Optimizing Data Locality by Array Restructuring

Shun-Tak Leung and John Zahorjan

Department of Computer Science and Engineering
University of Washington

Technical Report 95-09-01
September 1995

Optimizing Data Locality by Array Restructuring

Shun-Tak Leung and John Zahorjan *
Department of Computer Science & Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350
{*shuntak, zahorjan*}*@cs.washington.edu*

September 14, 1995

Abstract

It is increasingly important that optimizing compilers restructure programs for data locality to obtain high performance on today's powerful architectures. In this paper, we focus on *array restructuring*, a technique that improves the spatial locality exhibited by array accesses in nested loops. Specifically, we address the following question: Given a set of such accesses, how should the array elements be laid out in memory to match the access pattern and thus maximize locality?

Our approach is based on an invertible linear transformation of array index vectors. We present algorithms to choose a suitable transformation, and hence array layout, given the set of array accesses. Our analysis places no restrictions on the loop's nesting structure or dependence pattern. Although we focus on cases where the array indexing expressions are affine functions of loop variables, our techniques can be applied to the non-affine case as well.

We have implemented our technique in the SUIF compiler [17]. Experimental results show that array restructuring improves loop execution performance substantially, often with no or little runtime overhead. Moreover, the performance improvement of array restructuring compares favorably with that achieved by loop restructuring.

1 Introduction

To obtain high performance on today's powerful processor architectures, optimizing compilers increasingly have to restructure programs for data locality. This paper focuses on locality exhibited by array accesses in nested loops. There are several approaches a compiler might take to enhance this kind of locality. Since the access pattern results from an interaction between the order of the accesses and the data layout, a compiler can optimize for locality by reordering the accesses, changing the data layout, or doing both. Let us consider these possibilities.

One common approach to locality optimization is to reorder the accesses through restructuring of the loops. The iterations are reordered so that under the new order accesses to the same or nearby array elements

*This material is based upon work supported by the National Science Foundation (Grants CCR-9123308 and CCR-9200832), the Washington Technology Center, and Digital Equipment Corporation (the External Research Program and the Systems Research Center).

occur closer together in time than they would under the original program order. A number of techniques that follow this approach can be found in the literature [11, 10, 2, 18, 15, 14].

This paper focuses on a less widely studied approach: restructuring arrays. Data locality is improved by selecting the layouts of the array elements to better match the order of the accesses. For example, we might choose between row-major and column-major layouts (as well as possibly other layouts) depending on the access pattern exhibited in the loop. In general, the goal is that under the new array layouts, elements that are placed in the same cache line would be accessed within a shorter period of time than they would under the original layouts. A different form of data structures reorganization has been used to reduce false sharing on shared-memory multiprocessors [8, 12].

Array restructuring is attractive in several ways. First, it is not constrained by loop-carried dependences or imperfect loop nesting, which complicate and sometimes frustrate loop restructuring. Moreover, given a loop nest that accesses multiple arrays, each array can be restructured independently for optimal locality, whereas restructuring the loop nest unavoidably affects accesses to all arrays and therefore may necessitate some tradeoffs. On the other hand, unlike loop restructuring, the array restructuring decisions made in one loop nest can significantly affect successive loop nests. This can be addressed by choosing a globally advantageous layout, or by dynamically restructuring array data between loop nests.

The problem addressed in this paper can be stated as follows: Given a loop nest that accesses one or more multidimensional arrays, how should elements of these arrays be laid out in memory so as to improve spatial locality? We make no assumption on the loop structure: for example, it can be perfectly or imperfectly nested. We focus on, but do not restrict ourselves to, loop nests in which array indexing expressions are affine functions of the loop variables. Affine indexing expressions allow us to analyze the access patterns and determine appropriate array layouts within a linear framework.

Our approach to solving this problem hinges on linearly transforming array index vectors. The linear transformation determines the memory layouts of array elements. By choosing it properly, we can enhance the spatial locality exhibited by the array accesses in the loop nest while preserving the loop structure and avoiding extra overhead to locate elements in the restructured arrays. Our general transformation subsumes previous techniques that are, in principle or in practice, restricted to the permutation of array dimensions [9, 3]. Such generality is useful in the optimization of banded matrix computations, for example [3]. We have implemented our technique in the SUIF compiler [17], an experimental parallelizing compiler form Stanford. In this paper, we present our array restructuring techniques and report experiments to evaluate their effect on performance. Our results show that array restructuring can substantially speed up loop execution and compares favorably to alternative approaches.

In addition to restructuring loops and restructuring arrays, a compiler might also restructure both simultaneously [9, 3]. Although this hybrid approach is potentially more powerful than the other two, how to efficiently and fully exploit its potential is still unclear. We will return to this question in Section 7, after presenting array restructuring, and discuss the roles of pure loop and array restructuring in the hybrid approach.

The rest of this paper is organized as follows. Section 2 presents the framework on which our array restructuring techniques are based. Section 3 discusses how to optimize for data locality within this framework by finding a suitable transformation of array index vectors. In Section 4, we discuss the problem of mapping the transformed index vectors to scalar addresses, which is made non-trivial by the more sophisticated forms of array restructuring that we might perform. Section 5 explains how non-affine array accesses are handled. Experimental results are reported in Section 6. In Section 7, we reconsider the different approaches to locality optimization, and how they compare and interact. Section 8 summarizes related work. Finally, Section 9 concludes this paper.

2 Array Restructuring

In this section, we discuss how to restructure an array by means of a linear transformation of index vectors, and how such a transformation affects the code of loop nests accessing the restructured array. We assume for the moment that the transformation has already been chosen. The question of how to choose it to enhance locality is the subject of the next section.

The discussion throughout this section is illustrated by the simple example in Figure 1. First, we briefly describe an intuitively obvious array transformation that would improve the data locality of the example loop. The upper half of the diagram shows the original and transformed codes. Using this simple array transformation as a specific example, we then discuss our array restructuring framework in general. The matrices in the lower half of the diagram pertain to this discussion and are explained later in this section.

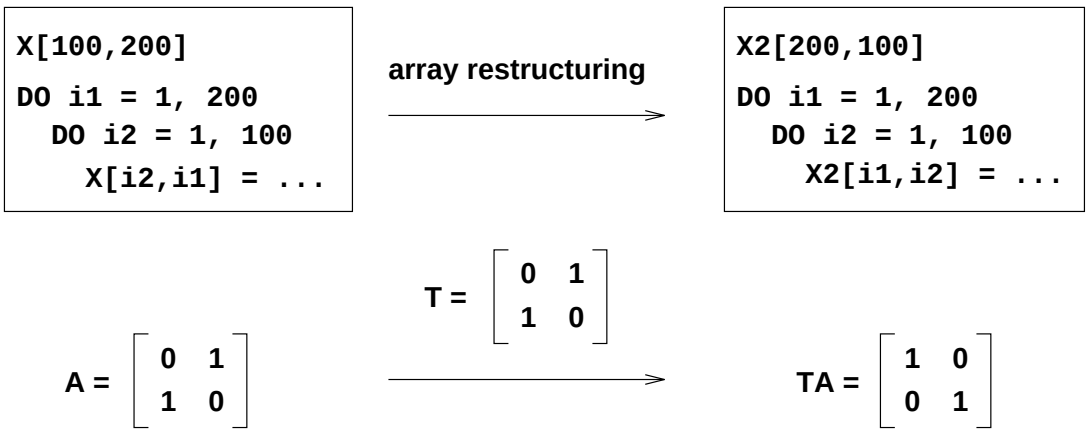

Figure 1: Simple Example of Array Restructuring

In Figure 1, a double loop accesses a two-dimensional array called X, assumed to be in row-major order. The original code, shown on the left, has poor spatial locality because the loop nest accesses the row-major array X by column. Consecutive iterations of the inner loop access elements that are far apart in memory, whereas adjacent elements are accessed by different iterations of the outer loop. A remedy through array restructuring is, in effect, to transpose the array X and rewrite the array indexing expressions to correspond to the new layout. The transformed code, shown on the right, accesses the row-major array X2 — the transpose of X — row by row. This code exhibits better spatial locality than the original version.

