An Integer Linear Programming Approach for Optimizing Cache Locality

M. Kandemir* P. Banerjee* A. Choudhary* J. Ramanujam† E. Ayguadé‡

Abstract

The actual performance of programs 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 multi-dimensional arrays from the language-defined default such as column-major storage in Fortran. These memory layout optimizations affect the spatial locality characteristics of loop nests.

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. 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.

1 Introduction

The speed of microprocessors has been steadily improving at a rate of between 50% and 100% every year, over the last decade. Unfortunately, the memory speed has not kept pace with this, improving only at the rate of about 10% per year during the same period [8]. Memory hierarchies in the form of one or more levels of cache have been used extensively in current processors in order to mitigate 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 using the caches effectively has been a difficult problem and will only get more difficult given the increasing gap between processor and memory speeds.

*CPDC, Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208, USA.

†Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803, USA.

‡Centre Europeu de Parallelisme de Barcelona (CEPBA) Dept. d'Arquitectura de Computadors, Univ. Politecnica de Catalunya Jordi Girona 1-3, Modul D6, Barcelona 08034, SPAIN.

ICS '99 Rhodes Greece

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 non-portable and difficult-to-maintain programs. The lack of automatic tools has led to much work on compiler optimizations over the last decade [26]. 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; therefore, the key issue is the conversion of this reuse into locality on the target architecture. Several compiler optimization techniques such as loop interchange [26], unimodular [25, 19] and non-unimodular [19] loop transformations, loop fusion [20], and loop tiling [17, 26, 16] 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 re-ordering 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 are constrained by data dependences [26]. In addition, transformations for imperfectly nested loops require a different solution as demonstrated by Kodukula et al. [16].

Some of these drawbacks of loop transformations have led researchers to consider changing memory layouts of arrays. Recent work [4, 10, 13, 18, 21] has addressed the use of different layouts for different arrays in a program. This has resulted in 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 as they do not change execution order [4, 10]. 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. 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-hard. Finally, some problematic constructs like array aliasing and pointers in C and the EQUIVALENCE statement in Fortran may prevent automatic data layout modifications.

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 [4, 11, 12, 22]; 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 [11] and have shown

results on using, for each loop nest, loop transformations followed by data transformations [12].

In this paper, we present a new approach that uses integer linear programming (ILP). 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. We have shown in an earlier paper that the problem of deciding the best data layouts for a single loop nest (assuming *no* loop transformations) corresponds to finding a path in a specific type of graph structure for that loop nest [13]. In this paper, we show that deciding the best combination of loop and data transformations corresponds to the solution of a certain path problem on parallel and series compositions of graphs associated to individual loops in a nest and loop nests, respectively. 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%, sometimes by as much as a factor of 8.

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 and cache locality, 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. Section 5 presents our conclusions along with a discussion of work in progress.

2 Important Concepts

2.1 Memory Layouts and Loop Transformations

We assume that the memory layout of an m-dimensional array can be in one of $m!$ forms each corresponding to the traversal of the array dimensions in some predetermined order. For a two-dimensional array, there are only two possible such 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 and similar layouts.

For each layout we associate a *fastest changing dimension* or FCD which is the dimension whose indices vary most rapidly in a sequential traversal of the array elements in memory. For example, for a row-major array the last dimension is the FCD. 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. The order of the remaining dimensions may not be as important.

As for loop transformations, we focus on general permutations of loops [26, 20]. 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 nesting order of the remaining loops in the nest can also be determined if desired.

2.2 Data Reuse and Cache Locality

When a data item (e.g., an array element) is used more than once we say that there is *temporal reuse*. A *spatial reuse* occurs when nearby data items are used. It should be noted that data reuse (temporal or spatial) is an intrinsic property of a given program and is independent of the cache architecture.

Cache locality refers to the actual realization of the inherent data reuse in a program and is strongly dependent on the cache memory structure [8, 25]. Data reuse translates to cache locality only if the subsequent uses of data occurs before the data are replaced from the cache. Since in current sequential and parallel architectures the data access time changes dramatically from level to level, it is vital to convert data reuse into cache locality.

To see how data reuse can translate to cache locality, consider the program fragment shown in Figure 1(a). This fragment contains two loop nests accessing five arrays. In this and the following program fragments used in this paper, the references inside a loop nest will be enclosed by { and }. 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. More specifically, locality for the 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 the loop i innermost in the first nest and j the 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 the locality for all references except for the 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 noted 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 an FCD for it such that the transformed innermost loop index (i.e., *after* the loop transformation) will appear either in none of the subscript positions (leading to temporal locality) or only in the subscript position corresponding to the FCD (resulting in spatial locality).

