A Compiler Algorithm for Optimizing Locality in Loop Nests*

M. Kandemir
EECS Dept.
Syracuse University
Syracuse, NY, 13244
mtk@ece.nwu.edu

J. Ramanujam
ECE Dept.
Louisiana State University
Baton Rouge, LA 70803
jxr@ee.lsu.edu

A. Choudhary
ECE Dept.
Northwestern University
Evanston, IL, 60208-3118
choudhar@ece.nwu.edu

Abstract

This paper describes an algorithm to optimize cache locality in scientific codes on uniprocessor and multiprocessor machines. A distinctive characteristic of our algorithm is that it considers loop and data layout transformations in a unified framework. We illustrate through examples that our approach is very effective at reducing cache misses and tile-size sensitivity of blocked loop nests; and can optimize nests for which optimization techniques based on loop transformations alone are not successful. An important special case is the one in which data layouts of some arrays are fixed and cannot be changed. We show how our algorithm can handle this case, and demonstrate how it can be used to optimize multiple loop nests.

1 Introduction

Minimizing the time spent in data accesses is an important issue in the efficient execution of nested loops on both uniprocessors and multiprocessors. Although caches are capable of reducing the average memory access time and optimizing compilers are able to detect significant parallelism, the performance of scientific programs on both uniprocessors and multiprocessors can be rather poor due to not exploiting the full potential locality in these programs [13].

We present a compiler approach to enhance the cache performance of these programs on uniprocessors and multiprocessors. In a unified framework, our approach considers modifying array layouts in memory and transforming loop nests suitably to exploit locality. We simulate miss rates for several nests in order to demonstrate that our approach is very effective at reducing number of cache misses, and report execution times on Sun SPARCstation 5, IBM RS/6000 and SGI Challenge. We also compare our optimization strategy to a representative method [10] from a class of approaches which consider only loop transformations to optimize locality and show that fixing the memory layouts for all arrays—as in C and Fortran—limits performance that could otherwise have been obtained from the programs.

A recent study shows that a group of highly parallelized benchmark programs spend 39% of their cycles stalled in memory access [11]. In order to eliminate the memory bottleneck, spatial locality should be exploited. One way of achieving this is to transform the loop nest such that the innermost loop exhibits unit-stride accesses for array references. While this approach produces satisfactory results for several cases, we show in this paper that there is still room for significant improvement, if the fixed array layout strategy adopted by the conventional compilers is relaxed.

In this paper we make the following contributions:

- We present a new algorithm to optimize the locality characteristics of nested loops. The algorithm applies both data and control transformations.
- We show that the known approaches considering only control transformations (e.g. loop permutations, tiling, etc.) are insufficient for many cases.
- We demonstrate the effectiveness of our approach by both simulation results and execution-time measurements.

Since our approach increases the spatial locality and the percentage of conflict misses and reduces the percentage of capacity misses; it is generally more effective with large block (cache line) sizes and set-associative caches.

This paper is organized as follows. Section 2 reviews basic loop transformation theory. Section 3 presents related work. Section 4 discusses the algorithm for optimizing locality in a single loop nest. Section 5 extends this algorithm to multiple loop nests. Section 6 presents experimental results which illustrate the efficacy of our approach. Section 7 presents a discussion of our work on false sharing. Section 8 presents summary and concludes the paper.

2 Preliminaries

The memory layout for an h-dimensional array can be in one of the $h!$ forms, each of which corresponding to layout of data on memory linearly by a nested traversal of the axes in some predetermined order. The innermost axis is called the *fastest changing dimension*. As an example for row-major memory layout the second dimension is the fastest changing dimension. We focus on loop nests where both array subscripts and loop bounds are affine functions of enclosing loop indices. A reference to an array X is represented by $X(\mathcal{L}\vec{I}+\vec{b})$ where $\mathcal{L}$ is a linear transformation matrix called *array reference matrix*, $\vec{b}$ is offset vector and $\vec{I}$ is a column vector representing the loop indices i_1, i_2,...,i_n starting from the outermost loop.

Linear mappings between iteration spaces of loop nests can be modeled by non-singular transformation matrices [10]. If $\vec{I}$ is the original iteration vector, after applying linear transformation T, the new iteration vector is $\vec{J}=T\vec{I}$. Similarly if $\vec{d}$ is the distance/direction vector, on applying T, $T\vec{d}$ is the new distance/direction vector. A transformation is *legal* if and only if $T\vec{d}$ is lexicographically positive for every $\vec{d}$ [15]. On the other hand, since $\mathcal{L}\vec{I}=\mathcal{L}T^{-1}\vec{J}$, $\mathcal{L}T^{-1}$ is the new array reference matrix after the transformation. We denote T^{-1} by Q. An important characteristic of our

*This work is supported in part by NSF Young Investigator Award CCR-9357840, NSF CCR-9509143. The work of J. Ramanujam is supported in part by an NSF Young Investigator Award CCR-9457768.

algorithm is that using the array reference matrices, the entries of $Q = [q_{ij}]$ are derived systematically. For the rest of the paper, the reference matrix for array X will be denoted by $\mathcal{L}^X$ whereas the i^{th} row of the reference matrix for array X will be denoted by $\vec{\ell}_i^{\,X}$.

3 Related Work

3.1 Fixed Layout Approach

Loop transformations have been used for optimizing cache locality in several papers [10, 14, 7]. Results have shown that on several architectures the speedups achieved by loop transformations alone can be significant.

Li [10] describes a data reuse model and a compiler algorithm called *height reduction* to improve cache locality. He introduces the concept of a *data reuse vector* and defines its *height* as the number of dimensions from the first non-zero entry to the last entry. The non-zero entries of a reuse vector indicate that there are reuses carried by the corresponding loops. The individual reuse vectors constitute reuse matrices which in turn constitute the global reuse matrix. The algorithm assigns priorities to reuse vectors depending on the number of times they occur, and tries to reduce the height of the global reuse matrix starting from the reuse vector of highest priority. Apart from reducing the execution time, the height reduction algorithm serves two purposes:

- it reduces the sensitivity of tiling to the tile size; and
- it places the loops carrying reuse into innermost positions; thus, when the outermost loops are parallelized, the chances of false sharing will be low.

