A Graph Based Framework to Detect Optimal Memory Layouts for Improving Data Locality

Mahmut Kandemir* Alok Choudhary* J. Ramanujam† Prith Banerjee*

Abstract

In order to extract high levels of performance from modern parallel architectures, the effective management of deep memory hierarchies is very important. While architectural advances in caches help in better utilization of the memory hierarchy, compiler-directed locality enhancement techniques are also important. In this paper we propose a locality improvement technique that uses data space (array layout) transformations in contrast to most of the previous work based on iteration space (loop) transformations. In other words, rather than changing the order of loop iterations, our technique modifies the memory layouts of multi-dimensional arrays. In comparison with previous work on data transformations it brings two novelties. First, we formulate the problem on a special graph structure called the layout graph (LG) and use integer linear programming (ILP) methods to determine optimal layouts. Second, in addition to static layout detection, our approach also enables the compiler to determine optimal dynamic layouts; that is, the layouts that can be changed across loop nest boundaries. We believe that this is the first attempt to determine optimal dynamic memory layouts. We also present preliminary experimental results on the SGI Origin 2000 distributed shared memory multiprocessor. Our results so far are encouraging and indicate that the additional compilation time taken by the solver is tolerable.

1 Introduction

Users of shared-memory parallel machines realize that the performance of programs strongly depends on how effectively the memory hierarchy characteristics are exploited. Cache memory utilization is good whenever data blocks brought from memory to cache are reused in the near future. Unfortunately a straightforward coding of many scientific applications does not exploit cache locality despite the significant potential for reuse. Most of the data (mainly arrays) that scientific codes manipulate get reused; but if the time between two uses of the same piece of data is not short enough, the chances are very high that the data will be displaced from cache due to a combination of factors such as limited associativity, limited cache size and memory access patterns of the program.

Recently a large body of work has been done to enhance the cache performance of scientific codes using compiler-based techniques. These rely on the ability of an optimizing compiler to have a global view of the program access patterns and data structures; therefore, a compiler can modify the access patterns to attain high levels of performance. The techniques that focused on access pattern modifications generally target loop nest structures where most of the execution time is spent. Along these lines, techniques such as loop interchange, distribution, fusion and tiling [21, 16, 22, 14, 11, 12, 4, 2, 15] have found their way into commercial compiler products. A common characteristic of these approaches is that they change the execution order of loop iterations by applying some kind of iteration space transformations. The main problem here is that changing the order of iterations may not always be legal; that is, the resulting program may not have the same semantics as the original program. Another problem is that changing the order of loop iterations affects the locality properties of all the data structures (e.g., arrays) referenced in the nest. It may not be trivial to find a loop transformation such that cache locality of all the data structures referenced in the nest will be improved. And lastly the scope of the iteration space transformations is limited, because they are not easily applicable to imperfectly-nested loops and explicitly-parallel loops [3].

More recently researchers have concentrated on data space oriented approaches to improve cache locality characteristics of scientific codes [3, 1, 18, 13, 9, 8]. Rather than changing the iteration space traversal order, data space transformations focus on multi-dimensional arrays and transform their memory layouts such that the new layouts will better match the memory access patterns imposed by the loop nest. An important advantage of these techniques is that they are not affected by data dependences; therefore, they are constrained only by the sequence and association rules of the language in question and not the program being transformed. The main problem in modifying the memory layout of an array, however, is that this change should be propagated to all the loop nests that reference this array. There is a distinct possibility that a memory layout that is suitable for a given array in a loop nest may be highly inappropriate for the same array in another loop nest. Then comes the question of selecting a memory layout for a given array which will satisfy as many loop nests as possible. In order to solve this problem several heuristics have been proposed. For example, one may view multiple loop nests as a single imperfectly-nested loop nest with an imaginary outermost loop that iterates only once [13, 8]. Since data transformations are applicable to imperfectly loop nests as well (as they do not change the loop structures), such a view is possible. The problem with this approach is that when the array in question is referenced in two loop nests with different optimal layout requirements, we need to employ a conflict resolution scheme [8]. Such a scheme in general favors one of the references over the other, resulting sometimes in a sub-optimal memory layout for the latter. An alternative way of handling this global array layout optimization problem is based on propagating memory layouts across loop nests. In an earlier paper [7], we proposed a technique that starts with the most costly nest and optimizes it using data transformations. After this step, the optimal memory layouts for the arrays referenced in this loop nest are determined. These layouts are then propagated to the next most costly nest. The potential negative impact of these new layouts in this second loop nest is decreased using iteration space transformations [18, 7]. This approach has three main problems, though. First, it uses iteration space transformations for all but the first nest, thereby bringing the issue of legality into the picture again. Second, it may not always be possible to find a loop transformation to lessen the negative impact of data transformations, especially when more than one array is involved. And finally, the success of this method depends greatly on the order in which the loop nests are handled. It may not always be easy to determine a suitable order to process the loop nests.