With this example, we now discuss our array restructuring framework. First, let us relate an iteration vector and the index vector of the element accessed by the iteration. Consider an access made by a perfectly nested loop to an array X, such as the access in Figure 1. Let n be the number of loop levels and m the number of array dimensions. Suppose that all the array indices are affine functions of loop variables[1]. Denoting the iteration vector by i and the index vector of the accessed element by v, we have

$$v = Ai + a \tag{1}$$

where the $m \times n$ matrix A, called the *access matrix*, and the m-dimensional vector a are constants specific to this access. Each row of A and of a correspond to one dimension of the array index vector. In Figure 1, for example, the original access matrix A is shown below the original code; the constant vector a happens to be zero and is not shown.

[1] An affine function is a linear combination of its arguments plus a constant, such as $f(x, y, z) = 2x + 3y - z + 7$.

The key idea of our array restructuring techniques is to transform the index vectors with an invertible linear mapping. Elements of the restructured array are stored in row-major order according to their transformed, rather than original, index vectors. The mapping must be invertible because otherwise multiple array elements might be assigned the same location in the restructured array. Formally, each vector v in the original array index space is mapped to a unique vector v' in the *transformed* array index space by

$$v' = Tv \tag{2}$$

where T, which we call the *index transformation matrix*, is an $m \times m$ nonsingular matrix. The transformed index vector v', instead of the original v, is used to index the restructured array, whose elements are stored in row-major order. In Figure 1, for example, we transpose the original array by multiplying any original index vector v with the matrix T shown in the middle to obtain a transformed index vector v', which is then used to index the row-major restructured array X2. This particular matrix T, in effect, exchanges the two dimensions of v. It should be noted, however, that a transposition, or any permutation of more than two array dimensions, is only a special case. This framework allows us to apply the arbitrary invertible linear transformation that best enhances locality. In some cases, such as many banded matrix computations, simple transformations like permuting array dimensions are not sufficient.

Array restructuring requires only small changes to the code implementing the loop nest. In particular, it has no effect on the loop structure, including loop headers, bounds, and nesting. Only the array accesses are affected.

Array restructuring does not make the transformed array accesses less efficient than the original ones, despite the appearance of an extra level of indirection in the preceding discussion. Conceptually, the original array X is replaced by another array X2. To access a certain element of X, we need to compute the transformed index vector v' from the original v and use v' to index the corresponding X2 element instead. This view makes it appear that array index computation has to be done in two steps: one to compute v from the iteration vector and one to compute v' from v. However, in practice this is not necessary since v' can in fact be expressed directly in terms of the iteration vector i:

$$v' = T(Ai + a) = (TA)i + (Ta) \tag{3}$$

where TA and Ta can both be computed at compile time or, at worst, outside the loop nest at run time. The extra level of indirection between the two index vectors therefore exists only in concept but not in practice. For instance, in Figure 1, the original access `X[i2,i1]` (access matrix $A = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$) is transformed to `X2[i1,i2]` (transformed access matrix $TA = \left[\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right]$). The two array index expressions are simply interchanged, as expected. Furthermore, since the array indexing expressions after array restructuring, like the original ones, are affine, a compiler can apply the usual code optimizations to eliminate most of the indexing cost.

To summarize, we have presented the framework for our array restructuring techniques and discussed the implications to the generated code, especially the code for array accesses.

Two questions remain. The first, obvious question is how to choose a transformation, specifically an index transformation matrix T, that optimizes locality for a given set of array accesses. This is discussed in Section 3. The second question is more subtle. To access an array element identified by a given index vector, we need to compute from that index vector the scalar address at which the element is stored in memory. This is normally easy. However, the issue is complicated by the more sophisticated forms of array restructuring that we might apply. We explain these complications and present our solution in Section 4.

3 Choosing an Index Transformation

Given a set of array accesses in a loop nest, we want to restructure the arrays so as to improve the data locality exhibited by these accesses. Specifically, we focus on spatial locality, since temporal locality is inherently independent of whatever array restructuring we might perform. We have already presented the framework for our array restructuring techniques. Its essence is to linearly transform index vectors by means of a nonsingular index transformation matrix. The transformation matrix for each array is chosen based on the pattern of the accesses to that array.

This section discusses how to choose an appropriate transformation matrix given the array accesses. First, we formulate the problem: we list a number of requirements for the desired transformation matrix and explain why they are necessary. Then, we present our solution: we describe an algorithm to find a transformation matrix that meets these requirements. To simplify the following discussion, we first consider a restricted case: a single access (to the array in question) in a perfectly nested loop. At the end of this section, we outline how to handle cases without these two restrictions.

3.1 Requirements for Index Transformation Matrix

Given a single array access, with access matrix A, in a perfectly nested loop, we wish to find an index transformation matrix T for the array so as to enhance the spatial locality exhibited by this access. In this section, we state two requirements for T and explain the rationale behind them, leaving to the next section the question of how to find such a matrix. The first requirement concerns the form of the transformed access matrix TA and is intended to enhance locality. The second requirement concerns the transformation matrix T itself and is motivated by implementation issues.

3.1.1 Transformed Access Matrix

The index transformation matrix T allows us to turn the original access matrix A into the transformed access matrix TA. To optimize for locality by a suitable choice of T, we need to relate the locality exhibited by an access to the forms of its access matrix. With this relationship, we can express the goal to improve locality in terms of the desired form of the transformed access matrix TA — the first of our two requirements for T. Specifically, the focus is on the nonzero structures of these matrices.

We use the example in Figure 2 to aid the following discussion, as well as discussion in following sections. Different parts of this figure will be explained as they become relevant to our discussion. This code, adapted from [14], implements a symmetric rank 2k update (SYR2K) for banded matrices. (SYR2K is one of the Level 3 Basic Linear Algebra Subprograms [5].) The original and transformed codes are shown at the top, the access and transformation matrices in the middle, and graphical depictions of array layouts at various stages of transformation at the bottom. This figure shows the restructuring of array X. (Since the access patterns to arrays X and Y are identical, array Y is similarly restructured. Array Z is not restructured.) The upper and lower access matrices represent respectively the first and second accesses to X.

We consider first why the original code has poor locality, then how the problem relates to the nonzero structure of the access matrix, and finally how locality can be improved by transforming the access matrix. We focus on the first access to X (`X[j-k+b,k]`). The problem with the second access is similar, but we ignore it for the moment since we currently assume only a single array access.

The first access exhibits poor spatial locality mainly because consecutive iterations of the innermost loop read array elements that are far apart in memory (assuming row-major storage). These elements are

