Improving Cache Locality by a Combination of Loop and Data Transformations

Mahmut Kandemir, *Student Member, IEEE* J. Ramanujam, *Member, IEEE*
Alok Choudhary, *Member, IEEE*

Abstract—**Exploiting locality of reference is key to realizing high levels of performance on modern processors. This paper describes a compiler algorithm for optimizing 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. Our approach is very effective at reducing cache misses and can optimize some 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 accommodate this case, and demonstrate how it can be used to optimize multiple loop nests. Experiments on several benchmarks show that the techniques presented in this paper result in substantial improvement in cache performance.**

Keywords— **caches, data reuse, locality, loop and data transformations, optimizing compilers.**

I. Introduction

In most computer systems, exploiting locality of reference is key to achieving high levels of performance. It is well known that increasing the cache hit rates is one of the most important factors in reducing the average memory latency. Fine tuning of the cache coherence protocol and selecting an appropriate block (line) size are very important techniques [17], [16] for improving cache performance. Recent advances include the work of Gonzalez et al. [6] who have developed a dual data cache and a selective data cache; Sanchez et al. [15] have proposed a static locality analysis technique oriented towards optimize programs that exhibit high conflict miss ratios and applied the results of their technique to these new caches developed in [6]. Software techniques [16] complement the advances in cache hardware and organization [17], and are capable of delivering additional performance. It has been observed that compiler techniques are useful for optimizing locality in both uniprocessors and multiprocessors, and in reducing the number of coherence related misses [3].

From a software point of view, programmers and compiler writers often attempt to change the access patterns of a program so that a majority of accesses are satisfied from the cache memory. Several efforts have been aimed at using iteration space (loop) transformations and scheduling techniques to improve locality [3], [13], [14], [20]; these techniques improve data locality *indirectly* as a result of modifying the iteration space traversal order.

This work is supported in part by NSF Young Investigator Award CCR-9357840, NSF CCR-9509143. The work of J. Ramanujam is supported by NSF Young Investigator Award CCR-9457768 and NSF grant CCR-9210422.

Mahmut Kandemir is with the Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY 13244.

J. Ramanujam is with the Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803.

Alok Choudhary is with the Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208.

We offer a compiler approach to enhance the cache performance of scientific codes on uniprocessors and multiprocessors. In a unified framework, our approach considers modifying array layouts in memory and transforming loop nests suitably to exploit spatial locality. We simulate miss rates for several programs in order to demonstrate that our approach is very effective at reducing number of cache misses, and report execution times on the SGI Challenge shared memory multiprocessor. We conclude that fixing the memory layouts for all arrays—as in C and Fortran—limits performance that could otherwise have been obtained from the programs. In this paper we make the following contributions:

- We present a new algorithm for optimizing the spatial locality characteristics of nested loops. This algorithm applies *both* data and loop transformations. When the data transformation part is not activated (i.e., when the layouts are fixed), in most cases, it obtains the same results as the existing loop transformation techniques.
- We argue that the known approaches which consider only loop transformations (e.g., loop permutations [14], [13], [20], tiling [12], [2], [11], etc.) might be insufficient for some cases.
- We demonstrate the effectiveness of our approach by using both simulation results and execution-time measurements, and show that our approach is effective on both uniprocessors and multiprocessors.

Since our approach is oriented towards optimizing spatial locality, it is generally more effective with large cache block sizes. Since the architectural trend is anyway towards larger block sizes and higher associativities, we believe that our approach will be suitable for future architectures as well.

This paper is organized as follows. Section II presents a brief summary of the necessary background. In Section III, we discuss related work on cache locality. Section IV discusses the algorithm for optimizing locality in a single loop nest. In Section V, we extend this algorithm to multiple loop nests. Section VI presents a set of experimental results which illustrate the efficacy of our approach. Section VII presents our work on false sharing. In Section VIII, we conclude with a summary and discussion.

II. Preliminaries

We represent each iteration of a loop nest of depth n using a loop iteration vector $\vec{I} = (i_1, i_2, ..., i_n)$ where i_k is the value of the index of the kth loop from the outermost. The subscript function in each reference to an m-dimensional array U in such a loop nest is assumed to be an affine function of the iteration represented by $\vec{I}$, i.e., $\vec{I}$ is mapped onto data element $L^U\vec{I} + \vec{b^U}$.