Although the reuse and locality concepts involving array references and loop orders used by the previous researchers can capture the general intuition, we need a finer granularity definition for our 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 $j_1, \cdots, j_n$ starting from the outermost loop to the 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 appears in no subscript position (including r), we say that the reference exhibits *temporal locality* with respect to rth dimension.
- If j_n appears only in 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 rth dimension[1].
- In all other cases, we say that the reference exhibits *no locality* with respect to rth dimension.

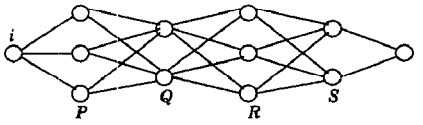
Based on these definitions, it is possible that a reference will have spatial locality with respect to a subscript position r but will not have spatial locality with respect to any other subscript position r'. In contrast, if the reference has temporal locality with respect to r,

[1] In this paper we assume that the condition $c \leq$ *cache_line_size* is always satisfied and we do not consider it further. However, it should be noted that different values of c lead to different degrees of spatial locality. The cost model we use [24] captures this aspect.

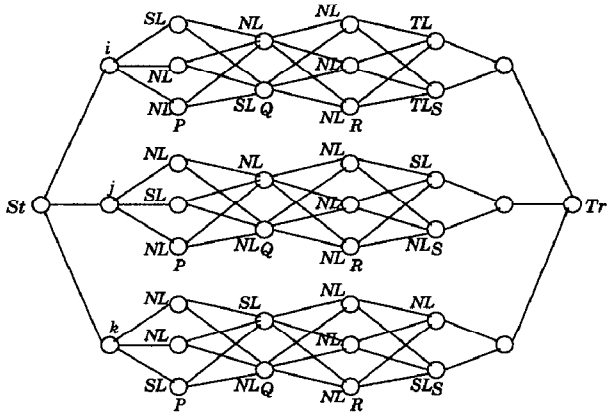
```
for i = li, ui
 for j = lj, uj
  for k = lk, uk
   {P(i, j, k), Q(j + k, i + j), R(i + j, j + k, i + k), S(j, k)}
  endfor
 endfor
endfor

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
```

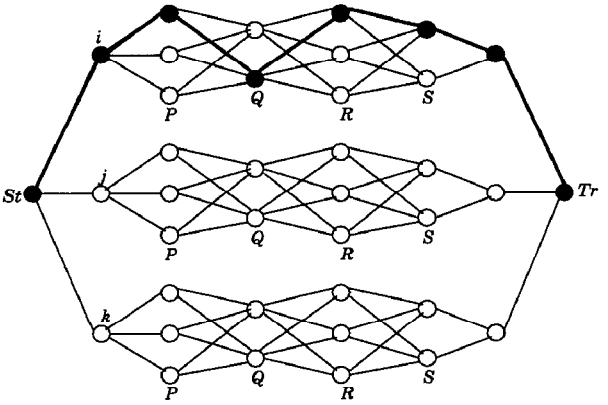
(a)



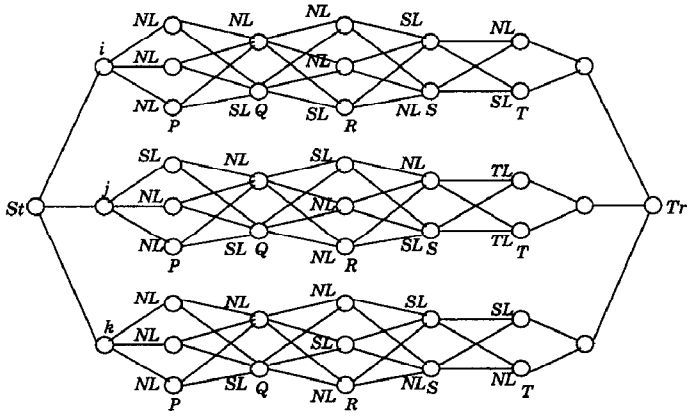
(b)



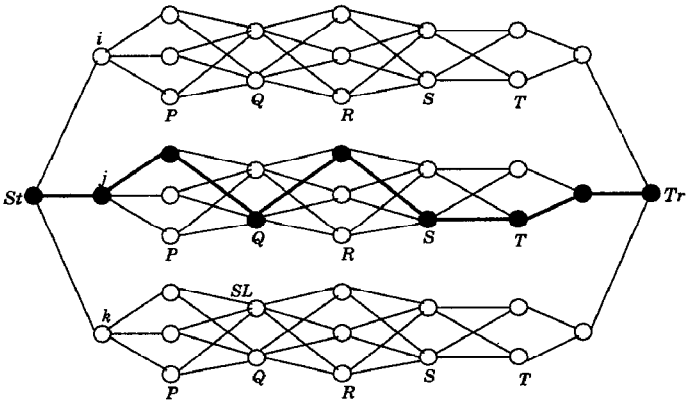
(c)