In comparison, our algorithm (Sections 4 and 5) tries to exploit the spatial locality by also considering different memory layouts for different arrays. Since Li's approach is representative of a class of algorithms that use only control transformations to exploit locality [14, 7, 10], for the rest of the paper we use Li's algorithm (denoted W-Opt) and compare it with our algorithm.

3.2 Data and Loop Transformations

For programs that are not conducive to loop transformations, data transformations should also be taken into account. Only a few works have considered data and loop transformations together to optimize locality. Ju and Dietz [6] present a systematic approach that integrates data layout optimizations and loop transformations to reduce cache coherence overhead. Anderson *et al.* [1] offer a simple algorithm to transform data layout to make the region accessed by each processor contiguous.

Cierniak and Li [3] present a unified approach to optimize locality that employs both data and control transformations. The notion of a *stride vector* is introduced and an optimization strategy is developed for obtaining the desired mapping vectors and transformation matrix. At the end, the following equality is obtained:

$$T^T v = A^T m$$

In this formulation only A, data reference matrix, is known. The algorithm tries to find T, the transformation matrix; m, a mapping vector which can assume $h!$ different forms for an h-dimensional array; and v, the desired stride vector. Since this optimization problem is difficult to solve, the following heuristic is used: First it is assumed that the transformation matrix contains only values 0 and 1. Second, the value of the stride vector v is assumed to be known beforehand. Then the algorithm constructs the matrix T row by row by considering all possible legal mappings. When compared to our strategy (Sections 4 and 5), we argue that our approach is more accurate, as it does not restrict the search space of possible loop transformations. Also our approach is simpler to embed in a compilation system, since it does not require a priori knowledge of any vector such as v; and more importantly it does not depend on any new reuse abstraction. The approach presented by Cierniak and Li [3] is a heuristic whereas our approach for single nest is exact solution which finds all possible optimized transformations and memory layouts. This last point is important as will be demonstrated in Section 6.

4 Algorithm for Optimizing Locality

Since accessing data from memory is usually an order of magnitude slower than accessing data in cache, optimizing compilers must reduce the number of memory accesses. We present an algorithm which automatically transforms a given loop nest to exploit spatial locality and assigns appropriate memory layouts for arrays, in a unified framework.

The algorithm is shown in Figure 1. In the algorithm, C is the array reference on the LHS whereas A represents an array reference from the RHS. The symbol $\times$ denotes the *don't care* condition. Let i_1, i_2,...,i_n be the loop indices of the original nest and j_1, j_2,...,j_n be the loop indices of the transformed nest, starting from outermost loop. The following is a brief explanation of our algorithm:

- Our transformation matrix should be such that the LHS array of the transformed loop has the innermost index as the only element in one of the array dimensions and that index should not appear in any other dimension for this array. In other words, after the transformation, the LHS array C should be of the form $C(*, *, ..., j_n, ..., *, *)$ where j_n (the new innermost loop index) is in the r^{th} dimension and $*$ indicates a term independent of j_n. This means that the r^{th} row of the transformed reference matrix for C is $(0, 0, ..., 0, 1)$ and all entries of the last column, except the one in r^{th} row, are zero. After that, the LHS array can be stored in memory such that the r^{th} dimension will be the fastest changing dimension. This approach exploits the spatial locality for this reference. Notice that all possible values for r should be considered.
- Then the algorithm works on one reference from the RHS at a time. If a row s in the data reference matrix is identical to r^{th} row of the original reference matrix of the LHS array, then the algorithm attempts to store this RHS array in memory such that the s^{th} dimension will be the fastest changing dimension. We note, however, that having such a row s does not guarantee that the array will be stored on memory such that the s^{th} dimension will be the fastest changing dimension.
- If the condition above does not hold for a RHS array A, that means this array cannot be stored in memory such that the new innermost loop index appears only in the fastest changing dimension. In that case the algorithm tries to transform the reference to $A(*, *, ..., \mathcal{F}(j_{n-1}), ..., *, *)$, where $\mathcal{F}(j_{n-1})$ is an affine function of j_{n-1} and other indices except j_n, and $*$ indicates a term independent of both j_{n-1} and j_n. This helps to exploit the spatial locality at the second innermost loop. If no such transformation is possible, the j_{n-2} is tried and so on. If all loop indices are tried unsuccessfully, then the remaining entries of Q are set arbitrarily, observing the data dependences and non-singularity.

Step 1 Initialize $i = 1$.

Step 2 Set $\vec{\ell}_i^{\,C}.Q = (0,0,...,0,1)$ and $\vec{\ell}_k^{\,C}.Q = (\times,\times,...,\times,0)$ for each $k \neq i$.

Step 3 Set memory layout for C such that i^{th} index position will be the fastest changing dimension.

Step 4 For each array reference A on the RHS that has $\vec{\ell}_l^{\,A} = \vec{\ell}_i^{\,C}$ for some l, try to set memory layout for A such that the l^{th} dimension will be the fastest changing dimension.

Step 5 Choose an array reference A for which the equality in **Step 4** does not hold. Initialize $j = 1$.

Step 6 Set $\vec{\ell}_j^{\,A}.Q = (0,0,...,1,0)$ and $\vec{\ell}_k^{\,A}.Q = (\times,\times,...,\times,0,0)$ for each $k \neq j$. If this step is consistent with the previous steps go to **Step 7**, otherwise increment j and go to the beginning of this step. If there exist inconsistencies for all j values, then initialize $j = 1$, and set $\vec{\ell}_j^{\,A}.Q = (0,0,...,1,0,0)$ and $\vec{\ell}_k^{\,A}.Q = (\times,\times,...,\times,0,0,0)$ for each $k \neq j$, and repeat **Step 6** and so on. If no T^{-1} is found then fill the remaining entries arbitrarily observing the *dependences* and *non-singularity*.

Step 7 Repeat **Step 6** for all reference matrices of a particular A (Of course, all reference matrices for a particular A should have the same memory layout).