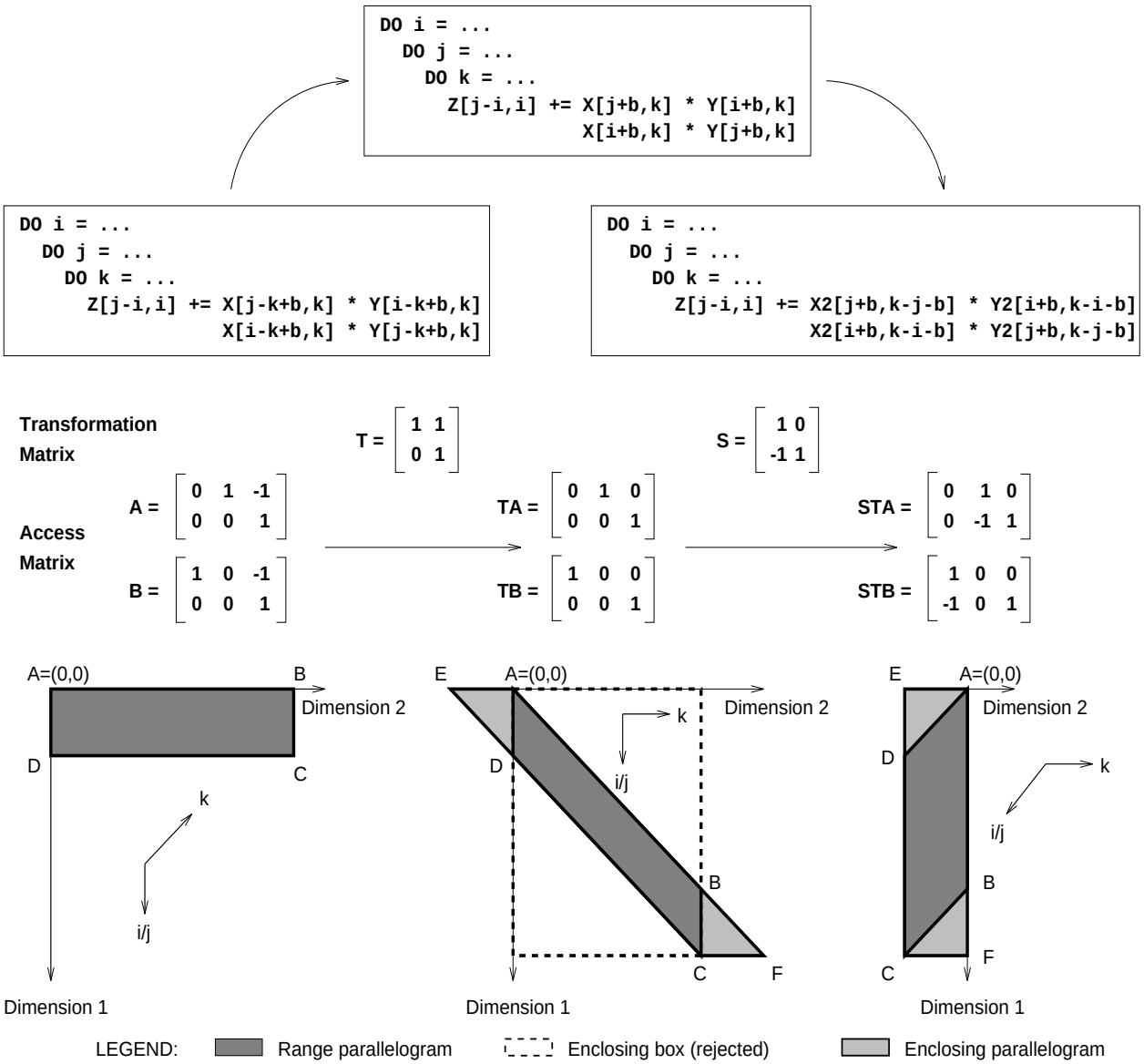

Figure 2: Example: Banded SYR2K

separated by almost an entire row, which most likely would occupy multiple cache lines. The accesses in consecutive iterations therefore cannot reuse the same cache line. Simple array transformations like permuting array dimensions do not solve the problem, as we shall see below.

The diagram at the bottom left of Figure 2 depicts this problem graphically. This diagram shows the original array index space with the first array dimension vertical and the second horizontal. The shaded region represents valid index vectors for array X. As execution goes through iterations of the innermost loop (the k-loop), the access moves through the array index space in the direction marked k, crossing one row per iteration. This indicates poor spatial locality. Simply storing the array in column-major order offers little help since the k-direction crosses columns as well. Spatial locality would improve, however, if the array

index space were transformed in such a way that the k-direction becomes horizontal. A horizontal k-direction means that iterations of the innermost loop access elements in the same row (which are stored contiguously in memory, assuming row-major storage) and therefore implies better spatial locality.

We can also view this problem from the perspective of the nonzero structure of the access matrix A (the upper one in Figure 2, since we are discussing the first access). The innermost loop corresponds to the rightmost column in the access matrix. As execution proceeds from one iteration to the next, the iteration vector i is incremented in the last dimension and thus the index vector of the accessed element ($v = Ai + a$) changes by A's rightmost column. This column determines the k-direction in the diagram. Since the first position of this column is nonzero, consecutive iterations access elements in different rows of the array. If, however, we could make the corresponding position in the transformed access matrix TA zero, then iterations of the innermost loop would access elements in the same row and spatial locality would be much better.

Therefore, to improve spatial locality, we require that the transformation matrix T eliminate this first nonzero in A's rightmost column. Figure 2 shows such a T and the resultant TA, with the corresponding code shown in the middle. (Note that this code is not final; the final version, shown on the right, has undergone further changes that will be explained later in this paper.) We can also state this requirement in terms of the column's *height*, defined as the position (counted from the bottom) of the column's top nonzero: The height of A's rightmost column, namely 2, is too large; we wish to reduce it to 1 in TA. In other words, we want to *flatten* the rightmost column.

In general, the key requirement for the index transformation matrix T is that the height of TA's rightmost column is 1. (Strictly speaking, we are concerned about the rightmost *nonzero* column. All zero columns of A can be ignored when we choose T because they never affect our choice: any nonsingular transformation matrix would transform a zero column to a zero column.) In other words, the transformation should eliminate all nonzeros in A's rightmost column, except one in the last position. If this column is brought into such a form, array elements accessed by consecutive iterations would be much closer together than if there are nonzeros in higher positions of the column. In particular, if this single nonzero in the last position of the column is 1 or -1, as is often the case, those array elements are contaguous.

So far, we have focused on the rightmost column of the access matrix because it corresponds to the innermost loop, which is most important for locality optimizations. To maximize spatial locality, it is desirable to flatten the other columns as well. However, with an invertible linear transformation, we cannot arbitrarily flatten an arbitrary set of columns. In Figure 2, for example, since A's rightmost column $\begin{bmatrix} -1 \\ 1 \end{bmatrix}$ is transformed to $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$, A's middle column $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$, being linearly independent of $\begin{bmatrix} -1 \\ 1 \end{bmatrix}$, cannot possibly be transformed to $\begin{bmatrix} 0 \\ x \end{bmatrix}$, regardless of x and regardless of our particular choice for T. Therefore, when choosing the index transformation matrix T, we always seek first and foremost to flatten A's rightmost column (to a height of 1), and flatten other columns to the left in decreasing priority.

In summary, our first requirement for the index transformation matrix T is that it reduces the heights of A's columns — the rightmost column to a height of 1 and others to the left as much as possible but in decreasing priority.

3.1.2 Unimodular Index Transformation Matrix

In addition to the desired form of TA just discussed, we require, without loss of generality, that T itself be unimodular. (A unimodular matrix is an integral matrix whose determinant is either 1 or -1.) This requirement is motivated by two implementation issues: efficient array index computation and efficient memory usage. They lead to two conditions that are satisfied if and only if T is unimodular.

- We want to keep array index computation efficient: it must involve only integers. This is guaranteed if T (and hence the transformed access matrix TA) is integral.
- We do not want to allocate unneeded memory: the transformed index space must not have "unoccupied holes" — a memory location that is allocated, because it corresponds to an integral vector in the transformed index space, but does not store an element, because it does not correspond to an integral vector in the original index space. For this purpose, we require that T maps the set of all integral vectors to the set of *all* integral vectors. In other words, every integral vector in the transformed index space is the image of some integral vector in the original index space. This is guaranteed if T^{-1} is integral.

In short, we want both T and T^{-1} to be integral. This is true if and only if T is unimodular. See Lemma 1 in Appendix A for a proof.

Restricting to unimodular matrices does not cause a loss of generality in the search for a locality enhancing index transformation matrix. This is because if there is a non-unimodular matrix, say T, such that the columns of TA have the desired heights, there is also a unimodular matrix, say U, such that the corresponding columns of UA and TA have the same heights. (A proof for the existence of such a unimodular matrix is given in Lemma 2 in Appendix A.) A search restricted to unimodular matrices would exclude the non-unimodular T but still include the unimodular U, which is equally good in terms of our optimization criteria.

3.2 Finding an Index Transformation Matrix

Given an access matrix A, the requirements for the index transformation matrix are summarized as follows: a unimodular matrix that flattens A's columns — the rightmost column to a height of 1 and others to the left as much as possible. We now outline an algorithm for finding such a matrix.

The algorithm consists of two steps. In the first, we find a nonsingular (but not necessarily unimodular) matrix R such that RA has the desired form in terms of the heights of its columns. In the second, we compute from R a unimodular matrix U such that corresponding columns of UA and RA have the same heights. We can choose U as the index transformation matrix because it meets both our requirements: U is unimodular and UA, like RA, has the desired form.

