Static and Dynamic Locality Optimizations Using Integer Linear Programming

Mahmut Kandemir, *Member*, *IEEE*, Prithviraj Banerjee, *Fellow*, *IEEE*, Alok Choudhary, *Fellow*, *IEEE*, J. Ramanujam, *Member*, *IEEE*, and Eduard Ayguadé

Abstract—The delivered performance on modern processors that employ deep memory hierarchies is closely related to the performance of the memory subsystem. Compiler optimizations aimed at improving cache locality are critical in realizing the performance potential of powerful processors. For scientific applications, several loop transformations have been shown to be useful in improving both temporal and spatial locality. Recently, there has been some work in the area of data layout optimizations, i.e., changing the memory layouts of multidimensional arrays from the language-defined default such as column-major storage in Fortran. The effect of such memory layout decisions is on the spatial locality characteristics of loop nests. While data layout transformations are not constrained by data dependences, they have no effect on temporal locality. On the other hand, loop transformations are not readily applicable to imperfect loop nests and are constrained by data dependences. More importantly, loop transformations affect the memory access patterns of all the arrays accessed in a loop nest and, as a result, the locality characteristics of some of the arrays may worsen. This paper presents a technique based on *integer linear programming* (ILP) that attempts to derive the best combination of loop and data layout transformations. Prior attempts to unify loop and data layout transformations for programs consisting of a sequence of loop nests have been based on heuristics not only for transformations for a single loop nest but also for the sequence in which loop nests will be considered. The ILP formulation presented here obviates the need for such heuristics and gives us a bar against which the heuristic algorithms can be compared. More importantly, our approach is able to transform memory layouts *dynamically* during program execution. This is particularly useful in applications whose disjoint code segments demand different layouts for a given array. In addition, we show how this formulation can be extended to address the false sharing problem in a multiprocessor environment. The key data structure we introduce is the *memory layout graph* (MLG) that allows us to formulate the problems as path problems. The paper discusses the relationship of this ILP approach based on the memory layout graphs to other work in the area including our previous work. Experimental results on a MIPS R10000-based system demonstrate the benefits of this approach and show that the use of the ILP formulation does not increase the compilation time significantly.

Index Terms—Data reuse, cache locality, memory layouts, compiler optimizations, cache miss estimation, integer linear programming.

1 Introduction

The speed of microprocessors has been steadily improving at a rate of between 50 percent and 100 percent every year over the last decade. Unfortunately, the memory speed has not kept pace with this, improving only at the rate of about 7 percent to 10 percent per year during the same period [19]. Memory hierarchies in the form of one or more levels of cache have been used extensively in current processors in order to minimize the impact of this speed gap. The performance of applications is determined to a great extent by the memory access characteristics rather than simple instruction and operation counts. Therefore, exploiting the memory hierarchy effectively is key to achieving good performance on modern computers. But, the effective use of caches has traditionally been a difficult problem for scientific applications and will only continue to become more difficult given the increasing gap between the speeds of processors and memories.

Current approaches to deal with this problem include the manual restructuring of programs in order to change the memory access patterns; this requires a clear understanding of the impact of memory hierarchies on the users' part. Such an approach is tedious and can lead to nonportable and difficult-to-maintain programs. The lack of automatic tools has led to much work on compiler optimizations over the last decade [59]. Compiler analysis can result in useful global memory access information and can be used to restructure scientific programs in order to improve the memory access characteristics. Regular scientific codes exhibit significant amounts of data reuse (due to their affine references). Therefore, the key issue is the conversion of this reuse into locality [36], [9] on the target architecture. Several compiler optimization techniques, such as loop interchange [1], [59], unimodular [9], [29], [36], [56] and nonunimodular [36] loop transformations, loop fusion [38], and loop tiling [6], [10], [58], [34], [59], [33],

- *M. Kandemir is with the Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16802. E-mail: kandemir@cse.psu.edu.*
- *P. Banerjee and A. Choudhary are with the Department of Electrical and Computer Engineering, Northwestern University, IL 60208. E-mail: {banerjee, choudhar}@ece.nwu.edu.*
- *J. Ramanujam are with the Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803. E-mail: jxr@ee.lsu.edu*
- *E. Ayguadé is with the Centre Europeu de Parallelisme de Barcelona (CEPBA) Department d'Arquitectura de Computadors, Universidada Politècnica de Catalunya, Jordi Girona 1-3, Modul D6, Barcelona 08034. E-mail: eduard@ac.upc.es.*

Manuscript received 1 June 2000; revised 14 Mar. 2001; accepted 17 Mar. 2001.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 112224.

have been proposed to improve memory performance of loop nests in scientific codes that access large arrays.

These loop transformation techniques attempt to change the memory access patterns of arrays in loops by reordering the iterations and can improve both temporal and spatial locality. Since a single transformation is used for each nest, the effect of the transformation on each array may be different, perhaps resulting in good locality for some arrays at the cost of poor locality for other arrays. Also, loop transformations must be legal and hence are constrained by data dependences [41], [59]. In addition, transformations for imperfectly nested loops require a different (and also more sophisticated) solution as demonstrated by Kodukula and Pingali [32] and Kodukula et al. [33].

Some of these drawbacks of loop transformations have led researchers to consider changing memory layouts of arrays. Note that spatial locality is exploited by moving a block of contiguous words from memory to cache and that, for multidimensional arrays, the storage (or layout) order determines what the contiguous words are. In Fortran, arrays are stored in memory using the column-major order while C uses a row-major order. Recently, some work [9], [21], [22], [24], [27], [35], [43] has addressed the use of different layouts for different arrays in a program. This has led to the development of *data transformations*, i.e., deriving good layouts for different arrays. Data transformations can improve spatial locality significantly and are not constrained by data dependences unlike loop transformations, since execution order is unchanged [29], [9]. An added advantage of data transformations is that they are applicable irrespective of the nesting patterns (perfectly or imperfectly nested). But data transformations have no effect on temporal locality [35]. In addition, the memory layout of an array influences the locality behavior of every loop nest that accesses the array. Therefore, deciding the memory layouts of arrays requires a *global view* of the memory access pattern of the whole program and not just a single loop nest. Not surprisingly, deciding the optimal layouts is NP-complete [28]. Finally, some problematic constructs, like array aliasing and pointers in C and the EQUIVALENCE statement in Fortran, may prevent automatic data layout modifications. We refer the reader to Chandra et al. [7] for a study of techniques for ensuring the legality of memory layout transformations.

It seems natural to try and combine the benefits of loop and data transformations in improving the memory performance of programs. There have been some efforts aimed at unifying loop and data transformations [2], [22], [25], [26], [29], [44]; all these efforts have used some form of heuristics. For example, these heuristics are used to decide such things as the order of processing the nests in deciding layouts and the order in which loop or data transformations are applied in each nest. In earlier work, we have presented a heuristic for deciding the order of processing loop nests [25] and have shown results on using, for each loop nest, loop transformations followed by data transformations [26]. Obviously, the use of heuristics leads to no guarantee of optimality and, more importantly, it is not clear how far the result obtained through a heuristic approach is from the optimal solution under a given set of assumptions. Therefore, it would be beneficial to have a framework that gives us the optimal combination of loop transformations and array layouts.

In this paper, we present a new approach that uses integer linear programming (ILP) [42]. We use a structure called the *memory layout graph* (MLG) to model locality characteristics as a function of loop order. The problem of determining loop and data transformations is then formulated as an ILP problem, which is equivalent to finding optimal paths in the MLG that satisfy different constraints. Unlike other solutions, this approach allows us to derive optimal solutions (within the bounds of our cost model and transformation space) and consider layout changes between parts of a whole program. That is, we handle both static and dynamic memory layout transformations. Dynamic layout optimizations might be particularly useful in large codes where different program segments require different memory layouts for the same group of arrays. In this paper, we show that deciding the best combination of loop and data transformations corresponds to the solution of a path problem on parallel and series compositions of graphs associated to individual loops in a nest and loop nests, respectively. In addition, we also show how the proposed ILP-based solution can be extended to address the false-sharing problem in multiprocessor architectures with shared memory support.

Our program domain includes perfectly nested loops with array subscripts and loop bounds being affine functions enclosing loop indices and loop-invariant constants. Our current implementation uses a preprocessing pass in which the imperfectly nested loops are converted to perfectly nested ones using a combination of loop fusion, loop distribution, and code sinking [58]. When we have choices, we prefer loop fusion and distribution over code sinking as the latter introduces "if statements" within loops (which might degrade the performance).

We used several programs to evaluate the approach presented in this paper. The experiments show that our technique is very effective, improving the performance on the average by 27.5 percent over the original codes, sometimes by as much as a factor of 8. Since the use of an ILP solver may in general require significant time, we also measured the increase in the compilation time needed for our approach over other solutions. We have found in our experiments that the compilation time does not increase by more than 16 percent.

The rest of this paper is organized as follows: Section 2 presents the important concepts used in our approach, such as the memory layout graph, and reviews the relevant background, including data reuse, cache locality, and loop and data transformations. Section 3 presents our approach in detail, highlighting the ILP formulations we derive. Experimental results are presented and discussed in Section 4. In Section 5, we present a discussion of related work in the areas of loop and data transformations. Section 6 presents our conclusions along with a discussion of work in progress.

2 Important Concepts

2.1 Memory Layouts and Loop Transformations

In this work, we assume that the memory layout of an m-dimensional array can be one of $m!$ forms, each corresponding to the traversal of the array dimensions in a predetermined order. For a two-dimensional array, there are only two possible memory layouts: row-major (as in C) and column-major (as in Fortran). For a three-dimensional array, there are six alternatives and so forth. It should be noted that this layout space subsumes the classical row-major and column-major layouts found in the conventional programming languages and higher dimensional equivalents of them and excludes diagonal-like layouts. Although, we believe that the diagonal-like layouts might be useful for some banded-matrix applications [3], [36], [24], we postpone their inclusion in our graph-based solution framework to a future work. For each layout we associate a *fastest changing dimension* (FCD) which represents the dimension whose indices vary most rapidly in a sequential traversal of the array elements in memory. For example, for a three-dimensional row-major array the third dimension is the FCD. This can easily be seen by observing the indices of a sequence of consecutive array elements such as

$$(1,3,1),(1,3,2),(1,3,3),...,(1,3,10).$$

Similarly, for the column-major memory layouts the first dimension is the FCD. It should be stressed that our approach, if desired, can determine the entire dimension order (the complete memory layout) for a given array. The idea behind focusing only on the FCDs is that in many cases it is sufficient from the locality point of view to determine the FCD correctly [29]. The order of the remaining dimensions may not be as important.

As for the loop transformations, we focus on general permutations of loops [59], [38]. From a given loop nest of depth n, we can construct $n!$ permutations (loop orders), though, some of them may not be legal. As with the memory layouts, we mainly focus on correctly determining the innermost loop. The orders for the remaining loops in the nest can also be determined if desired. McKinley et al. [38] state that the loop permutations are sufficient for a significant portion of the loop nests found in practice and the most important issue there is to place the innermost loop correctly.

2.2 Data Reuse and Cache Locality

When a data item (e.g., an array element) is used more than once, we say that a *temporal reuse* occurs [59], [56], [36]. A *spatial reuse*, on the other hand, occurs when nearby data items are used [59], [56], [36]. It should be noted that, whether spatial or temporal, data reuse is an intrinsic property of a given program and is independent from the underlying memory hierarchy and associated hardware parameters.

Cache locality, on the other hand, means capturing data reuse in cache memory and is strongly dependent on a given cache topology [19], [56]. Data reuse translates to cache locality only if the subsequent uses of data occurs before the data are displaced from cache. Since, in current sequential and parallel architectures the cost of a data access changes dramatically from level to level, it is vital to convert data reuse into cache locality. Improving cache locality can also reduce the contention for resources on parallel systems; therefore, it has a systemwide benefit.

To see how data reuse can translate to cache locality consider the program fragment shown in Fig. 1a. This fragment contains two loop nests accessing five arrays. In this and the following program fragments, the references used by a loop nest will be enclosed by { and }. The actual computations performed inside the nests are irrelevant for our purposes. Assuming that the total size of the arrays is larger than the cache capacity and the default memory layout is *column-major* for all arrays, the cache locality is poor for all references except for $Q(j+k,i+j)$ in the first nest and for $S(i+k,j)$ and $T(k,i)$ in the second nest. As a specific example, the locality for reference $Q(i,j+k)$ is poor in the second nest as the successive iterations of the innermost k loop access *different* columns. The locality for this program can be improved by making loop i innermost in the first nest and loop j innermost in the second nest (provided it is legal to do so) and by assigning the following memory layouts to the arrays: column-major for P and R and row-major for Q, S, and T. Doing so improves locality for all references except for reference $R(i+j,j+k,i+k)$ in the first nest. Thus, an appropriate combination of loop and data transformations can change the locality behavior of a program significantly. It should be noticed that neither pure loop nor pure data transformations alone can achieve this performance. This small example also shows that the most important aspect of optimizing cache locality for an array is to select a FCD for it such that the innermost loop index (*after* the loop transformation) will appear either in none of the subscript positions (temporal locality, as in $S(j,k)$ in the first nest and $T(k,i)$ in the second one) or only in the subscript position corresponding to the FCD (spatial locality, as in $P(i,j,k)$ in the first nest and $R(j,k,i)$ in the second one).

