. Language Support for Morton-order Matrices. In *Proceedings ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, Utah, UT, 2001.
14. David S. Wise and Jeremy D. Frens. Morton-order Matrices Deserve Compilers' Support. TR533, Computer Science Department, Indiana Univ., 1999.
15. Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Canada, 1991.