(d)



(e)



(f)

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

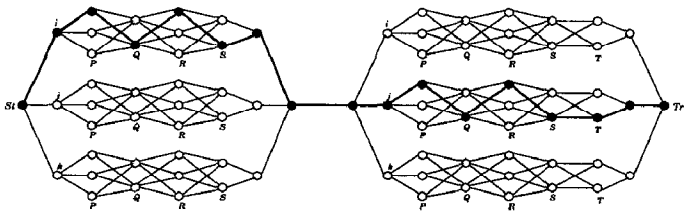


Figure 2: The MLG and an optimal solution for the program fragment shown in Figure 1(a).

it will have temporal locality with respect to all the other dimensions. It is possible to detect the localities with respect to subscript positions during the analysis of the program.

2.3 Memory Layout Graph

A graph that we call the *memory layout graph* (MLG) is the main structure in our representation and forms the basis for the ILP formulation. 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 have a node-column in the LG. The nodes in each node-column denote array subscript positions (dimensions). For 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 now 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 the 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. In other words, the edges between P and Q connect every subscript position of P to every subscript position of Q. In addition to the node-columns, an 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. Figure 1(b) shows the LG for the loop i of the first nest shown in Figure 1(a).

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

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 ith NG is connected to the terminal node of the $(i-1)$th NG. If the program in question contains only a single nest then its MLG is the same as the NG of the said nest. Figure 2 shows the MLG for the program fragment shown in Figure 1(a). 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. [7] and Kennedy and Kremer [15] 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, Figure 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. Sometimes, we use the notation $l \in x$ to indicate that the 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.

2.4 Node Costs

The cost of a node 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 the notation ${V_Q}^{xl}[j]$ to denote the jth node of a node-column for the 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 the loop l is placed in the innermost position in the nest x.

Although several methods can be used to estimate $Cost$ values (e.g., see [20], [5], [24], [6], [25]), in our experiments we use a slightly modified form of the approach due to Sarkar et al. [24]; their approach is relatively easy to implement and results in good estimations for the codes encountered in practice. 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 is defined as

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

This metric is adapted from the concept of the *weighted cache-hit ratio* used by Chong et al. [3]. Here δ is a system-dependent parameter and gives the relative access latency of the L2 cache with respect to the L1 cache. 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 the array Q is set to j and the 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 to $Cost({V_Q}^{xl}[j])$.

Once the node cost estimations have been made, the rest of our approach is independent of 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 cost 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 the said loop is innermost. However, 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. [24] and the metric given above.

Figure 1(c) shows the node costs for the first nest shown in Figure 1(a). Notice that no costs are associated with the start and terminal nodes and with the nodes used to connect individual LGs

to make up an NG. In Figure 1(c), the cost of the first node of the column for P in the LG i is SL since the reference $P(i,j,k)$ has spatial locality when i is the innermost loop and its FCD is the first dimension. The remaining costs are computed similarly.

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 the FCD for each multi-dimensional array accessed. 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. 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.

In regular scientific codes where large multi-dimensional arrays are accessed in different fashions in different loop nests, *array remapping* actions between loop nests can increase the efficiency of the solution. Taking this observation into account, our approach also considers changes in array layouts. Notice that the meaning of the term 'remapping' here is quite different from that of the same term used in the context of compilers for distributed memory message-passing machines. Here a re-mapping action is implemented as a simple *copy loop*, copying one array into another, thereby creating the effect of a layout transformation. 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 profiling data.

3.2 Integer Variables and Objective Function

We use the 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. The notation ${Y_{PQ}}^{xl}[i,j]$, on the other hand, denotes the *edge* between the i^{th} subscript position of P and the j^{th} 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, ${Y_{PQ}}^{xl}[i,j]$ 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 (i.e., it is chosen) to the locality problem in question. We define

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

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, where Q iterates over all the arrays accessed in x, and $l \in x$. That is, one needs to select a path such that the total cost of the selected nodes is minimized.