Although, the reuse and locality concepts that are centered around array references and loop orders as used by the previous researchers are able to capture the general intuition, we need a finer granularity definition for this study. Let us concentrate on a reference to an m-dimensional array in an n-deep loop nest. Assume that the loops in the nest are ordered as $j_1, j_2, \ldots, j_n$ from outermost to innermost. Assume further that r is a dimension (subscript position) of this array where $1 \leq r \leq m$. We define the locality of the said reference with respect to the dimension r as follows:

- If j_n does not appear in a subscript position (including r), we say that the reference exhibits *temporal locality* with respect to the rth dimension.
- If j_n appears only in the rth subscript position with a coefficient c where $c \leq cache\_line\_size$ and does not appear in any other subscript position, we say that the reference exhibits *spatial locality* with respect to the rth dimension.[1]
- In all other cases, we say that the reference exhibits *no locality* with respect to the rth dimension.

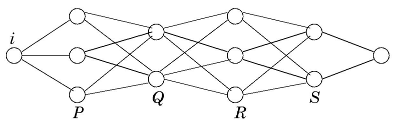
According to these definitions, it is possible that a reference will have spatial locality with respect to a subscript position r but will not have spatial locality with respect to another

1. In this paper, we assume that the condition $c \leq cache\_line\_size$ is always satisfied and we do not consider it further. Notice, however, that different values of c lead to different degrees of spatial locality.

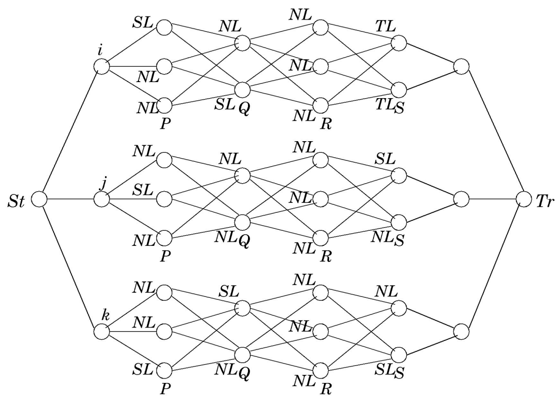
```
for i = li, ui
  for j = lj, uj
    for k = lk, uk
      ${P(i,j,k), Q(j+k, i+j), R(i+j, j+k, i+k), S(j,k)}$
    endfor
  endfor
endfor

for i = li, ui
  for j = lj, uj
    for k = lk, uk
      ${P(j+k, i+k, i-k), Q(i, j+k), R(j,k,i), S(i+k, j), T(k,i)}$
    endfor
  endfor
endfor
```

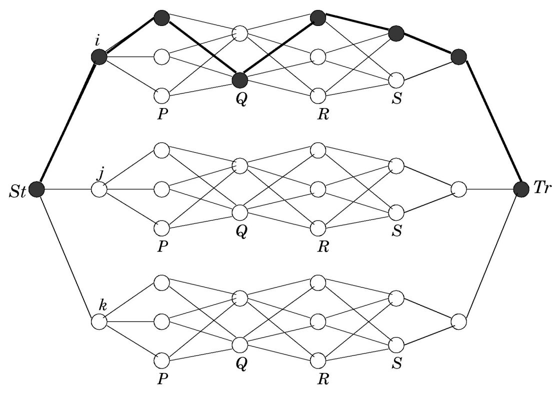
(a)



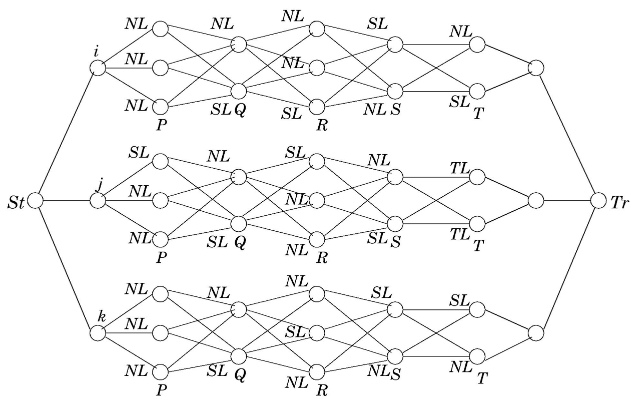
(b)



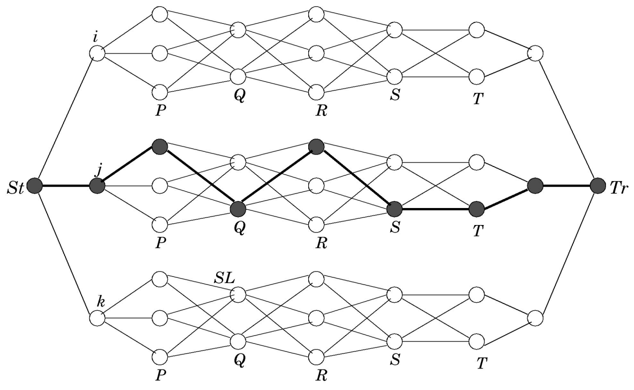
(c)



(d)



(e)



(f)

Fig. 1. (a) An example program fragment. (b) The loop graph (LG) for loop *i* in the first nest. (c) The nest graph (NG) for the first nest. (d) An optimal solution for the first nest. (e) The nest graph (NG) for the second nest. (f) An optimal solution for the second nest.

subscript position r' that is different from r. In contrast, if the reference has temporal locality with respect to r, it will have temporal locality with respect to any other dimension.

It is possible to detect the localities with respect to subscript positions during the analysis of the program. In order to achieve this, we use a function, called *appears*, of the following form:

$$appears(Loop\_Nest, Loop\_Index, Array, Ref, Subs).$$

This function returns one (true) if *Loop_Index* appears in the subscript position, *Subs*, of the reference, pointed by *Ref*, to the array, *Array*, in the nest, *Loop_Nest*; otherwise it returns zero (false). We determine the locality of a reference Rf to an *m*-dimensional array Q in a nest x with respect to a subscript position r assuming that j_n is the innermost loop as follows: If

$$appears(x, j_n, Q, Rf, r) = 1$$

and $appears(x, j_n, Q, Rf, r') = 0$ for all $r' \neq r$, reference Rf has spatial locality (*SL*). If $appears(x, j_n, Q, Rf, r') = 0$ for all $1 \leq r' \leq m$, reference Rf has temporal locality (*TL*). Otherwise, it has no locality (*NL*). We optimize these function

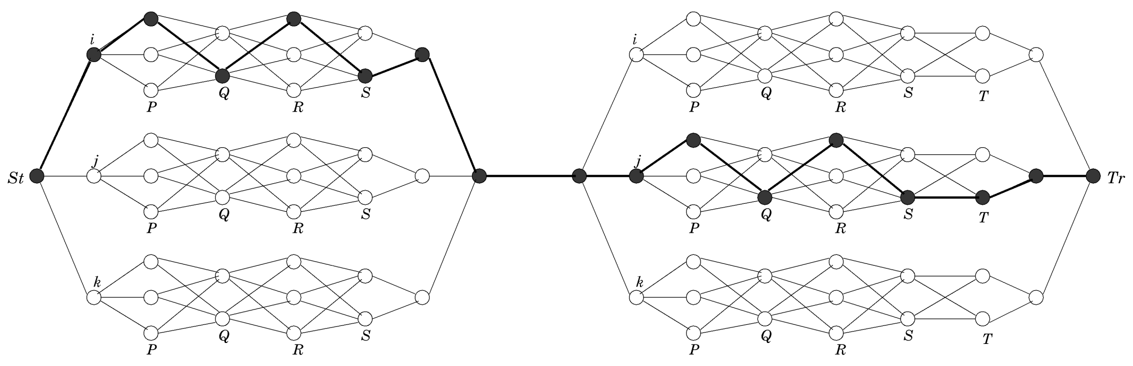


Fig. 2. The MLG and an optimal solution for the program fragment shown in Fig. 1a.

calls such that the number of the calls necessary to determine the localities of a given reference with respect to each subscript position and each candidate innermost loop is minimized. As an example, consider reference $P(i,j,k)$ in the first loop nest shown in Fig. 1a. Supposing that we are considering loop k as the candidate for the innermost position,

$$appears(N,k,P,P(i,j,k),1)=0$$
$$appears(N,k,P,P(i,j,k),2)=0,$$

and

$$appears(N,k,P,P(i,j,k),3)=1.$$

Consequently, $P(i,j,k)$ has spatial locality (*SL*) with respect to the third subscript position and has no locality (*NL*) with respect to the first and second subscript positions. If we consider loop i as the innermost, however, reference $P(i,j,k)$ has *SL* only in the first dimension.

2.3 Memory Layout Graph

The main data structure in our representation, that is, the basis for the formulation of the ILP model, is a graph called *memory layout graph* (MLG).[2] An MLG is built from several *nest graphs* (NGs), each corresponding to a loop nest in the program. The nest graphs, in turn, are constructed from *loop graphs* (LGs). An LG is built using *node-columns* which correspond to arrays accessed in the nest that contains the loop in question. For each array, we insert a node-column into the LG. The nodes in each node-column denote array subscript positions (dimensions). As an example, for a three-dimensional array, the node-column has three nodes; the first node from the top corresponds to the first (leftmost) dimension, the second node corresponds to the second (middle) dimension, and the third node corresponds to the third (rightmost) dimension. For the sake of presentation, we assume that, in a given nest, each array is referenced only once. We show how to deal with the general case in Section 3.3.

In a given LG, the node-columns are placed one after another. The relative order of the columns is not important for the purposes of this paper. Between the node-columns P and Q (that corresponds to arrays P and Q, respectively), there are $dim(P)\times dim(Q)$ edges where $dim(.)$ returns the dimensionality (the number of subscript positions) for a given array. Put another way, the edges between P and Q connect all subscript positions of P to all subscript positions of Q. In addition to the node-columns, the LG has a start node (marked with the loop index) and a terminal node. Each node in the first node-column is connected to the start node and each node in the last node-column is connected to the terminal node. Fig. 1b shows the LG for loop i of the first loop nest, shown in Fig. 1a.

An NG, on the other hand, is obtained by replicating the LG for each loop in the nest and connecting the start nodes and the terminal nodes of the individual LGs to build a single connected graph. Fig. 1c shows the NG for the first loop nest, given in Fig. 1a. The nodes St and Tr, in Fig. 1c denote the start and terminal nodes for the NG. Similarly, Fig. 1e depicts the NG for the second loop nest given in Fig. 1a.

Finally, an MLG is constructed from the NGs, each corresponding to a nest in a set of consecutive nests in the program. Thus, an MLG can be thought of as a series (or a chain) of NGs such that the start node of the *i*th NG is connected to the terminal node of $(i-1)\text{th}$ NG. Notice that, if the program in question contains only a single nest, then its MLG is the same as the NG of the said nest. Fig. 2 shows the MLG for the program fragment, shown in Fig. 1a. Note that the MLG, once built, contains all the memory access information for every array accessed in every loop nest. It is inspired by the graph structures used by Garcia et al. [16], [17] and Kennedy and Kremer [31] to solve the automatic data distribution problem for distributed-memory message-passing architectures.

A *path* in an MLG is defined as a series of connected paths in each NG. The LG visited by the path on a specific NG, corresponds to the innermost loop in the nest in question for best locality and the nodes touched by the path correspond to the selected FCDs for the arrays accessed in the nest. As an example, Fig. 2 shows a path on the MLG from St to Tr.

In the rest of the paper, we use the terms loop (nest) and loop (nest) graph interchangeably. We abuse notation $l\in x$ to indicate that loop l belongs to the nest x. When the context is clear, we also use the terms node-column, column, and (its associated) array interchangeably.

2. In all the graphs shown in this paper, we assume that the direction of the edges is from left to right.

2.4 Node Costs

It is important to estimate the costs of the nodes in a given MLG as accurately as possible. A *node cost* in our problem is the estimation of the cache misses and the cost of a path is the sum of the costs of the nodes it contains; the edges have no costs associated with them (unless dynamic layout transformations are considered). We use notation ${V_Q}^{xl}[j]$ to denote *j*th node of a node-column for array Q in the loop graph l of the nest graph x. Then, we define $Cost({V_Q}^{xl}[j])$ as the *number of cache misses* incurred due to array Q when the dimension j is its FCD and loop l is placed in the innermost position in the nest x.

Although, several methods can be used to estimate $Cost({V_Q}^{xl}[j])$ (e.g., see [38], [14], [54], [51], [15], [56], [48] and the references therein), in our experiments, we use a slightly modified form of the approach proposed by Sarkar et al. [51] as this approach is relatively easy to implement and results in good estimations for the codes encountered in practice. Sarkar et al.'s approach assumes a perfectly nested loop nest where array references contain affine subscript expressions. The approach estimates the total number of cache misses (compulsory, capacity, and conflict misses) that will be incurred. It also takes into account the nonlocal memory accesses in a multiprocessor environment. For each array variable, it first estimates the number of distinct accesses using a heuristic bound method or an exact method depending on the specific instance of the problem. The approach determines how many loop iterations will fill the whole cache (called "locality group") and estimates the number of misses within a locality group. Then, if the reuse among distinct locality group instances is conservatively ignored, the total number of misses can be easily estimated. A more detailed description of Sarkar et al.'s approach (including a more precise definition of a "locality group") can be found in [51]. Our approach in estimating misses is also capable of performing symbolic arithmetic which allows us to handle compile-time unknown loop bounds and array sizes. Although, our miss estimation approach mainly focuses on the innermost loop, in many cases encountered in practice, the innermost loop is the loop that dictates the cache behavior.