Step 8 Repeat **Step 6** for all distinct array references.

Step 9 Record the obtained transformation matrix. Also record, for each array, the loop index position which appears in the fastest changing position for that array.

Step 10 Increment i and go to **Step 2** (try a different memory layout for the LHS array C).

Step 11 Compare all the recorded transformation matrices and their associated layouts, and choose the best alternative.

Figure 1: Algorithm for optimizing locality.

• After a transformation and corresponding memory layouts are found, they are recorded and the next alternative memory layout for the LHS is tried and so on. Among all feasible solutions, the one which exploits most spatial locality in the innermost loop is chosen.

5 Global Locality Optimization: Multiple Loop Nests

5.1 General Problem

Let $\{\mathcal{N}_1, \mathcal{N}_2, ..., \mathcal{N}_m\}$ denote different loop nests in the program; and $\{\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_k\}$ denote different arrays. In general each nest can access a subset of the arrays. We assume that our algorithm described before is run for each nest, and a number of possible optimized layout combinations are obtained for each nest. In [8], the authors show that problem of *finding a global array layout combination that satisfies all the nests* is NP-complete even for the restricted case where only row-major and column-major arrays are considered. We present a heuristic for the global layout optimization problem.

5.2 Locality Optimization under Layout Constraints

During the compilation of a program it may be possible that the compiler, due to data dependences or some other constraints, is not able to apply loop transformations or modify memory layouts of some arrays. Each unmodifiable information constitutes a constraint for the compiler. An important case is the explicitly parallel programs where loop transformations are generally not possible since the programmer has already decided the parallelization [3].

Any transformation matrix must have a full rank and should not violate any data dependences. In the algorithm, after a candidate Q is built, it is checked against data dependences, and discarded if it violates any dependences or its rank is not full.

We now focus on the problem of optimizing locality when some or all array layouts are fixed. We note that each fixed layout requires that the innermost loop index should be in the appropriate array index position (dimension), depending on layout of the array. For example, suppose that the memory layout for a h-dimensional array is such that the dimension k_1 is the fastest changing dimension, the dimension k_2 is the second fastest changing dimension, k_3 is the third etc. The algorithm should first try to place the new innermost loop index j_n only to the ${k_1}^{th}$ dimension of this array. If this is not possible, then it should try to place j_n only to the ${k_2}^{th}$ dimension and so on. If all dimensions up to and including k_h are tried unsuccessfully, then j_{n-1} should be tried for the ${k_1}^{th}$ dimension and so on. In the next subsection we show that this *constrained layout* algorithm is important for global locality optimization.

5.3 Global Optimization Algorithm

In this subsection we show how our algorithm can be extended to work on multiple nests. Since a number of arrays can be accessed by a number of nests and each of these nests may require a different layout for a specific array, the algorithm should find a memory layout for that array that satisfies the majority of the nests.

In the following we present a sketch of a simple heuristic. Our approach is based on the concept of *most costly nest*. Intuitively, this is the nest which takes the most time. Different methods can be adopted to choose this nest. For example the programmer can use *compiler directives* to give hints about this nest. We can also use a metric such as multiplication of the number of loops and the number of arrays referenced in the nest. The nest which has the largest resulting value can be marked as the most costly nest. Then the algorithm proceeds as follows: First, the most costly nest is optimized by using the algorithm presented in Figure 1. After this step, memory layouts for some of the arrays will be determined. Then each of the remaining nests can be optimized using the approach presented for the *constrained layout* case in the previous subsection. After each nest is optimized, new layout constraints will be obtained, and these will be propagated for optimizing the remaining nests.

6 Examples, Simulation Results and Experimental Results

This section presents several examples to illustrate the algorithm. Our experimental suit comprises of some kernels and several representative nests extracted from NAS Benchmarks [2].

```
DO i = 1, n
 DO j = 1, n
  DO k = 1, n
   DO l = 1, n
    A[i,j]+=B[k,i]+C[l,k]
   ENDDO l
  ENDDO k
 ENDDO j
ENDDO i
      (A)
```

```
DO u = 1, n
 DO v = 1, n
  DO w =1 ,n
   DO y = 1, n
    A[w,y]+=B[v,w]+C[u,v]
   ENDDO y
  ENDDO w
 ENDDO v
ENDDO u
      (B)
```

```
DO u = 1, n
 DO v = 1, n
  DO w =1 ,n
   DO y = 1, n
    A[y,u]+=B[v,y]+C[w,v]
   ENDDO y
  ENDDO w
 ENDDO v
ENDDO u
      (C)
```

```
DO u = 1, n
 DO v = 1, n
  DO w =1 ,n
   DO y = 1, n
    A[u,y]+=B[w,u]+C[v,w]
   ENDDO y
  ENDDO w
 ENDDO v
ENDDO u
      (D)
```

```
DO j = 1, n2
 DO i = 1, n1
  Y[j,i]=X[i,j]
 ENDDO i
ENDDO j
   (E)
```

```
DO jj = 0, n2-1, nc
 DO ii = 0, n1-1, nc
  DO j = 1, nc
   DO i = 1, nc
    Z[j,i]=X[i+ii,j+jj]
   ENDDO i
  ENDDO j
  DO i = 1, nc
   DO j = 1,nc
    Y[j+jj,i+ii]=Z[j,i]
   ENDDO j
  ENDDO i
 ENDDO ii
ENDDO jj
 (F)
```

```
DO k = f1(c), f2(c)
 DO i = f3(c), f4(c)
  DO j = f5(c), f6(c)
   .....
    cv[j]=vs[i,j,k,c]
   .....
  ENDDO j
  DO j = f7(c), f8(c)
   lhs[i,j,k,1,c]=0.0
   lhs[i,j,k,2,c]=g1(cv[j-1],rhoq[j-1])
   lhs[i,j,k,3,c]=g2(rhoq[j])
   lhs[i,j,k,4,c]=g1(cv[j+1],rhoq[j+1])
   lhs[i,j,k,5,c]=0.0
  ENDDO j
 ENDDO i
ENDDO k
       (G)
```

```
DO k = 1, nz
 DO j =1, ny
  buf[1,(k-1)*ny+j]=g[1,nx-1,j,k]
  .....
  buf[5,(k-1)*ny+j]=g[5,nx-1,j,k]
  buf[1,(k-1)*ny+j+ny*nz]=g[1,nx,j,k]
  .....
  buf[5,(k-1)*ny+j+ny*nz]=g[5,nx,j,k]
 ENDDO j
ENDDO k
         (H)
```

```
 DO i3 = 2, n3-1
  DO i2 = 2, n2-1
   buff_len=buff_len+1
   buff[buff_len,buff_id]=u[2,i2,i3]
  ENDDO i2
 ENDDO i3
.....
 DO i3 = 2, n3-1
  DO i2 = 2, n2-1
   buff_len=buff_len+1
   buff[buff_len,buff_id]=u[n1-1,i2,i3]
  ENDDO i2
 ENDDO i3
.....
 DO i2 = 1, n2
  DO i1 = 1, n1
   buff_len=buff_len+1
   buff[buff_len,buff_id]=u[i1,i2,2]
  ENDDO i2
 ENDDO i3
       (I)
```

```
DO k = 1, n
 DO j = 1, n-1
   u[k,j]=0.0
   l[j,k]=0.0
 ENDDO j
 l[k,k]=1.0
 DO j = k, n
   u[k,j]=a[k,j]
   DO p = 1, k-1
     u[k,j]-=l[k,p]*u[p,j]
   ENDDO p
 ENDDO j
 IF (k ≤ n-1) THEN
 DO i = k+1, n
  l[i,k]=a[i,k]
  DO p = 1, k-1
   l[i,k]-=l[i,p]*u[p,k]
  ENDDO p
  l[i,k]=l[i,k]/u[k,k]
 ENDDO i
 ENDIF
ENDDO k
       (J)
```

