A Framework for Interprocedural Locality Optimization Using Both Loop and Data Layout Transformations

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

Abstract

There has been much work recently on improving the locality performance of loop nests in scientific programs through the use of loop as well as data layout optimizations. However, little attention has been paid to the problem of optimizing locality in whole programs, particularly in the presence of procedures. Current techniques do not propagate layout optimizations across procedures boundaries; this is critical for realistic scientific codes, since the cost of explicitly transforming memory layouts across procedure boundaries might be very high. In this paper we present a locality optimization framework that uses both loop and data transformations to improve cache locality program-wide. Our framework propagates layout (or locality) constraints as a system of equalities across procedures and involves two traversals in the call graph representation of the program. Preliminary experimental results obtained on an R10000 based system demonstrate the power of the framework.

1 Introduction

A key challenge in achieving high levels of performance on modern computer systems is the reduction of the time spent stalled waiting for data from memory as much as possible. Several architectural advances in memory hierarchy design has led to systems with multiple levels of memory hierarchy. Exploiting the memory hierarchy has become the most important problem in realizing the performance potential of modern machines. In the area of scientific computation, efforts have been aimed at the development of portable library routines such as LAPACK [3] in order to alleviate the difficulty if exploiting the memory hierarchy. Nevertheless, getting good performance remains difficult and we believe that this task is best left to optimizing compilers.

There has been much work on cache locality optimization techniques for loop nests. One group of these optimizations aim to reorder the iterations of a loop to improve both temporal and spatial locality; these include unimodular [31] and non-unimodular [25] linear loop (iteration space) transformations, loop distribution [27], fusion [27], and tiling [32, 31, 6, 22, 23]. These are limited by dependence constraints and are not readily applicable to imperfectly nested loops [8]. A second group consists of transformations that modify the memory storage order [8, 29, 30, 24, 17] of multi-dimensional arrays, referred to as *data transformations.* These are not constrained by dependences and are applicable to imperfect nests but have no effect on temporal locality; in addition, the effect of changing the memory layout of an array is program-wide. This has led to the use of combined loop and data transformations [29, 21, 18, 19].

Except for a few papers [10, 30], the impact of data reorganization necessitated at procedure boundaries has not received much attention. This is unfortunate, since practical codes contain procedure calls and the cost of data reorganization is very high. In this paper we present a locality optimization framework that uses both loop and data transformations to improve cache locality program-wide. Our framework propagates layout (or locality) constraints as a system of equalities across procedures and involves two traversals in the call graph representation of the program. We also show how to handle the cases where the callers of the same procedure demand conflicting memory layouts for the same array. Preliminary experimental results obtained on an R10000 based system demonstrate the power of the framework.

The rest of this paper is organized as follows. In Section 2 we present an outline of our approach along with details on the intra-procedural optimization framework. Section 3 describes in detail our solution—the bottom-up and the top top-down traversal techniques—to the inter-procedural locality optimization problem. In Section 4 we report performance results obtained on an SGI Origin 2000 distributed-shared-memory multi-processor. In Section 5 we review related work on compiler-based locality optimizations. In Section 6 we present our conclusions which are followed by a brief outline of on-going work.

2 Our Approach

Our approach performs two traversals on the *call graph* representation of the program. A call graph $G_c = (V_c, E_c)$ is a multi-graph where each node $p_i \in V_c$ represents a procedure and there is an edge $e \in E_c$ between p_i and p_j if p_i calls p_j [2]. In such a graph the leaves represent the procedures that do not contain any calls. If desired, the edges can be annotated by suitable information related to call sites such as the actual parameters passed to the procedure, the line number where the call occurs and so on.

Before the first traversal, we run an intra-procedural locality optimization algorithm on each leaf node. The details of this algorithm are explained in Section 2.1. In the first traversal, called *bottom-up*, we start with the leaves and process each node in the call graph if and only if all the nodes it calls have been processed. After all the callee nodes for a given caller have been processed, we propagate a system of equalities (called the layout or locality constraints) to the caller. The caller adds this system to its own local set of equalities (obtained using the intra-procedural locality optimization algorithm) and propagates the resulting system to its callers and so on. This bottom-up traversal is discussed in greater detail in Section 3.1. When we reach the root (the main program), we have all the locality constraints of the program. We solve these constraints at root and determine the layouts of the (global and local) arrays accessed by the root. The next step is the top-down traversal; in this traversal, each caller propagates down the layouts determined so far to its callees. The algorithm terminates when all the leaf nodes have been processed. The details of the top-down traversal are given in Section 3.2. Note that in this paper we assume that either array re-shaping does not occur or when it occurs it is possible to undo its effect using de-linearization [26].

2.1 Intra-procedural Locality Optimization Algorithm

This subsection presents our intra-procedural *static* locality optimization algorithm. While it resembles the previous approaches in

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

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

spirit [18, 19], it has the advantage of looking at the big picture before starting to solve the problem; if desired, the previous approaches can also be represented in our framework. The algorithm determines static memory layouts in the sense that there is a single memory layout for each array throughout the entire procedure being analyzed.

