Fast Indexing for Blocked Array Layouts to Reduce Cache Misses

Evangelia Athanasaki and Nectarios Koziris National Technical University of Athens School of Electrical and Computer Engineering Computing Systems Laboratory e-mail: {valia, nkoziris}@cslab.ece.ntua.gr

Abstract—The increasing disparity between memory latency and processor speed is a critical bottleneck in achieving high performance. Recently, several studies have been conducted on blocked data layouts, in conjunction with loop tiling to improve locality of references. In this paper, we further reduce cache misses, restructuring the memory layout of multi-dimensional arrays, so that array elements are stored in a blocked way, exactly as they are swept by the tiled instruction stream. A straightforward way is presented to easily translate multi-dimensional indexing of arrays into their blocked memory layout using quick and simple binary-mask operations. Actual experimental results on three hardware platforms, using 5 different benchmarks with various array sizes, illustrate that execution time is greatly improved when combining tiled code with blocked array layouts and binary, mask-based translation functions for fast indexing. Finally, simulations verify that our enhanced performance is due to the considerable reduction of cache misses in all levels of memory hierarchy, and especially due to their concurrent minimization, for the same tile size.

keywords: Cache Locality, Loop Tiling, Blocked Array Layouts, Fast Indexing, Code Optimization.

I. INTRODUCTION

As microprocessors become increasingly fast, memory system performance begins to determine overall performance [1], since a large percentage of execution time is spent on memory stalls, even with large on-chip caches. Computer architects have been battling against this memory wall problem [2] by designing even larger but slightly faster caches.

In order to overcome the disparity between cpu and cache memory speeds, software techniques that exploit locality of references for iterative codes are widely used. These techniques can be categorized into control (code) transformations, data transformations, combination of control and data transformations, and, lately, transformations of the array layouts in memory (non-linear, hybrid linear and non-linear).

As far as control transformations are considered, they change loop iteration order and, thereby, data access order. Loop permutation, loop reversal and loop skewing attempt to restructure the control flow of the program to improve data locality. Loop unrolling and software pipelining are exploiting multiple registers and pipelined datapaths to improve temporal locality [3]. Such transformations, when used in combination with software-controlled prefetching [4], [5], [6], [7], help tolerating memory latency, as long as prefetching happens soon enough, but not too early so that prefetched data are

evicted prior to their use. Loop fusion and loop distribution can indirectly improve reuse by enabling control transformations that were previously not legal [8]. Loop tiling, widely used in codes that include dense multidimensional arrays, operates on submatrices (tiles), instead of entire rows or columns of arrays, so that data loaded into the faster levels of memory hierarchy are reused. In this context, tile size selection [9], [10] plays a critical role, since an efficient execution requires elimination of capacity and interference (both self and cross interference) cache misses, along with avoiding underutilization of the cache.

Locality analysis [11] allows compilers to identify and quantify reuse and locality, and is used to guide the search for the best sequence of transformations. Unimodular control transformations, described in the most cited work of Wolf and Lam [12], and compound transformations of McKinley et al in [8], attempt to find the best combination of control transformations which, when used with tiling, ensure the correct computation order, while increasing locality of accesses to cache memory.

In a data-centric approach [13], array unification maps multiple arrays into a single array data space, in an interleaved way, grouping together arrays accessed in successive iterations, to reduce the number of conflict misses. Copying [14] is a method for reducing the intra-tile and inter-tile interference in the cache. However, it can not be used for free, as it incurs instruction overhead and additional cache misses, while data is copied from arrays to buffers. Padding [15], [16] is a data alignment technique that involves the insertion of dummy elements in a data structure for improving cache performance by eliminating severe conflict misses.

Nevertheless, increasing the locality of references for a group of arrays may affect the number of cache hits for other referenced arrays. Combined loop and data transformations were proposed in [17] and [18]. Cierniak and Li in [17] present a cache locality optimization algorithm which combines both loop (control) and linear array layout transformations. Another unified, systematic method, presented by Kandemir et al in [19], [20], [21], [18], aims at utilizing spatial reuse in order to obtain good locality.

The previous approaches assumed linear array layouts. Programming languages provide with multidimensional arrays which are finally stored in a linear memory layout, either column-wise or row-wise (canonical order). However, such linear array memory layouts can produce unfavorable memory access patterns, that cause interference misses and increase memory system overhead. Wise et al in [22] and [23] used quad-tree layouts in combination with level-order, either Ahnentafel or Morton, indexing. Quad-tree layouts seem to work well with recursive algorithms due to their efficient element indexing. Nevertheless, no locality gain can be obtained when quad-tree layouts are applied to non-recursive codes. According to Lam et al in [24], quad-tree (recursive) layouts need not to reach at the finest (element) level, since, if a tile is small enough to totally fit in cache, it can be organized in a canonical order, without any additional depth of recursion for its layout. In this way, the quad-tree decomposition can be pruned, well before reaching the element level and coexist with tiles organized in a canonical manner, thus creating hybrid layouts. Although quad-tree layouts are not efficient, their indexing is fast enough to be adopted for non-recursive layouts.

Hybrid quad-tree and canonical layouts were explored by Chatterjee et al in [25] and [26]. In these papers, the implementation cost was quantified in terms of execution time. Although they claimed for increasing execution-time performance, using four-dimensional arrays, any gain obtained by data locality due to blocked layouts seems to be counterbalanced by the slowdown caused when referring to four-dimensional array elements (in comparison to access time using two- or onedimensional arrays). Lin et al in [27] proposed that these fourdimensional arrays should be converted to two-dimensional ones, mapping array elements through Karnaugh representation scheme. However, indexing the right array elements still requires for tedious time calculations.

In order to quantify the benefits of adopting nonlinear layouts to reduce cache misses, there exist several different approaches. In [28], Tseng considers all levels of memory hierarchy to reduce L2 cache misses as well, rather than reducing only L1 ones. He presents even fewer overall misses, however performance improvements are rarely significant. This is because L2 misses are not as many as L1 ones, although much more cycles are lost per L2 miss. Targeting only the L1 cache, nearly all locality benefits can be achieved. In another approach, TLB misses should also be considered, along with L1 & L2 misses. As problem sizes become larger, TLB thrashing may occur ([29]), so that the overall performance can be seriously degraded. In this context, TLB and cache misses should be considered in concert. Park et al in [30] analyze the TLB and cache performance for standard matrix access patterns, when tiling is used together with block data layouts. Such layouts with block size equal to the page size, seem to minimize the number of TLB misses. Considering both all levels of cache (L1 and L2) and TLB, a block size selection algorithm calculates a range of optimal block sizes.

As related work has shown, the automatic application of nonlinear layouts in real compilers is a really time tedious task. It does not suffice to identify the optimal layout either blocked or canonical one for each specific array. For blocked layouts, we also need an automatic and quick way to generate the mapping from the multidimensional iteration indices to the correct location of the respective data element in the linear memory. Blocked layouts are very promising, subject to an efficient address computation method. Any method of fast indexing for non-linear layouts will allow compilers to introduce such layouts along with row or column-wise ones, therefore further reducing memory misses.

In this paper, in order to facilitate the automatic generation of tiled code that accesses blocked array layouts, we propose a very quick and simple address calculation method of the array indices. We can adopt any out of four different proposed types of blocked layouts, and apply a dilated integer indexing, similar to Morton-order arrays. Thus, we combine additional data locality due to blocked layouts, with fast access per any array element, since simple boolean operations are used to find its right location in linear physical memory. Since array data are now stored block-wise, we provide the instruction stream with a straightforward indexing to access the correct elements. Our method is very effective at reducing cache misses, since the deployment of the array data in memory follows the exact order of accesses by the tiled instruction code, achieved at no extra runtime cost. Experimental results were conducted using the Matrix Multiplication, LU-decomposition, SSYR2K (Symmetric Rank 2k Update), SSYMM (Symmetric Matrix-Matrix Operation) and STRMM (Product of Triangular and Square Matrix) codes from BLAS3 routines. We ran two types of experiments, actual execution of real codes and simulation of memory and cache usage using SimpleScalar. We compare our method with the ones proposed by Kandemir in [18] and by Cierniak in [17], that propose control tiling and arbitrary but linear layouts, and show how overall misses are further reduced and thus final code performs faster. Comparing with Chatterjee's implementation, which uses non-linear fourdimensional array layouts in combination with control transformations, we prove that limiting cache misses is not enough, if address computation for these layouts is not efficient.