*CPDC, Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208. e-mail: {mtk,choudhar,banerjee}@ece.nwu.edu

†Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803. e-mail: jxr@ee.lsu.edu

In this paper we describe a new approach to determine the optimal memory layouts for the multi-dimensional arrays referenced in a given program automatically. Our approach differs from the previous work on cache locality optimization in two important aspects. First, we formulate the problem on a graph structure that we call the *layout graph* (LG) and solve it using an *integer linear programming* (ILP) [17] optimally. Secondly, we consider memory layout changes (i.e., dynamic memory layouts) between loop nests. The first aspect enables the compiler to solve the problem without resorting to heuristics based on linear algebra equations that need to be built from the program structure, solved using techniques based on matrix arithmetic, and converted back to the compiler's internal data structures. Instead, our approach works on a graph based representation, thus requiring minimum additional help from the compiler, and finds optimal solutions. The second aspect mentioned above might be interesting since scientific programs in general have a number of loop nests with complex access patterns so that static memory layouts may be insufficient to exploit the cache hierarchy fully. The dynamic memory layout changes may be attractive alternatives for such programs. Of course, the added cost of changing memory layouts between loop nests must be taken into account as well.

We have evaluated our approach using a number of programs, and the results so far show that it successfully optimizes the codes in our experimental suite. We have also found that the extra time taken by the solver used to determine optimal memory layouts was never more than 19% of the original compilation time. Our current approach works on a single procedure at a time and does not take inter-procedural analysis into account. If on-going work on inter-procedural data transformations [19] proves to be successful, we plan to extend our approach to an inter-procedural setting as well.

The remainder of this paper is organized as follows. Section 2 defines our notion of locality with respect to a reference and introduces the layout graph. In Section 3 we formulate the memory layout detection problem on the LG and show how our approach can determine optimal static memory layouts for a given program. In Section 4 we explain our solution to dynamic layout detection. We present performance numbers for several programs on the SGI Origin 2000 distributed shared memory multiprocessor in Section 5. The paper concludes with a summary along with pointers to work in progress in Section 6.

2 Fundamental concepts in our framework

Our approach is based on building a graph called the *layout graph* for a given program and using integer linear programming to determine optimal memory layouts for multi-dimensional arrays. Strictly speaking, when we attempt to solve the problem with integer linear programming, building a graph is not necessary. If we decide to build a graph, then a shortest path algorithm will suffice. However, in this paper, we present both the views; this is because, a simple shortest path formulation does not suffice when issues related to parallelism and loop transformations are also taken into account. For example, when, in addition to optimizing cache locality, explicit data distribution across memories of processors is involved, we need to consider the cost of managing the data across memories as well. This, in its most general form, can be expressed as a *facility location problem* [17].

We restrict ourselves to regular dense array codes. An important characteristic of these codes is that their memory access patterns can be determined to a certain extent at compile time. We also confine ourselves to affine array bounds and affine subscript functions as most scientific codes contain such references. We assume that all loops have a unit step (which can be obtained using *loop normalization* [22]). Since it is usually the case that programs accessing large multi-dimensional arrays have serious cache locality problems, throughout this paper we are concerned primarily with optimizing cache locality for large multi-dimensional arrays. We ignore references to scalars, to one-dimensional arrays, and to small-sized arrays.