Our algorithm to find R in the first step resembles the Gaussian elimination algorithm for inverting nonsingular matrices. The key is to transform the access matrix A into the desired form using a systematically constructed series of elementary row operations (i.e., scaling a row, exchanging two rows, or adding a multiple of one row to another). Any series of elementary row operations transforms A to QA, where Q is a nonsingular matrix determined by the operations but independent of A. Thus, if QA has the desired form, Q can serve as our R. The detailed algorithm to construct an appropriate series of row operations and compute the corresponding Q efficiently can be found in Appendix B.

Computing U from R in the second step hinges on finding the Hermite normal form [16] of R. Let H be this Hermite normal form. We choose $U = H^{-1}R$. The proof of Lemma 2 in Appendix A shows that this U is unimodular and corresponding columns of UA and RA have the same heights. An algorithm to find the Hermite normal form of a nonsingular matrix is in the literature [16].

3.3 Multiple Accesses in Imperfectly Nested Loops

Finally, we briefly discuss how to handle cases that do not satisfy the two simplifying restrictions imposed at the beginning of this section: single access and perfect loop nesting.

To handle multiple accesses to the same array, we merge columns from the individual access matrices into a *combined access matrix* and find a transformation to flatten its columns as we do for a single access matrix. In the combined access matrix, columns are ordered according to the potential performance gain of flattening a column: the rightmost column is the one for which we expect the greatest benefit, as in the case of a single access matrix. Currently, we use a heuristic measure based on the loop level(s) to which a column corresponds. Roughly speaking, the deeper is the loop level, the more important (for performance) it is to flatten the column. With this heuristic measure, the combined access matrix degenerates to the single access matrix when there is only one access.

When constructing the combined access matrix from columns of individual access matrices, certain kinds of columns are treated specially. Zero columns are omitted. They are transformed to zero columns regardless of the transformation matrix, and so we need not consider them when choosing the transformation matrix. Moreover, columns that are scaled versions of one another are treated as one because their images have the same height under any transformation matrix. For the example in Figure 2, the combined access matrix for array X would be $\begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$, with $\begin{bmatrix} -1 \\ 1 \end{bmatrix}$ on the right because it corresponds to the innermost loop in both individual access matrices.

Accesses in imperfectly nested loops are handled similarly. Conceptually, all individual access matrices (for accesses at various loop levels to the array in question) are augmented on the right with zero columns so that they have the same number of columns. In practice, the added zero columns have no effect because the algorithm ignores all zero columns anyway.

Our method can be similarly extended to handle multiple loop nests. We can treat them as if they were enclosed by an imaginary loop with only one iteration. To represent this, a zero column is added to the left of all access matrices, but this has no effect in practice. The basic optimization strategy remains the same: flatten the columns most important for locality. The set of loop nests grouped together for analysis might range from a single loop nest, a series of consecutive loop nests that are always executed together (what one might call a "basic block" of loop nests), or all loop nests in a procedure or beyond. A future direction of our investigation is grouping loop nests to achieve an optimal tradeoff between locality within a group and dynamic restructuring overhead incurred at group boundaries.

All the above discussion applies to accesses to the same array. If a loop nest accesses different arrays, each array is restructured independently.

4 Mapping The Transformed Index Vector to A Scalar Address

To access an array element identified by a given index vector, we need to determine from that vector a scalar address at which the element is stored in memory. Index vectors are normally mapped to scalar addresses by means of an affine function, which can be efficiently evaluated at the time of access. How to compute this affine mapping function from array bounds is well understood. However, the issue is complicated by some of the more sophisticated forms of array restructuring that we might apply. In this section, we explain the problem and present our solution.

4.1 The Problem

To understand why it is non-trivial to compute a mapping function from *transformed* index vectors to scalar addresses, let us first briefly review how index vectors are traditionally mapped to addresses, assuming row-major storage. An array specification normally includes (perhaps implicitly) a lower bound and an upper

bound for each dimension, which delimit the range of valid index values in that dimension. As depicted in the bottom left diagram in Figure 2 for the original array X, the bounds for all dimensions together define a rectangular box in the index space for valid index vectors: Each integral point in the box identifies an array element. These points are numbered in lexicographic order (since we assume row-major storage) by an affine function computed from the array bounds: $(u_2 - l_2 + 1)(x_1 - l_1) + (x_2 - l_2)$, where l_k, u_k, and x_k are respectively the lower bound, upper bound, and index value in the k-th dimension. This can be easily generalized to more array dimensions.

Now we consider the problem of mapping *transformed* index vectors to addresses — to number valid transformed index vectors in lexicographic order by means of an affine function. Again, we assume that elements are stored in row-major order, as we did in Section 3.

This problem is trivial if the index transformation merely permutes the array dimensions. Components of a valid transformed index vector must fall within the original, similarly permuted array bounds. The region for valid transformed index vectors therefore remains a rectangular box. We can compute an index-to-address mapping from the permuted array bounds in the same way as in the traditional case.

For more general index transformations, however, the problem is beyond the scope of the traditional method, which applies only if valid index vectors are delimited by a rectangular box. In Figure 2, for example, the valid transformed index vectors are delimited by a parallelogram (the deeply shaded $ABCD$ in the middle diagram) rather than a rectangular box. This parallelogram, which we call the *range* parallelogram, is the image of the rectangular box representing the original array bounds ($ABCD$ on the left) under the index transformation T.

To help explain our solution, presented in the next section, let us first consider a straightforward but unsatisfactory solution. We could enclose the range parallelogram with a rectangular box, as illustrated by the dashed box in Figure 2. We can compute an index-to-address mapping for this enclosing box in the traditional manner. This mapping would number valid transformed index vectors, corresponding to integral points within the range parallelogram and thus also within the enclosing box, in lexicographic order. However, this technique wastes memory. It allocates memory as if all integral points in the enclosing box correspond to array elements. Since the enclosing box often does not bound the range parallelogram tightly (except in the trivial case where the range parallelogram is itself a rectangular box, i.e., the index transformation simply permutes the array dimensions), a large portion of the allocated memory is not used. In Figure 2, for example, only the deeply shaded region inside the dashed box corresponds to array elements; memory allocated for the rest of the dashed box is unused.

In short, the traditional method of computing an affine index-to-address mapping requires that valid index vectors are delimited by a rectangular box. It does not directly apply to our case because our transformed index vectors might be delimited by a parallelogram (the range parallelogram) rather than a rectangular box. A simple solution is to enclose the range parallelogram with a rectangular box, but this wastes memory. In the next section, we show how to arrive at a better solution using this as a starting point.

4.2 Our Solution

We now present our solution to the problem of mapping transformed index vectors to scalar addresses. It builds on the simple but unsatisfactory solution previously discussed. In the interest of space, we do so only intuitively, at the risk of some simplifications.

Instead of enclosing the range parallelogram directly with a rectangular box, we enclose it tightly with a parallelogram (e.g., the lightly shaded $AFCE$ in the middle diagram in Figure 2) that can be further transformed (through S in Figure 2) to a rectangular box ($AFCE$ on the right) without jeopardizing the

locality optimizations already performed. There are three aspects to this that fit together to solve our problem. First, the enclosing parallelogram is further transformed to a rectangular box. This allows us to compute an index-to-address mapping for this rectangular box and combine that mapping with the other two affine mappings — the locality optimizing index transformation (T) discussed earlier and the transformation (S) mentioned here — into one affine mapping directly from the original index vector to a scalar address. Second, the additional transformation must not, of course, jeopardize prior optimizations. For example, although we could always revert to the original array bounds (which form a rectangular box) using T^{-1} as the additional transformation, this is unacceptable because it effectively undoes prior optimizations. Third, to conserve memory, the region enclosing the range parallelogram should bound the latter as tightly as possible. Using a parallelogram rather than a rectangular box gives us greater flexibility to achieve this.

We now outline how such an enclosing parallelogram is chosen. We first discuss the two-dimensional case with reference to the example in Figure 2 and then the higher-dimensional case in more general terms.