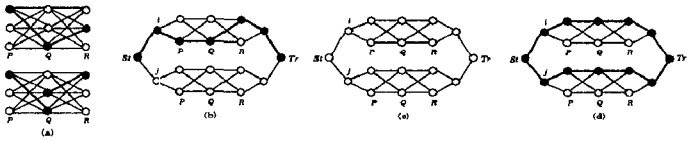


Figure 3: (a) (Top) An acceptable solution and (Bottom) an unacceptable solution (Q has no unique FCD). (b) An acceptable solution. (c) An unacceptable solution (No path is selected). (d) An unacceptable solution (Multiple paths are selected).

3.3 Single Nest

We first focus on a single loop nest and formulate the conditions that must be satisfied by any feasible 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 as

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

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 the 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 Figure 3(a). The bottom part of Figure 3(a), however, shows an unacceptable case.

(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 only from a single LG. We formalize this condition as

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

where $l_1, ..., l_n$ are the loops in x. As an example, Figure 3(b) shows an acceptable case whereas Figure 3(c) and Figure 3(d) depict unacceptable cases. Similarly, two optimal solution paths computed by the solver for the first and second loop nests from Figure 1(a) are shown in Figure 1(d) and Figure 1(f), respectively. Let us now interpret these solutions. The optimal solution in Figure 1(d) indicates that the loop i should be the innermost in the first nest and the layouts of P, R, and S should be column-major and the layout of Q should be row-major. Note that this optimal solution is not unique as the arrays R and S can assume any memory layout. The solution in Figure 1(f), on the other hand, indicates that the loop j should be the innermost loop in the second nest and Q, S, and T should be row-major and P and R should be column-major. Again, the array T can assume any layout.

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.

Another issue is the effect of data dependences on the legality of loop transformations [26]. Unfortunately, arbitrary permutations of the loops in a given nest may be illegal. The data dependence theory [26] can be used to determine what permutations are legal. Our approach uses this information to *prune* the search space of

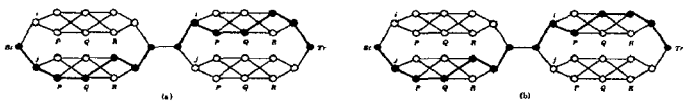


Figure 4: (a) An acceptable solution. (b) An unacceptable solution (the array Q is assigned different layouts in different nests).

possible loop permutations. For example, if it is determined that loop l in a given nest x cannot be placed in the innermost position we can omit the LG for the 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. We start by observing that 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 an array Q is selected in nest x, the same node should also be selected in every nest x' (different from x) that accesses the array Q. This is expressed as

$$\forall j \in [1...dim(Q)]:$$

$$\sum_{i=1}^{dim(P_1)} {Y_{P_1Q}}^{x_1 l_1{}^1}[i,j] + \sum_{i=1}^{dim(P_1)} {Y_{P_1Q}}^{x_1 l_2{}^1}[i,j] + \cdots =$$

$$\sum_{i=1}^{dim(P_2)} {Y_{P_2Q}}^{x_2 l_1{}^2}[i,j] + \sum_{i=1}^{dim(P_2)} {Y_{P_2Q}}^{x_2 l_2{}^2}[i,j] + \cdots =$$

$$\cdots = \sum_{i=1}^{dim(P_n)} {Y_{P_vQ}}^{x_v l_1{}^v}[i,j] + \sum_{i=1}^{dim(P_n)} {Y_{P_vQ}}^{x_v l_2{}^v}[i,j] + \cdots.$$

Here $P_1, ..., P_v$ are the arrays whose node-columns are connected to that of the array Q in the nests $x_1, ..., x_v$, respectively, and ${l_1}^k, {l_2}^k, ...$ are the loops in the nest x_k. Figure 4(a) shows an acceptable case whereas Figure 4(b) shows an unacceptable one. The problem in Figure 4(b) is that for array Q different nodes are selected in different nests. Similarly, Figure 2 shows a static optimal solution for the program fragment shown in Figure 1(a).

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 an 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 ith node of the node-column for array Q in loop l of nest x and 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 the layout of Q is dynamically transformed between x and x' from the FCD i to the FCD j; otherwise its value is zero.

As in the static layout selection case, the conditions (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.

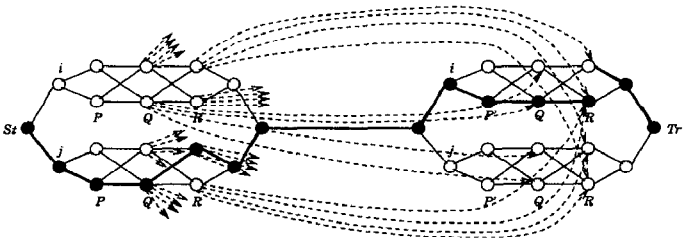


Figure 5: A solution with the dynamic layouts.