The remainder 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 and our 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.

II. THE PROBLEM: IMPROVING CACHE LOCALITY FOR ARRAY COMPUTATIONS

In this section, we elaborate on the necessity for both control (loop) and data transformations, to fully exploit data locality. We present, stepwise, all optimization phases to improve locality of references with the aid of the typical matrix multiplication kernel.

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. Thus, useful data are not evicted

	<pre></pre> <pre><<pre><<pre><<pre><<pre><<pre><<pre><<pre><<pre>i<</pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre><pre></pre></pre></pre></pre></pre></pre></pre></pre></pre>
for (j=0; j <n; (i="0;" fo<="" for="" i++)="" i<n;="" j++)="" th=""><th></th></n;>	
for (<i>kk</i> =0; <i>kk</i> < <i>N</i> ; <i>kk</i> += <i>step</i>)	(jj=0; jj < N; jj + step) or $(i=0; i < N; i++)$

Fig. 1. Matrix multiplication

from the register file or the cache before being reused. The tile size (*step*) should be selected accordingly, to allow reused data to fit in the specific memory hierarchy level (i.e. L1, L2, etc), that was selected to be optimized.

Unified Loop and linear Data Transformations: Since, loop transformations alone can not result in the best possible data locality, a unified approach that utilizes 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 [18] (since self-temporal reuse can be considered as a subcase of selfspatial, while group spatial reuses 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 the LHS (Left Hand Side) array, namely array C, because, in every iteration the elements of this array are both read and written, while the elements of arrays A and B are only read. 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 exploit self-temporal reuse in the second innermost loop for array C. Thus, A[i, k] should also be stored by rows (Ar).

Unified Loop and non-linear Data Transformations: In order to evaluate the merit of non-linear data transformations, let us consider the code shown in figure 1d. We assume that the elements of all three arrays are stored exactly as they are swept by the program (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. This simplifies the computations needed to find the location for each array element.

III. MORTON ORDER (RECURSIVE) ARRAY LAYOUTS

In the sequel, in order to capture the notion of Morton order [31], i.e. a recursive storage order of arrays, the basic elements

of the dilated integer algebra are presented. Morton defined a way to index two-dimensional arrays and pointed out the conversion to and from cartesian indexing available through bit interleaving (figure 2).

The following definitions are used only for two-dimensional arrays but can be easily generalized for *d*-dimensional ones. (A *d*-dimensional array is represented as a 2^d -ary tree).

Definition 3.1: The integer $\overrightarrow{b} = \sum_{k=0}^{w-1} 4^k$ is the constant 0x55555555 and is called *evenBits*. Similarly, $\overleftarrow{b} = 2\overrightarrow{b}$ is the constant 0xaaaaaaaa and is called *oddBits*.

The hexadecimal evenBits = 0x55555555 has all even bits set and all odd bits cleared. Likewise, oddBits = evenBits << 1 has the value 0xaaaaaaaaa.

Definition 3.2: 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 \vec{j} . The odd-dilated representation of $i = \sum_{k=0}^{w-1} i_k 2^k$ is $2 \vec{i}$ and is denoted \vec{i} .

Theorem 3.1: The Morton index for the $\langle i, j \rangle^{th}$ element of a matrix is $\langle i \vee j \rangle$, or $\langle i + j \rangle$.

Thus, the loop with range

for
$$(i=0; i < N; i++)$$
 ...

will be modified to

for $(i_m = 0; i_m < N_m; i_m = (i_m - evenBits)\&evenBits) \dots$

where $i_m = \overrightarrow{i}$ and $N_m = \overrightarrow{N}$

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.

When using a dilated integer representation for the indexing of an array, if an index is only used as a column index, then its dilation is odd. Otherwise it is even-dilated. If used in both roles, then, doubling its even-dilated value, produces the odddilation, 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:

 $\begin{array}{ll} \texttt{#define } evenIncrement(i)(i = ((i - evenBits)\&evenBits)) \\ \texttt{#define } oddIncrement(i)(i = ((i - oddBits)\&oddBits)) \\ \texttt{for } (i=0; \ i < colsOdd; \ oddIncrement(i)) \\ \texttt{for } (j=0; \ j < rowsEven; \ evenIncrement(j)) \\ \texttt{for } (k=0; \ k < rowsAEven; \ evenIncrement(k)) \\ C[i+j]+=A[i+k]*B[2*k+j]; \end{array}$

where rowsEven, colsOdd and rowsAEven are the bounds of arrays when transformed in dilated integers. Notice that k is used both as column and as row index. Therefore, it is translated to \vec{k} and for the column indexing of array B, 2 * k is used, which represents \vec{k} .

IV. BLOCKED ARRAY LAYOUTS

In this section we introduce the non-linear, blocked array layout transformation. Since the performance of the proposed methods in [17], [18], [12] is better when tiling is applied, perhaps we could achieve even better locality, if also, array data are stored neither column-wise nor row-wise, but follow a blocked layout. Such layouts were used by Chatterjee et al in [25], [26], in order to obtain better interaction with cache

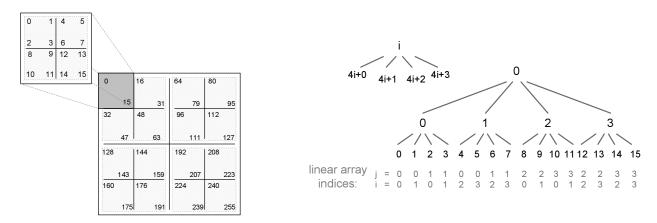


Fig. 2. Morton-order layout of a 4×4 matrix, and Morton indexing of the order-4 quadtree

memories. According to [25], 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) . 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. However, referring to 4-dimensional arrays comes up with long assembly codes, thus, repetitive load and add instructions, which, as seen in experimental results, are too time consuming and, thus, degrade total performance.

A. Different Proposed Types of Blocked Layouts

We expand the notion of blocked layouts, storing array elements exactly in the same order as they are accessed when loop iterations are executed. This storage layout is presented in figure 3 for an 8×8 array which is split into tiles of size 4×4 . The grey line shows the order by which the successive program iterations sweep the array data, while numbering illustrates the order, data are actually 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 this example as "ZZ", because the inner of the tiles is scanned row-wise (in a Z-like order) and the shift from one tile to another is Zlike as well. The first letter (Z) of transformation denotes the shift from one tile to another, while the second indicates the sweeping within a tile. The sweeping of array data in ZZ-order is done using the first portion of code shown below:

"ZZ sweeping" for
$$(ii=0; ii < N; ii+=step)$$

for $(jj=0; jj < N; jj==step)$
for $(i=ii; (i < ii + step \&\& i < N); i++)$
for $(j = jj; (j < jj + step \&\& j < N); j++)$
 $A[i, j] = \dots;$

In a similar way, the other three types of transformations are shown in figures 4, 5 and 6. For example, according to the "NN" transformation, both the sweeping of the array data within a tile and the shift from one tile to another are done columnwise (N-like order). The respective codes are found below.