Figure 2: (A) An example four-deep loop nest. (B) Optimized loop nest assuming fixed row-major arrays. (C) Optimized loop nest. (D) Optimized loop nest. (E) An example from the FT benchmark. (F) An example from the FT benchmark. (G) An example from the SP benchmark. (H) An example from the LU benchmark. (I) An example from the MG benchmark. (J) LU decomposition kernel.

We demonstrate the simulation results obtained by using an enhanced version of DineroIII [5], a trace-driven uniprocessor cache simulator. We simulate the miss rates over a range of cache sizes (4K, 8K, 16K, 32K, 64K, 128K), block (cache line) sizes (8, 16, 32, 64, 128, 256) and set-associativities (direct-mapped, 2-way, 4-way, full-associative).

Also presented are empirical results obtained on SPARCstation 5, RS/6000 and SGI Challenge. SPARCstation 5 has a 16K direct-mapped data cache and a 32 MB memory. RS/6000 Model 590 has 256 KB data cache. SGI Challenge has a logically and physically shared memory system. It uses snoopy write-invalidate cache coherence. Each node has 1 MB data cache attached to it. In SGI, during the multiprocessor experiments static scheduling has been employed. Due to space concerns, we do not show the steps or parts of steps which lead to unsuccessful trials; and we only present a subset of our simulation and experimental results.

6.1 Example: A four-deep loop nest

Figure 2:A shows a four-deep loop nest which can benefit from the layout flexibility. The array reference matrices for this nest are as follows. $L^A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$, $L^B = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$ and $L^C = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$.

The algorithm works as follows:
$L^A.Q = \begin{pmatrix} 0 & 0 & 0 & 1 \\ \times & \times & \times & 0 \end{pmatrix}$. Therefore, $q_{11} = q_{12} = q_{13} = q_{24} = 0$ and $q_{14} = 1$.
$L^B.Q = \begin{pmatrix} \times & \times & \times & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$. Therefore, $q_{34} = 0$.
$L^C.Q = \begin{pmatrix} \times & \times & 1 & 0 \\ \times & \times & 0 & 0 \end{pmatrix}$. Therefore, $q_{33} = q_{44} = 0$ and $q_{43} = 1$.
At this point $T^{-1} = Q = \begin{pmatrix} 0 & 0 & 0 & 1 \\ q_{21} & q_{22} & q_{23} & 0 \\ q_{31} & q_{32} & 0 & 0 \\ q_{41} & q_{42} & 1 & 0 \end{pmatrix}$. We set the unknowns to the following values: $q_{22} = q_{23} = q_{31} = q_{41} = q_{42} = 0$ and $q_{21} = q_{32} = 1$ and obtain $T^{-1} = Q = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$. Arrays A and C are column-major whereas the array B is row-major; and the resulting code is shown in Figure 2:C.
Next the compiler tries the other alternative layout (row-major) for A.
$L^A.Q = \begin{pmatrix} \times & \times & \times & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$. Therefore, $q_{14} = q_{21} = q_{22} = q_{23} = 0$ and $q_{24} = 1$.
$L^B.Q = \begin{pmatrix} \times & \times & 1 & 0 \\ \times & \times & 0 & 0 \end{pmatrix}$. Therefore, $q_{13} = q_{34} = 0$ and $q_{33} = 1$.
$L^C.Q = \begin{pmatrix} \times & \times & 0 & 0 \\ \times & \times & 1 & 0 \end{pmatrix}$. Therefore, $q_{43} = q_{44} = 0$.
By setting $q_{12} = q_{31} = q_{32} = q_{41} = 0$ and $q_{11} = q_{42} = 1$, $T^{-1} = Q = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$. Arrays A and C are row-major whereas the array B is column-major; and the resulting code is shown in Figure 2:D.

6.2 Example: A constrained-layout nest

We revisit the example shown in Figure 2:A, this time assuming fixed row-major memory layouts for all arrays.