Since we also want to consider both the first level cache (L1) and second level cache (L2) misses in a single formulation, (as node costs) we use a metric called *weighted cache misses*, which can be defined as

$$\text{L2 misses} + \frac{1}{\delta}\text{L1 misses}.$$

This metric is adapted from the concept of the weighted cache–hit ratio used by Chong et al. [8]. Note that this formulation can be extended to include an L3 cache as well. In this formulation, δ is a system dependent parameter and gives the relative access latency of the L2 cache with respect to the L1 cache. In the R10000 microprocessor that we use, δ is approximately 11. In other words, using Sarkar et al.'s [51] formulation we first estimate the L1 misses, then estimate the L2 misses, compute the metric given above, and use its value as the cost of the node in question. As an example, suppose that we have a two-level cache hierarchy, each cache having a different topology. Assume that for a reference to an array Q, we want to estimate $Cost({V_Q}^{xl}[j])$. This is the number of weighted cache misses when the FCD of array Q is set to j and loop l is placed in the innermost position in the nest x. Using the cache parameters (associativity, capacity, line size) and the bounds of the loops enclosing the reference, Sarkar's formulae can compute the number of L1 misses as, say, M_1 and the number of L2 misses as M_2. Afterwards, we can compute the number of weighted cache misses as $M_2 + (M_1/\delta)$ and assign this value as $Cost({V_Q}^{xl}[j])$. We are currently working on a more sophisticated cache miss estimation technique that can also estimate the TLB and false sharing misses accurately on shared memory parallel architectures [52].

Once the node cost estimations have been made, the rest of our approach is independent from how the estimations are made. Therefore, in order to make the description of our approach more clear and independent of the cost model, we assume that a node can have only one of three possible values: *TL*, corresponding to the number of weighted cache misses when the subscript position has *temporal locality* in the loop assuming that the said loop is innermost, *SL*, corresponding to the number of weighted cache misses when the subscript position has *spatial locality* in the loop assuming that the said loop is innermost, and *NL*, corresponding to the number of weighted cache misses when the subscript position has *no locality* in the loop, assuming that said loop is innermost.

It should be noted, however, that this is an oversimplification as, even for two arrays accessed within the same loop, the number of weighted cache misses can be drastically different depending on the sizes and the dimensionalities of the arrays as well as the memory layouts. Even two costs that appear as *SL* in our figures can be of different orders. For example, two references, such as $P(i, j)$ and $Q(i)$, will have spatial locality in the first dimension if the innermost loop in question is i. In the remainder of the paper, we refer the costs for both of these references as *SL*. In reality, depending on the array sizes, loop bounds, and cache topology, these costs can be quite different. The point here, however, is to show how to formulate the cache locality problem and solve it optimally using ILP. The detailed analysis of the cache miss estimation techniques is beyond the scope of this paper. It should be kept in mind that all the costs shown here as *TL*, *SL*, and *NL* actually correspond to the number of weighted cache misses and in our experiments the nodes are assigned appropriate costs using the techniques in Sarkar et al. [51] and the metric given above.

Fig. 1c shows the node costs for the first nest shown in Fig. 1a. Notice that no costs are associated with the start and terminal nodes and with the nodes used to connect individual LGs to make up a NG. In Fig. 1c, the cost of the first node of the column for P in the LG i is *SL* as reference $P(i, j, k)$ has spatial locality when i is innermost loop and its FCD is the first dimension. In a similar way, the first node of the column for Q in the LG j has an *NL* cost as, when j is the innermost, Q has no locality (j appears in both the subscript positions). Similarly, the node costs for the second loop nest, in Fig. 1a, are shown on the NG, in Fig. 1e.

3 Our Approach

3.1 Problem Statement

Our goal in this paper is to minimize the number of weighted cache misses thereby reducing the time spent due to memory stalls. We achieve this goal by selecting an innermost loop for each loop nest in the program and selecting a FCD for each multidimensional array accessed. Our experiments and experience show that determining the innermost loops and the FCDs is the most important step in achieving decent cache locality. The approach described here constructs a set of linear equalities and inequalities and solves the locality problem optimally using $0-1$ integer linear programming (ILP). ILP provides a set of techniques that solve those optimization problems in which both the objective function and constraints are linear functions and the variables are restricted to be integers. The $0-1$ ILP is an ILP problem in which each variable is restricted to be either 0 or 1 [42]. Our ILP formulation allows us to solve the locality problem optimally. It should be stressed, however, that this optimality is within our transformation space and cost model. As we have said, the transformation space currently contains loop permutation as loop transformation and dimension reindexing (array dimension permutation) as data transformation. Therefore, for a nest that requires diagonal-like layout(s) for the array(s) it accesses, our approach cannot derive the optimal solution. Similarly, a more accurate cost model than [51] (e.g., [57]) can result in better solutions.

In regular scientific codes, where large multidimensional arrays are accessed in different fashions in different loop nests, *remapping* actions between loop nests can increase the efficiency of the solution. Taking this observation into account, our approach also considers the dynamic layouts; that is, the layouts that can be changed during the course of a program. Notice that the meaning of the term "remapping" here is quite different from that of the same term in message-passing compilers context. Here, a remapping action is implemented as a simple *copy loop*, copying one array into another, thereby creating the effect of a layout transformation (dimension permutation). This has nothing to do with *redistributing* array portions across memories of multiple processors in a distributed-memory message-passing environment.

We also assume a linear flow of control through the loop nests of the program. While this is a common case, our approach can also be extended to handle the conditional control flow by assigning probabilities to each loop nest based on profile data. Notice that these probabilities can then be multiplied by the node costs to get *effective node* costs for the technique to use. In future, we plan to use a graph structure similar to the one used Garcia et al. [17] for automatic data-distribution problem.

3.2 Integer Variables and Objective Function

We use notation ${Y_{PQ}}^{xl}$ to denote all the $dim(P) \times dim(Q)$ edges between columns P and Q for a loop graph l of a nest graph x. Notation ${Y_{PQ}}^{xl}[i,j]$, on the other hand, denotes the *edge* between the ith subscript position of P and the jth subscript position of Q for a loop graph l of a nest graph x. We also use ${Y_{PQ}}^{xl}[i,j]$ to denote the $0-1$ *integer variable* associated with the edge in question. Given a path on the MLG (note that this means a series of paths, one for each NG x), ${Y_{PQ}}^{xl}[i,j]$ 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 ${Y_{PQ}}^{xl}[i,j]$ variable indicates whether the corresponding edge belongs to the optimal solution to the locality problem in question.

We define

$$Cost'(V_Q^{xl}[j]) = \begin{cases} Cost(V_Q^{xl}[j] & \text{if a } Y_{PQ}^{xl}[i,j] \text{ is selected} \\ & \text{where } P \text{ is the array} \\ & \quad \text{connected to} \\ & \quad Q(1 \le i \le dim(P)) \\ 0 & \text{otherwise.} \end{cases}$$

The *objective* of the locality optimization problem is then to select a *path* in the given MLG such that

$$\sum_x \sum_l \sum_Q \sum_{j=1}^{dim(Q)} Cost'(V_Q^{xl}[j]) \qquad (1)$$

is minimized,[3] where Q iterates over all the arrays accessed in x and $l \in x$. That is, the compiler should select a path such that the total cost of the selected nodes is minimum. Notice that this corresponds to selecting a data layout (actually only a FCD) for each array and selecting an innermost loop such that the total number of weighted cache misses will be minimized.

3.3 Single Nest

We first focus on a single loop nest and investigate the conditions that are required to find a correct solution. In this particular case, the MLG is simply the NG of the nest in question. There are two conditions that need to be satisfied by any correct solution.

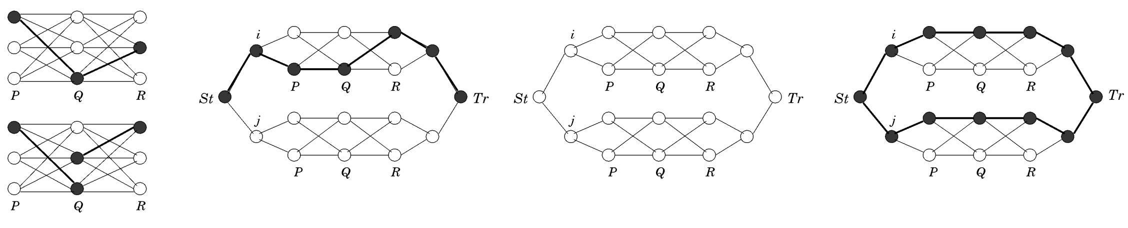
(c1) Loop Graph Condition. The selected edges and nodes should form a path. That is, whenever two nodes from two consecutive node-columns are included in a path, we should also include the edge between them. We can express this condition in terms our integer variables as

$$\forall j \in [1...dim(Q)] \qquad \sum_{i=1}^{dim(P)} Y_{PQ}^{xl}[i,j] = \sum_{k=1}^{dim(R)} Y_{QR}^{xl}[j,k].$$

This condition should be satisfied for each l of each x. Here, P, Q, and R are three arrays corresponding to three consecutive node-columns in the LGs. The nodes connected to start and terminal nodes are handled using separate equations (not given here for clarity). For example, an acceptable case is shown on the top part of Fig. 3a. The bottom part of Fig. 3a, however, shows an unacceptable case. The problem here is that for array Q two nodes are selected.

(c2) Nest Graph Condition. For a given array Q and a nest graph x of n loops, only a single loop in x can contain the selected edges. That is, the selected path should come from a single LG only. We can formalize this condition as

3. In our implementation, we have also defined ${V_Q}^{xl}[j]$ variables as 0-1 integer variables and set their value to 1 if and only if one of the incoming edges is selected. Then, $Cost'({V_Q}^{xl}[j])$ can be defined as ${V_Q}^{xl}[j]Cost({V_Q}^{xl}[j])$. Since we assume that the *Cost* values are known at compile-time, the objective function is clearly linear.



(a) (b) (c) (d)

Fig. 3. (a) (Top) An acceptable solution and (Bottom) an unacceptable solution (*Q* has no unique FCD). (b) An acceptable solution. (c) An unacceptable solution (No path is selected). (d) An unacceptable solution (Multiple paths are selected).

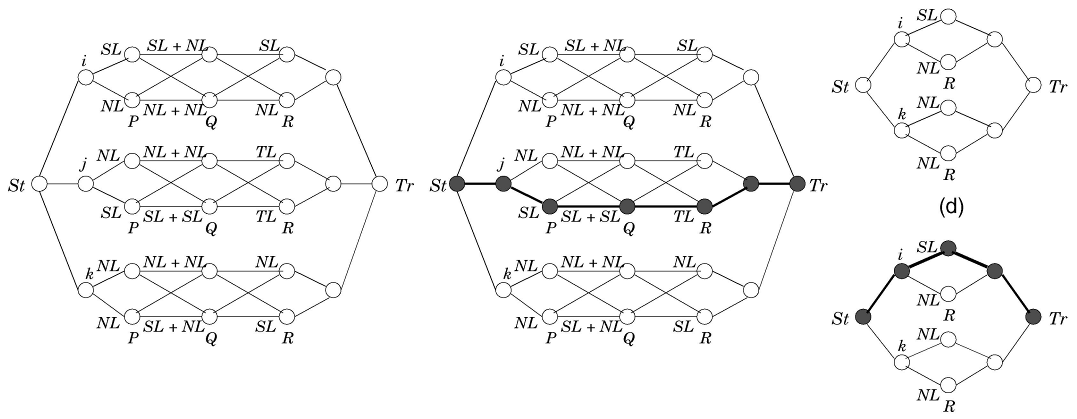
$$\sum_{i=1}^{dim(P)} \sum_{j=1}^{dim(Q)} {Y_{PQ}}^{xl_1}[i,j] + \sum_{i=1}^{dim(P)} \sum_{j=1}^{dim(Q)} {Y_{PQ}}^{xl_2}[i,j] + \cdots + \sum_{i=1}^{dim(P)} \sum_{j=1}^{dim(Q)} {Y_{PQ}}^{xl_n}[i,j] = 1,$$

where $l_1, l_2, ..., l_n$ are the loops in x. As an example, Fig. 3b shows an acceptable case, whereas Fig. 3c and Fig. 3d depict unacceptable cases. In Fig. 3d, edges from both loops (i and j) are selected (multiple paths), whereas, in Fig. 3c, no path is selected. Similarly, two optimal solution paths found by the solver for the first and second loop nests in Fig. 1a are shown in Fig. 1d and Fig. 1f, respectively. Let us now interpret these solutions. The optimal solution, in Fig. 1d, indicates that loop i should be the innermost in the first nest, layouts P, R, and S should be column-major, and layout Q should be row-major. Notice that this optimal solution is not unique as arrays R and S can assume any memory layout. The solution in Fig. 1f, on the other hand, indicates that loop j should be the innermost loop in the second nest, Q, S, and T should be row-major, and P and R should be column-major. Again, array T can assume any layout.

There are three important issues that we need to address. First, so far we have assumed that each array is referenced only once in a given loop nest. In practice this may not necessarily hold. Therefore, we need a mechanism to take the multiple-reference case into account. Our solution to this problem is rather simple. We continue to represent each array using a single node-column, but the costs of the nodes now reflect the aggregate costs of all references to the array in question. Consider the nest shown in Fig. 4a. This nest accesses array Q using two references. In the MLG shown in Fig. 4b the costs for these two references are summed. Notice that depending on the subscript functions and loop bounds, even two references to the same array can incur very different costs. Of course, in our experiments, we sum the actual costs (weighted cache misses) obtained using Sarkar et al.'s approach [51]. Fig. 4c shows an optimal solution which makes loop j innermost and assigns row-major memory layouts for all three arrays.

```
for i = li, ui
   for j = lj, uj
      for k = lk, uk
         {$P(i+k, j+k), Q(i, j+k), Q(i+k, i+j+k), R(i+k, k)$ }
      endfor
   endfor
endfor
```

(a)



(b) (c) (e)
(d)

Fig. 4. (a) An example loop nest. (b) The MLG. (c) An optimal solution. (d) The reduced MLG. (e) An optimal solution for the reduced MLG.