"NZ sweeping" for (jj=0; jj < N; jj+=step)

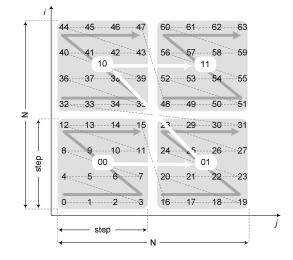


Fig. 3. ZZ-transformation

for
$$(ii=0; ii < N; ii+=step)$$

for $(i=ii; (i < ii + step \&\& i < N); i++)$
for $(j=jj; (j < jj + step \&\& j < N); j++)$
 $A[i,j]=\ldots;$

'NN sweeping" for
$$(ii=0; ii < N; ii+=step)$$

for $(jj=0; jj < N; jj+=step)$
for $(i=ii; (i < ii+step \&\& i < N); i++)$
for $(j=jj; (j < jj+step \&\& j < N); j++)$
 $A[j,i]=\ldots;$

"ZN sweeping" for
$$(jj=0; jj < N; jj+=step)$$

for $(ii=0; ii < N; ii+=step)$
for $(i = ii; (i < ii + step \&\& i < N); i++)$
for $(j = jj; (j < jj+step \&\& j < N); j++)$
 $A[j,i]=\ldots;$

Since compilers support only linear layouts (either columnorder or row-order) and not blocked array layouts, sweeping the array data when stored in one of the four aforementioned ways, can be done using one-dimensional arrays and indexing them through dilated integers. Morton indexing [23] cannot be applied, since it is applicable only when we have a fully recursive tiling scheme, whereas, in our case, tiling is applied only once (1-level tiling). Thus, instead of *oddBits* and *evenBits*, we introduce binary masks.

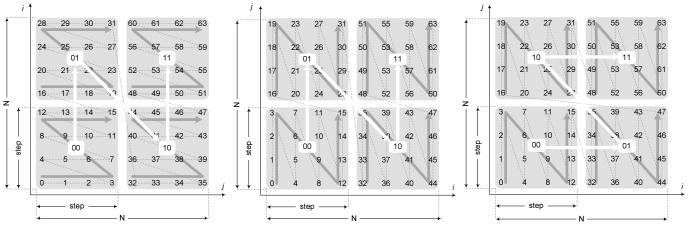


Fig. 4. NZ-transformation

Fig. 5. NN-transformation

Fig. 6. ZN-transformation

B. Mask Theory for Fast Address Computation

The form of the introduced masks, used to calculate the address of an element A[i, j], for an array of size $N_c \times N_r$ and tile size $step_c \times step_r$, is illustrated in table I.

The number of subsequent 0 and 1's that consist every part of each mask is defined by the functions $m_x = log(step_x)$ and $t_x = log\left(\frac{N_x}{step_x}\right)$, where step is 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 with arbitrary values. Such padding does not aggravate the execution time, since padded elements are not scanned by the algorithm. Rowwise scanning (Z-like order), requires for a mask of the form 0-1 for row indices, while the respective one for columns is 1-0. The opposite stands for the case of column element scanning (N-like order).

Similar to quad-tree indexing, every array element A[i, j] can be found in the one-dimensional array in the position $[i_m + j_m] = [i_m | j_m]$, where i_m, j_m are generated by i, j, when appropriately masked.

C. Implementation

Let us store elements of array A[i, j] in an one-dimensional memory space, in the same order as they are swept by the instruction stream of a tiled code, similar to ZZ-sweeping of section IV-A. Such storage order is illustrated in figures 7 and 8, where array A is of size 32×32 , tiled in blocks of size 8×8 .

To access any element of an array stored according to a blocked layout, loop indices that control the element locations should take non-successive values. In our example (ZZ layout), the right values for indices that control columns and rows of the first tile (as shown in figure 8) are:

row in	idex :	0	1	2	3	4	5	6	7
column in	dex :	0	8	16	24	32	40	48	52

These are the storage locations of the first column and the first row elements (figure 8). To find the storage location of an arbitrary element, e.g. A[4, 6] (5-th row and 7-th column), we can just add 6+32=38 (we selected the 5-th column index (=32) and the 7-th row index (=6)). Location #38 is the requested

one. For an element that belongs in the 3-rd row of tiles and in the 4-th column of tiles, which is tile #11(=3+8), as shown in figure 7), the element position will have $1011_{<2>}(=11_{<10>})$ as prefix.

The appropriate binary masks for the automatic calculation of such non-linear indices are given by the forms of section IV-B. For our example (32×32 array with 8×8 tiles), the masks for the row index : 0011000111

are: for the column index : 1100111000

If values 0-31 of linear indices (which are the values given to row and column indices according to standard compiler array handling), are filtered, using these masks, the desired values will arise (figure 9).

As shown in figure 9, element A[20, 30] which belongs to tile $\#11_{<10>}$, is stored in position:

"20"+"30"=10 00 100 000 + 00 11 000 110 = 1011 100 $110 = 742_{<10>}$, which has indeed $1011_{<2>}$ as prefix.

D. Example: Matrix Multiplication

According to Kandemir et al in [18], the best data locality is achieved when the code has the following form:

$$\begin{array}{l} \text{for } (ii=0;\;ii<32;\;ii+=8) \\ \text{for } (kk=0;\;kk<32;\;kk+=8) \\ \text{for } (jj=0;\;jj<32;\;jj+=8) \\ \text{for } (i=ii;\;(i$$

We use a modified version of [18] (as explained in section II), with a nested loop of depth 6, instead of depth 5, since the implementation of our mask theory is simpler in this case. All three arrays A, B, C are scanned according to ZZ-sweeping.

In the nested code of our example, the three inner loops (i, j, k) control the sweeping within the tiles. The 6 least significant digits of the masks are adequate to sweep all iterations within these loops, as shown in figures 8 and 9. The three outer loops (ii, kk, jj) control the shifting from one tile to another, and the 4 most significant digits of the masks can define the movement from a tile to its neighboring one.

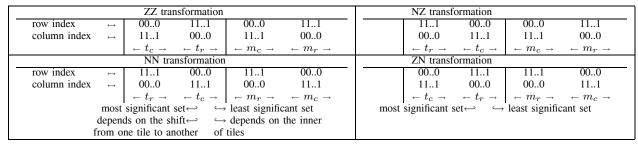


TABLE I

BINARY MASKS FOR INDEXING THE FOUR DIFFERENT TYPES OF BLOCKED LAYOUTS

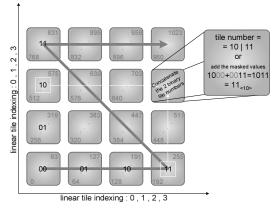
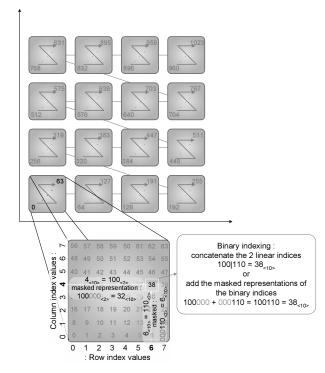
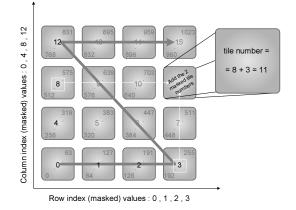


Fig. 7. Indexing Tiles (calculations using binary and decimal indices)





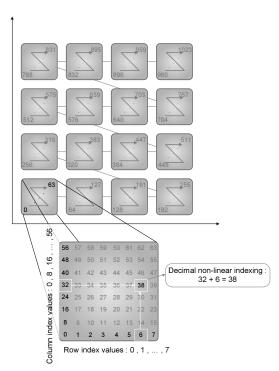


Fig. 8. Indexing the inner of Tiles (calculations using binary and decimal indices)

Thus, *i* takes values in the range of xxx000 where x = (0 or 1) and *ii* takes values in the range of xx00000000, so i|ii = xx00xxx000 gives the desired values for the columns of matrix A and C. Similarly, *j* takes values in the range of 000xxx and *jj* in the range of 00xx000000, so j|jj =