Here L^U is an $m \times n$ matrix called the *reference (access) matrix* and the m-vector $\vec{b^U}$ is called the *offset vector* [20].

Linear mappings between iteration spaces of loop nests can be modeled by non-singular transformation matrices [13]. On applying a transformation T to a loop with index $\vec{I}$, the transformed loop index becomes $\vec{I'} = T\vec{I}$ and the transformed reference matrix becomes $L^U T^{-1}$. Similarly if $\vec{d}$ is the distance (direction) vector, on applying T, then $T\vec{d}$ is the new distance (direction) vector. A transformation is *legal* if and only if $T\vec{d}$ is lexicographically positive for every $\vec{d}$ [21]. In this paper we denote T^{-1} by Q. An important characteristic of our algorithm is that the entries of $Q = [q_{ij}]$ are derived systematically using the array reference matrices.

In order to obtain good performance from programs running on a machine that contains some sort of cache memory, cache locality should be exploited. That is, a data brought into cache should be reused as much as possible before it is replaced. The reuse of the same data while it is still in the cache is termed as *temporal locality*, whereas the use of the nearby data in a cache line is called *spatial locality* [20]. We have to stress that a program may have data reuse, but due to the replacement of the data, it might not be able to exploit cache locality.

The scope of our work is dense matrix programs with affine subscript functions and affine loop bounds; we focus mainly on self-spatial reuse since the cases where the group-spatial reuses bring an additional reuse dimension over the self-spatial reuse space are rare [21]. We assume that the memory layout of an m-dimensional array can be in any of the $m!$ forms, each of which corresponding to layout of data in memory linearly by a nested traversal of the axes in some predetermined order. In other words, the data storage schemes we consider can be expressed as permutations of the dimensions of the array. For a two-dimensional array, these are only row-major and column-major layouts. For a three-dimensional array, there are six possible storages, and so on. Each layout that we consider in this paper has a *fastest changing dimension*; that is, the innermost dimension in the traversal of array in memory. For instance, for row-major layouts, the last dimension is the fastest changing dimension. Our algorithm only determines the fastest changing dimension; this is because the relative order of the other dimensions may not be as important, assuming large array bounds.

III. Related Work

A. Loop Transformations for a Fixed Layout

Wolf and Lam [20] present definitions of different types of reuse and propose an algorithm to optimize locality. Their algorithm evaluates a subset of legal loop transformations and transforms the loop nest such that the locality is maximized. They focus on tiling the innermost loops. In contrast, Li [13] uses the concept of *reuse distance.* His algorithm can represent the reuse vectors precisely and the transformations are operated directly on reuse vectors. McKinley et al. [14] offer a unified optimization technique consisting of loop permutation, loop fusion and loop distribution. None of these consider data space (memory layout) transformations. In this paper, we show that data space transformations can also make difference on the locality properties of the programs. Moreover, by unifying data space transformations with iteration space transformations, locality can be exploited in a better way, which is not possible using the loop and data transformations by themselves.

Unlike the pure loop transformations discussed above, Anderson et al. [1] propose a data transformation technique for distributed shared memory machines. By using two types of data transformations (strip-mining and permutation), they try to make the data accessed by the same processor contiguous in the shared address space. Their algorithm inherits parallelism decisions made by a previous phase of the SUIF compiler [19]; so, in a sense, their approach does not lend itself to a direct comparison with ours, which attempts to come up with both loop and data transformations to improve locality.

B. Combined Loop and Data Transformations

Cierniak and Li [3] present a unified approach like ours 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* representing layouts and transformation matrix. At the end, the following equality is obtained: $T^T\vec{v} = L^T\vec{m}$. In this formulation, only the reference matrix L is known. The algorithm tries to find T, the iteration-space transformation matrix; $\vec{m}$, a mapping vector that can assume $h!$ different forms for a h-dimensional array; and $\vec{v}$, 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 0s and 1s. Second, the value of the stride vector $\vec{v}$ is assumed to be known beforehand. The algorithm constructs the matrix T row by row by considering a restricted set of legal mappings. In comparison, our approach is more accurate, as it does not restrict the search space of possible loop transformations. Also our approach is simpler for embedding in a compilation system, since it does not require a prior knowledge of any vector such as $\vec{v}$. Our extension to multiple nests is also different from the one proposed by Cierniak and Li [3] for global optimization. Cierniak and Li [4] also used data transformations for optimizing JAVA byte-codes.