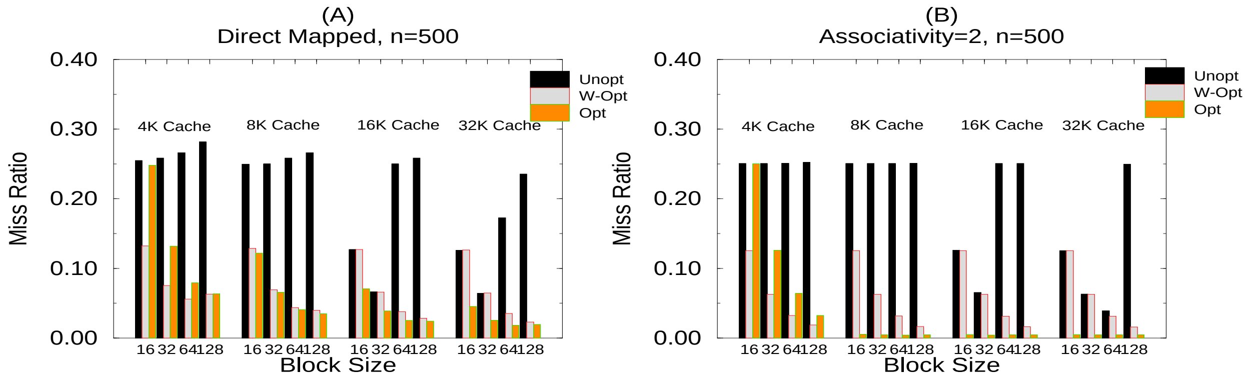

Figure 3: (A) Simulation results of a four-deep nest for different block and cache sizes on a direct-mapped cache. (B) Simulation results of a four-deep nest for different block and cache sizes on a set-associative cache.

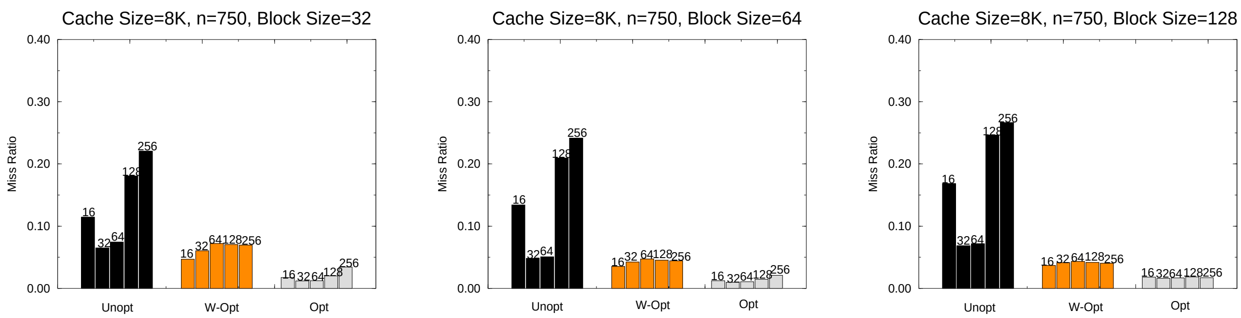

Figure 4: Tile size sensitivity for a four-deep nest.

$L^A.Q = \begin{pmatrix} \times & \times & \times & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$. Therefore, $q_{14} = q_{21} = q_{22} = q_{23} = 0$ and $q_{24} = 1$.
$L^B.Q = \begin{pmatrix} \times & \times & 0 & 0 \\ \times & \times & 1 & 0 \end{pmatrix}$. Therefore, $q_{33} = q_{34} = 0$ and $q_{13} = 1$.
$L^C.Q = \begin{pmatrix} \times & 0 & 0 & 0 \\ \times & 1 & 0 & 0 \end{pmatrix}$. Therefore, $q_{42} = q_{43} = q_{44} = 0$ and $q_{32} = 1$.
We set the unknowns to the following values $q_{11} = q_{12} = q_{31} = 0$ and $q_{41} = 1$ and obtain $T^{-1} = Q = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$.
The resulting code is shown Figure 2:B. Notice that this is the nest that would be obtained for row-major memory layouts, had we used the W-Opt.

Figure 3:A demonstrates miss ratios for this nest with 500×500 double arrays on a direct-mapped cache. We ran experiments with three different versions: unoptimized version (Figure 2:A, Unopt), optimized version by fixing row-major layouts for all arrays (Figure 2:B, W-Opt) and one of the versions obtained by our approach (Figure 2:C, Opt). The results shown indicate that except for the 4K cache, our approach outperforms the W-Opt (Figure 2:B). In order to further understand the sources of the misses in the optimized programs we breakdown the misses into compulsory, capacity and conflict misses. The results indicate that the majority of the misses in the optimized program are due to conflicts which can, in principle, be eliminated by increasing the set-associativity. Figure 3:B shows the miss ratios for this example with 500×500 double arrays on a 2-way set associative cache. As expected, except for the 4K cache, our approach eliminates almost all misses; whereas the W-Opt does not improve the performance at all for some cases.

Tiling (also known as *blocking*) is a technique to improve the locality, and is a combination of strip-mining and loop permutation [14, 15]. Due to interference misses it is difficult to select a suitable tile size. In other words, unless the tile size is tailored according to the matrix size and cache parameters, the performance of tiling may be rather poor [4, 9].

Figure 4 illustrates the insensitivity of the optimized tiled versions to the tile size. The numbers above the bars denote the tile sizes. Notice that while the miss ratio of the unoptimized tiled version is very unstable, those of the optimized versions (Figure 2:B and Figure 2:C) are stable. Notice also that our version outperforms the W-Opt for all tile, block (cache line) and cache sizes.