00xx000xxx are the desired values for rows of B and C. Index k, does not control the same dimension in the arrays in which it is involved, as shown in table II. So, for array A, the proper mask is a row one, namely $k_A \in 000xxx$ and $kk_A \in 00xx000000$. On the other hand, the mask for

row index		column index		
linear values		linear values masking		
0011 000111	00 xx 000 xxx	1100111000	xx 00 xxx 000	
0 = 00000 _{<2>} >	00 00 000 000 <2>= 0	0 = 00000 _{<2>} >	00 00 000 000 _{<2>} = 0	
1 = 00001 _{<2>} >	00 00 000 001 _{<2>} = 1	1 = 00001 _{<2>}	00 00 001 000 _{<2>} = 8	
6 = 00110 _{<2>}	00 00 000 110 _{<2>} = 6	4 = 00100 _{<2>} >	00 00 100 000 _{<2>} = 32	
7 = 00111 _{<2>} >	00 00 000 111 <2>= 7			
8 = 01000 _{<2>} >	00 01 000 000 <2>= 64	20 = 10100 _{<2>} >	10 00 100 000 _{<2>} = 544	
30 = 11110 _{<2>}	00 11 000 110 _{<2>} = 198	30 = 11110 _{<2>}	11 00 110 000 <2> = 816	
31 = 11111 _{<2>}	00 11 000 111 <2>= 199	31 = 11111 _{<2>}	11 00 111 000 <2>= 824	
which	n tile + inner of tile	which	n tile + inner of tile	

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

indices :	control	of array(s)	appropriate mask
i, ii:	columns	A & C	column mask (1100111000)
j, jj:	rows	B & C	row mask (0011000111)
k, kk:	columns	B	column mask (for k_B, kk_B)
k,kk:	rows	A	row mask (for k_A, kk_A)

TABLE II INDEXING OF ARRAY DIMENSIONS

array B is a column one. Thus, $k_B \in xxx000$ and $kk_B \in xx00000000$. Note that $k_B = k_A \ll logstep$ and $kk_B = kk_A \ll log \left(\frac{N}{step}\right)$.

In our example,

$$\begin{split} ibound = & \text{column_mask} = 1100111000_{<2>} = 824, \\ iiincrement = & 1000000_{<2>} = 64 = 8 \times 8 << \frac{N}{step}, \\ iincrement = & 1000_{<2>} = 8 = step \end{split}$$

and $ireturn = min\{\text{column}_\text{mask}, ii|111000_{\leq 2>}\}$.

```
for(ii=0; ii < ibound; ii+=iiincrement) {</pre>
    itilebound = (ii|imask) + 1;
    ireturn=(ibound < itilebound?ibound : itilebound);
    for (kk=0; kk < kjbound; kk+=kkjjincrement) {
         ktilebound = (kk|kjmask) + 1;
         kreturn = (kjbound < ktilebound?kjbound : ktilebound);
         kkB = kk << logNxy;
         for (jj=0; jj < kjbound; jj+=kkjjincrement) {
              jtilebound = (jj|kjmask) + 1;
              jreturn=(kjbound < jtilebound?kjbound : jtilebound);
              for (i=ii; i < ireturn; i+=iincrement)
                  for (k=kk; k < kreturn; k+=kjincrement) {
                       kB \texttt{=} (k \texttt{ \& } (step\_1)) << logstep;
                       ktB=kkB|kB;
                       xA=i|k;
                       for(j=jj; j < jreturn; j+=kjincrement)</pre>
                           C[i|j] + = A[xA] * B[ktB|j];
                  }
         }
    }
}
```

E. The Algorithm to select the Optimal Layout per Array

In order to succeed in finding the best possible transformation which maximizes performance, we present a method, based on the algorithm presented in [18], adjusted to allow for blocked array layouts. Notice that our algorithm finds the best data layout of each array, taking into account all its instances throughout the whole program (instances of the same array in different loop nests are also included). This way, no replicates of an array are needed.

Figure 10 gives a graphic representation of the following algorithm. As easily derived from the three loops of the flow chart, the worst case complexity is $O(A \cdot r \cdot D)$, where A is the number of different array references, r is the number of dimensions each array reference contains and D is the loop depth (before tiling is being applied).

• Create a matrix R of size $r \times l$, where r = numberof different array references and l = nested loop depth (before tiling is applied). Columns of R are ordered from left to right, according to the loop nest order (from the outermost to the innermost loop). If there are two identical references to an array, there is no need for an extra line. We can just add an indication to the row representing this array, thus putting double priority to its optimization. For the example of matrix multiplication

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

Notice that the Left Hand Side (LHS) array of an expression (for our example this is array C) is more important, because in every such access, the referred element is both read and written.

Each column of R represents one of the loop indices. 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 number of references in the loop body. (In our example this is array C). The applied loop transformations should be such that one of the indices that control an array dimension of it, is brought in the innermost position of the nested loop. Thus, the C row of R should come to 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.

8

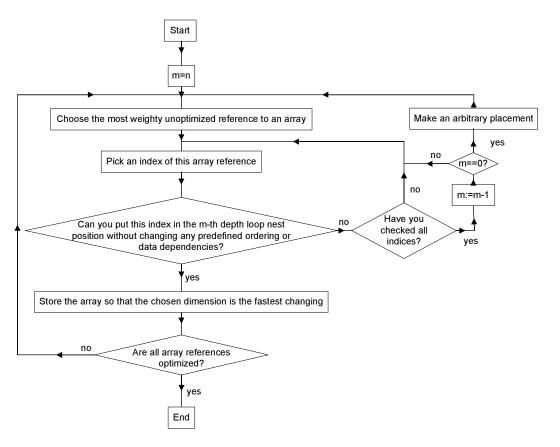


Fig. 10. Flow Chart of the proposed optimization algorithm

Thus, the reference to array C should have the form: $C[*, \ldots, *, i_{in}, *, \ldots, *]$, where i_{in} is the innermost index of the transformed code and controls the y-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 y-th dimension is the fastest changing dimension. Notice that all possible values for y should be checked.

• Then, fix the remaining references to arrays by priority order. Our goal is to bring as many *R*-rows in the form $(x, \ldots, x, 1)$ as possible. If the *z*-th dimension of array *A*, is controlled by index i_{in} only (namely *A* is in the form $A[*, \ldots, *, i_{in}, *, \ldots, *]$), then store *A* such that its fastest changing dimension is the *z*-th one.

If there is no such dimension for A, then we should try to transform the reference so that it is brought to the form $A[*, \ldots, *, f(i_{in-1}), *, \ldots, *]$, where $f(i_{in-1})$ is a function of the second innermost loop i_{in-1} and other indices except i_{in} , and * indicates a term independent of both i_{in-1} and i_{in} . Thus, the *R*-row for *A* would be $(x, \ldots, x, 1, 0)$ and 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.

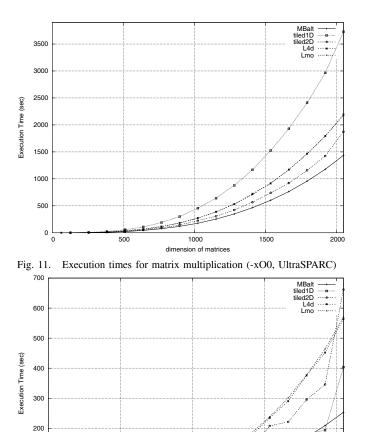
• After a complete loop transformation and a respective memory layout are found, they are stored, and the next

alternative solution is tried. Among all feasible solutions, the one which exploits spatial locality in the innermost loop for the maximum number of array references, is selected.

When the nested loop has been completely reordered, we then apply tiling. In this stage, the complete layout type is defined. For each one of the 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 of the elements within each tile is concerned. This means, that for two-dimensional arrays, if the form of the reference is $C[*, i_{in}]$, the storing order inside the tile should be row-major, thus, we should use the xZ-order (namely ZZ- or NZ-order) blocked array layout. If the form of the reference is $C[i_{in},*]$, the storing order inside the tile should be column major, thus, we should use the xN-order (namely ZN- or NNorder) blocked array layout. The shifting from one tile to another, and therefore the first letter of the applied transformation, is defined by the type of tiling we will choose. For the two dimensional example, if no tiling is applied to the dimension marked as * then we will have a NZ or ZN transformation, respectively. 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.