IV. Algorithm for Optimizing Locality

In this section we explain our algorithm that automatically transforms a given loop nest to exploit spatial locality and assigns appropriate memory layouts for arrays, both in a unified framework.

We assume that C is the array accessed on the LHS with the access function $L^C\vec{I} + \vec{b^C}$, whereas A is an array reference from the RHS with the access function $L^A\vec{I} + \vec{b^A}$. Let $j_1, j_2, ..., j_n$ be the loop indices of the *transformed* nest, starting from outermost position. The following is a brief explanation of our algorithm. More technical details can be found in [10], [9]. In comparison to [10], the techniques presented in this paper handle false sharing as well.

• *Fix the Layout of the LHS Array:* Our loop transformation matrix should be such that the LHS array of the transformed loop should have 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 $(0, ..., 0, 1)$ and all entries of the last column except the one in r^{th} row are zero. Although this is a stronger requirement than necessary, it is suitable for our purposes in this work. After that, the LHS array can be stored in memory such that the r^{th} dimension is the fastest changing dimension. This approach effectively exploits the spatial locality for this reference. Notice that all possible values for r should be considered.

• *Fix the Layouts of the RHS Arrays:* 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 row r row of the original reference matrix of the LHS array, then the algorithm attempts to store this RHS array in memory such that its dimension s will be the fastest changing dimension. Note 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. In the ideal case, each RHS array will have a row identical to the r^{th} row of LHS array and can be stored on memory such that the corresponding dimension will be the fastest changing dimension.

If the condition stated above does not hold for a RHS array A, it 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 $A(*, ..., *, f(j_{n-1}), *, ..., *)$, where $f(j_{n-1})$ 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 in exploiting the spatial locality at the second innermost loop. If no such transformation is possible, the transformed loop index 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*.

• *Pick up the Best Alternative:* After a transformation and corresponding memory layouts are found, these are recorded and the next alternative memory layout for the LHS is tried and so on. Among all the feasible solutions, the best one is chosen. The best alternative is the one that exploits spatial locality in the innermost loop for the *maximum* number of array references.

It should be emphasized that the algorithm determines the transformation matrix T (actually inverse of it) and memory layouts *together*. That is, depending on the resultant access matrices from the selected loop transformation, we determine the fastest changing dimension for each array. The following points should also be mentioned. First, it should be noted that the algorithm first optimizes the LHS array. Although this is not strictly necessary, we found it useful as the LHS array is read and written, whereas the other arrays are only read. Second, a special case occurs when an array is referenced more than once. If all of these references belong to the same uniformly generated set [5] (i.e., have the same L matrix) then it is enough to consider only one of them. If, on the other hand, there are references to the same array with different access matrices, then a reasonable heuristic might be to concentrate on the most frequently occurring reference. Third, we use the method given in [13] with appropriate modifications for completing a partial matrix to a full non-singular transformation matrix such that all data dependences are observed. Our algorithm performs an exhaustive search in the worst case. Using the structure of the LHS reference allows us to explore the search space of layouts in a controlled manner; in a sense, we use an enumeration tree of the hierarchically structured search space, exploring a current partial configuration if and only if it is a feasible partial configuration. The worst-case complexity of the algorithm is $\theta(m^v n^3)$ where m is the maximum array dimensionality, v is the number of arrays, and n is the number of loops in the nest ($n \geq m$). In practice, however, the algorithm is very fast.

A. Illustration of the Algorithm

Figure 1(a) shows the ijk matrix-multiply routine. The reference matrices are as follows:

$$L^C = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, L^A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, L^B = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

For the sake of clarity, we only show the successful steps of the algorithm which proceeds as follows ($\times$ denotes a *don't care* entry):

The compiler first tries column-major layout for array C.

$L^C Q = \begin{pmatrix} 0 & 0 & 1 \\ \times & \times & 0 \end{pmatrix}$. Thus $q_{11} = q_{12} = q_{23} = 0$ and $q_{13} = 1$.

$L^A Q = \begin{pmatrix} 0 & 0 & 1 \\ \times & \times & 0 \end{pmatrix}$. Therefore $q_{33} = 0$.

$L^B Q = \begin{pmatrix} \times & 1 & 0 \\ \times & 0 & 0 \end{pmatrix}$. Therefore $q_{22} = 0$ and $q_{32} = 1$.