2.1.1 Background

An n-deep loop nest with loop indices i_1,i_2,...,i_n in the program is represented by an integer polyhedron bounded by the loop limits. Each point in this polyhedron is represented by a vector $(i'_1, i'_2, ..., i'_n)^T$ and corresponds to an execution of the loop body when $i_k = i'_k$ for all $1 \leq k \leq n$; $\bar{I} = (i_1, i_2, ..., i_n)^T$ is called the *iteration vector*, where i_1 is the outermost loop index and i_n is the innermost loop index. Similarly the memory storage of an m-dimensional array can also be viewed as a (rectilinear) polyhedron. The extents of the array determine the bounds of the polyhedron and each point (array element) can be indexed using a column vector $(j_1, j_2, ..., j_m)^T$.

We assume that the array subscript expressions and loop bounds are affine functions of enclosing loop indices and loop-invariant constants; that is, we are assuming an affine loop nest. Under this assumption, in an n-deep loop nest, each reference to an m-dimensional array can be modeled by an *access matrix* $\mathcal{L}$ of size $m \times n$ and an m-dimensional *offset vector* $\bar{o}$ [25, 31, 33], i.e., modeled as $\mathcal{L}\bar{I} + \bar{o}$, where $\bar{I}$ is the iteration vector. In this paper $\mathcal{L}_{uij}$ ($\bar{o}_{uij}$) denotes the jth reference matrix (offset vector) for array U in nest i.

For such a loop nest, we consider an iteration space transformation [31, 25, 33] that can be represented by integer $n \times n$ non-singular square transformation matrix T. Such an invertible loop transformation matrix realizes the following transformation $\mathcal{L}\bar{I} + \bar{o} \rightarrow \mathcal{L}T^{-1}\bar{I'} + \bar{o}$, where $\bar{I'} = T\bar{I}$ is the new iteration vector (after the transformation). Similarly, for a m-dimensional array, an $m \times m$ non-singular data transformation matrix M has the following effect [24, 29, 17]: $\mathcal{L}\bar{I} + \bar{o} \rightarrow M\mathcal{L}\bar{I} + M\bar{o}$.

Consequently, applying *both* loop and data transformations to a reference represented by $\mathcal{L}$ and $\bar{o}$ gives us the following transformation: $\mathcal{L}\bar{I} + \bar{o} \rightarrow M\mathcal{L}T^{-1}\bar{I'} + M\bar{o}$. Since we are not interested in shift-type (alignment-like) data transformations in this paper, we only focus on the transformed access matrix $M\mathcal{L}T^{-1}$. Most of the previous approaches to loop and data transformations can be cast as problems of determining either or both of the matrices T and M (with some legality conditions) such that $M\mathcal{L}T^{-1}$ will have some desired form for a given objective such as optimizing locality [31] or maximizing parallelism [33]. An iteration space transformation matrix T is legal if it preserves all data dependences in the original loop nest [33]. Also, the data transformation matrix M should be applied to all the references to the array in question and and to all its aliases [7]. In this paper we use T_i to denote the loop transformation matrix for the nest i; we use M_u to refer to the data transformation matrix for the array U in a given procedure.

An element is said to be *reused* if it is accessed by more than once in a loop nest. There are two types of reuses: *temporal* and *spatial*. [31, 25]. Temporal reuse occurs when two references (not necessarily distinct) access the same memory location; and spatial reuse arises between two references that access nearby memory locations (e.g., elements mapped on the same cache line) [33]. In this paper, we focus primarily on self-reuses (i.e., reuses originating from individual references [31]); we do not discuss the extension to handle group-reuses.

It is important to note that the most important reuses (whether temporal or spatial) are the ones exhibited by the *innermost* loop. If the innermost loop exhibits temporal reuse for a reference (e.g., the reference $U(i)$ in a loop nest where i is *not* innermost), then the element accessed by that reference can be kept in a register throughout the execution of the innermost loop. Similarly, spatial reuse is most beneficial when it occurs in the innermost loop (as in $U(i, j)$ assuming U is column-major and i is innermost); because, in that case it *may* enable unit-stride accesses to consecutive locations in memory.

2.1.2 Problem Definition for Intra-procedural Optimization

Let $\mathcal{U}$ be the set of arrays accessed in a procedure P and $N_1, ..., N_l$ are the nests in the said procedure. We want to determine a M_u for each $U \in \mathcal{U}$ and a T_i for each N_i ($1 \leq i \leq l$) such that the overall cache locality of the procedure P will be improved. We also insist that T_i should observe the data dependences in N_i and each M_u ($U \in \mathcal{U}$) should be applied taking legality considerations [7] into account.

2.1.3 Approach

Our approach to the intra-procedural locality optimization problem is based on forming a set of locality constraints (equalities) and solving them using a heuristic so that the solution of the constraints will produce loop and data transformation matrices that collectively achieve the desired cache locality. We are mainly interested in exploiting temporal and spatial locality in the *innermost* loops where they are most useful, although our approach can be extended and/or integrated with tiling to exploit locality in higher loop levels. Assuming that the array layouts are *column-major*, in order to have a good locality in the *innermost* loop

$$M_u \mathcal{L}_{uij} \bar{q}_i = (\times, 0, ..., 0, 0)^T$$

should hold for all $1 \leq i \leq l$, $U \in \mathcal{U}$, and $1 \leq j \leq s_{ui}$, where s_{ui} is the number of references to the array U in the nest N_i. In this formulation, which we call a *layout* or *locality constraint*, $\bar{q}_i$ is the last column of ${T_i}^{-1}$, the inverse of the loop transformation matrix for the nest N_i. For the rest of the paper we use $\bar{c}$ instead of $(\times, 0, ..., 0, 0)^T$ for clarity. Notice that if $\times = 0$ we have temporal reuse in the innermost loop; if $\times \neq 0$ and $\times <$*cache-line-size*, we have spatial reuse in the innermost loop. As an example, consider the procedure P shown in Figure 1(a). For this procedure we have the following set of layout (or locality) constraints:

$$\{M_u \mathcal{L}_{u11} \bar{q}_1 = \bar{c} \qquad M_v \mathcal{L}_{v11} \bar{q}_1 = \bar{c}$$
$$M_u \mathcal{L}_{u21} \bar{q}_2 = \bar{c} \qquad M_w \mathcal{L}_{w21} \bar{q}_2 = \bar{c}\},$$

where $\mathcal{L}_{u11} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $\mathcal{L}_{v11} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$,
$\mathcal{L}_{u21} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$, and $\mathcal{L}_{w21} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$.

This set of equalities is represented as a bipartite graph $G = (V_l, V_a, E)$, called the *locality* (or *layout*) *constraint graph* (LCG), as shown in Figure 1(b). In a locality constrained graph G, V_l is the set of loop nests in the procedure and V_a is the set of arrays (global or local) accessed by the procedure. There is an edge $e \in E$ between a $v_a \in V_a$ and $v_l \in V_l$ if and only if the array represented by v_a is accessed in the nest represented by v_l.

Notice that such a set of equations can be solved in a number of ways and this fact will be used later in the paper to handle different cases. For our current example, six alternative solutions are shown in Figure 1(c). The numbers associated with arrows indicate the processing order; the arrows with the same number can be processed in any order. For example, on the upper-leftmost solution, we first apply a loop transformation to the first nest (we start with the vertex marked *1*). This loop transformation, in turn, allows us to determine the layouts of U and V (step 1). Next, using the layout of array U, we determine an appropriate loop transformation

for the second nest (step 2). In the last step (step 3), using the new loop order of the second nest we find an appropriate memory layout for the array W. Of course, different solutions have different qualities and some solutions can cause potential conflicts. Consider the solution given on the lower-rightmost. We first transform layouts of V and W for locality, which in turn determine appropriate transformations for the first and second nests. However, since both these nests access the same array U, in determining its layout we *may* have a conflict; that is, the two nests may require *different* layouts for the same array. Notice that this is a potential conflict, not a certain one as it may happen such that the two nests agree on the same layout. It is also interesting to see how the previous approaches to procedure-wide locality optimization problem map on the locality constrained graph. As an example, the solution proposed in [18] first orders the loop nests according to a cost criterion. It then optimizes the layouts of the arrays accessed by the most-costly nest. Afterwards using the layouts found so far it optimizes the next most-costly nest and so on. Assuming that the nest *1* in our example is costlier than the nest *2*, this solution corresponds to the one shown on the upper-leftmost of Figure 1(c). If, on the other hand, the nest *2* is the most-costly, the solution is the one shown on the lower-leftmost of Figure 1(c).

In general, in order to solve the problem, we can adopt the following *graph-theoretical* solution strategy on the LCG.[1] First, we convert each edge in the locality constraint graph to a bidirectional arc (arrow) so that the end points can be visited in either order. Then we run a *maximum-branching algorithm*[2] on the resulting graph and determine all the nodes that can be covered in a conflict-free manner. For the example LCG shown in Figure 1, the upper-leftmost figure given in Figure 1(c) depicts one possible solution obtained using maximum-branching. As another example, consider the locality constraint graph shown in Figure 2(a). After the edges have been converted to arrows (Figure 2(b)), the maximum-branching algorithm generates the solution shown in Figure 2(c) (assume again that the numbers on arrows denote the processing order); note that optimal solution here is not unique. In this solution, only two edges shown in Figure 2(d) (corresponding to two locality constraints) are left unsatisfied. Of course, whether these two unsatisfied constraints will really cause any conflict or not depends on actual access matrices. As a convention, we put the directions of these unsatisfied edges from nest nodes to array nodes. To sum up, in this example, two references can go unoptimized. Figure 2(e) shows the complete solution. We refer to this complete solution as a *maximum-branching solution*.

In case we have some edges whose directions have already been selected (decided), we have a *restricted* LCG, or RLCG for short. In that case, the solution is in general imposed by these selected edges (corresponding to the locality constraints that have already been solved so far). Consider Figure 2(a) again, this time assuming that the layout of U and the loop transformations for the nests *2* and *4* have already been determined; i.e., we have a fixed arrow going from node U to node *2* and another arrow from U to *4*. The problem now is to find a maximum branching solution such that when combined with the arrows between U and *2* and between U and *4* will lead to *minimum* number of conflicts. Such a solution is shown in Figure 2(f). As another example, let us assume that the edge between nodes W and 2 has already been selected. A maximum-branching solution in that case is given in Figure 2(g). Figures 2(h) and (i), on the other hand, depict, respectively, the unsatisfied constraints for the solutions shown in Figures 2(f) and (g). And finally, Figure 2(j) gives the final solution for Figure 2(f).

[1] Dion, Randriamaro, and Robert [12] uses a similar graph-based strategy for the automatic data alignment problem.