Figures 6:A and B present execution times for this example with different input sizes on SPARCstation 5 and a single node of SGI respectively. Opt-1 and Opt-2 denote the optimized versions obtained by our algorithm (Figures 2:C and D). Figures 6:C and D, on the other hand, show the execution times on multiple nodes of SGI Challenge with 150×150 and 200×200 double arrays respectively. It can be seen that although the approach based on loop transformations alone can improve the performance, our approach gives the best results on both uniprocessor and multiprocessors. In SPARC, for example, with 250×250 double matrices, our approach (Opt-2) runs in almost 800 seconds less than the W-Opt. On four nodes of the SGI Challenge, with 200×200 double arrays, our version (Opt-2) saves 36 more seconds than W-Opt. This example clearly shows that relaxing the memory layouts can save substantial amounts of time for some nests.

```
DO i = 1, n
 DO j = 1, n
  A[i,j]=B[j,i]*C[i,j]+D[i,j]*LOG(E[j,i])
 ENDDO j
ENDDO i

DO i = 1, n
 DO j = 1, n
  B[i,j]=A[j,i]+E[i,j]
 ENDDO j
ENDDO i
```

(A)

```
DO j = 1, n
 DO i = 1, n
  A[i,j]=B[j,i]*C[i,j]+D[i,j]*LOG(E[j,i])
 ENDDO i
ENDDO j

DO i = 1, n
 DO j = 1, n
  B[i,j]=A[j,i]+E[i,j]
 ENDDO j
ENDDO i
```

(B)

```
DO i = 1, n
 DO j = 1, n
  DO k = 1, n
   C[i,j]+=A[i,k]*B[k,j]
  ENDDO k
 ENDDO j
ENDDO i

DO i = 1, n
 DO j = 1, n
  DO k = 1, n
   F[i,j]+=E[i,k]*C[k,j]
  ENDDO k
 ENDDO j
ENDDO i
```

(C)

```
DO j = 1, n
  DO i = 1, n
    DO k = 1, n
     C[i,j]+=A[i,k]*B[k,j]
    ENDDO k
  ENDDO i
 ENDDO j

DO j = 1, n
  DO i = 1, n
    DO k = 1, n
     F[i,j]+=E[i,k]*C[k,j]
    ENDDO k
  ENDDO i
 ENDDO j
```

(D)

```
DO u = 1, n
  DO v = 1, n
   DO w = 1, n
    C[u,w]+=A[u,v]*B[v,w]
   ENDDO w
  ENDDO v
ENDDO u

DO u = 1, n
  DO v = 1, n
   DO w = 1, n
    F[u,w]+=E[u,v]*C[v,w]
   ENDDO w
  ENDDO v
ENDDO u
```

(E)

Figure 5: (A) A simple benchmark. (B) Optimized version of (A). (C) MxMxM program. (D) A transformed version of (C). (E) Transformed version of (C) by our approach.

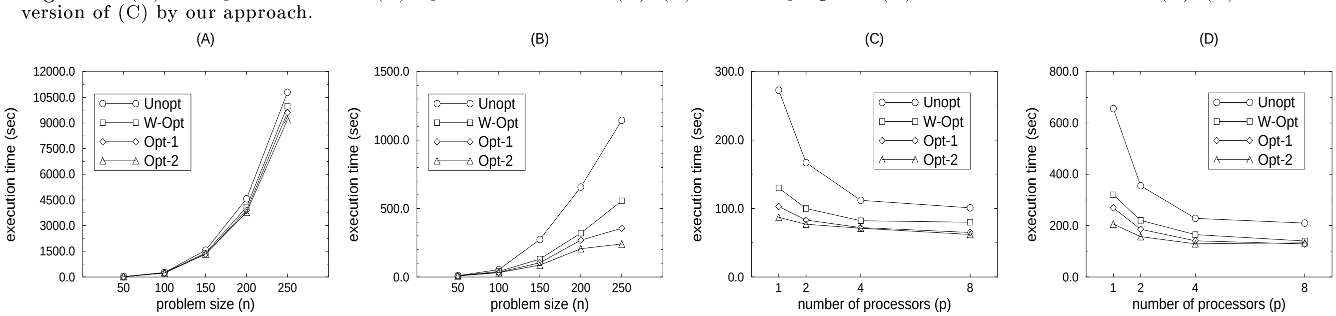

Figure 6: Execution times (A) on SPARCstation 5. (B) on a single node of SGI. (C) on multiple nodes of SGI with 150 × 150 double arrays. (D) on multiple nodes of SGI with 200 × 200 double arrays.

6.3 Example nests from NAS Benchmarks

The NAS Parallel Benchmarks [2] are a set of programs designed to help evaluate the performance of parallel supercomputers. To utilize the cache effectively, the benchmarks generally access data with unit stride. Default layout for the nests is column-major. It should be stressed that the examples considered here are only representative nests.

• FT benchmark uses *simple-transpose* and *complicated-transpose* nests shown in Figure 2:E and F respectively. In Figure 2:E, spatial locality for the array Y is poor; our algorithm attaches row-major layout for Y and column-major layout for X, retaining the original loop order. In Figure 7:A the leftmost group of bars show the performance improvement obtained by our approach for different block sizes on a 16K direct-mapped cache. Notice that the effectiveness of the approach increases with larger block sizes. In Figure 2:F, on the other hand, the reference $Z[j,i]$ in the first loop has poor locality. Our locality optimization algorithm attaches row-major layouts for Z and Y, and column-major layout for X; and interchanges the loops in the second nest placing the i-loop into innermost position. The middle and rightmost bar-charts in Figure 7:A show the improvement obtained by our approach for $nc = 64$ and $nc = 150$, respectively. Since when $nc = 64$, the data used by the innermost loop fits in the cache anyway, our algorithm does not add much.

• An example nest from the SP benchmark is given in Figure 2:G. In order to apply our algorithm, we first distribute the second j-loop over the individual statements. Then our approach attaches layouts for the arrays vs and lhs such that the second dimension in both arrays will be the fastest changing dimension exploiting the spatial locality in the innermost loops. Figure 7:B demonstrates the reduction in cache misses.

• Figure 2:H presents an example nest from the LU benchmark. After distributing the j-loop, our algorithm offers two alternatives: retain the original loop order and make the third dimension of the array g the fastest changing dimension; or apply the loop interchange and make the fourth dimension of the array g the fastest changing dimension. The performance improvement is similar for both the alternatives. Figure 7:C shows the performance improvement. With a block (cache line) size of 128, more than half of the misses are eliminated.