2.1 Memory layouts for multi-dimensional arrays

We assume that the memory layout of an m-dimensional array can be in one of $m!$ forms, each of which corresponds to an ordering of the dimensions. For example, for a two-dimensional array there are two possible memory layouts; namely, row-major (as in C) and column-major (as in Fortran). For three-dimensional arrays, we have six memory layouts, two of which are the row-major and the column-major orders. For a given memory layout, we define the *fastest changing dimension* (FCD) as the subscript position (dimension) whose indices change faster than any other dimension as we traverse the elements in contiguous memory locations. As an example, for a three-dimensional row-major array the third subscript position is the FCD. Given large array bounds and large number of loop iterations (trip counts), it is sufficient to determine just the FCD for a given array. The ordering of the remaining outer array dimensions is of secondary importance. Once an optimal FCD is determined for each array in the program, transforming the program to implement these FCDs within a language with a fixed memory layout for all arrays is quite mechanical; therefore, we do not discuss the code generation issue here and refer the reader to [18] and [8].

2.2 Locality with respect to a subscript position

We now define the notion of *locality with respect to a subscript position*. Given an n-dimensional loop nest (not necessarily perfectly-nested) with j_n as the innermost loop and an m-dimensional array, we can say whether the memory accesses induced by j_n exhibits locality with respect to a given subscript position. Let l be a subscript position for the array in question $(1 \leq l \leq m)$. If j_n appears only in the l^{th} subscript position and does not appear in any other subscript positions, we say that the reference in question exhibits *spatial locality* in the l^{th} position. If j_n does not appear in any subscript position, we say that the reference exhibits *temporal locality* in the l^{th} position (in fact, in every subscript position). If none of these two conditions holds, then we say that the reference *does not* exhibit any locality in the l^{th} position. Notice that all these definitions are only with respect to the innermost loop (j_n) and a specific subscript position (l). Therefore, a reference may have spatial locality with respect to a subscript position while it may not have locality with respect to another. In this paper, we refer to the cases with temporal locality, spatial locality, and no locality as *TL*, *SL*, and *NL*, respectively. We will also use these abbreviations to refer to the *costs* (in terms of *cache misses*) of having temporal locality, spatial locality, or no locality, respectively, in the innermost loop.

To clarify these locality concepts, let us consider the loop nest shown in Figure 1 that accesses five different three-dimensional arrays: U, V, W, X and Y. In this and the following program fragments, the references used in a loop nest will be enclosed by $\{$ and $\}$. The actual computation performed inside the nest is irrelevant for our purposes. Considering the reference to array *U*, since the innermost loop index k appears in all the subscript positions of this reference, it has no locality with respect to any subscript position. For the reference to array *V*, on the other hand, since k appears only in the first subscript position, this reference exhibits spatial locality with respect to this subscript position, but does not exhibit locality with respect to the other subscript positions. The reference to array *Y* exhibits temporal locality with respect to all the subscripts as k does not appear in any of them. The localities exhibited by the remaining references can be determined in a similar manner.

for *i = li, ui*
  for *j = lj, uj*
    for *k = lk, uk*
      { *U(i+k,k,j+k), V(k,i,j), W(i,j,k), X(i,k+i,i+j), Y(i,i+1,j-i)* }

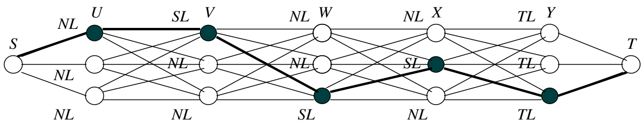

Figure 1: Top: A three-deep loop nest that accesses five three-dimensional arrays. Bottom: The layout graph (LG) for this loop nest. The thick lines and black nodes show an optimal solution found by the solver.

2.3 Layout graph (LG)

After obtaining the locality information, our approach builds an LG. The LG is essentially a graph in which the nodes correspond to subscript positions of arrays, and the edges carry the locality information about the nodes. The nodes are grouped into the columns where each column represents an array (assuming for now a single loop nest and that each array in the nest is referenced only once). In other words, each node in a given column represents an array subscript position. The columns are placed one after another; the order of the columns is not important. Between columns *U* and *V* (corresponding to arrays *U* and *V*) there are $dim(U) \times dim(V)$ edges where *dim*(.) returns the dimensionality for a given array. That is, the edges between *U* and *V* connect every subscript position of *U* to every subscript position of *V*. We define the *cost* of an edge *going into* a subscript position *i* of an array *U* as the cost (in terms of cache misses) of selecting *i* as the fastest changing dimension (FCD) for *U*. This means that all the edges going into the same node have the same costs associated with them (therefore, we write each cost only once).