[2] An *arborescence* is defined as a *tree* in which no two arcs are directed into the same node. A *branching* is defined as a *forest* in which each tree is an arborescence. Now associate a unit *weight* to each arc. *A maximum-branching* of a graph is any branching of the same graph with the largest possible weight [28]. Notice that within our problem domain this corresponds to satisfying as many locality constraints as possible.

3 Inter-procedural Locality Optimization

3.1 Bottom-up Traversal

In bottom-up traversal, we take a slightly different approach from what was described above. In the leaf nodes, we collect all locality constraints, but we do not attempt to solve them immediately. Instead, for each procedure R which calls procedure P, the locality constraints are propagated from P to R. Notice that we need to propagate only the constraints on global variables and formal parameters. Of course, the latter should be *re-written* in terms of actual parameters passed to P (Recall that we do *not* allow array re-shaping). As an example, consider the code fragment shown in Figure 3(a). After the procedure P (the callee) has been processed, we have the following locality constrains:

$$\{M_u\mathcal{L}_{u11}\bar{q}_1 = \bar{c} \qquad M_x\mathcal{L}_{x11}\bar{q}_1 = \bar{c}$$
$$M_y\mathcal{L}_{y11}\bar{q}_1 = \bar{c} \qquad M_z\mathcal{L}_{z11}\bar{q}_1 = \bar{c}\},$$

where $\bar{q}_1$ is the last columns of the inverse of the loop transformation matrix for the nest in P. Notice that the first constraint is on the global array variable U and the last constraint is on the local array variable Z. The second and the third constraints, on the other hand, are on the formal parameters X and Y. When we process the caller R, the second and the third constraints are *re-written* in terms of the actual parameters V and W passed to the callee P whereas the first constraint is propagated as it is. There is no need to propagate the last constraint as Z is a local variable. Thus, the total constraints in R are as follows:

$$\{M_u\mathcal{L}_{u11}\bar{q}_1 = \bar{c}; M_v\mathcal{L}_{v11}\bar{q}_1 = \bar{c}; M_w\mathcal{L}_{w11}\bar{q}_1 = \bar{c};\ \text{and}$$
$$M_u\mathcal{L}_{u21}\bar{q}_2 = \bar{c}; M_v\mathcal{L}_{v21}\bar{q}_2 = \bar{c}; M_w\mathcal{L}_{w21}\bar{q}_2 = \bar{c}\},$$

where $\bar{q}_2$ is the last columns of the inverse of the loop transformation matrix for the nest in R. The last three constraints are the local constraints to R. As can be seen, the call statement in R is treated as a program construct (e.g., a loop nest) that somehow generates the first three constraints. Note that our propagation technique is also able to handle the cases where *aliasing* between the formal parameters occur. For example, consider the program fragment shown in Figure 3(b). Notice that in procedure P we have two constraints $\{M_x\mathcal{L}_{x11}\bar{q}_1 = \bar{c}; \quad M_y\mathcal{L}_{y11}\bar{q}_1 = \bar{c};\}$, where $\bar{q}_1$ is the last columns of the inverse of the loop transformation matrix for the nest in P. This system, if considered alone, can assume many (equivalently optimized) solutions. (e.g., we can apply an identity loop transformation and select row-major layout for X and column-major layout for Y or alternatively we can apply loop interchange and select column-major layout for X and row-major layout for Y). However, after the propagation of these constraints (and re-writing), we have a 'more constrained' set: $\{M_v\mathcal{L}_{v11}\bar{q}_1 = \bar{c}; \quad M_v\mathcal{L}_{v12}\bar{q}_1 = \bar{c};\}$, where

$$\mathcal{L}_{v11} = \mathcal{L}_{x11} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \text{ and } \mathcal{L}_{v12} = \mathcal{L}_{y11} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

The solution now is to *skew* [31] the loop nest and assign diagonal layout for U; that is, we select

$$M_v = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \text{ and } T = \begin{pmatrix} 1 & 1 \\ 0 & -1 \end{pmatrix}.$$

We continue to propagate the constraints from children (callees) to parents (callers) until we reach the root (the main program). As we move up the call graph, the locality constraints from callee procedures are propagated to the caller procedures. The static locality

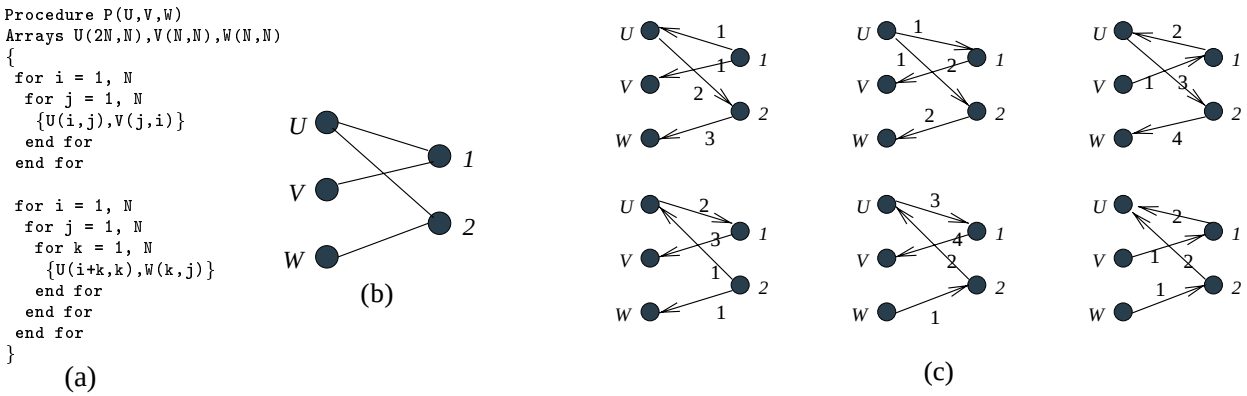


Figure 1: (a) A procedure that contains two nests (Arrays U, V, and W are formal parameters). (b) Locality constraint graph. (c) Example solution strategies.