(c3') **Multiple nest condition**: An edge ${Z_Q}^{xlx'l'}[i,j]$ will be selected if and only if both ${Y_{PQ}}^{xl}[k,i]$ and ${Y_{P'Q}}^{x'l'}[k',j]$ 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

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

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 re-stated 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

$$\sum_{x}\sum_{l}\sum_{Q}\sum_{j=1}^{dim(Q)} Cost'({V_Q}^{xl}[j]) +$$

$$\sum_{x,x'}\sum_{l \in x}\sum_{l' \in x'}\sum_{Q}\sum_{i=1}^{dim(Q)}\sum_{j=1}^{dim(Q)} Cost'({Z_Q}^{xlx'l'}[i,j]) \quad (2)$$

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

As an example consider the solution given in Figure 5 for a program that consists of two nests. The first nest accesses the arrays P, Q, and R while the second nest accesses the arrays P', Q, and R. Notice that the layout of the 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 the loop j of the first nest and the second node of the node-column for R in the loop i of the second nest is selected. Note 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 the array Q.

An 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. [24] to estimate its cost. Notice, however, that this loop nest is a pure overhead and its cost should be minimized as much as possible.

4 Experimental Results

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 R10K microprocessors from MIPS Technologies. Each processor has a two-level cache hierarchy. Located on the microprocessor chip is a 32 KB, two-way set associative L1 data cache. Off-chip is a two-way set associative L2 cache which is 4 MB. The latency ratio between the L1 and L2 caches is about 1:5.

For this study we selected 12 programs whose characteristics are shown in Table 1. All of these are C programs and are compiled using the native compiler MIPSpro version 7.2.1.1m. `MxM` is classical ijk matrix-multiply code and `LU` is an LU-decomposition program. The `Array Sizes` column gives the total size of the declared arrays in MBytes. The 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 [14]. Our initial evaluation is that these times are not very high and (except for `Vpenta` and `ADI`) they will bring at most a 16% increase in the compilation times as compared to a faster heuristic approach [11] 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% 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 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 optimizations are turned on, we let the commercial compiler to 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[2]. That is, in that case our approach acts as a front-end to the commercial compiler.

The performance results are presented in Tables 3 and 4. The first column in these tables (`V`) gives the version number (see Table 2). The remaining columns show the total execution cycles (`Cyc`), L1 miss rates (`L1`), L2 miss rates (`L2`), weighted miss rates (`W`), and the achieved Mflops rates (`Mflops`). All the numbers are obtained using the hardware performance counters in the R10K. In all of the applications except `ADI` and `Aps` the L1 and L2 cache misses were the main performance bottlenecks. In Tables 3 and 4, for each column (corresponding to cycles, misses or Mflops rates) the first (left) sub-column denotes the versions obtained using the original (unoptimized) code as input and the second (right) sub-column denotes the versions obtained using the code optimized using our approach as input. 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 4(c) the first (left) sub-column refers to the original case and the second (right) is the result of dynamic layout optimization. Similarly, in Table 4(a) the first sub-column corresponds to the unoptimized case and the second 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 [11], [21], [4], and [12] which do not take the possibility of dynamic memory layouts into account. From these results, we infer the following:

- The performance of unoptimized codes (the original codes with `NO` version) is extremely poor. In seven out of twelve codes the performance is below 10 Mflops.
- With no optimization turned on (`NO`), our approach improves the performance of the original codes in 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`) outperforms the best compiler optimized version (`LPUT`) of the original code. The main reason for this result is that the commercial compiler could not improve the locality of arrays for which the layout transformations are necessary (e.g., `Transpose` and `Htribk`); and also in some codes (e.g., `Aps` and `LU`) the imperfect nest structure prevented the loop transformations including tiling.
- Applying loop unrolling and tiling does not always improve the performance. In the versions that start from the unoptimized programs, tiling and unrolling could not improve the performance in four cases over the `LP` version. In the versions that start 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. [23] also observe similar problems in codes optimized using tiling and prefetching together.
- When we consider the best optimized versions (`LPUT`), the versions that start with our optimized code outperform the versions starting with the unoptimized codes by an average 27.5%, 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.

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

1. They should consider data layout optimizations. In cases where data layout optimizations are necessary for the best performance, the non-linear optimizations such as tiling and unrolling could not enhance the poor performance of linear loop-level transformation techniques. The codes such as `Transpose`, `Aps`, `LU`[3], and `Htribk` are examples supporting this claim.