• Three typical loop nests from the MG benchmark are shown in Figure 2:I. In the first nest, since $i2$ is the innermost loop, our global locality algorithm makes second dimension of u the fastest changing dimension. This choice is appropriate for the second nest as well; and for the third nest our algorithm interchanges the loops $i2$ and $i1$. The performance improvement illustrated in Figure 7:D is substantial.

We ran experiments on RS/6000 and SPARCstation 5. Due to space concerns, we only present the execution times for *simple-transpose* nest, in Figures 8:A and B for RS/6000 and SPARCstation 5 respectively. When $n_1 = n_2 = n =$

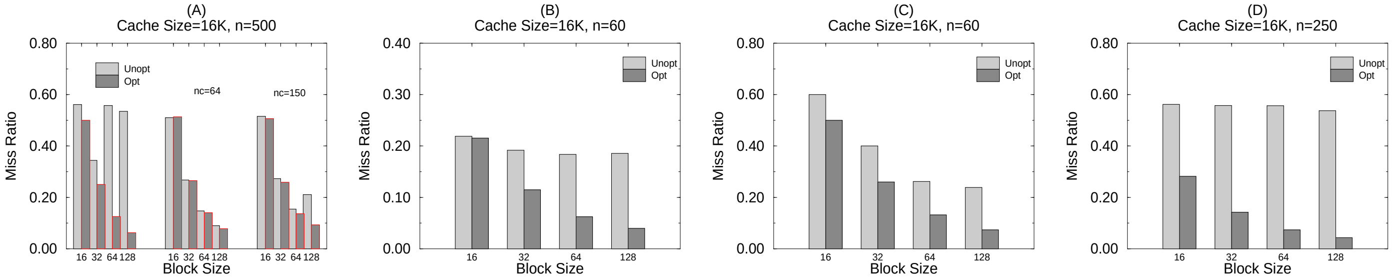

Figure 7: (A) Miss ratios for two example nests from FT. (B) Miss ratios for an example nest from the SP. (C) Miss ratios for an example nest from the LU. (D) Miss ratios for an example nest from the MG.

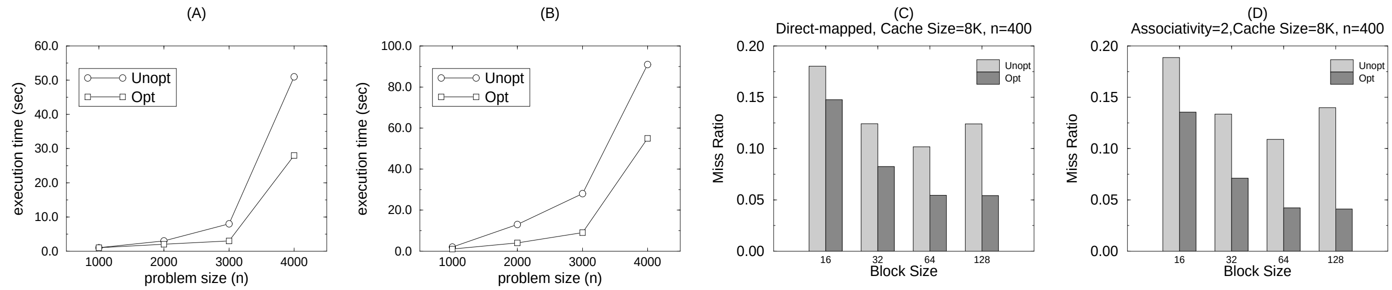

Figure 8: (A) Execution times for *simple-transpose* on RS/6000. (B) Execution times for for *simple-transpose* on SPARCstation 5. (C) Miss ratios for the LU decomposition. (D) Miss ratios for the LU decomposition.

4000, on RS/6000, there is 45% performance improvement.

6.4 LU Decomposition

Figures 2:J shows an LU decomposition algorithm. Our global locality algorithm identifies the nests containing the innermost p-loops as the most costly nests, and attaches row-major layout for the array l and column-major layout for the array u. Figures 8:C and D show the miss rates of the unoptimized and optimized nests for an 8K data cache with direct-mapping and a set-associativity of size 2 respectively. As can be seen, our algorithm reduces the original miss rates by 7% to 40%.

6.5 Other examples

Figure 9:A shows the *normalized* miss rates for the *dgemm* routine from BLAS. This routine performs the following operation: $C = \alpha f(A) f(B) + \beta C$, where $f(X) = X$ or X^T, and α and β are scalars. Both the unoptimized and the optimized versions have been called four times, each of which with different operation, and the average miss rates have been computed. Below each pair of bars is given the triple *cache size, block size, associativity*. In the simulation 500×500 double precision matrices are used.

Figure 9:B demonstrates the performance improvement on *dtrsl*, a routine from LINPACK which solves the systems of the form $Tx = b$ or $T^T x = b$ where T is a triangular matrix of order n. While for optimizing the *dgemm* both data and loop transformation are used, for *dtrsl* only data transformations are used.

Finally, we show the impact of our algorithm on two programs from [3]. The program shown in Figure 5:A is a simple benchmark. Figure 9:C shows the improvement obtained by our approach. For each cache size, the three bars from left to right correspond to unoptimized version with column-major layouts, unoptimized version with row-major layouts and version optimized by our approach. In the optimized version, the loops in the first nest are interchanged; and the following layouts are assigned: A, C, and D are column-major; B and E are row-major. With these optimizations, the spatial locality for every reference is exploited in the innermost loop and the optimized program is given in Figure 5:B. 400×400 double matrices are used.

Figure 5:C shows a program named MxMxM from [3]. This program computes the product of three square matrices. The version obtained by using the method in [3] is presented in Figure 5:D. The layouts for A and E are row-major, whereas those for the other arrays are column-major. For the four of the total eight references, the spatial locality is exploited in the innermost loop; and for the remaining four references it is exploited in the second loop. In comparison, our global approach transform this program to the one shown in Figure 5:E. All arrays are row-major (a version with all column-major arrays is also possible). For six of the total eight references, the spatial locality is exploited in the innermost loop; for the two references it is exploited in the second loop. The normalized miss rates for an 8 KB direct-mapped cache is are shown in Figure 9:D. These results reveal that our approach is better in the sense that it finds all possible transformations and layouts; and selects the most optimal one.