The LG also has a start node and a terminal node. The start node *S* is connected to each subscript position of the first column whereas the terminal node *T* is connected to each subscript position of the last column. Figure 1 also shows the LG for our example loop nest. The LG, once built, contains all the memory access information for all the arrays with respect to the innermost loop, and inspired by the graph structure used by Garcia et al. [6] to solve the automatic data distribution problem for message-passing machines. In Figure 1 the five columns correspond to the five arrays referenced in the nest. The nodes in a specific column correspond to the subscript positions; e.g., the first node denotes the first subscript position and so on.

3 Static memory layout detection

The 0-1 integer programming problem is a linear integer programming problem in which each variable is restricted to have a value from the set $\{0, 1\}$ [17].

Like Garcia et al. [6], we use the notation E_{UV} to denote all the $dim(U) \times dim(V)$ edges between *U* and *V*. $E_{UV}[i, j]$, on the other hand, denotes the edge between the i^{th} subscript position of *U* and the j^{th} subscript position of *V*. We also use $E_{UV}[i, j]$ to denote the 0-1 *integer variable* associated with the edge in question. Given a path on the LG, $E_{UV}[i, j]$ has a value of 1 if the edge belongs to the path; otherwise its value is 0. In other words, the final value for each $E_{UV}[i, j]$ variable indicates whether the corresponding edge belongs to the optimal solution. Let *Cost*($E_{UV}[i, j]$) be the cost

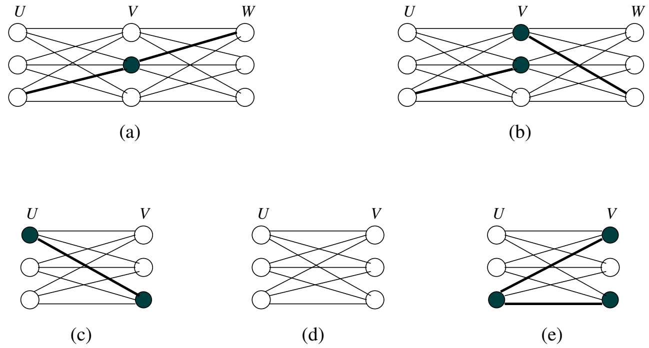

Figure 2: Acceptable and unacceptable LG segments. (a) acceptable, as the selected edges constitute a path. (b) unacceptable, as the selected edges do not constitute a path. (c) acceptable, as there is a single selected edge between two neighboring columns. (d) unacceptable, as no edge is selected between two neighboring columns. (e) unacceptable as more than one edge are selected between two neighboring columns.

of the edge that connects the i^{th} node of *U* with the j^{th} node of *V*. In practice the edge costs should be computed as accurately as possible using the techniques based on miss rate estimations [20]. However, the derivation of exact cost expressions is beyond the scope of this paper. Once the costs are determined, the rest of the technique to be presented is fully automatic. Let $Cost'(E_{UV}[i, j])$ be *Cost*($E_{UV}[i, j]$) if $E_{UV}[i, j]$ is 1 (i.e., $E_{UV}[i, j]$ is selected for a given path on the LG); otherwise $Cost'(E_{UV}[i, j])$ is 0. The *objective* of the locality optimization problem is to select a *path* from S to T on a given LG such that

$$\sum_{U,V} \sum_{i=1}^{dim(U)} \sum_{j=1}^{dim(V)} Cost'(E_{UV}[i, j]) \qquad (1)$$

is *minimized*, where U, V denotes two adjacent columns. Notice that this corresponds to selecting a data layout (FCD) for each array such that the total number of misses will be minimized.

In the following discussion (for sake of simplicity of presentation) we assume that $Cost'(E_{UV}[i, j])$ can take only three possible values: *TL*, *SL*, and *NL*. However, as mentioned above, our technique can accommodate more accurate cost estimations [20]. We impose three conditions to ensure the correctness of the solution:

(1) We should select a path from S to T in the LG,

(2) We should select a single node from each column, and

(3) We should select the edge between two selected nodes.