V. EXPERIMENTAL RESULTS



dime Fig. 12. Execution times for matrix multiplication (-fast, UltraSPARC)

1000

sion of matrices

1500

2000

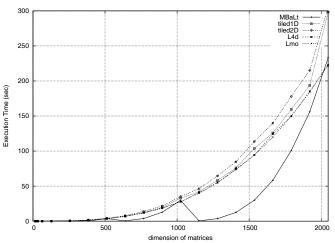
A. Execution Environment

500

100

0

In this section we present experimental results using 5 benchmarks: Matrix Multiplication, LU-decomposition, SSYR2K (Symmetric Rank 2k Udate), SSYMM (Symmetric Matrix-Matrix Operation) and STRMM (Product of Triangular and square Matrix). There are two types of experiments: actual execution times of optimized codes using non-linear layouts and simulations using the SimpleScalar toolkit [32]. Firstly, the experiments were performed on three platforms with different architectural characteristics: a Sun Enterprise 450 machine, an SGI/Cray Origin2000 multiprocessor, and an Athlon XP 2600+ PC. The Sun Enterprise has UltraSPARC II CPUs at 400MHz, each with a 16 KB 2-way set associative on-chip instruction L1 cache, a 16 KB direct-mapped on-chip data L1 cache, with 32 bytes cache line size and 8 cycles L1 miss



Execution times for matrix multiplication (-fast, SGI Origin) Fig. 13.

latency, a direct-mapped L2 external cache of 4 MB, with 64 bytes cache line size and 84 cycles L2 miss latency, and a 64-entry data TLB with 8 KB page size and 51 cycles miss penalty. The SGI Origin has MIPS R10000 processors, each of which operating at 250MHz and has a 32 KB 2-way set associative on-chip instruction L1 cache, with 32 bytes line size. It also has a 32 KB 2-way set associative on-chip data L1 cache, with 64 bytes line size, a 4 MB 2-way set associative unified external cache, with 128 bytes cache line size and a 64entry TLB, with 16 KB page size. The latency of the L1 data cache, L2 cache and TLB is 6, 100 and 57 cycles respectively.

We used the Workshop cc compiler 5.0, first without any optimization flags (cc -xO0), in order to study the clear effect of different data layouts, avoiding any possible interference in the results due to any compiler optimizations. We then used the highest optimization level (-fast -xtarget=native), which includes memory alignment, loop unrolling, software pipelining 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, to show the merit of our cache miss reduction method both on data sets that fit and do not fit in cache.

The Athlon XP CPU operates at 2,083GHz and has an 128KB, 2-way set associative on-chip L1 cache (64KB+64KB for data+instructions) with 64bytes line size and 3 cycles miss penalty. The unified L2 cache is also on-chip, with 64bytes line size, it is 16-way set associative, 256KB in capacity and has 20 cycles miss penalty. There are also two levels of TLBs: Instruction TLB has 24 entries in L1 and 256 entries in L2, while data TLB has 40 entries in L1 (with 3 cycles miss penalty and 4KB page size) and 256 entries in L2. In Athlon XP the gcc (version 3.3.4-13) compiler has been used, firstly without any optimization flags (gcc -O0) and then, at the highest optimization level (-O3).

We implemented 5 different versions per benchmark: our method, that is Blocked array Layouts using our Mask theory for fast address computation (MBaLt), L_{MO} (Lmo) and L_{4D} (L4d) both from Chatterjee [25], and the method [18] by Kandemir using both 2-dimensional arrays (tiled2D) and 1dimensional arrays (tiled1D). Measured execution times do

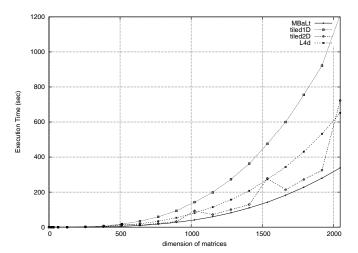


Fig. 14. Execution times for LU-decomposition (-xO0, UltraSPARC)

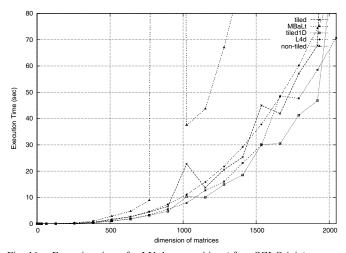


Fig. 16. Execution times for LU-decomposition (-fast, SGI Origin)

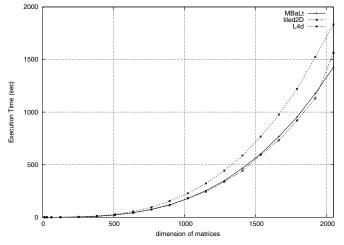


Fig. 18. Execution times for SSYR2K (-xO0, UltraSPARC)

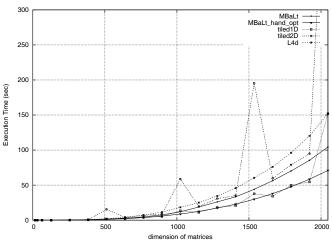


Fig. 15. Execution times for LU-decomposition (-fast, UltraSPARC)

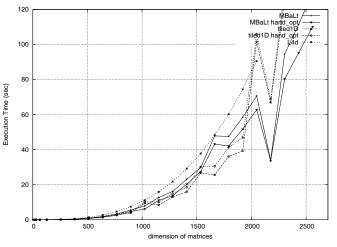


Fig. 17. Execution times for LU-decomposition for larger arrays and hand optimized codes (-fast, SGI Origin)

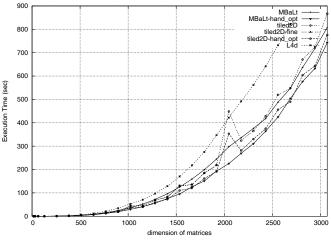
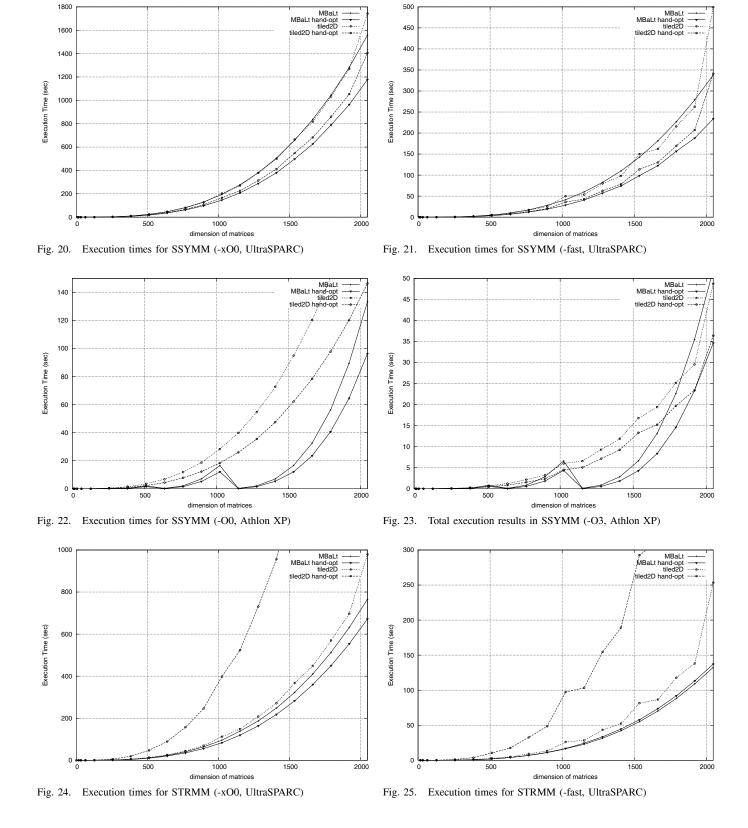


Fig. 19. Execution times for SSYR2K (-fast, UltraSPARC)

not include runtime layout conversions from linear to blocked ones, because a single layout for all instances of each specific array is selected, which optimizes all possible references for the array.