At this point $T^{-1} = Q = \begin{pmatrix} 0 & 0 & 1 \\ q_{21} & 0 & 0 \\ q_{31} & 1 & 0 \end{pmatrix}$. Setting $q_{21} = 1$, $q_{31} = 0, T^{-1} = Q = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$.

All the arrays are column-major and the resulting code is shown in Figure 1(b).

Next the compiler tries the other alternative memory layout (row-major) for array C.

$L^C Q = \begin{pmatrix} \times & \times & 0 \\ 0 & 0 & 1 \end{pmatrix}$. Thus $q_{13} = q_{21} = q_{22} = 0$ and $q_{23} = 1$.

$L^B Q = \begin{pmatrix} \times & \times & 0 \\ 0 & 0 & 1 \end{pmatrix}$. Therefore $q_{33} = 0$.

$L^A Q = \begin{pmatrix} \times & 0 & 0 \\ \times & 1 & 0 \end{pmatrix}$. Therefore $q_{12} = 0$ and $q_{32} = 1$.

At this point $T^{-1} = Q = \begin{pmatrix} q_{11} & 0 & 0 \\ 0 & 0 & 1 \\ q_{31} & 1 & 0 \end{pmatrix}$. Setting $q_{11} = 1$, $q_{31} = 0$, we obtain $T^{-1} = Q = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$.

Notice that our first optimized nest is the same as the nest obtained by earlier works [13], [14]. Our other optimized nest is the same nest used in Lam et al. [12] for row-major layouts. Since, even when the layouts are fixed as row-major or column-major, we exhaustively search for all possible loop transformations, in most cases (omitting temporal locality) we replicate the results obtained by pure loop oriented approaches such as Li [13].

```
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
 (a)
```

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

```
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
 (c)
```

Fig. 1. (a) Matrix-multiply nest. (b-c) Optimized versions of (a).

V. Global Locality Optimization: Multiple Loop Nests

In this section, we address the problem of optimizing a *collection (sequence) of loop nests* each accessing a subset of the arrays in the program. It is easy to show that the problem of finding a global array layout and loop order combinations that satisfy all the nests is NP-complete even for the restricted case where only row-major and column-major arrays are considered. Therefore, we present a heuristic for this problem.

A. 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 change memory layouts. In fact, the order of loops in the nest may only be partially changed or may not be changed at all. Similarly, the compiler may not be able to change the memory layouts of some arrays. Each unmodifiable information constitutes a constraint for the compiler.

We now focus on the problem of optimizing locality when some or all the 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 form of the array. For example, suppose that the memory layout for an m-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 compiler 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 very important for global locality optimization.

B. Global Locality Optimization Algorithm

The algorithm should find a memory layout for that array that satisfies the majority of the nests. Our approach is based on the concept of the *most costly nest*. Intuitively, this is the nest which takes the most (memory) time and should be optimized. Different methods can be adopted to choose this nest. For example, profiling may be used or the programmer can use *compiler directives* to give hints about this nest. Then the algorithm proceeds as follows: First, the most costly nest is optimized using the algorithm presented in Section IV. After this step, the 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 optimization of the next nest. Note that the order of processing for the remaining nests may be important. If the number of nests is small, a more aggressive approach can apply this heuristic by considering each nest in turn as the most costly nest. For a more formal discussion of the algorithm, we refer the reader to [9].

VI. Experimental Results

This section presents experimental results for a number of example programs. We demonstrate the simulation results obtained by using an enhanced version of DineroIII [7], a trace-driven uniprocessor cache simulator. We simulate the miss rates over a range of cache sizes, block sizes, and set-associativities. For the matrix-multiply nest, we also present the execution times obtained on an SGI Challenge multiprocessor. This machine uses snoopy write-invalidate cache coherence. Each node has 1 MB data cache attached to it. During the multiprocessor experiments static scheduling has been employed. Due to space concerns, we present only a subset of our results. More results as well as a quantitive comparison with Li's algorithm [13] can be found in the longer version of our paper [9].

A. Matrix-Multiply

In Section IV-A we showed how our algorithm optimizes this nest. Here we present experimental results. Figure 2 shows the miss ratios for the matrix-multiply nest with 500×500 double arrays on a direct-mapped cache. We present four different versions of the program: unoptimized, optimized (all arrays column-major), tiled [12] version of the unoptimized nest, and finally the tiled version of the optimized nest. The first thing to notice is that the tiled-optimized version outperforms the rest for all cache and block sizes. It is also important to note that for some cases, even the optimized nest without tiling performs better than the tiled-unoptimized version. The tile size is fixed at 32 for every loop in the tiled versions.