Recall that we look for an enclosing parallelogram that can be further transformed to a rectangular box without jeopardizing prior optimizations. In two dimensions, this condition is satisfied if the enclosing parallelogram has a pair of horizontal sides. For example, in the middle diagram of Figure 2, the enclosing parallelogram $AFCE$, which has two horizontal sides, is transformed to the rectangular box $AFCE$ on the right if we slide each row leftward in proportional to its vertical distance from the origin (point A). This transformation, represented by the matrix S, meets both conditions. First, it maps the enclosing parallelogram to a rectangular box by making the former's slanted sides vertical while keeping the horizontal sides horizontal. Moreover, it preserves prior optimizations because it keeps horizontal vectors horizontal. As we saw earlier, making the k-direction, which corresponds to the innermost loop, horizontal is key to good spatial locality. The first transformation T makes the k-direction horizontal; the second transformation S keeps it that way.

To find a tight enclosing parallelogram with horizontal sides, we apply the Fourier-Motzkin algorithm [16] to the range parallelogram and select for each dimension the closest pair of parallel lines from those generated by the algorithm. The two chosen pairs define the enclosing parallelogram.

In general, the enclosing parallelogram is defined by pairs of parallel hyperplanes that tightly bound the range parallelogram between them. There is one pair for each array dimension. In geometric terms, the pair for the k-th dimension must be parallel to the $(k+1)$-th dimension, $(k+2)$-th dimension, etc., but not the k-th dimension. When there are only two dimensions, this condition means that one pair (for the first dimension) must be parallel to the second dimension but *not* the first, whereas the other pair (for the second dimension) must *not* be parallel to the second dimension. In Figure 2, the former must be horizontal, whereas the latter must *not* be horizontal (but may be vertical or slanted), just as we saw before. In algebraic terms, the region bounded by the pair of hyperplanes for the k-th dimension is

$$b_k \leq x_k + \sum_{j=1}^{k-1} c_{jk} x_j \leq d_k \tag{4}$$

where b_k, c_{jk}, and d_k are constants and $b_k \leq d_k$. These hyperplane pairs are selected in the same way as in the two-dimensional case.

We look for such a parallelogram because it can always be transformed to a rectangular box without jeopardizing previous optimizations. The transformation involves a unit lower-triangular matrix (i.e., a lower-triangular matrix with all 1's on the main diagonal), such as S in Figure 2. Because S is lower-triangular, columns of STA must have the same heights as corresponding columns of TA. In other words, the additional transformation S preserves the effect of prior optimizations, which seek to keep the heights of TA's columns low and thus achieve good spatial locality.

To sum up, we compute an index-to-address mapping for transformed index vectors by first finding

a special form of parallelogram to tightly enclose the range parallelogram. We require that the enclosing parallelogram can be further transformed to a rectangular box through a transformation that does not nullify the locality improvement of prior transformations. Once the enclosing parallelogram and the additional transformation have been found, we can merge the various intermediate affine mappings into one that maps the original index vector directly to an address, thus eliminating the cost of calculating the intermediates at run time.

5 Handling Non-affine Accesses

We have so far assumed that all array index expressions are affine functions of loop variables, thus allowing analysis and transformation of array accesses in a linear framework. While affine accesses are most common, we wish to make our technique as generally applicable as possible: the presence of non-affine accesses among affine ones should not frustrate the application of our technique despite its focus on the affine case.

In this section, we show that our techniques can handle non-affine index expressions as well. Two issues are discussed. First, we show that the mechanism of array restructuring — transforming array accesses with a given index transformation — applies as easily and efficiently to non-affine accesses as it does to affine ones. Second, we consider the policy aspect — selecting the index transformation itself — and discuss how the analysis accomodates non-affine index expressions.

5.1 Applying an Index Transformation

We first consider the runtime overhead of computing a transformed index vector when indexing expressions are non-affine. Recall that the original and transformed index vectors (v and v' respectively) are related by $v' = Tv$ (see (2)). In the affine case v', like v, can be expressed directly in terms of the iteration vector i with the help of simplifications that exploit the associativity of matrix multiplication (see (3)). As a result, v' can be computed as efficiently at run time as v. In the non-affine case, however, similar simplifications are not possible. Thus, it might appear that, with array restructuring, additional computation is needed to calculate Tv from v. In fact, this is not necessary.

When indexing expressions are non-affine, the original index vector v is computed explicitly. From v, the address of the accessed element, d, can be calculated directly without first computing v'. Without array restructuring, the relationship between d and v is simply

$$d = \alpha v + \alpha_0 \tag{5}$$

where the constant m-dimensional row vector α (m being the number of array dimensions) and the constant scalar α_0 together represent the index-to-address mapping. If array restructuring is applied, the address d is related to the transformed index vector v' via another linear transformation, represented by the matrix S, and an index-to-address mapping that maps the doubly transformed index vector (denoted v'' here) to the address. In other words,

$$v'' = Sv' \qquad \text{and} \qquad d = \beta v'' + \beta_0 \tag{6}$$

Combining this equation with $v' = Tv$, we can express the address d in terms of the original index vector v as follows:

$$d = \beta v'' + \beta_0 = (\beta S)v' + \beta_0 = (\beta ST)v + \beta_0 \tag{7}$$

In both cases, the address d equals the dot product of a constant vector (βST or α) and the original index vector v plus a constant scalar offset (β_0 or α_0). The constants in the two cases might be different,

but they are all loop-invariant. At each access inside the loop nest, essentially the same code can be used to calculate the element's address. Therefore, even for non-affine accesses, array restructuring does not incur any indexing overhead beyond what would be needed anyway.

5.2 Choosing an Index Transformation

We now consider how to select an index transformation matrix T when some accesses are non-affine. The strategy is to extract as much information as we can from non-affine index expressions and ignore what we cannot analyze. Note that the inability to fully analyze the array accesses never makes the transformed program incorrect. At worst, we choose a transformation that does not improve locality as expected. In contrast, loop restructuring based on a partial dependence analysis may lead to an erroneous program.

One simple approach to analyzing non-affine index expressions is to treat them as the sums of affine and non-affine terms and consider only the former. Recall that for affine accesses, the accessed element's index vector (v) is related to the iteration vector (i) by $v = Ai + a$, where the matrix A, called the access matrix, and the vector a are loop-variant. For non-affine accesses, we can separate the affine and non-affine terms in the index expressions and represent the latter by a vector-valued function $f(i)$:

$$v = Ai + a + f(i) \tag{8}$$

For example, consider the access `X[i2,im[i1]]`, adapted from a two-dimensional fast Fourier transform (FFT) code (see Section 6). The second array index, which involves the indirection table `im`, is non-affine. The access is represented by

$$v = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ im(i_1) \end{bmatrix}$$

Completely affine accesses or completely non-affine accesses are just special cases: The former correspond to an $f(i)$ always equal to zero and the latter to an A and a that are zero.

Having isolated the affine terms, we can choose the index transformation based on the access matrix A alone, as just discussed. In our example, from the column $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ in A we know that the innermost loop iterations access different rows, causing poor spatial locality for a row-major array. To alleviate the problem, we would flatten the column with $T = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, in effect transposing the array.

This might not be good enough, however, if the second, non-affine array index also varied with the innermost loop variable `i2` (although fortuitously this is not the case here). We would know nothing about how the second index varies if the non-affine terms are just ignored. In our approach, we extract more information from the non-affine terms. To explain our technique, let us first briefly review, in the affine case, what information is needed to select the index transformation and how that information is obtained. We then generalize to the non-affine case.

In Section 3.1.1, we looked for the direction in which the array access moves in the index space or, equivalently, the change in the index vector as execution goes through the innermost loop, as well as similar information for the other loops. The index transformation matrix was chosen to flatten these vectors. Affine index expressions readily yield the needed information because they are linear combinations, with loop-invariant coefficients, of loop variables.