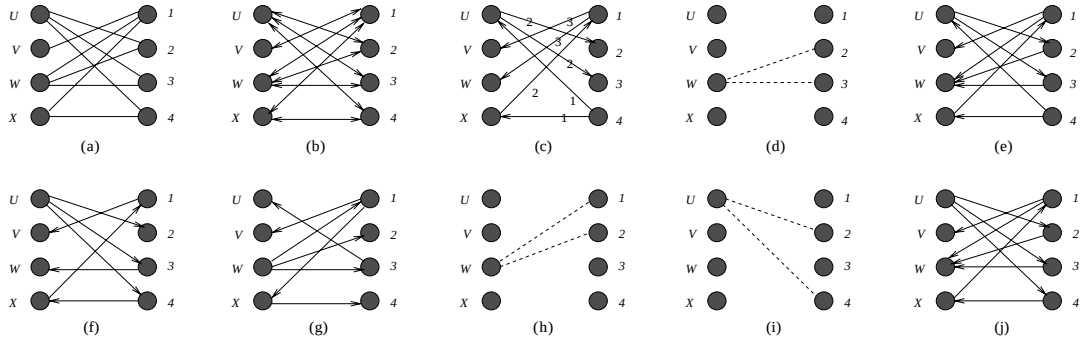


Figure 2: (a) An example locality constraint graph. (b) After the edges have been converted to bidirectional arcs (arrows). (c) A maximum-branching solution. (d) Unsatisfied edges (constraints). (e) A complete (maximum-branching) solution. (f) The solution starting from the vertex U. (g) The solution starting from the vertex W. (h) The unsatisfied edges for the solution shown in (f). (i) The unsatisfied edges for the solution shown in (g). (j) Complete solution for (f).

optimization algorithm is then run on the caller procedures. Afterwards the relevant locality constraints are passed up the call graph to their callers and so on.

When we reach the root, we have all the necessary locality constraints. In the root node, we construct a *global layout* (*locality*) *constraint graph* (GLCG) and attempt to solve these locality constraints using the graph theoretical approach discussed earlier. Notice that this GLCG has all the arrays accessible from the root node and all the nests in the root. It can also have some nests belonging to the other procedures which also reference the arrays accessed by the root. When the system has finally been solved, we have (1) the loop transformation matrices for (some of) the nests in the program and (2) the data transformation matrices for the global and local arrays accessible from the root. For example, consider again the code given in Figure 3(a). Figures 4(a) and (b) show the LCGs for the procedures P and R, respectively, when they are considered separately. Figure 4(c), on the other hand, illustrates the GLCG after all locality constraints have been gathered in the root R. Note also that the nest in P (numbered *1*) also appears in this GLCG. A maximum-branching solution is for this GLCG depicted in Figure 4(d). Using our convention, Figure 4(e) gives the complete solution. Notice that this solution determines all memory layouts as well as the loop transformations for the two nests in the program.

3.2 Top-down Traversal

In this traversal, the layouts found so far are propagated from caller procedures to callee procedures. In receiving the layouts, a callee procedure computes the loop transformation matrices for the nests it contains (if they have not been determined so far) as well as the memory layouts of its *local* arrays. In the example shown in Figure 3(a), the callee procedure P takes the layouts of U, V, and W from its caller, R. The layouts of V, and W automatically determine the layouts of X and Y, respectively. Then using these layouts, the algorithm determines the layout of the local array Z.

In terms of our graph-theoretical solution, we represent the constraints inherited from a parent as a RLCG. This RLCG contains the nodes corresponding to all the arrays accessed by the procedure being analyzed, its nests as well as (some of) the nests accessed by other procedures below this procedure in the call graph. The problem now is to find a maximum-branching solution on this RLCG. Returning to the example code shown in Figure 3(a), after the root procedure (R) has been processed, the locality constraints solved are transferred to P as an RLCG as shown in Figure 4(f). The complete maximum-branching solution for the procedure P is shown in Figure 4(g). The only thing performed when working in the RLCG of P was to determine the layout for Z, considering the loop transformation already found for the nest it contains.