Figures 3(a) shows the execution times for the matrix-multiply nest with different input sizes on a single node of the SGI Challenge and Figure 3(b) gives the execution times on different number of processors on the SGI Challenge with $2,000 \times 2,000$ double precision arrays. We note that there is a 20% performance improvement over the unoptimized nest on single node. For both the unoptimized code (`Unopt`) and the optimized code (`Opt`), only the outermost loop is parallelized.

B. Example Nests from NAS Benchmarks

The NAS Parallel Benchmarks are a set of programs designed to help evaluate the performance of parallel supercomputers. To

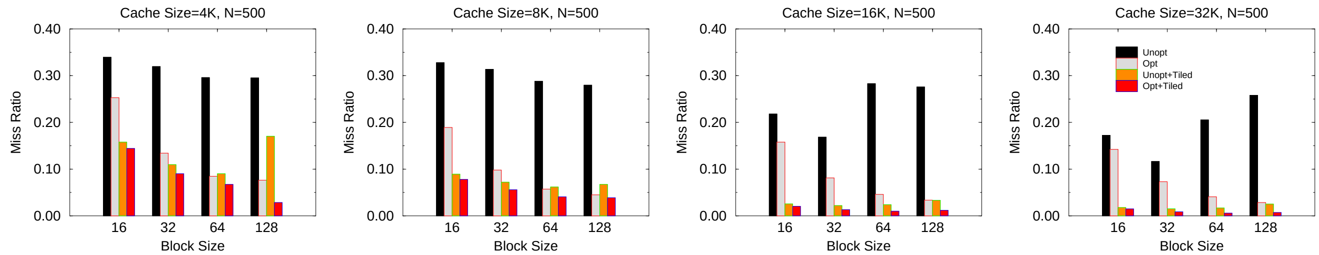

Fig. 2. Simulation results for the matrix-multiply nest.

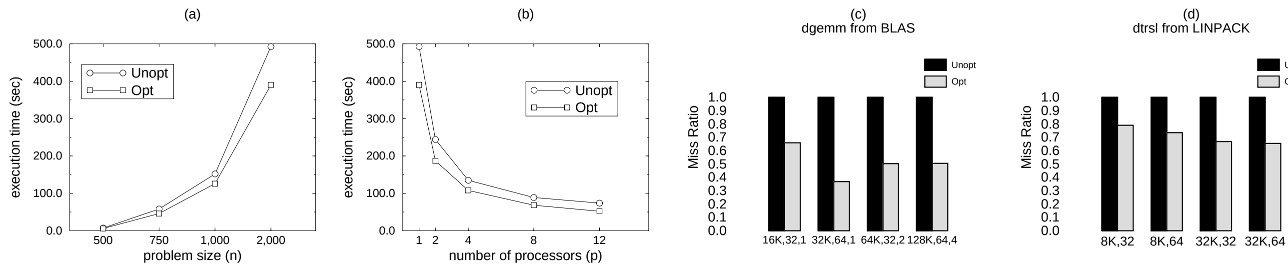

Fig. 3. (a) Execution times for the matrix-multiply nest on a single node. (b) Execution times for the matrix-multiply nest on multiple nodes. (c) Miss rates for *dgemm*. (d) Miss rates for *dtrsl*.

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, not the whole programs.

FT Benchmark: This kernel uses *simple-transpose* and *complicated-transpose* nests. In Figure 4(a) the leftmost group of bars show the performance improvement for the *simple-transpose* obtained by our approach for different block sizes. Notice that the effectiveness of the approach increases with larger block sizes. The middle and rightmost bar-charts show the improvement for the *complicated-transpose* obtained by our approach for the tile sizes of 64×64 and 150×150, respectively. Since when tile size is 64×64, the data used by the innermost loops fit in the cache, our algorithm does not add much. It should be emphasized that this *complicated-transpose* nest is specifically meant for exploiting the cache memory. This example shows that the performance of a blocked (or tiled) loop nest can sometimes be further improved if proper data layout optimizations are applied.

SP Benchmark: Figure 4(b) illustrates the reduction in cache misses for typical loop nest from the SP benchmark after loop distribution has been applied. As can be seen, for a block (cache line) size of 64, the miss rate of the optimized program is 35% of that of the unoptimized. We chose this example to illustrate that sometimes, transformations such as loop distribution enable the applicability of our techniques.