Transformations needed to optimize the various versions of

the benchmark codes (MBaLt, tiled, Lmo, L4d) were applied by hand. Moreover, constants needed for accessing arrays stored in blocked layouts (such as column_masks, *bounds* of arrays, *tilebounds*, and so on - see section IV-D), are calculated through in-line library functions. All tested versions



of benchmarks are exhaustively optimized according to section IV-E (all possible transformations are applied). For each separate benchmark (Matrix Mul, LU, SSYR2K, SSYMM, STRMM), the only difference, in the various versions tested, is the applied data layout. Best data layout selection in the MBaLt version is determined by considering the specific loop ordering and the priority of array references (step 5 of the flow chart). Both factors are determined from earlier steps of our algorithm. So, almost no extra time would be spent by a compiler for determining the optimal block layout. The most time consuming part of our algorithm $(O(A \cdot r \cdot D))$ is unavoidable no matter what sort of layout (linear or not) is

used. Adding block layout selection does not incur any extra complexity. Only constant calculations for the MBaLt versions could add some time delay. However, such constants are calculated just once, which, in any case, takes negligible time, compared to thousands of iterations of real array computations.

B. Execution Time Measurements

As far as the Matrix Multiplication benchmark is concerned, we compared our method (MBaLt) having selected the best tile size that achieves maximum performance, with the ones proposed in the literature [18], where the respective optimal tile size has been selected as well. Optimal tile sizes are chosen experimentally, by making execution time measurements for various tile sizes. It is observed that tiles should fit in L1 cache and especially for blocked layouts step = $\sqrt{L1 \ cache \ capacity}$. In figure 11, 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 (tiled2D). Since current compilers do not support non-linear (blocked) array layouts, we used one-dimensional arrays throughout the implementation of our method (MBaLt). Storing elements in one-dimensional arrays, while two-dimensional ones are needed, required extra computations to find the desired array elements. Thus, a fair comparison should be done with one-dimensional arrays arrays of the tiled version (tiled1D). In this case, MBaLt over-exceeds in performance by 60%. Both Chatterjee's implementations $(L_{MO}$ (Lmo) and L_{4D} (L4d)) perform worse than tiled2D. Notice that the L_{4D} array layout is in practice the same as MBaLt, except for data are indexed through four-dimensional arrays instead of one-dimensional ones used in our approach. However, the performance degradation is due to the much more complicated memory location calculation scheme when dealing with four-dimensional arrays.

Using the -fast optimization level (figure 12), MBaLt gives a gentle slope curve as execution times increase smoothly when array sizes increase, due to independence on conflict misses. The tiled version of the benchmark, (especially in LUdecomposition and SSYR2K), is prone to such unpredictable types of misses, since steep fluctuations occur in the performance graph for specific matrix sizes. Furthermore, comparing figures 11 and 12, we conclude that the simpler the initial code is, the better optimization the -fast level can bring. Thus, standard compiler options are not enough when optimizing complex codes. Applying even more optimizations by hand (e.g. loop unrolling and data prefetching) in conjunction with -fast optimization level, proves to be the best technique for achieving top performance (see hand-optimized performance of MBaLt: MBaLt-hand_opt in figure 17). Extreme performance degradation for specific array sizes is possible due to self-conflict misses that take place when successive tile lines map to exactly identical cache locations.

Although the MBalt code is larger in terms of instruction lines (1.5 more instruction code lines, as seen in the example code of section IV-D), it takes less to execute, since boolean operations are used for address computations. Furthermore, array padding, which is necessary for rounding up array sizes in order to become a power of 2, does not affect performance, since execution time increases regularly, proportionally to the actual array sizes. The reader can verify that no sharp peaks occur in the execution time graphs. 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 nonlinear layouts usually needs a lot of computations, to find the right location. Consequently, what we achieved with the proposed method is handling one-dimensional arrays without introducing any additional delays due to complicated address computations.

The above results are also verified with the LUdecomposition, SSYR2K, SSYMM and STRMM benchmarks (figures 14 - 25). Additionally, we noticed that in LUdecomposition, the use of blocked array layouts in combination with our fast indexing (MBaLt version), not only provides with an average of 15% reduction in execution times (when no optimization is applied: figure 14) compared to the tiled version (tiled2D), but also smoothes any steep peaks that come up due to conflict misses in the simple tiled code. This reduction is even bigger (can reach even 30%) with -fast flag. After observing that tiled1D (in the matrix multiplication benchmark and LU-decomposition) outperforms MBaLt when -fast optimization (in SGI Origin platform) is applied for arrays smaller than 2048×2048 , we conducted experiments with larger ones (sizes bigger than 2048×2048). Figure 17 illustrates that MBaLt in this case, for large matrices, is much more efficient also using -fast optimization level.

In the SSYR2K benchmark (figures 18 and 19), we searched for the best tile size among a wide range of values (not only for values of power of 2), when linear layouts are used (tiled2Dfine). We notice that, in all previous tiled codes experiments were using tile sizes equal to a power of 2, in order to be in accordance with the MBaLt implementation. The fact that graphs for tiled and tiled-fine are practically identical, proves that no delay is introduced when tile sizes are a power of 2.

Finally, as far as different tile sizes are considered, MBaLt versions perform better for tile sizes, that keep data mainly in L1 cache. Successive tile lines are mapped in sequential cache lines. As a result, conflict misses are minimized and optimal tile sizes are fixed for any array size. On the other hand, tile sizes that give minimum execution times in the tiled versions (tiled1D and tiled2D) are difficult to predict. Conflict misses can, in this case, severely degrade performance. To avoid conflict misses, complex cache alignment analysis is required or else an exhaustive search of all possible tile sizes.

C. Simulation results

In order to verify the results of measured execution times, we applied the binary codes of matrix-multiplication (MBaLt, Kand2D), LU-decomposition (MBaLt, tiled) and SSYR2K (MBaLt, tiled) to the SimpleScalar 2.0 toolkit, simulating both the cache characteristics of the Sun Enterprise 450 and the SGI Origin machines. We measured the data L1 (dl1), unified L2 (ul2) cache and data TLB (dtlb) misses, for various values

Misses in D-L1 cache, L2 cache and TLB

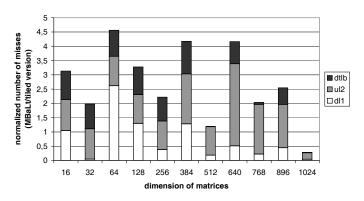


Fig. 26. Matrix Multiplication: misses (normalized values: MBaLt/tiled version) in all levels of Memory Hierarchy (UltraSPARC)

Ν	D-L1 (%)	L2 (%)	D-TLB (%)	total
16	-4,66	-8,74	0,00	-8,24
32	95,75	-6,59	13,33	66,14
64	-161,52	-3,22	8,33	-118,07
128	-29,80	-0,93	3,33	-27,70
256	62,15	-0,10	16,31	60,93
384	-27,92	-75,26	-13,98	-28,80
512	81,72	0,06	98,81	81,83
640	49,24	-187,69	22,30	45,43
768	78,05	-74,55	92,81	76,97
896	55,72	-52,06	41,67	53,62
1024	95,85	77,27	98,82	95,73
average:	26,78	-30,16	34,70	27,08

TABLE III

IMPROVEMENT ACHIEVED IN TERMS OF MISSES IN THE MBALT VS. TILED VERSION OF MATRIX MULTIPLICATION FOR EACH LEVEL OF MEMORY HIERARCHY (ULTRASPARC)

for N, ranging from 16 to 1024, and various values for *step*, ranging from 8 to N, at the standard optimization level.