We are looking for the same kind of information in the non-affine case. To obtain that information, we treat index expressions as linear combinations, with loop-invariant coefficients, of loop variables and *quasi variables* — non-affine expressions varying with one or more loop variable. For example, in the access

`X[i2,im[i1]]`, the expression `im[i1]` is a quasi variable that varies with `i1`. (Non-affine sub-expressions that are identical up to a constant factor are treated as multiples of the same quasi variable rather than distinct quasi variables. For instance, `X[i2+4*im[i1],2*im[i1]]` contains two occurences of one quasi variable `im[i1]`, not two quasi variables `4*im[i1]` and `2*im[i1]`.) The index vector v is expressed in terms of the iteration vector i and the *quasi iteration vector* q, composed of the quasi variables:

$$v = Ai + a + Bq \tag{9}$$

where the loop-invariant matrix B, called the *quasi access matrix*, contains the coefficients of the non-affine terms in the array index expressions. For the access `X[i2,im[i1]]`, we have

$$v = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} [im(i_1)]$$

From this equation we can deduce the change in the index vector between consecutive iterations, though less precisely than in the affine case. Specifically, as execution goes through the innermost loop, for example, both the loop variable and quasi variables varying with it, if any, change. The columns in A and B corresponding to these variables indicate how the index vector would change accordingly. (This knowledge is imprecise, though, because we do not analyze exactly how each quasi variable varies but only which loop variable(s) it varies with.) Therefore, we try to flatten all such columns, rather than only the one from A as in the affine case. If the loop variable is not involved in any non-affine expression, it corresponds to no column in B and thus only its column in A is considered. In our example, the inner loop corresponds to only the column $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ from A, and the outer loop to the column $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ from B. By choosing T as $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, we transform the access to `X2[im[i1],i2]`, which exhibits better spatial locality than the original. Measurements of the performance impact of this transformation are reported in Section 6.

6 Performance

We have implemented our array restructuring technique in the SUIF compiler [17] and performed a number of experiments to evaluate their effectiveness. The results are reported in this section. We first study the performance impact of array restructuring by itself. Then, array restructuring is compared with common loop restructuring techniques. Finally, we discuss how it interacts with loop tiling.

6.1 Experiments

We have implemented our array restructuring technique in the SUIF compiler [17]. For each loop nest, the compiler analyzes how arrays are accessed, chooses the index transformations, and modifies the array accesses. The compiler also generates calls to our runtime system, which dynamically restructures array data between loop nests when needed. We assume all arrays to be in a canonical layout on procedure entry and restore them on exit if necessary. This allows procedures that have been transformed by our compiler to be linked with separately compiled procedures that have not. The compiler outputs C code, which is then compiled by the native C compiler.

The experiments were done on a DEC3000 Model 400 workstation running DEC OSF/1 [4, 6]. The workstation is based on the DEC Alpha architecture. It has two levels of cache for data: an on-chip, write-through, 8 KB data cache, and an off-chip, write-back, 512 KB unified cache shared by instructions and data. Both levels of cache are direct-mapped. Cache blocks are 32 bytes long. On a read miss in the first-level

cache, 5 CPU cycles are required to read the accessed data from the second-level cache and another 5 cycles to fill the rest of the cache line. A write miss adds another 5 cycles. On a miss in the second-level cache, it takes 24 cycles to read the accessed data from main memory and another 6 cycles to fill the other half of the cache line.

Nine loop nests were used in the experiments.

The first is a matrix multiply (MATMUL), as shown in Figure 3. This form of the loop nest has good temporal locality and is the most straightforward implementation of the mathematical definition. Our compiler in effect decides to store `C` and `A` in row-major order and `B` in column-major order.

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

Figure 3: MATMUL — Matrix Multiply

The second loop nest, adapted from [14], implements a symmetric rank 2k update (SYR2K) for banded matrices. We have already discussed this loop nest extensively as our running example in Section 3 and Section 4.

The other loop nests come from the seven NASA kernels in the SPEC benchmark suite: MXM, GMTRY, CFFT2D, CHOLSKY, BTRIX, VPENTA, and EMIT. We measured the performance for different problem sizes to study how array restructuring scales. Most of the sizes we used are larger than those of the original benchmarks because the original ones are often too small for locality optimizations to have any significant effect: execution takes only fraction of a second and the data fit entirely in the cache.

Some loop nests in these kernels are not affected by array restructuring because our compiler estimates that there is little or no performance gain and therefore decides not to apply any transformation. To focus on the effect of array restructuring, we measured and report here the performance of only the affected loop nests in each kernel (which, as it happens, are also the most time-consuming ones). It should be noted, however, that we always compiled and executed the entire kernel and verified that the performance of the supposedly unaffected loop nests, though not reported, indeed did not change noticeably.

Specifically, in CHOLSKY, one loop nest computes the Cholesky decompositions of multiple banded matrices, and a second one performs triangular solves for multiple right-hand sides with these decompositions. This paper reports the performance of the latter. CFFT2D computes a two-dimensional fast Fourier transform (FFT) on a two-dimensional array of complex numbers in four phases. We focus on the two for which the canonical layout would lead to poor locality. GMTRY sets up equations for a vortex method solution and solves them with Gaussian elimination. We focus on the Gaussian elimination. None of EMIT's loop nests is transformed. Therefore, no results are reported here.

6.2 Array Restructuring

This section reports experiments that evaluate our array restructuring technique by itself. We consider the execution times as well as the memory overheads.

6.2.1 Execution Times

In this section, we study the performance impact of array restructuring, focusing on how it affects loop execution time and the impact of any runtime data restructuring that may be required. Results are shown in Figure 4.

For MXM, BTRIX, and VPENTA, our compiler finds index transformations that would improve locality but decides not to apply them because it estimates that the performance benefit does not justify the runtime restructuring overhead. In these experiments, however, we temporarily suspended the cost analysis so that our compiler applied those transformations anyway. This allows us to learn more about how array restructuring affects performance,

In all the loop nests except MXM, array restructuring significantly reduces loop execution times for problems of different sizes. In most of these loops, execution time is roughly halved. The performance improvement for SYR2K is even more dramatic: the loop execution time falls by over 80%. MXM, however, is an exception. It is a highly optimized, hand-tuned implementation of multiplying two matrices. Array restructuring neither improves nor degrades its performance.

The impact of the runtime restructuring overhead is small for most of these loop nests, and it is expected to remain small in even larger problems. In the cases of MATMUL, SYR2K, GMTRY, and MXM, this overhead is negligible compared with the loop execution time for all the problem sizes we have measured. In fact, it is so small that in Figure 4 the curve for loop execution time alone and the one that includes the restructuring overhead are indistinguishable. For CFFT2D and CHOLSKY, the overhead is non-trivial, but the benefits of restructuring the array data far outweigh the costs.

For these six loop nests, we expect the amount of computation to grow as fast as, if not faster than, the restructuring overhead, as shown in Table 1. Therefore, the performance impact of the overhead would be even smaller, relative to actual computation, in problems larger than those we have measured. Also, in our current implementation, array data are restructured by a generic runtime routine. We expect the overhead to decrease if the generic routine were replaced by specialized, compiler-generated code.

Loop nest	Problem size parameters	Computation	Overhead
MATMUL	Matrix order (N)	$O(N^3)$	$O(N^2)$
SYR2K	Matrix order (N) and bandwidth (B)	$O(NB^2)$	$O(NB)$
GMTRY	Matrix order (N)	$O(N^3)$	$O(N^2)$
CFFT2D	Array dimension (N)	$O(N^2 \log N)$	$O(N^2)$
CHOLSKY	Matrix order (N), etc.	$O(N)$	$O(N)$
MXM	Matrix order (N)	$O(N^3)$	$O(N^2)$
BTRIX	Array dimensions (J, K, L)	$O(JKL)$	$O(JKL)$
VPENTA	Matrix orders (A, B)	$O(AB)$	$O(AB)$

Table 1: Scaling with Problem Size

For BTRIX and VPENTA, as in the others, array restructuring reduces loop execution times significantly. However, the runtime restructuring overhead, if required, is also substantial. Overall performance improves only marginally. Larger problems are expected to behave similarly because the restructuring overhead and the amount of computation grow at the same rate, as shown in Table 1.

In summary, we found in these experiments that array restructuring can reduce loop execution time significantly. A runtime overhead may be incurred when there is no single array layout that matches the access patterns in separate loop nests and data have to be dynamically restructured. In many cases, this

overhead is outweighed by the performance gain. Our compiler's cost-benefit analysis has correctly identified those cases where it is not.

6.2.2 Memory Overhead

Table 2 lists the amounts of memory required by the original and restructured arrays. The figures are for the largest problems reported in the previous section and include only arrays that are restructured.