LU Benchmark: Figure 4(c) shows the performance improvement for a typical loop nest. With a block size of 128, more than half of the misses are eliminated.

MG Benchmark: The performance improvement illustrated in Figure 4(d) for a typical loop nest is substantial. It should be emphasized that for this program fragment, neither data transformations alone nor loop transformations alone can optimize spatial locality for all arrays. Therefore, a combination of data and loop transformations derived using a technique like ours is crucial for the best performance.

C. Additional Examples

Figure 3(c) shows the 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 for a matrix X, 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. The miss rates in the figure are normalized such that the miss rate for the unoptimized version is always 1. 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 3(d) demonstrates the performance improvement on *dtrsl*, a routine from LINPACK which solves the systems of the form $T\vec{x} = \vec{b}$ or $T^T\vec{x} = \vec{b}$ where T is a triangular matrix of order n. While for optimizing the *dgemm* both loop and data transformations have been used, for *dtrsl* only the data transformations have been applied.

VII. False Sharing

In shared-memory multiprocessors, when processors make references to different data items within the same cache block even though there is no dependence, *false sharing* may occur [8], [18]. Since cache coherence is maintained on a block basis, when one processor modifies a data item, it causes an invalidation in the other processors' cache. One of the main causes of the false sharing is the parallelization of a loop that carries spatial reuse [13]. Reducing the extent of false sharing can improve the scalability of parallel applications as well as the execution time. On the other hand, the larger the granularity of parallelism the better it is; because the synchronization overhead will diminish with the increasing parallelism granularity. Therefore, to get the maximum benefit from shared memory multiprocessors, parallelism and locality should be optimized and the false sharing should be reduced as much as possible. This optimization problem looks very hard because of the numerous factors

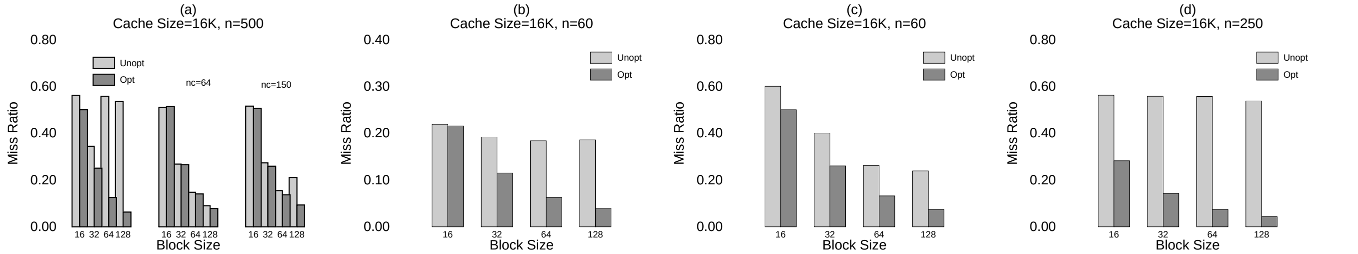

Fig. 4. (a) Miss rates for two example nests from FT. (b) Miss rates for an example nest from the SP. (c) Miss rates for an example nest from the LU. (D) Miss rates for an example nest from the MG.

involved in the optimization process.

We now discuss how our algorithm can be extended to work in a shared memory multiprocessor environment. We start by observing that so far our algorithm derives good loop and data transformations to optimize spatial locality in the innermost loops if possible. As a consequence, (in many cases) the outermost loops in the transformed nest will not carry any spatial reuse. Therefore, the compiler can safely parallelize these loops as since they carry no spatial reuse most probably they will not cause false sharing. As a rule, the compiler should always refrain from parallelizing a loop which carries spatial reuse. In the following we only focus on loop permutations.

Essentially, we want the transformed loop nest to be of the following generic form:

```
DO i_1 = ···, ···
 ...
  DOALL i_k = ···, ···
   ...
    DO* i_m = ···, ···
     ...
      DO* i_n = ···, ···
       loop body
      ENDDO* i_n
     ...
    ENDDO* i_m
   ...
  ENDDOALL i_k
 ...
ENDDO i_1
```

In this generic form, all the spatial reuse will be carried by only a subset of the loops between i_m and i_n, including the both and these loops and the loops between them will not be parallelized. There will be only a single parallel loop i_k where $1 \leq k \leq m - 1$. We want k to be as small as possible. The best possible loop nest will have $k = 1$ and $m = n$ and only i_n will carry all the spatial reuse in the program. The steps taken by our approach are as follows:

(1) Run the locality optimization algorithm explained in this paper to obtain a sequence of alternative transformed programs. The possible loop nests are permutations of the original loop nest. The possible array layouts are row-major, column-major, and higher equivalents of them.