Consider now Figure 2 in order to interpret these conditions on a few example LG segments. The first condition ensures that all the selected edges should be connected. For example, Figure 2(a) shows an acceptable solution where the two selected edges, $E_{UV}[3, 2]$ and $E_{VW}[2, 1]$, are connected and form a path. Figure 2(b), on the other hand, depicts an unacceptable case where two selected edges ($E_{UV}[3, 2]$ and $E_{VW}[1, 3]$) are not connected. In mathematical terms, we can state this condition as

$$\forall j \in [1...dim(V)] : \quad \sum_{i=1}^{dim(U)} E_{UV}[i, j] = \sum_{k=1}^{dim(W)} E_{VW}[j, k].$$

Notice that this condition should be satisfied for all neighboring triples $U, V,$ and W on the LG. The start and terminal nodes are treated as if they represent one-dimensional arrays.

The second and the third conditions, together, guarantee that each array will have one and only one FCD. For example, Figure 2(c) shows an acceptable situation where the first node of U and the last node of V (as well as the edge connecting them) are selected. These nodes correspond to the fastest changing dimensions for the respective arrays. Figure 2(d) illustrates an unacceptable case as no edge is selected between columns U and V. Similarly, the situation shown in Figure 2(e) is also unacceptable as there is more than one edge selected between two columns. We can enforce these two conditions as

$$\sum_{i=1}^{dim(U)} \sum_{j=1}^{dim(V)} E_{UV}[i,j] = 1$$

for each neighboring pairs U and V.

Returning to our example in Figure 1, after assigning the costs and formulating the conditions given above, if we run the solver we obtain the path shown with the thick lines on the LG. The cost of this path is *TL+3SL+NL*. Therefore, the FCD for arrays *U* and *V* is the first subscript position; for arrays *W* and *Y* the third subscript position; and for array *X* the second subscript position. Notice that this is not the only minimum cost path for this example. For arrays *U* and *Y* all the FCDs are equally acceptable.

It should be noted that this example represents the simplest case, where there is a single loop nest, and each array is accessed using a single reference. In fact, for this example we could have easily worked on each array individually without building an LG. However, we would like to keep our framework as general as possible so that the future extensions (e.g., loop transformations, data distribution, false sharing related issues) can be incorporated easily.

We now discuss more general cases. We define two references to the same array as *uniformly generated with respect to loop index* k if k appears in exactly the same subscript positions for both the references. For example, the references $U(i+j, j, k-1)$ and $U(j, k+j, i+1)$ are uniformly generated with respect to loop index j but are not uniformly generated with respect to i or k. An important observation is that if two references to the same array are uniformly generated with respect to the *innermost* loop index and if they both exhibit spatial locality, they require the same subscript position as their desired FCD. Therefore, if all the references to the same array are uniformly generated with respect to the innermost loop index, then there is no conflict in their layout demands. It should be noted that this concept of uniformity is a relaxed form of the concept of uniformly generated references (UGR) as defined by Gannon et al. [5]. For the rest of the paper, when we say (non-)uniform accesses we mean (non-)uniformly generated accesses with respect to the innermost loop.

If two or more references to the same array are non-uniform, then we have a conflict that should be resolved in favor of one of the references in question. In that case we can represent each reference with a column; that is, if an array is accessed using s non-uniformly generated references, we can use s columns for this array, each column corresponding to a reference. If the array is m-dimensional each pair of adjacent columns for the same array can have m edges between the corresponding subscript positions; we do not allow cross edges (the edges going from i^{th} node to j^{th} node where $i \neq j$) between the nodes belonging to the references to the same array. This is necessary for ensuring that a *single* memory layout (FCD) will be found for the array in question. In fact, in the static memory layout detection problem, all columns representing the same array can be replaced by a single column, summing the respective edge costs.

```
for i = li, ui
  for j = lj, uj
    { U(i,j), V(j,i), W(i+j,i) }

for i = li, ui
  for j = lj, uj
    { U(j,i), W(j,i) }
```

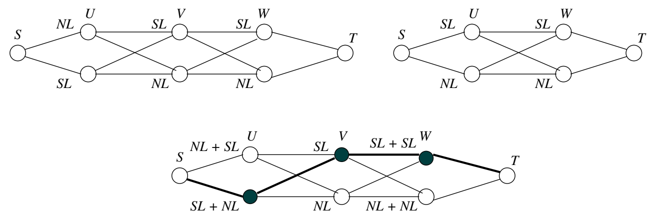

Figure 3: Top: A program fragment that consists of two nests. Middle: The layout graphs. Bottom: The combined LG.