[2] Our approach is currently being built (as a proof-of-concept implementation) using the Omega Library [14] 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 standard ILP solvers such as CPLEX or LpSolve [1]. 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 are of the order. We believe that using the *library version* of LpSolve will reduce the time spent in finding the optimal solution.

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

Table 1: Programs in our experiment set.

Program	Source	# of Arrays	Array Sizes	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	[4]	3	57.6	90	90	104	224	52	172	100	120	2.41	2.69
Vpenta	Spec	9	86.7	116	116	175	206	116	197	145	181	3.90	4.52
ADI	Livermore	6	201.3	70	70	157	319	108	210	69	87	3.94	4.15
Transpose	NWchem (PNL)	2	64.0	28	28	41	73	16	48	24	32	0.52	0.68
Amhmtm	Perfect Club	10	34.6	66	66	98	122	48	72	88	104	2.01	2.29
Bmcm	Perfect Club	11	24.1	46	46	82	98	36	52	48	60	0.80	0.93
Aps	Perfect Club	17	192.3	123	123	133	133	127	127	140	140	0.61	0.79
Tsf	Perfect Club	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	Spec	9	14.7	66	66	116	146	74	104	68	84	0.95	2.10
Htribk	Eispack	5	10.5	52	52	89	112	64	96	60	80	0.83	1.87

Table 2: Different versions (numbers) and the associated compiler flags.

Ver. No (V)	Ver.	Brief Explanation	Optimization Flags
1	NO	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
2	LP	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
3	LPU	Same as the L+P version with loop unrolling turned on.	-n32 -mips4 -Ofast=ip27 -OPT:IEEE_arithmetic=3 -LNO:blocking=off
4	LPT	Same as the L+P version with loop tiling turned on.	-n32 -mips4 -Ofast=ip27 -OPT:IEEE_arithmetic=3 -LNO:outer_unroll=1
5	LPUT	Same as the L+P version with unrolling and tiling turned on.	-n32 -mips4 -Ofast=ip27 -OPT:IEEE_arithmetic=3

2. 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. [16]) is extremely important for future architectures.

3. They should develop algorithms that couple loop unrolling and tiling with the linear loop and data transformations better. Even in a sophisticated commercial compiler like MIPSpro 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 [2] for combining optimizations for cache and instruction-level parallelism are very important.

5 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 computation. 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 sub-problems, 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.

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 inter-procedural optimization as well; we plan to study this problem, focusing in particular on the formulation proposed recently by O'Boyle and Knijnenburg [22]. In addition to spatial and temporal locality, the issues of parallelism and false sharing are extremely important in distributed shared memory (DSM) machines. In this case, we plan to investigate accurate estimation techniques for TLB and false sharing misses. In addition, we plan to work on extending our framework to deal with more general layouts such as diagonal layouts. We are working on extending our ILP model using the memory layout graphs to deal with these added issues. In the case of message-passing machines, we plan to solve the data mapping and the memory layout problems simultaneously. Thus, we see our work as an important step in a multi-frontal attack on the problem of optimizing the performance of large scientific codes on a variety of modern computing platforms.

Acknowledgments

The work of M. Kandemir and A. Choudhary were supported in part by NSF Young Investigator Award CCR-9357840, NSF grant CCR-9509143 and Air Force Materials Command under contract F30602-97-C-0026. P. Banerjee was supported in part by DARPA under contract F30602-98-2-0144 and by NSF grant CCR-9526325. J. Ramanujam was supported in part by NSF Young Investigator Award CCR-9457768. Eduard Ayguade was supported by the Ministry of Education of Spain under contract of the CICYT TIC98-511.

References

[1] Berkelaar, *lp_solve* version 2.1, Available through anonymous ftp from ftp://ftp.es.ele.tue.nl/pub/lp_solve.

[2] S. Carr. Combining optimization for cache and instruction-level parallelism. In *Proc. the 1996 International Confer-*

ence on Parallel Architectures and Compiler Techniques (PACT'96), Boston MA, Oct 1996.

[3] F. T. Chong, B-H. Lim, R. Bianchini, J. Kubiatowicz, and A. Agarwal. Application performance on the MIT Alewife machine. *IEEE Computer,* Vol. 29, No. 12, December 1996, pp. 57–64.

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

[5] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *Proc. Languages and Compilers for Parallel Computing (LCPC'91)*, pages 328–343, 1991.

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

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

[8] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Second edition, Morgan Kaufmann Publishers, San Mateo, CA, 1995.

[9] High Performance Computational Chemistry Group. NWChem: A computational chemistry package for parallel computers, version 1.1, Pacific Northwest Laboratory, Richland, WA 99352, 1995.