7 Impact on False Sharing

In shared-memory multiprocessors when processors make references to different data items within the same cache line, *false sharing* occurs [12]. Since cache coherence is maintained on a cache block (line) basis, when one processor modifies a data item, it causes an invalidation in the other processors' cache. It is well known that one of the main causes of the false sharing is the parallelization of a loop that carries spatial reuse [10, 15]. On the other hand, the larger the granularity of parallelism the better it is; because the synchronization overhead will diminish with the increas-

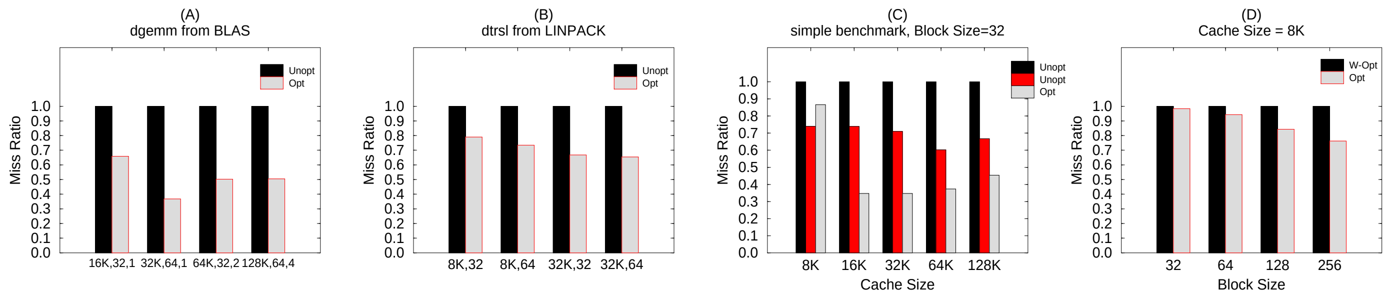

Figure 9: (A) Normalized miss rates for *dgemm*. (B) Normalized miss rates for *dtrsl*. (C) Normalized miss rates for a simple benchmark. (D) Normalized miss rates for the MxMxM.

ing parallelism granularity. A recent study shows that apart from affecting the synchronization cost, the granularity of application parallelism is also an important determinant of applications' memory behavior [13].

To summarize, in order to optimize the locality for parallel machines, the maximum granularity of parallelism is obtained and the outermost parallelized loops should not carry any spatial reuse. Since our algorithm tries to achieve this goal by both changing the loop orders and memory layouts, we believe that it will be very effective at eliminating false sharing on multiprocessors.

Let us consider the loop shown in Figure 2:A, this time assuming column-major layouts. Notice that applying the approaches like in [7] and [10] for this nest result in the same nest; that is this loop order is the most desirable one for the fixed column-major layouts. If the outermost loop (i) is parallelized then the reference $a[i,j]$ will cause false sharing. In comparison, for both of our optimized versions (Figure 2:C and Figure 2:D), the outermost loop (u) can now safely parallelized, without an apparent danger of false sharing.

8 Conclusions and Future Work

We designed a compiler algorithm which transforms the loop nests and changes the memory layouts of arrays in a unified framework. Our algorithm can either be employed alone, or can be combined with low-level locality optimizations such as tiling [9].

We have shown that our approach is more effective in reducing sensitivity of tiling to the tile size and at eliminating false sharing than the approaches based on loop transformations alone. In fact, when the memory layouts are fixed, our approach produces the same results as other approaches such as those of Li [10] and Wolf and Lam [14], if temporal locality is not considered. Both our simulation results and empirical results provide encouraging evidence that our approach is likely to be successful on both uniprocessors and multiprocessors. Work is in progress on evaluating the performance of our approach on full NAS benchmarks [2] on multicomputers. We also intend to apply our technique to the other levels of memory hierarchy.

References

[1] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In *Proc. Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

[2] D. Bailey, T. Harris, W. Saphir, R. van der Winjngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0., Technical Report NAS-95-020, NASA Ames Research Center, CA, December 1995.

[3] M. Cierniak and W. Li. Unifying Data and Control Transformations for Distributed Shared Memory Machines. *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, California, June 1995.

[4] S. Coleman and K.S. McKinley. Tile Size Selection Using Cache Organization and Data Layout, In *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[5] M. D. Hill and A. J. Smith. Evaluating Associativity in CPU Caches, *IEEE Trans. on Computers*, C-38, 12, December 1989, pages 1612–1630.

[6] Y.-J. Ju and H. Dietz. Reduction of Cache Coherence Overhead by Compiler Data Layout and Loop Transformations. *Proc. 4th Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[7] K. McKinley, S. Carr, and C.W. Tseng. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems*, 1996.

[8] M. Kandemir, J. Ramanujam, and A. Choudhary. A Compiler Algorithm for Optimizing Locality in Loop Nests. Technical Report, Northwestern University, Evanston, IL, April 1997.

[9] M. S. Lam, E. Rothberg and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proc. 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[10] W. Li. Compiler Optimizations for Cache Locality and Coherence. Technical Report 504, Dept. of Computer Science, University of Rochester, NY, April 1994.

[11] C.-W. Tseng, J. Anderson, S. Amarasinghe, and M. Lam. Unified Compilation Techniques for Shared and Distributed Address Space Machines. In *Proc. 1995 International Conference on Supercomputing (ICS'95)*, Barcelona, Spain, July 1995.

[12] J. Torrellas, M. S. Lam, and J. L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.

[13] E. Torrie, C-W. Tseng, M. Martonosi, and M. W. Hall. Evaluating the impact of advanced memory systems on compiler-parallelized codes. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, June 1995.

[14] M. Wolf and M. Lam. A Data Locality Optimizing Algorithm. In *Proc. ACM SIGPLAN 91 Conf. Programming Language Design and Implementation*, pages 30–44, June 1991.

[15] M. Wolfe. High Performance Compilers for Parallel Computing. Addison-Wesley Publishing Company, CA, 1996.