We now focus on the problem of layout detection when multiple loops are involved. Our approach is rather simple and is based on *graph combining*. The idea is that when multiple loop nests access the same array we combine the costs due to different nests into the same subscript position. As an example, consider the program fragment shown in Figure 3. In the middle portion of the figure, the individual LGs corresponding to the nests are shown. The bottom part, on the other hand, depicts the combined LG. Of course, for example, an *SL* cost coming from different nests will usually have different values (in terms of misses), depending on the number of enclosing loops and their trip counts and so on. When more accurate miss estimations [20] are available, our model incorporates them as edge costs. Finally, notice that (for handling the multiple-nest case) graph combining was made possible due to our assumption (in this section) that for each array we determine a single static layout.

4 Dynamic memory layout detection

For scientific codes with complex access patterns, it might be difficult to find static memory layouts that maximize locality for all the loop nests and arrays. Changing the memory layouts dynamically during the course of the program between loop nests may be a more appropriate choice. The LG described in the previous sections contains all the information required to determine optimal static memory layouts. In order to handle dynamic layouts, we need an extension that will basically allow us to select either of two options for each array in a loop nest boundary: we either keep the layout the same as in the previous loop nest, or we dynamically change the layout. Here by dynamic layout change what we mean is that we can change the FCD for the array. It should be emphasized that we attempt to dynamically change the memory layouts only at loop nest boundaries.

We extend our framework by adding *converter nodes* to accommodate layout changes. A converter node is required between the columns of the same array accessed in different nests (see Figure 4). The function of this node is to map an input layout to an output layout for an array. If the input and output layouts are the same no additional conversion cost is incurred. Otherwise, the input edges to the converter node are annotated with the cost(s) of conversion. The output edges of the converter node are connected to the input of the column for the same array in the other nest. In Figure 4, the

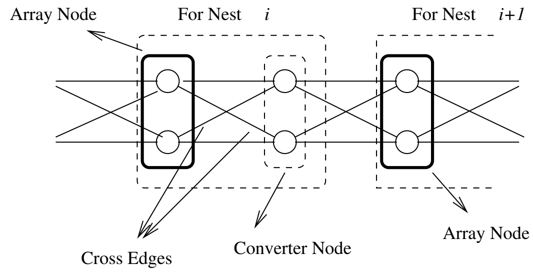

Figure 4: Converter node for dynamic layout detection.

cross edges are used for layout conversions and carry the associated (conversion) costs on them whereas the straight (non-cross) edges indicate that there will be no layout changes between the two nests in question and, of course, have zero costs. It should be stressed that although the nests shown in Figure 4 are consecutive (i.e., i and $i+1$), in general they do not need to be. For example, if an array is accessed in nests 1, 4, and 6, in the resulting LG this array will have three columns, and there will be converter nodes between nest 1 and nest 4 as well as between nest 4 and nest 6. Note that by treating the converter nodes as if they are ordinary nodes in the LG, the 0-1 integer programming method introduced in the previous section can be used for determining the optimal layouts.

5 Experimental results

In this section we present preliminary experimental results on an eight-node SGI Origin 2000 distributed shared memory multiprocessor at Northwestern University. This machine uses MIPS R10000 processors, each of which is a 4-way superscalar microprocessor operating at a clock frequency of 195 MHz. Each processor has a 32 KB on-chip instruction cache, and can issue instructions to its two integer and two floating-point functional units out-of-order. It also has a 32 KB 2-way set-associative on-chip data cache (primary), and 4 MB external cache (secondary). The cache line size is 128 bytes and page size is 16KB. For the primary cache hits, the latency is 2 cycles; and for the primary cache misses that hit in the secondary cache, the latency is 8 to 10 cycles. For the nonlocal misses, on the other hand, the latency is around 20 cycles.