The normalized values presented in tables and figures of this section are calculated through the fraction $\frac{value \ in \ the \ MBaLt \ version}{value \ in \ the \ tiled \ version}$. Percentage improvement in MBaLt is $\frac{(tiled \ performance) - (MBaLt \ performance)}{tiled \ performance} \cdot 100\%$.

The results show that dl1 misses are reduced in matrix multiplication for tiles of size 64×64 , because at this tile size, data of a whole tile for each one of the three arrays fits in the L1 cache. For the same reason, for step = 2048, ul2 misses increase sharply, compared to ul2 misses when step = 1024. The minimum value fluctuates with the problem size, because it depends on conflict misses which can not be easily foreseen, and thus avoided. Figure 26 shows the ratio of MBaLt misses vs. Kand2D misses for all memory hierarchy levels of the UltraSPARC architecture, for different array sizes. In all cases, when total height is less than 3, MBaLt has better cache performance, in all levels of memory hierarchy. The analytic formula used to calculate the height of bars in figures 26 to 29 is $\left(\frac{m_{1M}}{m_{1t}} + \frac{m_{2M}}{m_{2t}} + \frac{m_{TM}}{m_{Tt}}\right)$, where m_1 = number of D-L1 misses, m_2 = number of L2 misses, m_T = number of TLB misses, while the 2nd subscript letter stands for M = MBaLt version and t = tiled version. The most weighty factor of memory performance is L1 cache

Misses in D-L1 cache, L2 cache and TLB

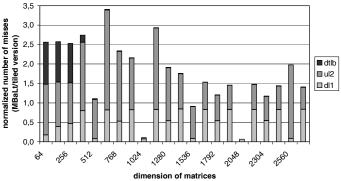


Fig. 27. LU-decomposition: misses (normalized values: MBaLt/tiled version) in all levels of Memory Hierarchy (UltraSPARC)

N	D-L1 (%)	L2 (%)	D-TLB (%)	total
16	-8,63	-47,20	-25,00	-42,21
32	-6,29	-44,11	-11,11	-38,29
64	82,36	-30,29	-8,33	31,68
128	61,09	-14,47	-4,17	54,16
256	52,92	-4,47	-1,39	51,67
384	20,34	-76,18	81,75	19,65
512	92,26	-1,11	97,85	92,22
640	18,03	-156,85	98,10	29,72
768	46,99	-79,33	98,65	53,35
896	17,58	-32,80	98,93	35,17
1024	92,53	96,77	100,00	99,11
1152	16,96	-109,56	99,08	34,46
1280	46,04	-36,46	99,20	54,02
1536	91,51	18,07	99,32	91,19
1664	16,74	30,35	99,36	41,45
1792	45,78	34,19	99,39	56,92
1920	17,02	37,78	99,42	43,00
2048	93,55	99,89	100,00	99,56
2176	16,82	35,62	99,39	43,39
2304	45,48	37,83	99,40	57,62
2432	16,76	39,78	99,39	44,19
2560	91,37	-89,05	100,00	99,35
2688	16,61	43,38	99,37	45,12
average:	41,65	-9,95	75,74	47,30

TABLE IV

IMPROVEMENT ACHIEVED IN TERMS OF MISSES IN THE MBALT VERSION OF LU-DECOMPOSITION FOR EACH LEVEL OF MEMORY HIERARCHY

(ULTRASPARC PLATFORM)

behaviour, as it can bring an extremely large number of misses. Thus, improvement in real time execution is achieved when the white bar of figures 26 to 29 is lower than 1, even if L2 and TLB misses increase. In any case, no L2 cache or TLB thrashing occurs (no major increase in the number of misses occurs), because we have to restrain the tile size so that L1 cache misses do not increase excessively. The total % improvement of memory hierarchy performance in MBaLt vs. tiled versions is tables III, IV (last column), is given by the formula: $\frac{(m_{1t}-m_{1M})\cdot p_1+(m_{2t}-m_{2M})\cdot p_2+(m_{Tt}-m_{TM})\cdot p_T}{m_{1t}\cdot p_1+m_{2t}\cdot p_2+m_{Tt}\cdot p_T}$, where $p_1 = D-L1$ miss penalty, $p_2 = L2$ miss penalty and $p_T =$ TLB miss penalty in clock cycles. The reduction of misses in %, achieved in the MBaLt code, is shown in table III. Negative percentage means that misses increase in MBaLt. Such values are met mainly in L2 cache. However, data L1

Misses in D-L1 cache, L2 cache and TLB

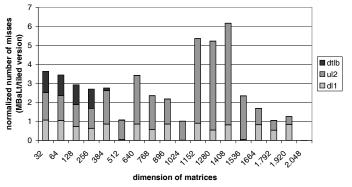
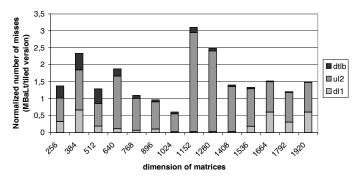


Fig. 28. LU-decomposition: misses (normalized values: MBaLt/tiled version) in all levels of Memory Hierarchy (SGI Origin)



Misses in D-L1 cache, L2 cache and TLB

Fig. 29. SSYR2K: misses (normalized values: MBaLt/tiled version) in all levels of Memory Hierarchy (UltraSPARC)

cache misses are 2 orders of magnitude more than that of L2 cache. So, although a L1 miss has much less performance cost (latency) than a L2 one, the total percentage (calculated by taking into consideration the latency of each memory level), gives an average of 27% miss improvement, and an even better result as array dimension grow bigger. As a matter of fact, small array sizes have no need of data layout or any other optimization, because the working set, in this case, can totally fit even in the higher (faster) level of cache hierarchy. Complicated transformations can be efficient when cache and TLB misses really degrade performance, which happens for large array sizes. For such large array sizes (N > 512) MBaLt achieves to decrease misses in all memory levels significantly. The average miss improvement is over 70%.

Furthermore, we observed that among the various tile sizes used in the matrix multiplication benchmark, in MBaLt code, the TLB misses remain within the same order of magnitude for all tile sizes. On the contrary, in Kand2D code, the best tile size (*step*) for TLB misses does not match to the one that minimizes the data L1 misses, while for other values of *step*, TLB misses increase rapidly.

The results are quite similar for the rest of the benchmarks. In the LU-decomposition benchmark, for example, the TLB misses of the masked version are almost zero compared to those that occur in the tiled version of the code, so, in figures 27 and 28, the TLB misses are, for the most array sizes, Finally, notice that for the problematic array sizes (sizes equal to some power of 2 and N^2 > cache capacity) MBaLt achieves a major reduction of misses in all memory levels. Blocked data layouts manage an efficient placement of array elements, mapping elements accessed by nearby iterations to adjacent, but not identical cache lines. Otherwise, linear layouts, when array dimensions are equal to some power of 2, align successive rows to exactly the same cache line, thus, bringing many conflicts.

VI. 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 using 5 benchmarks and have provided with a fast addressing scheme that uses simple binary masks, accomplished at low cost. Experimental results illustrate the efficiency of the proposed fast address computation method. Simulations prove that performance improvement is due to excessive reduction, mainly of data L1 cache misses and additionally of TLB ones.

ACKNOWLEDGMENTS

We wish to express our profound gratitude to the anonymous reviewers for their suggestions, which considerably increased the clarity and quality of the original manuscript.