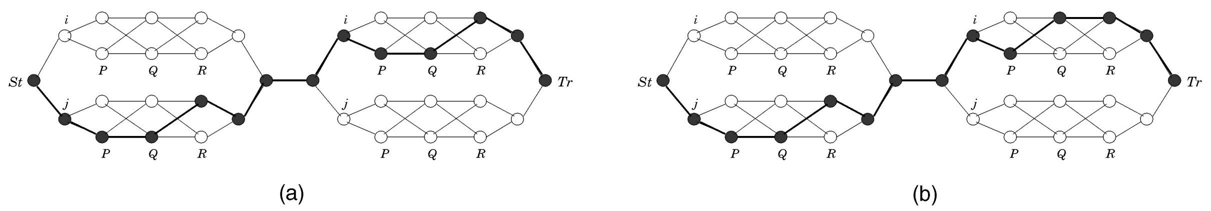


Fig. 5. (a) An acceptable solution. (b) An unacceptable solution (array Q is assigned different layouts in different nests).

The second issue to be addressed is related to temporal reuse. Notice that the memory layout for array R in Fig. 4c is selected somewhat arbitrarily as, when we make loop j innermost, reference $R(i+k,k)$ has temporal reuse in the innermost loop. We can extend our approach to determine the *next innermost* loop and to assign a possibly better layout for array R. We do this by eliminating the innermost loop (the j loop) from the MLG and all the node columns whose associated arrays have references with j in at least one of their subscript positions. The resulting *reduced MLG* is shown in Fig. 4d. When we run the solver on this reduced MLG, we obtain an optimal solution shown in Fig. 4e, which indicates that the layout of array R should be column-major rather than row-major and loop i should be the *second innermost* loop. This example shows how our approach can be used to determine a complete loop order. Notice that two applications of the solver are sufficient here to determine that the complete loop order should be k, i, and j, from outermost to innermost. It should also be noted that, after selecting an innermost loop and an FCD for a given array, if we omit the innermost loop (graph), omit *only* the FCD from the node-column of said array, and run the solver, we can determine the *second fastest changing* dimension for this array. To sum up, our approach can be easily extended to determine complete memory layout and complete loop order.

The third issue is about the data dependences [59]. Unfortunately, arbitrary permutations of the loops in a given nest can lead to incorrect programs. The data dependence theory [59], [41] can be used to determine what permutations are legal (i.e., semantics-preserving). Our approach uses this information to *prune* the search space for possible loop permutations. For example, if the compiler determines that loop l in a given nest x cannot be placed in the innermost position we can omit the LG for loop l from the NG of x, thereby reducing the size of the search space and the time to find an optimal solution.

3.4 Multiple Nests with Static Layouts

What we mean by static layouts is that the memory layouts of arrays will be fixed at specific forms for the *entire duration* of the program execution. Typically, it is very difficult to select appropriate memory layouts satisfying as many loop nests as possible simultaneously [29], [24], [43]. In the following, we discuss the conditions necessary for an optimal solution. We start by observing that the conditions (c1) and (c2) given above are also valid here for each nest (graph) in the MLG. In addition to these two conditions, in the multiple loop nest case we have the following condition that need to be satisfied by all the nests collectively:

(c3) Multiple Nest Condition. If a node of array Q is selected in nest x, the same node should also be selected in any nest x' that is different from x and that accesses array Q. In terms of our variables,

$$\begin{aligned}
&\forall j \in [1...dim(Q)]:\\
&\sum_{i=1}^{dim(P_1)} Y_{P_1Q}{}^{x_1 l_1{}^1}[i,j] + \sum_{i=1}^{dim(P_1)} Y_{P_1Q}{}^{x_1 l_2{}^1}[i,j] + \cdots\\
&= \sum_{i=1}^{dim(P_2)} Y_{P_2Q}{}^{x_2 l_1{}^2}[i,j] + \sum_{i=1}^{dim(P_2)} Y_{P_2Q}{}^{x_2 l_2{}^2}[i,j] + \cdots = \cdots\\
&= \sum_{i=1}^{dim(P_n)} Y_{P_vQ}{}^{x_v l_1{}^v}[i,j] + \sum_{i=1}^{dim(P_n)} Y_{P_vQ}{}^{x_v l_2{}^v}[i,j] + \cdots.
\end{aligned}$$

Here, $P_1, P_2, ..., P_v$ are the arrays whose node-columns are connected to that of array Q in the nests $x_1, x_2, ..., x_v$, respectively, and $l_1{}^k, l_2{}^k, ...$ are the loops in nest x_k. Fig. 5a shows an acceptable case, whereas Fig. 5b shows an unacceptable one. The problem in Fig. 5b is that, for array Q, different nodes are selected in different nests. Similarly, Fig. 2 shows a static optimal solution for the program fragment shown in Fig. 1a. One important point should be noted. In Fig. 1d when we consider the first nest alone (in an isolated manner), layout S is selected (arbitrarily) as column-major. Notice that, for this array, both column-major and row-major layouts are equally optimal as far as the locality in the innermost loop is concerned. In Fig. 2, however, the layout of this array in the first nest is forced to be row-major because the second nest requires a row-major layout for this array and we consider only static layouts. In general, an optimal solution for an MLG could be one with suboptimal solutions for some of the constituent NGs. Notice that the path shown in Fig. 2 corresponds to the solution discussed at the beginning of Section 2.2.

3.5 Multiple Nests with Dynamic Layouts

In a dynamic layout selection problem, we allow the same array to have different layouts in different loop nests provided that it is beneficial to do so from the cache locality viewpoint. Assume that x and x' are two nests (with $l \in x$ and $l' \in x'$) accessing array Q and there is *no* other nest between them which accesses Q. Let $Z_Q{}^{xlx'l'}[i,j]$ denote a *conversion edge* between the ith node of the node-column for array Q in loop l of nest x and the jth node of the node-column for array Q in loop l' of nest x'. The value of this edge is 1 if it is selected; that is, if layout Q is *dynamically transformed* between x and x', from the FCD i to the FCD j; otherwise, its value is 0.

As in the static layout selection case, (c1) and (c2) should be satisfied by the individual nests involved. In addition to those two conditions, the following condition needs to be satisfied by all the nests collectively:

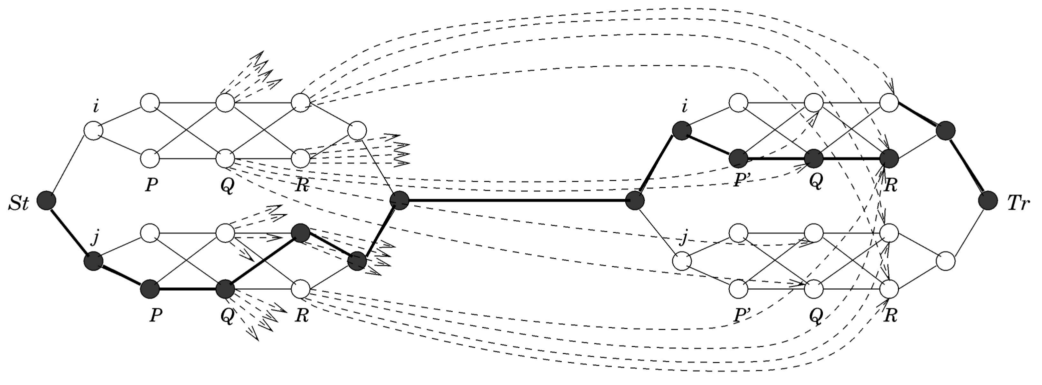


Fig. 6. A solution with the dynamic layouts.

(c3′) Multiple Nest Condition. Edge $Z_Q{}^{xlx'l'}[i,j]$ will be selected if and only if both $Y_{PQ}{}^{xl}[k,i]$ and $Y_{P'Q}{}^{x'l'}[',j]$ for a $k \in [1...dim(P)]$ and $k' \in [1...dim(P')]$ are selected. In terms of our Y and Z variables, this condition can be stated as[4]

$$
\begin{aligned}
Y_{PQ}{}^{xl}[k,i] + Y_{P'Q}{}^{x'l'}[k',j] - Z_Q{}^{xlx'l'}[i,j] &\le 1 \\
Z_Q{}^{xlx'l'}[i,j] &\le Y_{PQ}{}^{xl}[k,i] \\
Z_Q{}^{xlx'l'}[i,j] &\le Y_{P'Q}{}^{x'l'}[k',j].
\end{aligned}
$$

Here, P and P' are the arrays whose node-columns are connected to that of Q in x and x', respectively. The objective of the locality optimization problem needs to be restated now. We first define *Cost*$(Z_Q{}^{xlx'l'}[i,j])$ as the cost of converting the FCD of array Q from i (in loop l of nest x) to j (in loop l' of nest x'). Of course, if $i = j$ then the conversion cost is zero; that is, there is no dynamic layout conversion. Then, we define

$$
Cost'(Z_Q{}^{xlx'l'}[i,j]) = \begin{cases} Cost(Z_Q{}^{xlx'l'}[i,j]) & \text{if } Z_Q{}^{xlx'l'}[i,j] \\ & \text{is selected} \\ 0 & \text{otherwise.} \end{cases}
$$

The *objective* of the locality problem now is to select a path from each nest graph of a given MLG and a conversion edge for each array between pairs of nests that access the said array such that

$$
\begin{aligned}
&\sum_x \sum_l \sum_Q \sum_{j=1}^{dim(Q)} Cost'(V_Q{}^{xl}[j]) \\
&+ \sum_{x,x'} \sum_{l \in x} \sum_{l' \in x'} \sum_Q \sum_{i=1}^{dim(Q)} \sum_{j=1}^{dim(Q)} Cost'(Z_Q{}^{xlx'l'}[i,j])
\end{aligned} \tag{2}
$$

is minimized, where x, x' denotes two nests accessing array Q and have no other such nest between them.

As an example consider the solution given in Fig. 6 for a program that consists of two nests. The first nest accesses arrays P, Q, and R while the second nest accesses arrays P', Q, and R. Notice that the layout of array R is different in two nests (column-major in the first nest and row-major in the second nest). The figure also shows the conversion edges. (Due to clarity, not all the conversion edges are attached to their destinations). Assuming the optimal path shown in the figure, the conversion edge between the first node of the node-column for R in loop j of the first nest and the second node of the node-column for R in loop i of the second nest is selected. Notice that since Q and R are the only common arrays between these two nests the conversion edges are put only between their columns. In this example, we have assumed that the optimal solution does not involve any layout conversion for array Q between the nests.

An important issue now is to decide exactly at what point in the program layout transformations should take place. We believe that this problem is itself a research topic. In our experiments, we only considered two alternatives: *first point*, in which the transformations take place at the earliest point possible and *last point*, in which the transformations take place at the latest point possible. Although, there is no guarantee that these approaches will result in optimal solutions, the hope is that by transforming the data in points close to where they are used will help to improve internest temporal locality [39].

Another important issue is to determine the cost of dynamic layout transformation nest which is used to transform memory layout from a form (a FCD) into another. Since it is simply a copy loop nest, we treat this nest as an ordinary nest and use the approach proposed by Sarkar et al. [51] to estimate its cost. Notice, however, that this loop nest is a pure overhead and its cost should be minimized as much as possible. Our experience suggests two optimizations for this purpose. First, this loop nest should definitely be tiled. Notice that tiling this nest is always legal as there are no data dependences. Second, if possible, the transformations for multiple (dynamically transformed) arrays should be done using a single (or minimum number of) nest(s). Notice that this approach (which can reduce the loop overhead significantly) may contradict with the first point and last point approaches mentioned above. We believe that determining optimal transformation points for multiple arrays is an open research problem.

4. Note that the formulation problem here is simply to form a set of linear inequalities connecting three integer variables, a, b, and c, such that when the inequalities are solved, c (corresponding to the variable Z) will be one (i.e., will be selected) if and only if both a and b (corresponding to the variables Y) are one (i.e., are selected).

3.6 False Sharing

In shared memory multiprocessors, an important issue besides optimizing spatial locality is the elimination of false sharing [13], [55]. False sharing occurs when two processors access the same coherence unit (at least one of them writes) without sharing a data element. In other words, they share the coherence unit (e.g., cache line, page) but each accesses different elements in it [13], [21]. Most of the previous approaches aimed at optimizing spatial locality (e.g., [35], [24]) do not take the effect of false sharing into account. This does not cause much of a problem as long as the compiler (after locality optimizations) is able to find large granularity parallelism; that is, it is able to parallelize the outermost loop in the nest. Since, after the locality transformations, most of the reuse will be exploited in the innermost loops, parallelizing outermost loops will not cause severe false sharing. Obviously, this may not always work since it may not be possible to parallelize the outermost loops due to data dependence constraints. In those cases where a loop index that is present in the FCD of an (written) array is parallelized, it is very likely that the coherence unit will be false-shared by a number of processors. Therefore, an important rule is to *not* parallelize a loop that carries spatial reuse for an array being *written* in the nest. The effect of false sharing (when it occurs) is known to worsen with an increase in the number of processors [21] as well as with larger block sizes [55].

We now show how to extend our approach explained so far to take into account the effects of false sharing. Using data dependence analysis [59], [61], we can obtain parallelism information; that is, which loops can be parallelized. For the simplicity of presentation, we assume a single loop nest and each array that is subject to false sharing is referenced only once in the LHS and we do not consider the potential false sharing that may occur between different references. We also assume that we will parallelize at least a single loop in the nest. It is relatively straightforward to remove these restrictions. For each parallelizable loop i, whose index sits in a subscript position j of array U subject to false sharing, we associate variable $F_{i,U,j}$. The value of this variable is 1 if node j of U is selected (i.e., is part of the solution path); otherwise $F_{i,U,j}$ is 0.