Loop nest	Size of Original Array(s) (MB)	Size of Restructured Array(s) (MB)	Change (%)
MATMUL	0.95	0.95	0
SYR2K	6.10	6.41	+5
GMTRY	0.95	0.95	0
CFFT2D	8.00	8.00	0
CHOLSKY	9.54	9.54	0
MXM	0.95	0.95	0
BTRIX	11.6	11.6	0
VPENTA	6.10	6.10	0

Table 2: Memory Overhead

From Table 2, we see that the elements are stored almost as compactly in the restructured as in the unrestructured arrays in all our experiments. For all but one of the loop nests, the restructured and unrestructured arrays have exactly the same size. For SYR2K, the restructured arrays are at most 5% larger than the corresponding unstructured arrays, depending on the bandwidth of the banded matrices.

6.3 Comparing Array Restructuring and Loop Restructuring

To study how array restructuring compares with loop restructuring techniques, we manually transformed the loop nests used in earlier experiments and measured the execution times of the restructured loop nests. For each loop nest, we carefully selected, with the help of experimentation, what we believe to be the best sequence of loop transformations, such as loop permutation, reversal, skewing, scaling, fusion, and distribution[2]. For all the loop nests except CHOLSKY (which we further discuss below), a sophisticated optimizing compiler should be able to identify and apply these transformations.

The results are shown in Figure 5. All execution times are for the largest problems reported in Section 6.2. For each loop nest, the execution times are normalized with respect to that of the original loop.

Let us consider these loop nests in turn. They can be roughly divided into four categories.

For CFFT2D, which implements a two-dimensional FFT, there is no obvious way to restructure the loop nest for better locality, whereas restructuring the arrays speeds up execution significantly. Previous work on loop restructuring did not perform any optimizations on this loop nest [13, 19] or reported no performance improvement [2]. Loop restructuring is frustrated by imperfect loop nesting and non-affine array indexing expressions that involve indirection arrays. Neither of these, however, prevents us from restructuring the

[2]Loop tiling is not included here. The next section studies how it interacts with these loop transformations and with array restructuring.

array. The loop execution time is halved. The overall performance gain is substantial despite a modest restructuring overhead at run time.

For MATMUL, SYR2K, MXM, and CHOLSKY, array restructuring compares favorably with loop restructuring in terms of performance.

- In MATMUL (see Figure 3), interchanging the middle and innermost loops increases performance, but not as much as transposing array `B`. Previous work found this loop interchange to be optimal for overall locality [14, 11], and our experiments with all six possible loop permutations also confirm this. However, it sacrifices some temporal locality in the accesses to `C`, the product array, for much better spatial locality in the accesses to `B`, one of the two operand arrays. It does improve performance, but array restructuring achieves even better performance because we gain spatial locality without losing temporal locality. Runtime restructuring overhead, even if needed, is trivial, as we saw earlier.
- As for SYR2K, adapted from [14], array restructuring reduces execution time to 15% of the original, compared with about 25% for loop restructuring. This loop nest is particularly difficult to optimize because of its complex access pattern. The restructured loop nest that we used is obtained using the technique in [14] [3]. As in the case of MATMUL, array restructuring achieves better performance because it needs not trade temporal locality for gains in spatial locality.
- The performance of MXM (a highly optimized, hand-tuned implementation of multiplying two matrices) is not changed by array restructuring. Performance worsens, however, if the loops are permuted to an order that would appear optimal for locality if the manual optimizations in the original code were absent and that, in fact, yields the best performance among the five possible orders other than the one in the original loop nest.
- For CHOLSKY, array restructuring improves performance slightly more than a carefully chosen series of loop transformations (including loop permutation, distribution, and fusion) that amounts to completely rewriting the imperfectly nested loop. Although these are merely standard transformations, it seems questionable whether this combination can be automatically selected with existing techniques. (Carr et al., who work on improving locality with such loop transformations, report no performance gain for this loop nest [2].) In earlier experiments not reported here, we observed significant performance degradation when only some of the loop transformations were applied.

For GMTRY, BTRIX, and VPENTA, array restructuring and loop restructuring achieve comparable performance improvement, although data restructuring overhead at run time, if needed, might put array restructuring at a disadvantage. In all these routines, appropriate loop permutations improve performance substantially. Almost all the loop nests of interest are perfectly nested and have simple loop-carried dependence patterns. Thus, it should be easy for most loop restructuring techniques to automatically identify the optimization in each case. Array restructuring can achieve comparable performance gains by permuting the array dimensions. However, since such routines are typically only part of a program, the arrays might have to be restructured dynamically. In the cases of BTRIX and VPENTA, this overhead would make array restructuring unattractive.

For EMIT, the original code is already highly optimized for data locality. We have not performed any optimization on this routine. Neither have several others who have experimented with it [13, 2, 19]. No performance results are reported here.

To sum up, we have compared the performance of array restructuring with that of some common loop restructuring techniques. One loop nest that defies loop restructuring is optimized by restructuring the

[3] The restructured loop nest given in [14] contains loop bounds and array indices with integer divisions. We manually transformed the loop nest further, without reordering the iterations, to eliminate these divisions.

array instead. In other cases, array restructuring improves performance as much as or sometimes more than loop restructuring. In most of the loop nests we have examined, runtime data restructuring overheads are outweighed by performance gains, if not negligible. In some cases, however, they are too high for array restructuring to benefit performance if arrays have to be restructured dynamically.

6.4 Array Restructuring and Tiling

In this section, we study how array restructuring interacts with tiling, a powerful locality optimization technique with wide applicability [18]. We manually tiled the innermost loop(s) of three versions of each loop nest wherever appropriate: the original version, the one after array restructuring, and the loop transformed version used in earlier experiments. Each tiled loop nest's execution time was measured for a range of tile sizes.

The results are shown in Figure 6. All execution times are for the largest problems reported in Section 6.2. In each graph, the largest tile size equals the number of iterations in the loop being tiled (except in the case of SYR2K). For these tile sizes, the tiled loop nest has the same access pattern as the untiled version. In some graphs, we show horizontal lines, without individual data points, that represent the execution times of the untiled loop nests. This is done when tiling brings no improvement over what prior loop or array restructuring has achieved. (We confirmed it in each case by performing the tiling anyway and examining the resultant performance curves.) No results are reported for CFFT2D because its imperfect loop nesting and non-affine array indexing expressions would prevent the application of tiling by a compiler.

First, from Figure 6 we see that tiling generally speeds up the original loop nests, but the improvement depends on the tile sizes. Small tiles could cause excessive loop overhead while large tiles would degrade locality. With array restructuring, performance is less sensitive to tile sizes, provided that they are not too small. Therefore, in addition to performing array restructuring by itself, we can also apply it with tiling to reduce the potential performance impact of mistakenly choosing an inappropriately large tile size. The two techniques can always be applied together because array restructuring never changes the loop structure and therefore never makes an originally tilable loop nest untilable.

Second, we observe in Figure 6 that array restructuring, with or without tiling as appropriate, consistently achieves comparable or shorter loop execution times than either tiling the original loop nests or tiling the manually restructured loop nests. Except for BTRIX and VPENTA, this remains the case even when runtime restructuring overhead is taken into account.

7 Restructuring Loops, Arrays, or Both

As mentioned earlier, in order to improve the locality exhibited by nested loops with array accesses, a compiler might restructure the loops, the arrays, or both. In this section, we discuss how these three approaches compare and how they are related, in particular what roles the first two might play in the third, hybrid approach.

Array restructuring has a number of advantages compared with loop restructuring, as we briefly discussed in Section 1. First, array restructuring is not constrained by loop-carried dependences or imperfect loop nesting. It can be applied even when dependence information is limited, imprecise, or non-existent, whereas lack of precise information or complicated loop structures or dependences, common in many large-scale programs, often frustrates automatic application of sophisticated loop transformations [19]. Second, given a single loop nest that accesses multiple arrays, we can restructure each array for optimal locality without sacrificing the locality exhibited by accesses to the other arrays. Loop restructuring, on the other hand,

always affects accesses to all arrays. Third, when a compiler cannot find one layout that is optimal for every loop nest accessing a given array, it still has the option of dynamically restructuring the array data between loop nests. Dynamic restructuring, despite its runtime overhead, may be preferable to a mismatch between the array layout and the access patterns in some of the loop nests.