REFERENCES

- D. Patterson and J.Hennessy, *Computer Architecture. A Quantitative Approach*, 3rd ed. San Francisco, CA: Morgan Kaufmann Pub., 2002, pp. 373–504.
- [2] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *Computer Architecture News*, vol. 23, no. 1, pp. 20–24, 1995.
- [3] M. Jimenez, "Multilevel tiling for non-rectangular iteration spaces," Ph.D. dissertation, Universitat Politecnica de Catalunya, May 1999.
- [4] A. Aggarwal, "Software caching vs. prefetching," in *Proceedings of the third international symposium on Memory management*. ACM Press, 2002, pp. 157–162.
- [5] A.-H. A. Badawy, A. Aggarwal, D. Yeung, and C.-W. Tseng, "Evaluating the impact of memory system performance on software prefetching and locality optimizations," in *Proceedings of the 15th international conference on Supercomputing*. ACM Press, 2001, pp. 486–500.
- [6] C.-K. Luk and T. C. Mowry, "Automatic compiler-inserted prefetching for pointer-based applications," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 134–141, 1999.
- [7] T. Mowry and A. Gupta, "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 12, no. 2, pp. 87–106, 1991.
- [8] K. S. McKinley, S. Carr, and C.-W. Tseng, "Improving Data Locality with Loop Transformations," *ACM Transactions on Programming Languages and Systems*, vol. 18, no. 04, pp. 424–453, July 1996.
 [9] S. Coleman and K. S. McKinley, "Tile size selection using cache
- [9] S. Coleman and K. S. McKinley, "Tile size selection using cache organization and data layout," in *Proceedings of the ACM SIGPLAN* 1995 conference on Programming language design and implementation. ACM Press, 1995, pp. 279–290.
- [10] J. Chame and S. Moon, "A Tile Selection Algorithm for Data Locality and Cache Interference," in *Proc. of the 13th ACM Int. Conf. on Supercomputing (ICS '99)*, Rhodes, Greece, June 1999.

- [11] S. Ghosh, M. Martonosi, and S. Malik, "Precise miss analysis for program transformations with caches of arbitrary associativity," in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII).* ACM Press, Oct 1998, pp. 228–239.
- [12] M. E. Wolf and M. S. Lam, "A Data Locality Optimizing Algorithm," in Proc. of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada, June 1991, pp. 30–44.
- [13] M. T. Kandemir, "Array Unification: A Locality Optimization Technique," in Proc. of the International Conference on Compiler Construction (CC'01), ETAPS'2001, Genova, Italy, Apr 2001, pp. 259–273.
- [14] O. Temam, E. D. Granston, and W. Jalby, "To copy or not to copy: a compile-time technique for assessing when data copying should be used to eliminate cache conflicts," in *Proceedings of the 1993 ACM/IEEE conference on Supercomputing.* ACM Press, 1993, pp. 410–419.
- [15] P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau, "Augmenting loop tiling with data alignment for improved cache performance," *IEEE Trans. Comput.*, vol. 48, no. 2, pp. 142–149, 1999.
- [16] G. Rivera and C.-W. Tseng, "Data Transformations for Eliminating Conflict Misses," in Proc. of the 1998 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'98), Montreal, Canada, June 1998.
- [17] M. Cierniak and W. Li, "Unifying Data and Control Transformations for Distributed Shared-Memory Machines," in *Proc. of the ACM SIGPLAN*, La Jolla, California, USA, 1995.
- [18] M. Kandemir, J. Ramanujam, and A. Choudhary, "Improving Cache Locality by a Combination of Loop and Data Transformations," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 159–167, Feb 1999.
- [19] 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*, vol. 10, no. 2, Feb 1999.
- [20] M. Kandemir, J. Ramanujam, and A. Choudhary, "A Compiler Algorithm for Optimizing Locality in Loop Nests," in *Proc. of the International Conference on Supercomputing (ICS)*, Vienna, Austria, July 1997, pp. 269–276.
- [21] M. Kandemir, J. Ramanujam, A. Choudhary, and P. Banerjee, "A Layout-conscious Iteration Space Transformation Technique," *IEEE Transactions on Computers (TC)*, vol. 50, no. 12, pp. 1321–1336, Dec 2001.
- [22] D. S. Wise, G. A. Alexander, J. D. Frens, and Y. Gu, "Language Support for Morton-order Matrices," in *Proc. of the 2001 ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'01)*, Utah, USA, June 2001.
- [23] D. S. Wise and J. D. Frens, "Morton-order Matrices Deserve Compilers' Support," CS Dept., Indiana University," TR533, Nov 1999.
- [24] M. S. Lam, E. E. Rotheberg, and M. E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms," in *Proc. of the 1991 ACM SIGMOD Int. Conf. on Management of Data*, Denver, Colorado, May 1991.
- [25] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi, "Nonlinear Array Layouts for Hierarchical Memory Systems," in 13th ACM Int. Conf. on Supercomputing (ICS '99), Rhodes, Greece, June 1999.
- [26] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi, "Recursive Array Layouts and Fast Parallel Matrix Multiplication," in 11th ACM Symp. on Parallel Algorithms and Architectures (SPAA '99), Saint Malo, France, June 1999.
- [27] C.-Y. Lin, J.-S. Liu, and Y.-C. Chung, "Efficient Representation Scheme for Multidimensional Array Operations," *IEEE Transactions on Comput*ers, vol. 51, no. 03, pp. 327–345, Mar 2002.
- [28] G. Rivera and C.-W. Tseng, "Locality Optimizations for Multi-Level Caches," in *Proc. of the ACM/IEEE SC'99 Conference*, Portland, OR, Nov 1999.
- [29] N. Park, B. Hong, and V. Prasanna, "Analysis of Memory Hierarchy Performance of Block Data Layout," in *Int. Conf. on Parallel Processing* (*ICPP 2002*), Vancouver, Canada, Aug 2002.
- [30] —, "Tiling, Block Data Layout, and Memory Hierarchy Performance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 07, pp. 640–654, July 2003.
- [31] D. S. Wise, "Ahnentafel Indexing into Morton-ordered Arrays, or Matrix Locality for Free," in *1st Euro-Par 2000 Int. Workshop on Cluster Computing*, Munich, Germany, June 2001, pp. 774–783.
- [32] A. R. Lebeck and D. A. Wood, "Cache Profiling and the SPEC Benchmarks: A Case Study," *IEEE Computer*, vol. 27, no. 10, pp. 15–26, 1994.



Evangelia Athanasaki received her Diploma in Electrical and Computer Engineering from the National Technical University of Athens in 2002. She is currently a PhD candidate in the School of Electrical and Computer Engineering, National Technical University of Athens. Her research interests include Computer Architecture and High Performance Computing. She is a student member of the IEEE.



Nectarios Koziris received his Diploma in Electrical Engineering from the National Technical University of Athens (NTUA) and his Ph.D. in Computer Engineering from NTUA (1997). He joined the Computer Science Department, School of Electrical and Computer Engineering at the National Technical University of Athens in 1998, where he currently serves as an Assistant Professor. His research interests include Computer Architecture, Parallel Processing, Parallel Architectures (OS and Compiler Support, Loop Compilation Techniques, Automatic

Algorithm Mapping and Partitioning) and Communication Architectures for Clusters. He has published more than 60 research papers in international refereed journals and in the proceedings of international conferences and workshops. He has also published two Greek textbooks "Mapping Algorithms into Parallel Processing Architectures", and "Computer Architecture and Operating Systems". Nectarios Koziris is a recipient of the IEEE IPDPS 2001 best paper award for the paper "Minimising Completion Time for Loop Tiling with Computation and Communication Overlapping" (held at San Francisco, California). He is reviewer in International Journals and Conferences. He served as a Program Committee member in HiPC-2002 and CAC03 (organized with IPDPS03) Conferences, Program Committee co-Chair for both the ACM SAC 2003 and 2004 Symposiums on Applied Computing-Special Track on Parallel, Distributed Systems and Networking. He conducted research in several EU and national Research Programmes. He is a member of IEEE Computer Society, member of IEEE-CS TCPP and TCCA (Technical Committees on Parallel Processing and Computer Architecture), ACM and organized the Greek IEEE Chapter Computer Society.