Let $\{i_1, i_2, \cdots, i_h\}$ be the set of parallelizable loops where i_1 is the outermost and so forth and let $\{U_1, U_2, \cdots, U_g\}$ be the set of arrays subject to false sharing. For simplicity, we also assume that, for every array, there are m possible values for j mentioned above.

Now we impose two new conditions to ensure the correctness of the solution *in addition* to the conditions discussed earlier in the paper.

1'. A given loop i_k will be either parallelized or not. In terms of our variables, we can express this condition as

$$F_{i_k,U_l,j} = F_{i_k,U'_l,j'}$$
$$\text{for each } l, l' \in [1..g] \text{ and for each } j, j' \in [1..m].$$

2'. There will be *at least* one parallelized loop. In mathematical terms,

$$\sum_{k=1}^{h}\sum_{l=1}^{g}\sum_{j=1}^{m} F_{i_k,U_l,j} \geq 1.$$

For example, if two LHS references, $V(i,j)$ and $W(i,j,k+j)$, appear in a three-deep loop nest with loop indices i, j, k from the outermost to the innermost, we have the following conditions:

1'. $F_{i,V,1} = F_{i,W,1}$ and $F_{j,V,2} = F_{j,W,2} = F_{j,W,3}$.
2'. $F_{i,V,1} + F_{i,W,1} + F_{j,V,2} + F_{j,W,2} + F_{j,W,3} + F_{k,W,3} \geq 1$.

Let $FCost(i,U,j)$ be the additional cost in terms of false sharing misses [27] when loop i sits in subscript position j (which is the FCD) of an array U. Also let $FCost'(i,U,j)$ be $FCost(i,U,j)$ if $F_{i,U,j}$ is 1 (i.e., it is selected, meaning that loop i is parallelized and dimension j is the FCD for array U). Then, the objective is to *minimize*

$$\sum_{k=1}^{h}\sum_{l=1}^{g}\sum_{j=1}^{m} FCost'(i_k, U_l, j). \tag{3}$$

It is now clear that this cost formulation takes care of false sharing effects and can be added to the cost formulations derived earlier for good cache locality. Work is underway in deriving accurate cost formulations for the false sharing given specific architectural characteristics of the underlying parallel machine [52].

4 Experiments

In this section, we report our experimental results obtained on a single processor of an SGI Origin 2000 distributed shared memory multiprocessor. The Origin 2000 uses 195 MHz R10000 microprocessors from MIPS Technologies. Each processor has a four-way superscalar architecture, which can fetch and decode four instructions per cycle to be run on its five independent, pipelined execution units: a nonblocking load store unit, two 64-bit integer ALUs, a 32-/64-bit pipelined floating point adder, and a 32-/64-bit pipelined floating point multiplier. The R10000 has a two-level cache hierarchy. Located on the microprocessor chip are a 32 KB, two-way set associative L1 instruction cache and a 32 KB, two-way set associative, two-way interleaved L1 data cache. Off-chip is a two-way set associative, unified L2 cache which is 4 MB. The L1 data cache has a line size of 32 Bytes whereas the line size of the L2 cache is 128 Bytes. The latency ratio between the L1 and L2 caches is about 1:11.

For this study, we selected 12 programs whose characteristics are shown in Table 1. All of the programs manipulate double-precision arrays, are written in C, and compiled using the native compiler MIPSpro version 7.2.1.1. `MxM` is classic ijk matrix-multiply code and `LU` is an LU-decomposition program. The codes `Amhmtm`, `Bmcm`, `Aps`, and `Tsf` are subroutines from several programs in Perfect Club benchmarks. `Vpenta` and `Tomcatv` are from Spec. `MxMxM` is a routine from [9] that multiplies three matrices. `ADI` is one of Livermore kernels.; `Htribk` is from Eispack library. And, finally, `Transpose` is a routine from a large computational chemistry application [20]. The third column gives the number of arrays and the fourth column shows the total size of the declared arrays in MBytes. The

TABLE 1
Programs in Our Experiment Set

Program	Source	Number of Arrays	Array Sizes (MB)	Nodes		Edges		Var		Constr		Time	
				S	D	S	D	S	D	S	D	S	D
MxM	-	3	34.5	52	52	85	193	36	144	42	54	0.71	0.84
MxMxM	[9]	3	57.6	90	90	104	224	52	172	100	120	2.41	2.69
Vpenta	[11]	9	86.7	116	116	175	206	116	197	145	181	3.90	4.52
ADI	[40]	6	201.3	70	70	157	319	108	210	69	87	3.94	4.15
Transpose	[20]	2	64.0	28	28	41	73	16	48	24	32	0.52	0.68
Amhmtm	[46]	10	34.6	66	66	98	122	48	72	88	104	2.01	2.29
Bmcm	[46]	11	24.1	46	46	82	98	36	52	48	60	0.80	0.93
Aps	[46]	17	192.3	123	123	133	133	127	127	140	140	0.61	0.79
Tsf	[46]	2	45.0	36	36	69	95	36	72	46	58	0.81	0.90
LU	-	3	8.7	54	54	86	158	48	120	36	48	0.70	0.86
Tomcatv	[11]	9	14.7	66	66	116	146	74	104	68	84	0.95	2.10
Htribk	[12]	5	10.5	52	52	89	112	64	96	60	80	0.83	1.87

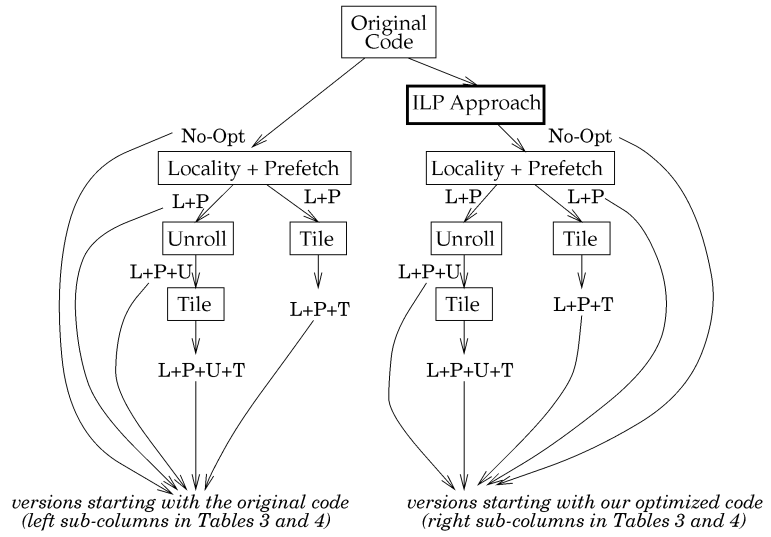

Original Code
ILP Approach
No-Opt
Locality + Prefetch
L+P
Unroll
Tile
L+P+U
L+P+T
L+P+U+T
No-Opt
Locality + Prefetch
L+P
Unroll
Tile
L+P+U
L+P+T
L+P+U+T
versions starting with the original code (left sub-columns in Tables 3 and 4)
versions starting with our optimized code (right sub-columns in Tables 3 and 4)

Fig. 7. Different versions used in the experiments.

next four columns give the number of nodes in the MLG (Nodes), the number of edges (Edges), the number of 0-1 integer variables (Var), and the number of constraints (Constr) for both static (S) and dynamic (D) optimization cases. These numbers are obtained by taking the data dependence information into account; otherwise, they would be much higher. The Time column gives the times (in *seconds*) required to find optimal solutions using the Omega library [30]. Our initial evaluation is that these times are not very high and (except for Vpenta and ADI) they will bring at most a 16 percent increase in the compilation times as compared to a faster heuristic approach [25] that uses both loop and data transformations to improve cache locality. In our experiments, the time taken to find a solution constituted at most 33 percent of the total compilation time (excluding Vpenta). Notice that the total array sizes used are larger than the L2 cache size but smaller than the node memory size, which is 256 MBytes.

Our experimental methodology, which is shown in Fig. 7, is as follows: First, we take the original (unoptimized) code and incrementally optimize it using the techniques shown in Table 2. When prefetching, tiling, and unrolling turned on, we let the native compiler select the best prefetch style, the best blocking factor, and the best unrolling factor. Then, we optimize the original (unoptimized) code using the approach proposed in this paper, and afterwards, we again apply all the optimizations shown in Table 2 taking this version as input.[5] That is, in that case, our approach acts as a front-end to the native compiler. In contrast to the previous studies, this methodology allows us to compare our layout-based approach with the production-quality loop tiling and loop unrolling techniques.

The performance results are presented in Table 3 and Table 4. The columns in these tables show the total execution cycles, miss rates (including the weighted misses—W Misses) as well as the achieved Mflops rates. All the numbers are obtained using the hardware performance counters [60] in the R10000. It is important to note that the R10000 is a complex superscalar processor (with

5. Our approach is currently being built (as a proof-of-concept implementation) on top of Parafrase-2 [47] using the Omega Library [30] as solver. It should be mentioned that the Omega library is not an ILP solver and generates only the loops that enumerate possible solutions. By putting the objective function as the first element of the tuples enumerated, after running the loops once, we obtain the solution. In future, we plan to connect our compiler to either the Cplex, Lingo, or LpSolve [4] integer programming tools. Our initial experiments with the LpSolve tool indicate that the time taken by the Omega library (to generate the loops) and the time taken by the ILP tools will be of the same orders. We believe that using the *library version* of LpSolve will reduce the time spent in finding the optimal solution.

TABLE 2
Different Versions and the Associated Native Compiler Flags

Version	Brief Explanation	Optimization Flags
No-Opt	Input program (This can be either the original code or the code obtained using our approach).	-n32 -mips4 -Ofast=ip27 -OPT:IEEE_arithmetic=3 -LNO:opt=0
L+P	A version with all loop level locality optimizations and prefetching turned on except loop unrolling and tiling.	-n32 -mips4 -Ofast=ip27 -OPT:IEEE_arithmetic=3 -LNO:blocking=off -LNO:outer_unroll=1
L+P+U	Same as the L+P version with loop unrolling turned on.	-n32 -mips4 -Ofast=ip27 -OPT:IEEE_arithmetic=3 -LNO:blocking=off
L+P+T	Same as the L+P version with loop tiling turned on.	-n32 -mips4 -Ofast=ip27 -OPT:IEEE_arithmetic=3 -LNO:outer_unroll=1
L+P+U+T	Same as the L+P version with unrolling and tiling turned on.	-n32 -mips4 -Ofast=ip27 -OPT:IEEE_arithmetic=3

nonblocking caches) in which service times for misses can get overlapped with the ongoing computation inside the chip. Therefore, the cache miss rates may not reflect the overall performance and the Mflops rates are the final criterion. However, we tried to ensure that the number of TLB misses are not significant by making the page size larger when necessary. Note, however, that, in cases where TLB misses constitute a large percentage of performance penalty, our cost formulations can be modified to take them into account. Consequently, in all of the applications, except `ADI` and `Aps`, the L1 and L2 cache misses were the main performance bottlenecks. In Table 3 and Table 4, for each column (corresponding to cycles, misses, or Mflops rates) the first (left) subcolumn denotes the versions obtained using the original (unoptimized) code as input and the second (right) subcolumn denotes the versions obtained using the code optimized using our approach as input (see Fig. 7). In two codes (`Bmcm` and `Tsf`), the solver selected the *dynamic layouts* as optimal. In `Tsf`, the static layout detection technique could not optimize the code, so in Table 4c, the first (left) subcolumn refers to the original case and the second (right) is the result of dynamic layout optimization. In Table 4a, on the other hand, the first subcolumn corresponds to the unoptimized case, the second to the static optimized version, and the third to the dynamic optimized version. We believe that this is the first approach that determines dynamic memory layouts *with* accompanying loop transformations and is a definite improvement over the previous unified approaches presented in [25], [43], [9], and [26] which do not take the possibility of dynamic memory layouts into account. We also expect that as scientific codes get larger, the importance of dynamic layout modifications (at runtime) will be more critical. This is because the larger the code, the higher the probability that we will have different code segments demanding different memory layouts for a given array. `Tomcatv` is the only example that our approach could not optimize because the locality of the original code was quite good on a single processor. From these results, we infer the following:

- The performance of unoptimized codes (the original codes with `No-Opt` version) is extremely poor. In seven out of twelve codes, the performance is below 10 Mflops.[6]
- With no optimization turned on (`No-Opt`), our approach improves the performance of the original codes on average by a factor of 15. In four codes (`Transpose`, `Aps`, `LU`, and `Htribk`) the code generated with our approach without any additional optimizations (`No-Opt`) outperforms the best compiler-optimized version (`L+P+U+T`) of the original code. The main reason for this result is that the native compiler could not improve the locality of arrays for which the layout transformations are necessary (e.g., `Transpose` and `Htribk`) and, in some codes (e.g., `Aps` and `LU`) the imperfect nest structure prevented the loop transformations, including tiling. Also, in some cases, the compiler-tiled program leads to poor intertile locality [36] as not all the spatial locality is exploited using loop transformations alone.
- Applying loop unrolling and tiling does not always improve the performance. In the versions starting with the unoptimized programs, tiling and unrolling could not improve the performance in four cases over the `L+P` version. In the versions starting with our optimized programs, the tiling could not improve the performance in two cases and the loop unrolling could not improve the performance in three cases. Saavedra et al. [50] also observed similar problems in codes optimized using tiling and prefetching together.
- When we consider the best optimized versions (`L+P+U+T`), the versions starting with our optimized code outperform the versions starting with the unoptimized codes by an average 27.5 percent, excluding two extreme cases, `Transpose` and `Aps`, in which the performance is improved by a factor of 8.5 and 7, respectively. This shows that optimizing data layouts is very important even in the cases where tiling and/or loop unrolling are applicable.
- Finally, in all the cases the best Mflops rates (shown in boldface) are obtained using the code generated taking our optimized version as input. Also, in all cases except `ADI`, `Aps`, and `Bmcm`, the best weighted miss rates (shown in boldface) correspond to the best Mflops rates indicating that the data locality plays a major role in the overall performance.