Notice that not all the nests in the program have to appear in the GLCG built for the root procedure. For example, consider a program with two procedures, R (main) and P (callee). Assume

```
Arrays U(N,2N)
Procedure R(V,W)
Arrays V(2N,N),W(N,N)
{
 for i = 1, N
  for j = 1, N
   {U(i,j),W(j,i),V(i+j,i)}
  end for
 end for
 ...
 call P(V,W)
 ...
}

Procedure P(X,Y)
Arrays X(2N,N),Y(N,N)
{
 Arrays Z(N,N)
 for i = 1, N
  for j = 1, N
   for k = 1, N
    {X(i+k,k),Y(k,j)
     Z(i,j),U(i,j+k)}
   end for
  end for
 end for
}
```

(a)

```
Procedure R(V)
Arrays V(N,N)
{
 ...
 call P(V,V)
 ...
}

Procedure P(X,Y)
Arrays X(N,N),Y(N,N)
{
 for i = 1, N
  for j = 1, N
   {X(i,j),Y(j,i)}
  end for
 end for
}
```

(b)

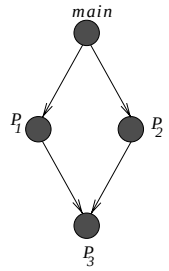
```
Procedure main(U1,U2,U3,U4)
Arrays U1(N,N),U2(N,N),
       U3(N,N),U4(N,N)
{
 call P1(U1,U2)
 for i = 1, N
  for j = 1, N
   {U1(j,i),U2(i,j),
      U3(i,j),U4(j,i)}
  end for
 end for
 call P2(U2,U3)
}

Procedure P1(V1,V2)
Arrays V1(N,N),V2(N,N)
{
Arrays V3(N,N)
 for i = 1, N
  for j = 1, N
   {V1(i,j),V2(j,i),V3(i,j)}
  end for
 end for
 call P3(V1,V3)
}

Procedure P2(W2,W3)
Arrays W2(N,N),W3(N,N)
{
Arrays W1(N,N)
 for i = 1, N
  for j = 1, N
   {W1(j,i),W2(i,j),W3(i,j)}
  end for
 end for
 call P3(W1,W2)
}

Procedure P3(X,Y)
Arrays X(N,N),Y(N,N)
{
Arrays Z(N,N)
 for i = 1, N
  for j = 1, N
   {X(i,j),Y(j,i),Z(j,i)}
  end for
 end for
}
```

(c)



(e)

```
Procedure main(U1,U2,U3,U4)
Arrays U1(N,N),U2(N,N),
       U3(N,N),U4(N,N)
{
 call P1(U1,U2)
 for j = 1, N
  for i = 1, N
   {U1(i,i),U2(i,j),
      U3(i,j),U4(i,i)}
  end for
 end for
 call P2(U2,U3)
}

Procedure P1(V1,V2)
Arrays V1(N,N),V2(N,N)
{
Arrays V3(N,N)
 for i = 1, N
  for j = 1, N
   {V1(j,i),V2(j,i),V3(j,i)}
  end for
 end for
 call P3copy2(V1,V3)
}

Procedure P2(W2,W3)
Arrays W2(N,N),W3(N,N)
{
Arrays W1(N,N)
 for j = 1, N
  for i = 1, N
   {W1(i,j),W2(i,j),W3(i,j)}
  end for
 end for
 call P3copy1(W1,W2)
}

Procedure P3copy1(X,Ycopy1)
Arrays X(N,N),Ycopy1(N,N)
{
Arrays Zcopy1(N,N)
 for i = 1, N
  for j = 1, N
   {X(j,i),Ycopy1(j,i),Zcopy1(j,i)}
  end for
 end for
}

Procedure P3copy2(X,Y)
Arrays X(N,N),Ycopy2(N,N)
{
Arrays Zcopy2(N,N)
 for j = 1, N
  for i = 1, N
   {X(j,i),Ycopy2(i,j),Zcopy2(i,j)}
  end for
 end for
}
```

(d)

Figure 3: (a-d) Example program fragments. (e) Call graph for (c).

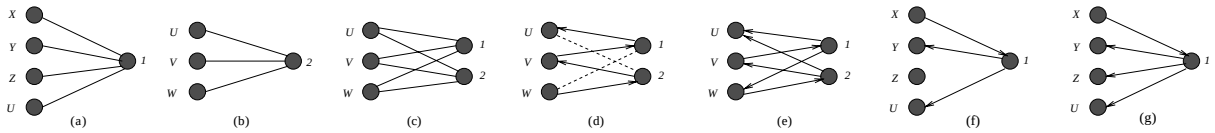


Figure 4: The locality constraint graphs and solutions for the program shown in Figure 3(a).

that R contains a single nest (nest *1*) that accesses three arrays U, V, and W. Assume further that R calls P passing V and W as parameters which, respectively, map to X and Y in P. Suppose that the procedure P contains four nests; the first one (nest *2*) accesses X, Y, and Z; the second one (nest *3*) accesses Z and L; and finally, the third one (nest *4*) accesses L and K. Figures 5(a) and (b) show the LCGs for R and P, respectively. Figure 5(c), on the other hand, illustrates the GLCG obtained after the bottom-up traversal. A complete solution to this GLCG is shown in Figure 5(d). Notice that this solution satisfies all the constraints but one. Finally, Figure 5(e) shows a maximum-branching solution on the RLCG for the procedure P. In this graph, the node *2* inherits a constraint from the solution given in Figure 5(d). This constraint corresponds to the edge between X and *2*. The rest of the solution in Figure 5(e) builts upon this constraint, and determine all the remaning layouts (of L, Z, and K) as well as the loop transformations for the nests *3* and *4*.

In case there are multiple paths from different callers to the same callee procedure, there is a possibility that the different callers may impose conflicting layouts for the same array in the callee. In this case, our approach uses *selective cloning* [11]. Selective cloning is a goal-directed approach that selectively decides which procedures to clone and how many different instances to create. Figure 3(c) shows an example program with its call graph given in Figure 3(e). Figure 3(d) shows the transformed code where the procedure P_3 is cloned. The first and the third nests are transformed using loop interchange [33] whereas the second nest is left unmodified. For the fourth nest, on the other hand, two copies have been created with different loop orders.