Table 1 provides information about the programs in our experimental suite. The `size` column gives the size of a dimension of any array used in the program. However, the small sized dimensions (e.g., with a fixed value of 4 or 5) are not modified. The `iter` column, shows how many times the outermost timing loop is iterated for each code. We have used four versions of each code: *CM* and *RM* are the original versions in which the memory layout for every array is column-major and row-major, respectively. The *LP* version denotes a code obtained using linear loop transformation techniques for locality (see [14]). In a sense this version represents the state-of-the-art (loop-nest-based) compiler technology for optimizing locality. Finally, the *DT* version is the one obtained using the approach proposed in this paper. For each version, the original codes are first transformed automatically using the respective technique, and the resulting programs are then compiled using the native compiler parallelizing only the outermost loops in each nest with all the scalar optimizations (-O2) turned on. We used the Omega library [10] as our solver (just to generate the enumerating loops) and the binary (or 0-1) conditions were formulated explicitly. In future we plan to link our compiler to either CPLEX or LINGO integer programming tools. We have noticed that the extra time taken by the library was never more than 19% of the original compilation time.

Table 1: Programs used in our experiments.

Program	Source	size	iter	arrays
mxm	Spec92/Nasa7	720	4	three 2-D
adi	Livermore	3000	4	three 1-D, three 3-D
vpenta	Spec92/Nasa7	920	25	seven 2-D, two 3-D
btrix	Spec92/Nasa7	150	10	twenty-five 1-D, four 4-D
syr2k	BLAS	1024	1	three 2-D
htribk	Eispack	1600	4	five 2-D
gfunp	Hompack	2500	10	one 1-D, five 2-D
trans	NWChem (PNL)	4000	10	two 2-D

The overall performance of different versions on 4 processors of the Origin 2000 is shown in Figure 5. In four out of eight programs *DT* outperforms *LT* whereas in three codes the reverse occurs. For comparison purposes, this figure also shows the MFLOPS rates obtained using hand optimized version of each code. In obtaining this last version, we used loop transformations (e.g., permutation, skewing, fusion, tiling) as well as data transformations (e.g., memory layout modifications, padding) in the best possible way we could do. For the codes in our experimental suite, the improvement brought by the hand optimizations upon *DT* is between 4% and 44%, averaging 22.2%. This result motivates us to seek ways of combining loop and data transformations in the LG representation and solve the problem optimally.

6 Conclusions and future work

In this paper we have presented a solution to the optimal memory layout detection problem. Our approach is based on two important elements: (a) formulation of the problem in a special graph structure (LG), and (b) the use of an ILP solver to determine memory layouts. Such an approach not only allows us to solve the global static layout detection problem optimally, but also helps us in the simple formulation of the dynamic layout detection problem. To the best of our knowledge, this is the first attempt to derive a framework allowing dynamic layout modifications for cache locality. We have shown in this paper that the framework is quite powerful. However, whether dynamic layout modifications will be useful for larger applications remains to be seen. Along this direction, we plan to complete our implementation and make extensive experiments to evaluate the impact of our approach on different codes from matrix computations as well as to quantify more thoroughly the time taken by our approach at compile-time.

We believe that the approach proposed here can be enhanced in a number of ways. First, we can unify the loop and data transformations in an enhanced LG representation. We can also take additional parallelism-related factors into the account when building the LG (e.g., data distribution across processor memories). In particular, we would like to use customized LGs for uniprocessors, UMA, NUMA, and message-passing architectures. Lastly, we plan to embed inter-procedural analysis [19] into our framework and seek the ways of transforming array layouts uniformly across processor boundaries.

Acknowledgments

The authors would like to thank Dr. Eduard Ayguade and the anonymous referees for providing helpful comments. The work of Mahmut Kandemir and Alok Choudhary was supported in part by NSF Young Investigator Award CCR-9357840, NSF grant CCR-9509143 and Air Force Materials Command under contract F30602-97-C-0026. J. Ramanujam was supported in part by NSF Young Investigator Award CCR-9457768. Prith Banerjee was supported in part