(2) Then the compiler starts handling these loops one at a time. For a given transformed loop, it checks which outermost loop can run parallel. Then it checks whether that loop carries any spatial reuse as well. If so, the compiler discards that alternative and focuses on the next. This process stops when the compiler finds the set of all alternative loop nests which fit in our generic form explained above.

(3) Among the candidate alternatives, the compiler chooses the one which has the outermost loop parallelism. We use spatial locality as a tie breaker if there is a need to do so. If there is still more than one such alternative, any of those will do the job. Consider the following example.

```
DO i = 1, N - 1
 DO j = 1, N
  DO k = 2, N
   A[i,j,k]=B[j,k,i]+A[i,j,k-1]+A[i,j,k]
   B[j,k,i+1]=A[i,j,k]+B[j,k,i]*B[j,k,i]
  ENDDO k
 ENDDO j
ENDDO i
```

TABLE I

POSSIBLE LOOP PERMUTATIONS AND THE BEST FASTEST CHANGING DIMENSIONS (DENOTED X).

	array A			array B		
order	1	2	3	1	2	3
i,j,k			X		X	
i,k,j		X		X		
j,i,k			X		X	
j,k,i	X					X
k,i,j		X		X		
k,j,i	X					X

Table I shows all the possible loop permutations and for each array the fastest changing dimension (marked with X) for each permutation. For example, for the i,j,k loop order, the third dimension of A and the second dimension of B should be the fastest changing dimensions. An application of dependence analysis [21] reveals that the only parallelizable loop is the j loop. Let us focus on the loop order k,i,j. This loop order results in very good spatial locality provided that for array A the second dimension is the fastest changing dimension and for array B the first dimension is the fastest changing dimension. Notice that in that case the spatial reuse for both the arrays are carried by the j loop. Since this loop is the only parallelizable loop in the nest, when it runs parallel there will be false sharing for both A and B. Since this alternative does not fit in our generic optimized loop nest explained above, the compiler discards it. With a similar analysis, we can eliminate the alternative i,k,j as well. The remaining four alternatives fit in our generic form. Since j,i,k and j,k,i have the outermost loop parallelism we select one of them as our transformed nest.

To sum up, after the second step of our approach we have four candidates and after the third step, we have only two candidates which are equally optimized from the points of view of parallelism and locality.

VIII. SUMMARY

This paper presents a new algorithm for improving cache locality in scientific computations. Our algorithm transforms the loop nests and changes the memory layouts of multi-dimensional arrays in a unified framework. Our algorithm can either be employed alone, or can be combined with other locality optimizations such as tiling and transformations such as loop fusion and loop distribution. Experimental results on several programs provide strong evidence that our approach is likely to be successful on both uniprocessors and multiprocessors.

We are currently exploring the possibility of dynamically changing data layouts and will be evaluating the relative merits of this considering the runtime overhead incurred in such a change. In addition, we plan to work on the problem of improving spatial locality in sparse computations.

REFERENCES

[1] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformations for multiprocessors. In *Proc. 5th ACM SIGPLAN Symp. Principles & Practice of Parallel Programming*, July 1995.

[2] S. Carr and R. Lehoucq. Compiler blockability of dense matrix factorizations. *ACM Trans. Mathematical Software* 23(3), September 1997.

[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*, June 1995.

[4] M. Cierniak and W. Li. Briki: an optimizing Java compiler. In *Proc. the IEEE CompCon '97* , San Jose, California, February 1997.

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

[6] A. Gonzalez, C. Aliagar, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *Proc. ACM International Conference on Supercomputing,* pages 338–247, July 1995.

[7] M. Hill and A. Smith. Evaluating associativity in CPU caches. *IEEE Trans. on Computers*, C–38(12):1612–1630, December 1989.

[8] T. Jeremiassen and S. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proc. 5th ACM SIGPLAN Symp. Principles & Practice of Parallel Programming*, July 1995.