Finally, in Fig. 8, we show the MFLOPS rates for three different versions on eight processors of the Origin. `opt` is the version that uses all native compiler optimizations (`L+P+U+T`) taking the original code as input. `opt+` is the version that uses all native compiler optimizations (`L+P+U+T`) taking as input the code that is optimized using

6. This is the ratio of the "graduated loating point instructions" and the total program runtime. Note that while a multiply-add carries out two floating operations, it only counts as one instruction in the R1000 processor, so the Mflops figures reported here may underestimate the number of floating point operations per second. In any case, however, these results are pathetic for a processor with 390 Mflops peak performance.

TABLE 3
Performance Results

Version	Cycles (B)		L1 Misses (B)		L2 Misses (M)		W Misses (M)		Mflops	
No-Opt	46.77	11.50	1.94	0.44	101.66	94.9	278.02	134.90	7.22	29.38
L+P	9.88	6.76	0.45	0.43	98.25	90.95	139.16	130.04	34.57	49.74
L+P+U	5.54	4.85	0.29	0.32	48.36	47.73	74.72	76.82	61.13	69.29
L+P+T	5.72	4.11	0.06	0.06	1.15	9.04	16.60	14.49	78.23	83.44
L+P+U+T	2.56	2.24	0.05	0.05	1.08	9.07	15.63	**13.61**	132.14	**150.09**

(a)

Version	Cycles (B)		L1 Misses (B)		L2 Misses (M)		W Misses (M)		Mflops	
No-Opt	88.00	23.63	3.91	0.87	209.89	189.65	565.34	268.74	7.72	28.14
L+P	20.77	19.79	0.89	0.88	200.86	186.84	281.76	266.84	33.11	34.13
L+P+U	8.76	8.81	0.59	0.63	92.64	89.62	146.28	146.89	77.29	75.16
L+P+T	11.50	10.02	0.11	0.14	102.37	93.27	112.37	105.95	78.67	87.54
L+P+U+T	5.12	4.78	0.12	0.10	12.66	12.31	23.57	**21.40**	131.99	**146.64**

(b)

Version	Cycles (B)		L1 Misses (M)		L2 Misses (M)		W Misses (M)		Mflops	
No-Opt	0.28	0.07	15.59	6.31	0.68	0.32	2.10	0.89	12.66	51.18
L+P	0.07	0.07	2.80	5.99	0.41	0.38	0.66	0.92	63.39	51.81
L+P+U	0.05	0.05	3.32	4.80	0.76	0.33	1.06	0.77	67.04	68.53
L+P+T	0.06	0.07	3.61	2.61	0.70	0.25	1.03	0.59	66.23	59.93
L+P+U+T	0.06	0.05	3.16	3.79	0.32	0.18	0.61	**0.52**	66.26	**68.91**

(c)

Version	Cycles (B)		L1 Misses (M)		L2 Misses (M)		W Misses (M)		Mflops	
No-Opt	0.22	0.04	10.44	3.35	0.67	0.02	1.62	0.32	17.95	151.46
L+P	0.03	0.03	2.62	2.58	0.01	0.01	0.25	0.24	162.19	169.54
L+P+U	0.03	0.02	2.61	2.04	0.01	0.01	0.25	**0.20**	164.14	180.71
L+P+T	0.04	0.04	3.26	2.16	0.01	0.01	0.31	0.21	103.76	110.12
L+P+U+T	0.04	0.02	3.30	2.17	0.02	0.01	0.32	0.21	108.10	**193.39**

(d)

Version	Cycles (B)		L1 Misses (M)		L2 Misses (M)		W Misses (M)		Mflops	
No-Opt	0.21	0.02	5.09	0.92	0.27	0.01	0.73	**0.08**	9.17	**107.84**
L+P	0.19	0.02	5.42	0.85	0.26	0.01	0.75	0.09	10.47	97.44
L+P+U	0.17	0.02	5.37	0.87	0.28	0.01	0.77	0.09	11.81	96.23
L+P+T	0.19	0.02	4.71	0.91	0.25	0.01	0.68	0.09	10.49	99.14
L+P+U+T	0.19	0.02	4.97	0.89	0.25	0.01	0.70	0.09	10.37	98.15

(e)

Version	Cycles (B)		L1 Misses (B)		L2 Misses (M)		W Misses (M)		Mflops	
No-Opt	41.73	11.05	2.24	0.44	93.59	92.90	297.23	132.90	8.08	30.62
L+P	6.94	7.79	0.45	0.44	28.37	62.40	64.78	102.40	48.90	43.23
L+P+U	3.25	2.71	0.24	0.33	12.78	2.16	34.60	32.16	105.70	128.17
L+P+T	4.27	4.22	0.29	0.19	12.28	11.43	39.14	36.88	79.85	79.82
L+P+U+T	2.66	2.44	0.09	0.06	3.08	2.50	21.26	**7.95**	126.03	**141.08**

(f)

For each column—`Cycles`, `L1 Misses`, `L2 Misses`, `W Misses`, and `Mflops`—the first (left) subcolumn denotes the versions obtained using the original (unoptimized) code as input and the second (right) subcolumn denotes the versions obtained using the code optimized by our approach as input. `(B)` means in billions and `(M)` means in millions. (a) `MxM`. (b) `MxMxM`. (c) `Vpenta`. (d) `ADI`. (e) `Transpose`. (f) `Amhmtm`.

TABLE 4
Performance Results

Version	Cycles (B)			L1 Misses (B)			L2 Misses (M)			W Misses (M)			Mflops		
No-Opt	21.04	7.85	5.76	0.99	0.26	0.25	47.40	46.88	44.94	137.40	70.52	67.67	9.26	24.58	33.98
L+P	6.34	4.07	4.55	0.26	0.26	0.26	36.83	34.57	34.74	60.47	58.21	58.38	30.94	48.18	43.82
L+P+U	3.90	1.43	1.35	0.14	0.14	0.13	14.05	0.62	1.11	26.77	13.35	12.93	49.83	137.72	**147.83**
L+P+T	3.10	2.57	2.57	0.13	0.19	0.18	12.40	0.60	0.59	24.22	17.87	16.95	62.99	76.64	77.14
L+P+U+T	1.65	1.43	1.39	0.05	0.10	0.10	11.04	0.57	0.51	15.59	9.66	**9.60**	110.05	135.64	140.01

(a)

Version	Cycles (B)		L1 Misses (M)		L2 Misses (M)		W Misses (M)		Mflops	
No-Opt	2.55	0.32	54.38	8.85	1.40	1.33	6.34	2.13	7.35	56.61
L+P	2.62	0.30	52.88	8.84	1.36	1.39	6.19	2.13	7.04	**61.49**
L+P+U	2.67	0.33	52.53	9.13	1.39	1.37	6.17	2.20	6.97	54.28
L+P+T	2.68	0.38	53.02	8.63	1.38	1.27	6.20	**2.05**	6.91	60.20
L+P+U+T	2.64	0.33	54.65	9.18	1.33	1.23	6.30	2.06	7.32	58.56

(b)

Version	Cycles (B)		L1 Misses (M)		L2 Misses (M)		W Misses (M)		Mflops	
No-Opt	0.12	0.09	4.89	3.59	0.18	0.36	0.62	0.69	7.61	15.85
L+P	0.06	0.08	1.89	4.52	0.21	0.27	0.38	0.69	22.54	20.74
L+P+U	0.06	0.08	1.82	4.34	0.17	0.40	0.34	0.79	46.48	26.87
L+P+T	0.04	0.03	1.85	1.19	0.20	0.11	0.37	0.22	43.51	73.48
L+P+U+T	0.03	0.03	1.81	1.92	0.19	0.07	0.35	**0.20**	76.93	**85.30**

(c)

Version	Cycles (B)		L1 Misses (M)		L2 Misses (M)		W Misses (M)		Mflops	
No-Opt	0.34	0.24	21.50	19.46	0.39	0.11	2.34	1.88	40.28	59.94
L+P	0.28	0.22	20.47	19.09	0.23	0.13	2.09	1.87	48.88	61.86
L+P+U	0.31	0.21	19.85	19.02	0.16	0.11	1.96	1.84	47.66	63.47
L+P+T	0.31	0.19	18.35	19.01	0.21	0.13	1.88	1.86	45.23	73.10
L+P+U+T	0.28	0.19	20.39	18.92	0.15	0.10	2.00	**1.82**	48.84	**81.87**

(d)

Version	Cycles (M)		L1 Misses (M)		L2 Misses (M)		W Misses (M)		Mflops	
No-Opt	44.62	44.62	2.30	2.30	0.51	0.51	0.72	0.72	61.80	61.80
L+P	43.27	43.27	1.43	1.43	0.19	0.19	0.31	0.31	66.83	66.83
L+P+U	45.31	45.31	3.59	3.59	0.26	0.26	0.59	0.59	58.02	58.02
L+P+T	43.18	43.18	1.31	1.31	0.16	0.16	0.30	**0.30**	67.29	**67.29**
L+P+U+T	52.23	52.23	3.96	3.96	0.23	0.23	0.59	0.59	54.99	54.99

(e)

Version	Cycles (B)		L1 Misses (B)		L2 Misses (M)		W Misses (M)		Mflops	
No-Opt	4.38	1.56	0.19	0.04	10.11	4.17	27.38	7.81	44.09	65.83
L+P	3.89	1.25	0.20	0.05	9.74	1.80	27.92	6.35	46.99	88.10
L+P+U	4.03	1.05	0.20	0.04	6.61	0.50	24.79	4.14	46.38	98.36
L+P+T	4.01	1.04	0.20	0.04	6.61	0.37	24.79	4.00	46.37	99.23
L+P+U+T	3.87	1.01	0.20	0.04	5.30	0.36	23.48	**3.86**	46.95	**102.47**

(f)

For each column—`Cycles`, `L1 Misses`, `L2 Misses`, `W Misses`, and `Mflops`—the first (left) subcolumn denotes the versions obtained using the original (unoptimized) code as input and the second (right) subcolumn denotes the versions obtained using the code optimized by our approach as input. `(B)` means in billions and `(M)` means in millions. (a) `Bmcm`. (b) `Aps`. (c) `Tsf`. (d) `LU`. (e) `Tomcatv`. (f) `Htribk`.

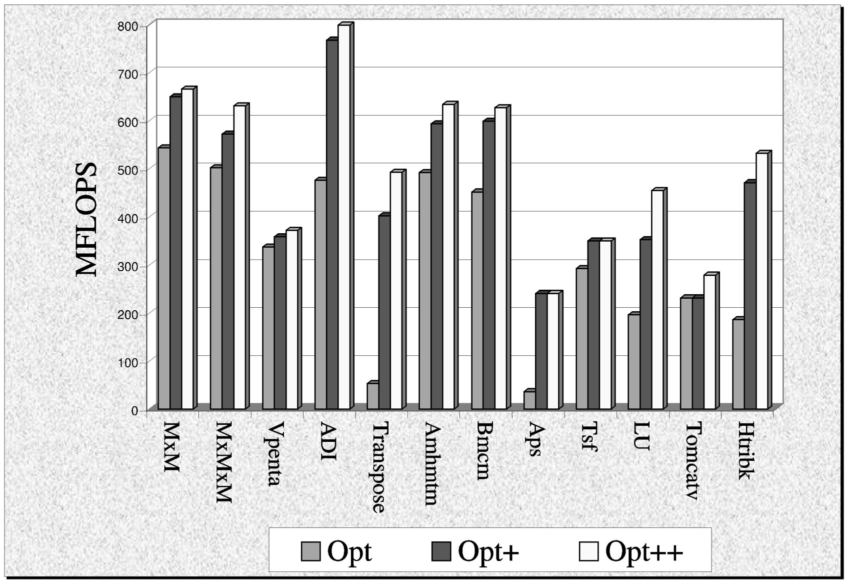


Fig. 8. MFLOPS rates for different versions.

the approach explained in this paper. It also uses dynamic layout transformations whenever necessary. These two versions do *not* perform any optimizations to specifically eliminate false sharing. `opt++` is, on the other hand, the same as `opt+` except that it also takes false sharing constraints into account as explained in Section 3.6. We observe from these results that `opt++` is roughly 60 percent better than `opt` and 9 percent better than `opt+`. The reason that there is not a large difference between the `opt+` and `opt++` versions is the fact that, in many cases, the best alternative from the false-sharing point of view is different from the data locality viewpoint. Consequently, the ILP-based approach favors one objective over the other (in many but not all cases) taking into account the costs.

From this experience, we emerge with the following suggestions for optimizing compiler implementors:

- They should consider data layout optimizations. In cases where data layout optimizations are necessary for the best performance, the nonlinear optimizations such as tiling and unrolling could not enhance the poor performance of linear loop-level transformation techniques. Codes, such as `Transpose`, `Aps`, `LU`,[7] and `Htribk` are examples supporting this claim. In particular, the cases where the spatial locality can be improved by data transformations without distorting the inherent temporal locality should be taken care of (as in `MxM` and `MxMxM` codes).
- They should focus more on imperfectly nested loops. In a number of codes in our experimental suite (e.g., `LU` and `Aps`), the potential of loop transformations could not be realized due to imperfect nests. We believe that the research for optimizing cache locality for imperfectly nested loops (e.g., Kodukula et al. [33]) is extremely important for future architectures.
- They should develop algorithms that couple loop unrolling and tiling with the linear loop and data transformations. Even in a sophisticated native compiler such as the one we used here, we have found that sometimes loop unrolling and tiling could not improve the performance over linear loop transformations. Therefore, the works such as the one done by Carr [5] for combining optimizations for cache and instruction-level parallelism are very important.
- They should attempt to combine the techniques designed for optimizing locality with those designed for minimizing false sharing.