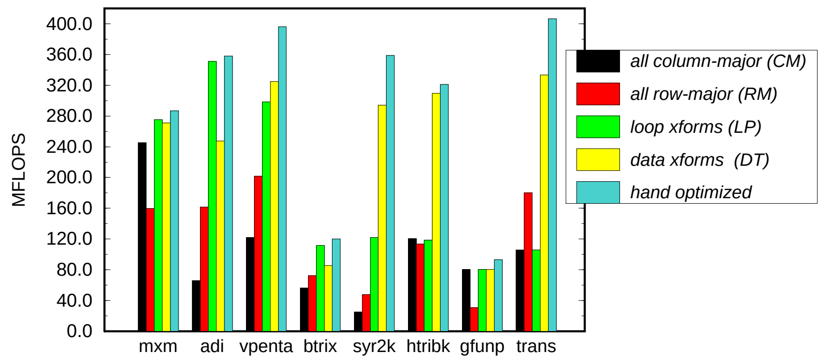

Figure 5: MFLOPS rates.

by DARPA under contract F30602-98-2-0144 and by the NSF grant CCR-9526325.

References

[1] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformations for multiprocessors. In *Proc. 5th ACM SIGPLAN Symp. Principles & Practice of Parallel Programming (PPoPP'95)*, pages 166–178, Santa Barbara, CA, July 1995.

[2] L. Carter, J. Ferrante, S. Hummel, B. Alpern, and K. Gatlin. Hierarchical tiling: a methodology for high performance. *UCSD Tech Report CS 96-508*, November 1996.

[3] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared memory machines. *Proc. SIGPLAN Conf. Programming Language Design & Implementation (PLDI'95)*, La Jolla, CA, pages 205–217, June 1995.

[4] S. Coleman and K. McKinley. Tile size selection using cache organization and data layout. In *Proc. the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI'95)*, ACM, New York.

[5] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. *Journal of Parallel & Distributed Computing*, 5(5):587–616, October 1988.

[6] J. Garcia, E. Ayguade, and J. Labarta. A novel approach towards automatic data distribution. In *Proc. Supercomputing'95*, San Diego, December 1995.

[7] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A matrix-based approach to the global locality optimization problem. To appear in *Proc. 1998 Intl. Conf. Parallel Architectures & Compilation Techniques (PACT'98)*, Paris, France, October 1998.

[8] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A hyperplane based approach for optimizing spatial locality in loop nests. In *Proc. 1998 ACM International Conference on Supercomputing (ICS'98)*, pages 69–76, Melbourne, Australia, July 1998.

[9] M. Kandemir, J. Ramanujam, and A. Choudhary. A compiler algorithm for optimizing locality in loop nests. In *Proc. 11th ACM International Conference on Supercomputing (ICS'97)*, pages 269–276, Vienna, Austria, July 1997.

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

[11] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proc. Programming Language Design and Implementation (PLDI'97)*, June 1997.

[12] M. Lam, E. Rothberg, and M. Wolf. The cache performance and optimizations of blocked algorithms. In *Proc. the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'91)*, ACM, New York.

[13] S.-T. Leung and J. Zahorjan. Optimizing data locality by array restructuring. Technical Report TR 95-09-01, Dept. Computer Science and Engineering, University of Washington, September 1995.

[14] W. Li. *Compiling for NUMA Parallel Machines*. Ph.D. Thesis, Cornell University, Ithaca, NY, 1993.

[15] N. Manjikian and T. Abdelrahman. Fusion of loops for parallelism and locality. In *Proc. the 24th International Conference on Parallel Processing (ICPP'95*), Oconomowoc, Wisconsin, August 1995.

[16] K. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages & Systems,* 18(4):424–453, July 1996.

[17] G. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization*, Wiley-Interscience Publications, John Wiley & Sons, New York, 1988.

[18] M. O'Boyle and P. Knijnenburg. Non-singular data transformations: definition, validity, applications. In *Proc. 6th Workshop on Compilers for Parallel Computers (CPC'96)*, pages 287–297, Aachen, Germany, 1996.

[19] M. O'Boyle and P. Knijnenburg. Integrating loop and data transformations for global optimisation. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, October 14–17, 1998, Paris, France.

[20] V. Sarkar, G. Gao, and S. Han. Locality analysis for distributed shared-memory multiprocessors. In *Proc. the Ninth International Workshop on Languages & Compilers for Parallel Computing (LCPC'96)*, Santa Clara, California, August 1996.

[21] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proc. SIGPLAN Conf. Programming Language Design & Implementation (PLDI'91)*, pages 30–44, Toronto, Canada, June 1991.

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