[9] M. Kandemir, J. Ramanujam, and A. Choudhary. A compiler algorithm for optimizing locality in loop nests. Technical Report, CPDC-TR-9802-010, Northwestern University, Evanston, IL, February 1998.

[10] M. Kandemir, J. Ramanujam, and A. Choudhary. A compiler algorithm for optimizing locality in loop nests, In *Proc. 11th ACM International Conference on Supercomputing*, pages 269–278, July 1997.

[11] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proc. 1997 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 346–357, June 1997.

[12] M. Lam, E. Rothberg and M. Wolf. The cache performance and optimizations of blocked algorithms. In *Proc. 4th Int. Conf. Architectural Support for Programming Languages & Operating Systems*, April 1991.

[13] W. Li. *Compiling for NUMA parallel machines.* Ph.D. Thesis, Cornell University, Ithaca, NY, 1993.

[14] K. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Trans. Programming Languages & Systems,* 1996.

[15] F. Sanchez, A. Gonzalez, and M. Valero. Static locality analysis for cache management. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT-97),* November 1997.

[16] M. Tartalja and V. Milutinovic. A survey of software solutions for cache consistency maintenance in shared memory multiprocessors. *IEEE Software,* Fall 1996.

[17] M. Tomasevic and V. Milutinovic. *Tutorial on the cache coherency problem in shared-memory multiprocessors: Hardware solutions.* IEEE Computer Society Press, Los Alamitos, California, 1993.

[18] J. Torrellas, M. Lam, and J. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Trans. Computers*, C-43(6):651–663, June 1994.

[19] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[20] 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.

[21] M. Wolfe. *High performance compilers for parallel computing.* Addison-Wesley Publishing Company, CA, 1996.

Mahmut Kandemir is a PhD candidate in electrical engineering and computer science department at Syracuse University. He received a BS and an MS, both in computer engineering, from Istanbul Technical University. His research interests include different aspects of locality improvement techniques for cache memories, optimizations for I/O-intensive applications, and computer architecture. He is a student member of the IEEE Computer Society.

J. Ramanujam received the B.Tech. degree in Electrical Engineering from the Indian Institute of Technology, Madras, in August 1983, and the M.S. and Ph.D. degrees in Computer Science from The Ohio State University, Columbus, Ohio, in August 1987 and December 1990, respectively. He is currently an an Associate Professor of Electrical and Computer Engineering at Louisiana State University, Baton Rouge. His research interests are in compilers for high-performance computer systems, program transformations, high-level synthesis, parallel input/output systems, parallel architectures and algorithms.

Dr. Ramanujam received the National Science Foundation's Young Investigator Award in 1994. He has served on the Program Committees of the 1997 International International Conference on Parallel Processing and the 8th International Conference on Supercomputing, 1995 and several other conferences. He is a member of the High Performance Fortran Forum. He has taught tutorials on compilers for high-performance computers at several conferences such as the International Conference on Parallel Processing (1998, 1996), Supercomputing 94, Scalable High-Performance Computing Conference (SHPCC 94), and the International Symposium on Computer Architecture (1993 and 1994).

Alok Choudhary received his Ph.D. from University of Illinois, Urbana-Champaign, in Electrical and Computer Engineering, in 1989, M.S. from University of Massachusetts, Amherst, in 1986 and B.E.(Hons.) from Birla Institute of Technology and Science, Pilani, India in 1982. He has been an associate professor in the Electrical and Computer Engineering Department at Northwestern University since September, 1996. From 1993 to 1996 he was an associate professor in the ECE department at Syracuse University and from 1989 to 1993 he was an assistant professor in the same department. He has worked in industry for computer consultants prior to 1984.

Alok Choudhary received the National Science Foundation's Young Investigator Award in 1993 (1993-1999). He has also received an IEEE Engineering Foundation award, an IBM Faculty Development award and an Intel Research Council award.

His main research interests are in high-performance computing and communication systems and their applications in many domains including multimedia systems, information processing and scientific computing. In particular, his interests lie in the design and evaluation of architectures and software systems (from system software such as runtime systems, compilers and programming languages to applications), high-performance servers, high-performance databases and input-output. He has published more than 100 papers in various journals and conferences in the above areas. He has also written a book and several book chapters on the above topics. His research has been sponsored by (past and present) DARPA, NSF, NASA, AFOSR, ONR, DOE, Intel, IBM, and Texas Instruments.