5 Related Work

Significant work related to optimizing cache locality has been done by several research groups. We discuss the most related of this in three categories.

Loop Transformations. Exploiting the memory hierarchy through loop transformations has been the subject of several papers. Abu-Sufah et al. [1] were among the first to address the systematic use of program transformations to improve locality in the context of virtual memory systems. Based on the notion of reuse space composed of reuse vectors, Wolf and Lam [56] show how to use unimodular loop transformations followed by tiling loops that carry some form of reuse in order to improve locality. Their method is expensive (uses exhaustive search) and approximate since the reuse space cannot always be computed exactly. Li [36] uses the notion of reuse distance and the level of the loop carrying reuse, which refines the notion of reuse space. He uses a simple heuristic to derive linear loop transformations to improve reuse and was the first to demonstrate that one way to decrease the sensitivity of tiling to tile sizes is to improve spatial locality first before applying tiling. McKinley et al. [38] present a method that considers loop fusion, distribution, permutation, and reversal for improving memory locality. In addition to

7. The `LU` code that we optimize uses three arrays; the version that uses only a single array is not amenable to layout optimizations.

these linear loop transformations, tiling [6], [10], [34], [33], [58], [59] has been found to be very useful in improving memory performance. Tiling is orthogonal to these loop transformations and is typically used after these. The effectiveness of tiling is critically dependent on the values of tile sizes chosen [34], [10]; the tile size selection problem is in general difficult to solve. It is important to note that none of these above mentioned approaches considers memory layout transformations for improving locality. Our results in this paper show that layout transformations can also play a major role in overall performance of scientific codes.

Data Transformations. Recently there has been some work in using memory layout optimizations to improve spatial locality. Leung and Zahorjan [35] demonstrate cases where loop transformations fail (for a variety of reasons) for which data transformations are useful. The data transformations they consider correspond to nonsingular linear transformations of the data space. O'Boyle and Knijnenburg [43] present code generation techniques for several data optimizations, such as linear transformations for memory layouts, alignment of arrays to page boundaries, and page replication. Ju and Dietz [22] use data transformations to reduce the overhead coming from coherence activity in shared memory machines. Rivera and Tseng [49] consider array padding to eliminate conflict misses to help the program to enjoy cache locality fully. Eggers and Jeremiassen [13], [21] discuss the use of data transformations for reducing the impact of false sharing. Anderson et al. [2] derive a layout transformation—-consisting only of array dimension permutation and strip mining—for shared memory machines that ensures that the data accessed by one processor are contiguous in the shared address space; this results in enhanced spatial locality. Kandemir et al. [24] present a hyperplane representation of memory layouts of multidimensional arrays and show how to use this representation to derive very general data transformation matrices (nonsingular). Recall that the the memory layout of an array has an impact on the spatial locality characteristic of all the loop nests in the program which access the array. This implies the need for a global view of memory access patterns. The approaches to handling a sequence of loop nests presented by Kandemir et al. [24] and Leung and Zahorjan [35] rely on a conflict resolution scheme that may fail to derive the best solution when there are competing layout preferences for an array due to different accesses to the array. It is important to note that the data transformation approaches discussed above do not consider loop transformations at all. Therefore, they may fail to improve temporal locality.

Combined Loop and Data Transformations. Some authors have addressed unifying loop and data transformations into a single framework, as done in this paper. Cierniak and Li [9] use loop permutations and array dimension permutations in an exhaustive search to determine the appropriate loop and data transformations for a single nest. In earlier work [29], [25], we have addressed this problem. The approach used is based on a heuristic for processing a sequence of loop nests; this heuristic orders loop nests based on cost. The loop nests are processed in order. In processing a loop nest, the memory layouts of some arrays may be determined, which are then propagated to the next nest considered and so on. There is no guarantee of optimality here. In addition, we consider the use of loop transformations for improving temporal locality, followed by data transformations for improving spatial locality for a single nest (on uniprocessors) in a recent paper [26]. Multiple loop nests are handled using the same heuristic as in [25]. Note that none of these approaches considers dynamic layout transformations which might be useful for large scientific codes.

Our Approach. Unlike all these solutions, the framework presented in this paper formulates the problem of simultaneously determining data and loop transformations for a sequence of loop nests using integer linear programming (ILP). Thus, it avoids the use of heuristics such as ordering nests for processing. It is useful to note that determining memory layouts bears similarity to the problem of automatic data distribution in distributed-memory message-passing machines [16], [17], [18], [31], [37], [45], [53]. The solution for data and loop transformations presented in this paper uses the memory layout graph that shares many features with the graph structure used in [16], [17], we believe that it is possible to include the effects of data distribution as well.

6 Conclusions and Future Work

The performance of applications on modern processors depends on the memory access patterns to a large extent. Loop (iteration space) and memory layout (data space) transformations have been shown to be useful in improving the memory performance of loops in scientific computations. This paper presented an integer linear programming (ILP)-based approach to the problem of *detecting memory layouts of different arrays along with the best loop permutation for each loop nest in a sequence of loop nests*. This allows the handling of whole programs. Unlike other approaches that rely on ad hoc heuristics for the various subproblems, the ILP approach used here allows us to derive exact solutions (without resorting to any heuristics) to the problem. In addition, the ILP formulation allows to infer when it is beneficial to change the memory layouts of some arrays *dynamically.* Since we handle both loop and data layout transformations in a single framework, the result is significant improvements in both spatial and temporal locality. Experimental results have shown significant improvement and demonstrate the clear superiority of our solution. We have also shown in this paper how the ILP-based framework can be made to work with false-sharing constraints as well.

We are in the process of including other loop transformations, such as tiling and fusion in our framework. The investigation of the effects of our approach on tile size selection is under way. Whole program compilation requires effective interprocedural optimization as well; we plan to study this problem, focusing in particular on the formulation proposed recently by O'Boyle and Knijnenburg [44]. In addition to spatial and temporal locality, the issues of parallelism and false sharing are extremely important in distributed shared memory (DSM) machines. We are working on extending our ILP model using the memory layout graphs to better deal with these added issues. We see our work as an important first step in a multifrontal attack on the problem of optimizing the performance of large scientific codes on a variety of modern computing platforms.

Acknowledgments

The work of A. Choudhary was supported in part by US National Science Foundation Young Investigator Award CCR-9357840, US National Science Foundation grant CCR-9509143 and Air Force Materials Command under contract F30602-97-C-0026. P. Banerjee was supported in part by Defence Advanced Research Project Agency under contract F30602-98-2-0144 and by the US National Science Foundation grant CCR-9526325. J. Ramanujam was supported in part by US National Science Foundation Young Investigator Award CCR-9457768. Eduard Ayguadé was supported by the Ministry of Education of Spain under contract of the Centro de Investigación Cientifica y Technológica TIC98-511.

References

[1] W. Abu-Sufah, "Improving the Performance of Virtual Memory Computers," Ph. D. thesis, Univ. of Illinois at Urbana-Champaign, Nov. 1978.

[2] J. Anderson, S. Amarasinghe, and M. Lam, "Data and Computation Transformations for Multiprocessors," *Proc. Fifth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '95),* pp. 166-178, July 1995.

[3] E. Ayguade and J. Torres, "Partitioning the Statement Per Iteration Space Using Non-Singular Matrices," *Proc. 1993 Int'l Conf. Supercomputing (ICS '93),* July 1993.

[4] Berkelaar, "*lp_solve* version 2.1,"Available from ftp://ftp.es.ele.tue.nl/pub/lp_solve, 2001.

[5] S. Carr, "Combining Optimization for Cache and Instruction-Level Parallelism," *Proc. 1996 Int'l Conf. Parallel Architectures and Compiler Techniques (PACT '96),* Oct. 1996.

[6] L. Carter, J. Ferrante, S. Hummel, B. Alpern, and K. Gatlin, "Hierarchical Tiling: A Methodology for High Performance," Technical Report CS 96-508, Univ. of California, Santa Barbara, Nov. 1996.

[7] R. Chandra, D. Chen, R. Cox, D. Maydan, N. Nedeljkovic, and J. Anderson, "Data-Distribution Support on Distributed-Shared Memory Multiprocessors," *Proc. SIGPLAN Conf. Programming Language Design and Implementation (PLDI '97),* pp. 334-345, 1997.

[8] F.T. Chong, B.-H. Lim, R. Bianchini, J. Kubiatowicz, and A. Agarwal, "Application Performance on the MIT Alewife Machine," *Computer,* vol. 29, no. 12, pp. 57-64, Dec. 1996.

[9] M. Cierniak and W. Li, "Unifying Data and Control Transformations for Distributed Shared Memory Machines," *Proc. SIGPLAN Conf. Programming Language Design and Implementation (PLDI '95),* pp. 205-217, June 1995.

[10] S. Coleman and K. McKinley, "Tile Size Selection Using Cache Organization and Data Layout," *Proc. SIGPLAN '95 Conf. Programming Language Design and Implementation (PLDI'95),* 1995.

[11] K.M. Dixit, "New CPU Benchmark Suites from SPEC," *Proc. COMPCON '92—37th IEEE Computer Soc. Int'l Conf.,* Feb. 1992.

[12] A.A. Dubrulle, "A Version of EISPACK for the IBM 3090VF," Technical Report TR G320-3510, IBM Scientific Center, Palo Alto, Calif., 1988.

[13] S. Eggers and T. Jeremiassen, "Eliminating false Sharing," *Proc. Int'l Conf. Parallel Processing (ICPP '91),* vol. I, pp. 377-381, Aug. 1991.

[14] J. Ferrante, V. Sarkar, and W. Thrash, "On Estimating and Enhancing Cache Effectiveness," *Proc. Languages and Compilers for Parallel Computing (LCPC '91),* pp. 328-343, 1991.

[15] D. Gannon, W. Jalby, and K. Gallivan, "Strategies for Cache and Local Memory Management by Global Program Transformations," *J. Parallel and Distributed Computing,* vol. 5, no. 5, pp. 587-616, Oct. 1988.

[16] J. Garcia, E. Ayguadé, and J. Labarta, "A Novel Approach Towards Automatic Data Distribution," *Proc. Supercomputing '95,* Dec. 1995.

[17] J. Garcia, E. Ayguade, and J. Labarta, "Dynamic Data Distribution with Control Flow Analysis," *Proc. Supercomputing '96,* Nov. 1996.

[18] M. Gupta and P. Banerjee, "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers," *IEEE Trans. Parallel and Distributed Systems,* vol. 3, no. 2, pp. 179-193, Mar. 1992.

[19] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach.* second ed., San Mateo, Calif.: Morgan Kaufmann, 1995.

[20] High Performance Computational Chemistry Group. *NWChem: A Computational Chemistry Package for Parallel Computers, Version 1.1.* Richland, Wash.: Pacific Northwest Laboratory, 1995.

[21] T. Jeremiassen and S. Eggers, "Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations," *Proc. Fifth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '95),* pp. 179-188, July 1995.

[22] Y. Ju and H. Dietz, "Reduction of Cache Coherence Overhead by Compiler Data Layout and Loop Transformation," *Proc. Languages and Compilers for Parallel Computing (LCPC '92),* U. Banerjee et al., eds., pp. 344-358, 1992

[23] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and E. Ayguadé, "An Integer Linear Programming Approach for Optimizing Cache Locality," *Proc. 1999 ACM Int'l Conf. Supercomputing (ICS '99),* pp. 500–509, June 1999.

[24] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam, "A Hyperplane Based Approach for Optimizing Spatial Locality in Loop Nests," *Proc. 1998 ACM Int'l Conf. Supercomputing (ICS '98),* pp. 69-76, July 1998.

[25] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee, "A Matrix-Based Approach to the Global Locality Optimization Problem," *Proc. 1998 Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '98),* Oct. 1998.

[26] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee, "ImprovingLocality Using Loop and Data Transformations in an Integrated Approach," *Proc. 31st Ann. ACM/IEEE Int'l Symp. Microarchitecture, MICRO-31,* Dec. 1998.

[27] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee, "A Graph Based Framework to Detect Optimal Memory Layouts for Improving Data Locality," *Proc. Int'l Parallel Processing Symp. 99,* Apr. 1999.

[28] M. Kandemir, A. Choudhary, J. Ramanujam, and M. Kandaswamy, "Locality Optimization Algorithms for Compilation of Out-of-Core Codes," *J. Information Science and Eng.,* vol. 14, no. 1, pp. 107-138, Mar. 1998.

[29] M. Kandemir, J. Ramanujam, and A. Choudhary, "A Compiler Algorithm for Optimizing Locality in Loop Nests," *Proc. 11th ACM Int'l Conf. Supercomputing (ICS '97),* pp. 269-276, July 1997.

[30] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott, "The Omega Library Interface Guide," Technical Report CS-TR-3445, Computer Science Dept., Univ. of Maryland, Mar. 1995.

[31] K. Kennedy and U. Kremer, "Automatic Data Layout for High Performance Fortran," *Proc. Supercomputing '95,* Dec. 1995.

[32] I. Kodukula and K. Pingali, "Transformations of Imperfectly Nested Loops," *Proc. Supercomputing 96,* Nov. 1996.

[33] I. Kodukula, N. Ahmed, and K. Pingali, "Data-Centric Multi-Level Blocking," *Proc. Programming Language Design and Implementation (PLDI '97),* June 1997.