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

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

[12] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated approach. In *Proc. MICRO-31,* Dallas, TX, December 1998.

[13] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A graph based framework to detect optimal memory layouts for improving data locality. In *Proc. IPPS 99,* San Juan, Puerto Rico, April 1999.

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

[15] K. Kennedy and U. Kremer. Automatic data layout for High Performance Fortran. In *Proc. Supercomputing'95*, San Diego, CA, December 1995.

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

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

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

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

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

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

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

[23] R. H. Saavedra, W. Mao, D. Park, J. Chame, and S. Moon. The combined effectiveness of unimodular transformations, tiling, and software prefetching. In *Proc. 10th International Parallel Processing Symposium (IPPS'96)*, Honolulu, Hawaii, April 15–19, 1996, pp. 39–46.

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

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

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

Table 3: Performance Results. For each column—Cyc, L1, L2, W misses and Mflops—the first (left) sub-column denotes the versions obtained using the original (unoptimized) code as input and the second (right) sub-column denotes the versions obtained using the code optimized by our approach as input. (B) means *in billions* and (M) means *in millions*.

(a): MxM

V	Cyc (B)		L1 (B)		L2 (B)		W (B)		Mflops	
1	46.77	11.50	1.94	0.44	0.10	0.10	0.28	0.23	7.2	29.4
2	9.88	6.76	0.45	0.43	0.10	0.09	0.14	0.13	34.6	49.7
3	5.54	4.85	0.29	0.32	0.05	0.05	0.08	0.08	61.1	69.3
4	5.72	4.11	0.06	0.06	0.01	0.01	0.02	0.01	78.2	83.4
5	2.56	2.24	0.05	0.05	0.01	0.01	0.02	**0.01**	132.1	**150.1**

(b): MxMxM

V	Cyc (B)		L1 (B)		L2 (B)		W (B)		Mflops	
1	88.0	23.6	3.91	0.87	0.21	0.19	0.57	0.27	7.7	28.1
2	20.8	19.8	0.89	0.88	0.20	0.19	0.28	0.27	**33.1**	34.1
3	8.8	8.8	0.59	0.63	90.09	0.09	0.15	0.15	77.3	75.2
4	11.5	10.0	0.11	0.14	0.10	0.09	0.11	0.10	78.7	87.5
5	5.1	4.8	0.12	0.10	0.01	0.01	0.02	**0.02**	132.0	**146.6**

(c): Vpenta

V	Cyc (B)		L1 (M)		L2 (M)		W (M)		Mflops	
1	0.28	0.07	15.59	6.31	0.68	0.32	2.10	0.89	12.7	51.2
2	0.07	0.07	2.80	5.99	0.41	0.38	0.66	0.92	63.4	51.8
3	0.05	0.05	3.32	4.80	0.76	0.33	1.06	0.77	67.0	68.5
4	0.06	0.07	3.61	2.61	0.70	0.25	1.03	0.59	66.2	59.9
5	0.06	0.05	3.16	3.79	0.32	0.18	0.61	**0.52**	66.3	**68.9**

(d): ADI

V	Cyc (B)		L1 (M)		L2 (M)		W (M)		Mflops	
1	0.22	0.04	10.44	3.35	0.67	0.02	1.62	0.32	18.0	151.5
2	0.03	0.03	2.62	2.58	0.01	0.01	0.25	0.24	162.2	169.5
3	0.03	0.02	2.61	2.04	0.01	0.01	0.25	**0.20**	164.1	180.7
4	0.04	0.04	3.26	2.16	0.01	0.01	0.31	0.21	103.8	110.1
5	0.04	0.02	3.30	2.17	0.02	0.01	0.32	0.21	108.1	**193.4**

(e): Transpose

V	Cyc (B)		L1 (M)		L2 (M)		W (M)		Mflops	
1	0.21	0.02	5.09	0.92	0.27	0.01	0.73	**0.08**	9.2	**107.8**
2	0.19	0.02	5.42	0.85	0.26	0.01	0.75	0.09	10.5	97.4
3	0.17	0.02	5.37	0.87	0.28	0.01	0.77	0.09	11.8	96.2
4	0.19	0.02	4.71	0.91	0.25	0.01	0.68	0.09	10.5	99.1
5	0.19	0.02	4.97	0.89	0.25	0.01	0.70	0.09	10.4	98.2