4 Experiments

In order to validate the approach presented in this paper, we conducted experiments with four common scientific programs that contain procedure calls: three programs from SPECfp92 benchmark and an alternate direction integral (ADI) code. For each code, we used three different versions: `Base` is the code with most classical locality optimizations (used in commercial compilers) except tiling and loop unrolling turned off. `Intra_r` is an optimized version using the intra-procedural optimization method described in Section 2.1. However, the arrays are *re-mapped* explicitly at procedure boundaries. Note that most of the published papers on data layout optimizations do not take this explicit array re-mapping costs into account. Our performance numbers reported below, however, indicate that these costs can easily outweigh any gains made by transforming array layouts for enhancing cache locality. Finally, `Opt_inter` is the version obtained using the approach described in this paper.

The experiments are performed on an SGI Origin 2000 architecture. Each node of the Origin contains two R10000 CPUs which run at a clock rate of 195 MHz. Each processor has an instruction cache of 32 Kbytes and a Level-1 (L1) data cache of 32 Kbytes as well as a 4 Mbyte secondary (L2) unified instruction/data cache. The node board is configured with 4 Gbytes of main memory. The R10000 CPUs operate on data that are resident in their caches. When programs use the caches effectively, the access time to memory is unimportant because the great majority of accesses are satisfied from the caches. The processors can also prefetch data that are not in cache. Independent work can be carried out while these data move from memory to cache, thus hiding the access time.

The intra-procedural optimization strategy based on maximum-branching described in Section 2.1 is currently being implemented on top of Parafrase-2. The bottom-up and top-down traversals, however, are performed by hand for the time being. For all three versions the transformed codes are compiled using the native optimizing compiler with the following optimization flags: `-n32 -mips4 -Ofast=ip27 -OPT:IEEE_arithmetic=3 -LNO:blocking=off -LNO:outer_unroll=1`.

The results are shown in Table 1 for the single-processor and eight-processor cases; we report Cache Line Reuses as well as MFLOPS rates (measured by using an outer timing loop). The L1 Cache Line Reuse is the number of times, on the average, that a primary data cache line is reused after it has been moved into the cache. It is calculated as 'graduated loads' plus 'graduated stores' minus 'primary data cache misses', all divided by 'primary data cache misses'. The L2 Cache Line Reuse is defined similarly for the L2 Cache. All these values have been collected through the hardware counters of the R10000. These cache line reuse values show that `Opt_inter` is more successful than the other two versions in exploiting cache locality.

The MFLOPS results show that propagating array layouts across procedures is very critical in fully exploiting the advantage of memory layout optimizations. This is true for both the single and eight processor cases. It should be noted that there is only a marginal difference in the MFLOPS rates between `Base` and `Intra_r`. Actually, in the `ADI` code using eight processors, the performance of `Intra_r` is even worse than that of `Base`. Good performance results are obtained only with the codes optimized using the approach described in this paper.

5 Related Work

The area of locality optimization has received much attention. Several researchers have proposed the use of loop restructuring techniques to improve locality in a loop nest [1, 31, 25, 27, 32, 33, 6, 23]. Recently, some research groups have advocated the use of memory layout transformations for multi-dimensional arrays [24, 29, 21, 17]. In addition, there have been proposed techniques aimed at exploiting the benfits of loo as well data transformations by combining the two [8, 4, 21, 18, 29].

Very few papers have addressed the problem of inter-procedural optimization of locality. Cierniak and Li [10] address the problem of performing data layout transformations that are propagated across procedure boundaries. But they do not consider loop transformations at all. Our work also relates to recent work on interprocedural data distribution [5]. Anderson [5] was the first to perform inter-procedural analysis to derive data distributions. Her work does not address cache locality explicitly; rather, it is intended to reduce the inter-processor communication incurred due to data distribution. In addition, her work on inter-procedural data distribution does not consider general linear loop and data transformations. It is important to note that the work presented in this paper is

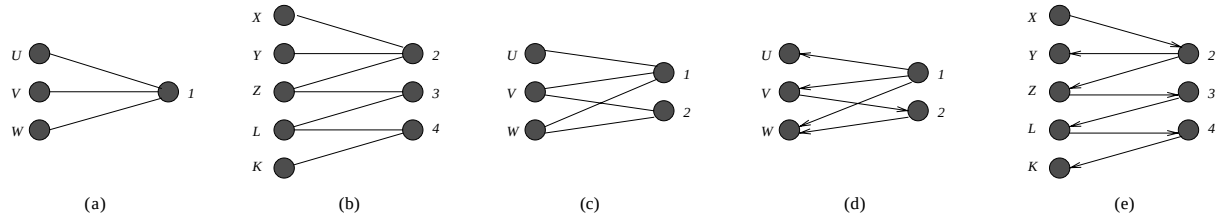


Figure 5: An example scenario which contains a main procedure whose LCG is shown in (a) and a callee whose LCG is shown in (b). (c) gives the GLCG and (d) shows a complete maximum-branching solution. (e) shows a solution for the RLCG for the callee.

closest in spirit to Anderson's work [5]. Our solution mechanism (using maximum branching) and the type of constraints we handle are different from hers, though.