[34] M. Lam, E. Rothberg, and M. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '91),* 1991.

[35] S.-T. Leung and J. Zahorjan, "Optimizing Data Locality by Array Restructuring," Technical Report TR 95-09-01, Dept. of Computer Science and Eng., Univ. of Washington, Sept. 1995.

[36] W. Li, "Compiling for NUMA Parallel Machines," PhD thesis, Cornell Univ., Ithaca, NY, 1993.

[37] J. Li and M. Chen, "Compiling Communication Efficient Programs for Massively Parallel Machines," *J. Parallel and Distributed Computing,* vol. 2, no. 3, pp. 361-376, 1991.

[38] K. McKinley, S. Carr, and C. Tseng, "Improving Data Locality with Loop Transformations," *ACM Trans. Programming Languages and Systems,* vol. 18, no. 4, pp. 424-453, July 1996.

[39] K.S. McKinley and O. Temam, "A Quantitive Analysis of Loop Nest Locality," *Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '96)*, 1996.

[40] F. McMahon, "The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range," Technical Report UCRL-53745, Lawrence Livermore Nat'l Laboratory, Livermore, Calif., 1986.

[41] S.S. Muchnick, *Advanced Compiler Design Implementation.* San Francisco: Morgan Kaufmann, 1997.

[42] G. Nemhauser and L. Wolsey, *Integer and Combinatorial Optimization,* New York: John Wiley and Sons, 1988.

[43] M. O'Boyle and P. Knijnenburg, "Non-Singular Data Transformations: Definition, Validity, Applications," *Proc. Sixth Workshop Compilers for Parallel Computers (CPC '96)*, pp. 287-297, 1996.

[44] M. O'Boyle and P. Knijnenburg, "Integrating Loop and Data Transformations for Global Optimisation," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '98)*, pp. 14-17, Oct. 1998.

[45] D. Palermo and P. Banerjee, "Automatic Selection of Dynamic Data Partitioning Schemes for Distributed-Memory Multicomputers," *Proc. Eighth Workshop Languages and Compilers for Parallel Computing (LCPC '95)*, pp. 392-406, 1995.

[46] Perfect Club, "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers," *Int'l J. Supercomputing Applications,* vol. 3, no. 3, pp. 5-40, 1989.

[47] C. Polychronopoulos, M.B. Girkar, M.R. Haghighat, C.L. Lee, B.P. Leung, and D.A. Schouten, "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors," *Proc. Int'l Conf. Parallel Processing (ICPP '89),* vol. II pp. 39-48, Aug. 1989.

[48] A.K. Porterfield, "Software Methods for Improving Cache Performance on Supercomputer Applications," PhD thesis, Dept. Computer Science, Rice Univ., Houston, Texas, May 1989.

[49] G. Rivera and C. Tseng, "Data Transformations for Eliminating Conflict Misses," *Proc. 1998 ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '98),* pp. 38-49, June 1998.

[50] R.H. Saavedra, W. Mao, D. Park, J. Chame, and S. Moon, "The Combined Effectiveness of Unimodular Transformations, Tiling, and Software Prefetching," *Proc. 10th Int'l Parallel Processing Symp. (IPPS '96),* pp. 39-46, Apr. 1996.

[51] V. Sarkar, G. Gao, and S. Han, "Locality Analysis for Distributed Ahared-Memory Multiprocessors," *Proc. Ninth Int'l Workshop Languages and Compilers for Parallel Computing (LCPC '96),* Aug. 1996.

[52] N. Shenoy, M. Kandemir, D. Chakarabarti, A. Choudhary, and P. Banerjee, "Estimating Memory Access Costs on Distributed Shared Memory Multiprocessors," technical report, Northwestern Univ., Evanston, Ill., 1998.

[53] S. Tandri and T. Abdelrahman, "Automatic Partitioning of Data and Computations on Scalable Shared Memory Multiprocessors," *Proc. 1997 Int'l Conf. Parallel Processing (ICPP '97),* pp. 64-73, Aug. 1997.

[54] O. Temam, C. Fricker, and W. Jalby, "Impact of Cache Interferences on Usual Numerical Dense Loop Nests," *Proc. IEEE,* vol. 81, no. 8, pp. 1103-1115, 1993.

[55] J. Torrellas, M. Lam, and J. Hennessey, "False Sharing and Spatial Locality in Multiprocessor Caches," *IEEE Trans. Computers,* vol. 43, no. 6, pp. 651-663, June 1994.

[56] M. Wolf and M. Lam, "A Data Locality Optimizing Algorithm," *Proc. SIGPLAN Conf. Programming Language Design and Implementation (PLDI '91),* pp. 30-44, June 1991.

[57] M. Wolf, D. Maydan, and D. Chen, "Combining Loop Transformations Considering Caches and Scheduling," *Proc. 29th Ann. ACM/IEEE Int'l Symp. Microarchitecture, MICRO-29,* pp. 274-286, Dec. 1996.

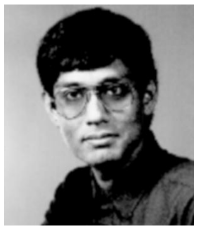
[58] M. Wolfe, More Iteration Space Tiling, *Proc. Supercomputing '89,* pp. 655-664, Nov. 1989.

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

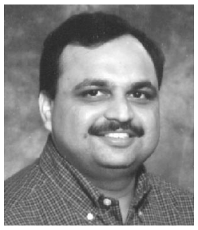
[60] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz, "Performance Analysis Using the MIPS R10000 Performance Counters," *Proc. Supercomputing '96,* Nov. 1996.

[61] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers.* New York: ACM Press, 1991.

Mahmut Kandemir received the BSc and MSc degrees in control and computer engineering from Istanbul Technical University, Istanbul, Turkey, in 1988 and 1992, respectively. He received the PhD from Syracuse University, Syracuse, New York in electrical engineering and computer science, in 1999. He has been an assistant professor in the Computer Science and Engineering Department at the Pennsylvania State University since August 1999. His main research interests are optimizing compilers, embedded computing, I/O-intensive applications, and power-aware computing. He is a member of the IEEE and the ACM.

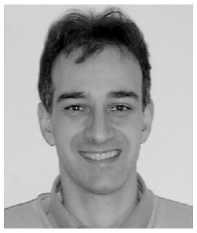


Prithviraj Banerjee received the BTech degree in electronics and electrical engineering from the Indian Institute of Technology, Kharagpur, India, in August 1981 and the MS and PhD degrees in electrical engineering from the University of Illinois at Urbana-Champaign in 1982 and 1984, respectively. He is currently the Walter P. Murphy Professor and chairman of the Department of Electrical and Computer Engineering and Director of the Center for Parallel and Distributed Computing at Northwestern University in Evanston, Illinois. Prior to which he was the director of the Computational Science and Engineering program and professor of the Electrical and Computer Engineering Department and the Coordinated Science Laboratory at the University of Illinois at Urbana-Champaign. His research interests are in Parallel Algorithms for VLSI design automation, distributed memory parallel compilers, and compilers for adaptive computing, and is the author of more than 300 papers in these areas. He leads the PARADIGM compiler project for compiling programs for distributed memory multicomputers, the ProperCAD project for portable parallel VLSI CAD applications, the MATCH project on a MATLAB compilation environment for adaptive computing, and the PACT project on power aware compilation and architectural techniques. He is also the author of a book entitled *Parallel Algorithms for VLSI CAD* (Prentice Hall 1994). He has supervised 27 PhD and 30 MS student theses so far. He has received numerous awards and honors during his career. He received the IEEE Taylor L. Booth Education Award from the IEEE Computer Society in 2001. He was the recipient of the 1996 Frederick Emmons Terman Award of ASEE's Electrical Engineering Division sponsored by Hewlett-Packard. He received the University Scholar award from the University of Illinois for in 1993, the Senior Xerox Research Award in 1992, the IEEE Senior Membership in 1990, the National Science Foundation's Presidential Young Investigators' Award in 1987, the IBM Young Faculty Development Award in 1986, and the President of India Gold Medal from the Indian Institute of Technology, Kharagpur, in 1981. He has served as the program chair of the High-Performance Computing Conference in 1999 and program chair of the International Conference on Parallel Processing for 1995. He has served as general chairman of the International Conference on Parallel and Distributed Computing Systems in 1997 and the International Workshop on Hardware Fault Tolerance in Multiprocessors, 1989. He has served on the Program and Organizing Committees of the 1988, 1989, 1993, and 1996 Fault Tolerant Computing Symposia, the 1992, 1994, 1995, 1996, and 1997 International Parallel Processing Symposium, the 1991, 1992, 1994, and 1998 International Symposia on Computer Architecture, the 1998 International Conference on Architectural Support of Programming Languages and Operating Systems, the 1990, 1993, 1994, 1995, 1996, 1997, and 1998 International Symposium on VLSI Design, the 1994, 1995, 1996, 1997, 1998, and 2000 International Conference on Parallel Processing, and the 1995, 1996, and 1997 International Conference on High-Performance Computing. He is an associate editor of the *IEEE Transactions on Parallel and Distributed Systems* and the *IEEE Transactions on Computers*. In the past he has served as associate editor of the *Journal of Parallel and Distributed Computing*, the *IEEE Transactions on VLSI Systems*, and the *Journal of Circuits, Systems and Computers*. He has been a consultant to many companies and was on the Technical Advisory Board of Ambit Design Systems. He became a fellow of the ACM in 2000 and a fellow of the IEEE in 1995.



Alok Choudhary received the PhD degree from University of Illinois, Urbana-Champaign, in electrical and computer engineering, in 1989, MS from University of Massachusetts, Amherst, in 1986 and BE (Hons.) from Birla Institute of Technology and Science, Pilani, India in 1982. He is a professor of electrical and computer engineering at Northwestern University. From 1993 to 1996, he was an associate professor in the Electrical and Computer Engineering Department at Syracuse University and from 1989 to 1993 he was an assistant professor in the same department. He has worked in industry for computer consultants prior to 1984. He received the US National Science Foundation's Young Investigator Award in 1993 (1993-1999). He also received an IEEE Engineering Foundation award, an IBM Faculty Development award, and an Intel Research Council award. His main research interests are in high-performance computing and communication systems and their applications in many domains, including multimedia systems, information processing, and scientific computing. In particular, his interests lie in the design and evaluation of architectures and software systems (from system software such as runtime systems, compilers and programming languages to applications), high-performance servers, high-performance databases, and input-output. He has published more than 130 papers in various journals and conferences in the above areas. He has also written a book and several book chapters on the above topics. His research has been sponsored by (past and present) Defence Advanced Research Project Agency, US National Science Foundation, NASA, US Air Force Office of Scientific Research, US Office of Naval Research, Department of Energy, Intel, IBM, and Texas Intruments. He served as the conference cochair for the International Conference on Parallel Processing and as a program chair and general chair for the International Workshop on I/O Systems in Parallel and Distributed Systems. He also served as program vice-chair for HiPC '1999. He is an editor of the *Journal of Parallel and Distributed Computing*, and an associate editor of *IEEE Transactions on Parallel and Distributed Systems*. He has also served as a guest editor for *Computer* and *IEEE Parallel and Distributed Technology*. He serves (or has served) on the program committee of many international conferences in architectures, parallel computing, multimedia systems, performance evaluation, distributed computing, etc. He also serves in the High-Performance Fortran Forum, a forum of Academia, Industry and Government Labs working on standardizing programming languages for portable programming on parallel computers. He is a fellow of the IEEE, a member of the IEEE Computer Society, and a member of the ACM.

J. Ramanujam (Ram) received the BTech degree in electrical engineering from the Indian Institute of Technology, Madras, India, in 1983 and the MS and PhD degrees in computer science from Ohio State University, Columbus, Ohio, in 1987 and 1990, respectively. He is currently an associate professor of electrical and computer engineering at Louisiana State University, Baton Rouge, Louisiana. His research interests are in embedded systems, compilers for high-performance computer systems, software optimizations for low-power computing, high-level hardware synthesis, parallel architectures and algorithms. He has published more than 90 papers in refereed journals and conferences in these areas in addition to several book chapters. He received the US National Science Foundation's Young Investigator Award in 1994. He has served on the Program Committees of several conferences and workshops such as the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '2001), the Workshop on Power Management for Real-Time and Embedded Systems, (IEEE Real-Time Applications Symposium, 2001), the International Conference on Parallel Architectures and Compilation Techniques (PACT 2000), the International Symposium on High Performance Computing (HiPC 99) and 1997 International International Conference on Parallel Processing. He is coorganizing a workshop on compilers and operating systems for low power, to be held in conjunction with the International Conference on Parallel Architectures and Compilation Techniques (PACT 2001) in October 2001. He coorganized the first workshop in this series as part of PACT 2000 in October 2000. He has taught tutorials on compilers for high-performance computers at several conferences such as the International Conference on Parallel Processing (1998, 1996), Supercomputing 94, Scalable High-Performance Computing Conference (SHPCC 94), and the International Symposium on Computer Architecture (1993 and 1994). He has been a frequent reviewer for several journals and conferences. He is a member of the IEEE.



Eduard Ayguadé received the engineering degree in telecommunications in 1986 and the PhD degree in computer science in 1989, both from the Universitat Politècnica de Catalunya (UPC), Spain. Since 1987, he has been lecturing on computer organization and architecture and optimizing compilers. Since 1997, he has been a full professor of the Computer Architecture Department at UPC. His research interests cover the areas of processor microarchitecture and ILP exploitation, parallelizing compilers for high-performance computing systems and tools for performance analysis and visualization. He has published more than 100 papers in these topics and participated in several long-term research projects with other universities and industries, mostly in the framework of the European Union ESPRIT and IST programmes.