(f): Amhmtm

V	Cyc (B)		L1 (B)		L2 (M)		W (B)		Mflops	
1	41.7	11.1	2.24	0.44	93.59	92.90	0.30	0.13	8.1	30.6
2	6.9	7.8	0.45	0.44	28.37	62.40	0.06	0.10	48.9	43.2
3	3.3	2.7	0.24	0.33	12.78	2.16	0.03	0.03	105.7	128.2
4	4.3	4.2	0.29	0.19	12.28	11.43	0.04	0.04	79.9	79.8
5	2.7	2.4	0.09	0.06	3.08	2.50	0.02	**0.01**	126.0	**141.1**

Table 4: Performance Results. For each column—Cyc, L1, L2, W misses and Mflops—the first (left) sub-column denotes the versions obtained using the original (unoptimized) code as input and the second (right) sub-column denotes the versions obtained using the code optimized by our approach as input. (B) means *in billions* and (M) means *in millions*.

(a): Bmcm

V	Cyc (B)		L1 (B)		L2 (M)		W (B)		Mflops	
1	21.0	5.8	0.99	0.25	47.40	44.94	0.14	0.07	9.3	34.0
2	6.3	4.6	0.26	0.26	36.83	34.74	0.06	0.06	30.9	43.8
3	3.9	1.4	0.14	0.13	14.05	1.11	0.03	0.01	49.8	**147.8**
4	3.1	2.6	0.13	0.18	12.40	0.59	0.02	0.01	63.0	77.1
5	1.7	1.4	0.05	0.10	11.04	0.51	0.01	**0.01**	110.0	140.0

(b): Aps

V	Cyc (B)		L1 (M)		L2 (M)		W (M)		Mflops	
1	2.55	0.32	54.38	8.85	1.40	1.33	6.34	2.13	7.4	56.6
2	2.62	0.30	52.88	8.84	1.36	1.39	6.19	2.13	7.0	**61.5**
3	2.67	0.33	52.53	9.13	1.39	1.37	6.17	2.20	7.0	54.3
4	2.68	0.38	53.02	8.63	1.38	1.27	6.20	**2.05**	6.9	60.2
5	2.64	0.33	54.65	9.18	1.33	1.23	6.30	2.06	7.3	58.6

(c): Tsf

V	Cyc (B)		L1 (M)		L2 (M)		W (M)		Mflops	
1	0.12	0.09	4.89	3.59	0.18	0.36	0.62	0.69	7.6	15.9
2	0.06	0.08	1.89	4.52	0.21	0.27	0.38	0.69	22.5	20.7
3	0.06	0.08	1.82	4.34	0.17	0.40	0.34	0.79	46.5	26.9
4	0.04	0.03	1.85	1.19	0.20	0.11	0.37	0.22	43.5	73.5
5	0.03	0.03	1.81	1.92	0.19	0.07	0.35	**0.20**	76.9	**85.3**

(d): LU

V	Cyc (B)		L1 (M)		L2 (M)		W (M)		Mflops	
1	0.34	0.24	21.50	19.46	0.39	0.11	2.34	1.88	40.3	59.9
2	0.28	0.22	20.47	19.09	0.23	0.13	2.09	1.87	48.9	61.9
3	0.31	0.21	19.85	19.02	0.16	0.11	1.96	1.84	47.7	63.5
4	0.31	0.19	18.35	19.01	0.21	0.13	1.88	1.86	45.2	73.1
5	0.28	0.19	20.39	18.92	0.15	0.10	2.00	**1.82**	48.8	**81.9**

(e): Tomcatv

V	Cyc (M)		L1 (M)		L2 (M)		W (M)		Mflops	
1	44.62	44.62	2.30	2.30	0.51	0.51	0.72	0.72	61.8	61.8
2	43.27	43.27	1.43	1.43	0.19	0.19	0.31	0.31	66.8	66.8
3	45.31	45.31	3.59	3.59	0.26	0.26	0.59	0.59	58.0	58.0
4	43.18	43.18	1.31	1.31	0.16	0.16	0.30	**0.30**	67.3	**67.3**
5	52.23	52.23	3.96	3.96	0.23	0.23	0.59	0.59	55.0	55.0

(f): Htribk

V	Cyc (B)		L1 (B)		L2 (M)		W (M)		Mflops	
1	4.38	1.56	0.19	0.04	10.11	4.17	27.38	7.81	44.1	65.8
2	3.89	1.25	0.20	0.05	9.74	1.80	27.92	6.35	47.0	88.1
3	4.03	1.05	0.20	0.04	6.61	0.50	24.79	4.14	46.4	98.4
4	4.01	1.04	0.20	0.04	6.61	0.37	24.79	4.00	46.4	99.2
5	3.87	1.01	0.20	0.04	5.30	0.36	23.48	**3.86**	47.0	**102.5**