6 Conclusions and Future Work

While a large number of recent papers have addressed the use of loop and data layout optimizations aimed at improving locality, very few have addressed such locality optimizations that work in whole program with procedure calls. If data layout decisions are not propagated across procedures, then either expensive data layout re-mapping is needed or one has to settle for the resulting poor performance. In this paper we present a locality optimization framework that uses both loop and data transformations to improve cache locality program-wide. Our framework propagates layout (or locality) constraints as a system of equalities across procedures and involves two traversals in the call graph representation of the program. We also show how to handle the cases where the callers of the same procedure demand conflicting memory layouts for the same array. Preliminary experimental results obtained on a R10000 based system demonstrate the power of the framework.

At this point, our approach does not consider array re-shaping. We are working on including array re-shaping in our framework. We note that the problem of inter-procedural data and loop transformations and the problem of inter-procedural data distribution are closely related in the case of multiprocessors. We are working on extending our framework to include the effects of parallelism and false sharing on the locality characteristics of whole programs with procedure calls.

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.

References

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

[2] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sonensen, editors. *LAPACK Users' Guide*, SIAM, 1995.

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

[5] J. Anderson. *Automatic Computation and Data Decomposition for Multiprocessors.* Ph.D. dissertation, Stanford University, March 1997.

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

[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 & Implementation (PLDI'97)*, pp. 334–345, 1997.

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

[9] M. Cierniak and W. Li. Recovering logical structures of data. In *Languages and Compilers for Parallel Computing*, C. Huang et al. (Eds.) Lecture Notes in Computer Science, Vol. 1033, Springer-Verlag, 1996.

[10] M. Cierniak and W. Li. Inter-procedural array re-mapping. In *Proc. the International Conference on Parallel Architectures and Compilation Techniques (PACT '97)*, Nov 1997.

[11] K. Cooper, M. W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2), April 1993.

[12] M. Dion, C. Randriamaro, and Y. Robert. Compiling affine nested loops: how to optimize the residual communications after the alignment phase. *Journal of Parallel and Distributed Computing (JPDC)*, 38(2):176–187, 1996.

[13] M. W. Hall. *Managing Interprocedural Optimization*. PhD thesis, Rice University, Houston, TX, May 1991.

[14] M. W. Hall and K. Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3), September 1992.

[15] M. W. Hall, J. M. Mellor-Crummey, A. Carle, and R. Rodriguez. FIAT: A framework for interprocedural analysis and transformation. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, 1993.

[16] M.W. Hall, S.P. Amarasinghe, B.R. Murphy, S. Liao, and M.S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95*, December 1995.

Table 1: Performance results on an SGI Origin 2000 multiprocessor. (The data sizes: Btrix, the parameters JD, KD, LD, and MD are set to 100. Vpenta, size parameter is set to 850. Cholsky, size parameter is set to 800. ADI, $1040 \times 1040 \times 3$ arrays. All these are in double precision elements).

(a) Btrix

1 proc

	Base	Intra_r	Opt_inter
L1 Cache Line Reuse	7.75	7.48	8.38
L2 Cache Line Reuse	27.15	31.29	39.86
MFLOPS	29.09	33.62	90.71

8 procs

	Base	Intra_r	Opt_inter
L1 Cache Line Reuse	16.89	11.43	18.11
L2 Cache Line Reuse	26.01	26.93	44.03
MFLOPS	158.16	164.12	660.16

(b) Vpenta

1 proc

	Base	Intra_r	Opt_inter
L1 Cache Line Reuse	3.34	3.40	4.52
L2 Cache Line Reuse	14.18	17.01	21.33
MFLOPS	44.01	55.04	71.82

8 procs

	Base	Intra_r	Opt_inter
L1 Cache Line Reuse	4.51	4.39	4.97
L2 Cache Line Reuse	13.82	15.13	24.79
MFLOPS	240.19	264.82	434.17

(c) Cholsky

1 proc

	Base	Intra_r	Opt_inter
L1 Cache Line Reuse	2.20	2.40	4.88
L2 Cache Line Reuse	11.06	12.13	20.51
MFLOPS	25.80	26.04	60.90

8 procs

	Base	Intra_r	Opt_inter
L1 Cache Line Reuse	2.11	2.39	4.82
L2 Cache Line Reuse	12.82	13.13	23.00
MFLOPS	128.62	134.82	383.10

(d) ADI

1 proc

	Base	Intra_r	Opt_inter
L1 Cache Line Reuse	2.73	2.86	3.56
L2 Cache Line Reuse	174.78	180.04	242.67
MFLOPS	93.45	98.65	139.00

8 procs

	Base	Intra_r	Opt_inter
L1 Cache Line Reuse	4.04	4.32	4.62
L2 Cache Line Reuse	109.00	104.55	201.73
MFLOPS	515.74	513.48	780.69

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

[18] 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)*, October 1998.

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

[20] 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*, April 1999.

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

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

[23] 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)*.

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

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

[26] V. Maslov. Delinearization: an efficient way to break multi-loop dependence equations. In *Proc. SIGPLAN Conf. Programming Language Design & Implementation*, pp. 152–161, June 1992.

[27] 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.

[28] E. Minieka. *Optimization Algorithms for Networks and Graphs*, Marcel Dekker, Inc., 1978.

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

[30] 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.

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

[32] M. Wolfe. More iteration space tiling. In *Proc. Supercomputing'89*, pp. 655–664, November 1989.

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