A major limitation of array restructuring is that it affects only spatial locality. Therefore, it cannot be used to improve temporal locality. However, for the very same reason, it also never degrades temporal locality. This property can be used to advantage in a hybrid approach that combines both loop and array restructuring, as suggested below.

Restructuring both loops and arrays at once is potentially more powerful than restructuring either loops or arrays alone. However, it is not yet clear how this potential can be fully and efficiently exploited. Ideally, the compiler should simultaneously decide how to restructure each array and loop nest to achieve optimal locality for the entire program. So far, this has proven to be too difficult. Cierniak and Li discussed a framework that can represent and evaluate a rich set of array and loop transformations [3]. However, finding the resultant general optimization problem too hard, they select specific transformations to apply to individual loop nests by exhaustively searching an *a priori* subset of the possibilities allowed by their framework [3]. Exhaustive searching can become prohibitively expensive because its cost grows exponentially with the number of arrays and loops analyzed. This limits how many options can be explored within a reasonable amount of time, even with heuristic pruning of unpromising options.

An alternative to exhaustive searching is to iteratively optimize along design space dimensions one at a time, until the solution does not improve any further. This approach can consider infinitely many options and direct the search where an optimum seems most likely. Although a global optimum is not guaranteed, this strategy is commonly used and is often effective in tackling otherwise intractable optimization problems. For our problem of improving data locality, the overall algorithm would alternate between phases of loop restructuring, given fixed array layouts, and array restructuring, given fixed loop structures.

One important advantage of this strategy is that it leverages the many existing, proven loop restructuring techniques, as well as array restructuring techniques like those we have presented here. In this context, however, existing loop restructuring techniques should be adapted to focus on temporal locality, rather than spatial locality or both. The rationale is based on the fact that array restructuring affects only spatial but not temporal locality while loop restructuring affects both. If we optimize for both types of locality when restructuring loops alone, we may have to trade some temporal locality for gains in spatial locality. Such a compromise becomes much less attractive when spatial locality can be improved by array restructuring *without harming temporal locality*. Moreover, the lost temporal locality can never be recovered by restructuring the arrays.

In short, array restructuring and loop restructuring each has its own strengths and limitations, which would make one or the other more effective in enhancing data locality in any given situation. Moreover, both kinds of restructuring techniques can contribute to a hybrid approach that is potentially more powerful.

8 Related Work

Various loop restructuring techniques to improve data locality have been proposed. Kennedy and McKinley developed a cost model to guide the selection of loop permutation transformations for enhancing data locality [11]. They also studied the use of loop fusion and distribution for this purpose [10]. Carr et al. reported an extensive performance study of these techniques [2]. Wolf and Lam use loop tiling, which combines stripmining and loop permutation, to increase the likelihood of reusing cached data in the innermost loops [18]. Li and Pingali proposed a linear algebraic framework for loop transformation [15]. Based on that

framework, Li also developed algorithms for selecting loop transformations to enhance locality and reduce false sharing, especially in banded matrix applications [14].

Data restructuring has also been used to improve spatial locality and reduce false sharing on shared-memory multiprocessors.

Ju and Dietz reduce cache coherence overhead by permuting array dimensions and by loop permutation [9]. They restrict their array restructuring options to permutations of array dimensions, which are subsumed by the linear mappings in our approach. On the other hand, they consider both array and loop restructuring while we focus on array restructuring.

Cierniak and Li propose a linear algebraic framework integrating control and data transformations to enhance locality and reduce false sharing [3]. Their framework allows index-to-address mappings to be any linear mapping, but in practice they restrict attention to those that, in effect, represent different permutations of array dimensions and exhaustively search the design space for an optimal solution. By contrast, we consider more general data transformations, select one algorithmically, and focus on array restructuring.

Anderson et al. restructure arrays to improve spatial locality and reduce false sharing on shared-memory multiprocessors [1]. Given a data distribution, they place array elements assigned to the same processor together in memory. In effect, the compiler computes a data distribution as if compiling for a distributed-memory machine and, for each array, uses contiguous regions of the shared address space as the "local memories" of individual processors. The array layout on each individual processor is not further optimized for locality. We, on the other hand, optimize array layouts for single-processor execution, whether on a uniprocessor or on one processor of a shared-memory multiprocessor.

Jeremiassen and Eggers transform array data structures to reduce false sharing [7, 8]. Their work differs form ours in several ways. First, they focus on coarse-grain, explicitly parallel programs while we are concerned with both sequential and parallelized loops. Second, their goal is to reduce false sharing while we primarily aim to improve spatial locality. Third, their data transformations mainly apply to one-dimensional arrays and are more ad hoc, whereas ours are designed for multidimensional ones and based on a general mathematical framework for access pattern analysis.

Leung and Zahorjan dynamically restructure arrays to reduce false sharing and improve spatial locality in runtime parallelized loops [12]. Since this work focuses on runtime parallelized loops for which the access patterns are irregular, the index-to-address mapping is represented by a runtime generated indirection table rather than a parametric form like a linear mapping. Therefore, array accesses incur extra indexing overhead.

9 Conclusion

Compilers increasingly have to optimize for data locality in order to achieve high performance. A compiler might do this by restructuring loops or the arrays they access. While only loop restructuring can improve temporal locality, array restructuring is more flexible: it is not constrained by imperfect loop nesting or loop-carried dependences; multiple arrays accessed by a single loop nest can be restructured independently for optimal locality; arrays can be dynamically restructured to match the access patterns at different points in program execution.

We have presented techniques to restructure arrays for locality. Our techniques are based on an invertible linear transformation of array index vectors, which is represented by an index transformation matrix. Such a linear transformation is much more general than simply permuting the array dimensions. We have developed algorithms to choose an index transformation matrix that would improve the spatial locality exhibited by a given set of array accesses in a loop nest. Our analysis places no restrictions on the loop structure or

dependence pattern. We focus on those cases where the array index expressions are affine functions of loop variables, but our techniques also apply to non-affine cases. Therefore, these techniques are applicable to a wide class of loop nests.

Results of performance experiments with our implementation show that array restructuring technique can improve loop execution performance substantially. Restructuring overhead at run time, even if needed, is often outweighed and sometimes even overwhelmed by the reduction in loop execution times. Moreover, array restructuring has achieved comparable, and in some cases superior, performance compared with many common forms of loop restructuring. Furthermore, when used with tiling, it can make performance less sensitive to the choice of the tile size.

A Mathematical Proofs

Lemma 1 *Let T be a nonsingular matrix. Both T and T^{-1} are integral if and only if T is unimodular.*

Proof: First consider the "if" part. If T is unimodular, it is integral by definition. Its inverse T^{-1} is also unimodular [16] and thus integral by definition.

Now, we turn to the "only if" part. Since T is integral, $|T|$ is an integer. Since T is nonsingular, $|T| \neq 0$. Thus, the absolute value of $|T|$ must be at least 1. Similarly for T^{-1}. $|T|$ and $|T^{-1}|$ are reciprocals of each other. Therefore, they must be either 1 or -1. Because T is integral and $|T|$ is either 1 or -1, T is unimodular by definition.

Lemma 2 *Let R be a rational nonsingular matrix. There exists a unimodular matrix U such that for any matrix A, corresponding columns of RA and UA have the same heights (i.e., the top nonzeros in each pair of corresponding columns are in the same row).*

Proof: Since R is nonsingular, it is of full row rank. There exists a unimodular matrix V such that RV (denoted H from now on) is in Hermite normal form [16]. The inverses of V and of H both exist. For V, this is because V is unimodular and therefore, by definition, nonsingular [16]. For H, the reason is that $H = RV$ and both R and V are nonsingular.

Let $U = H^{-1}R$. We now show that U is unimodular and columns of UA have the same heights as corresponding columns of RA. First, note that

$$\begin{array}{ll} & H = RV \\ \Rightarrow & H^{-1}(H) = H^{-1}(RV) \\ \Rightarrow & I = (H^{-1}R)V = UV \end{array}$$

Thus, U is the inverse of a unimodular matrix V and for that reason also unimodular [16]. Second, for any matrix A,

$$UA = (H^{-1}R)A = H^{-1}(RA)$$

Since H is in Hermite normal form, it is by definition lower-triangular and therefore so is its inverse H^{-1}. The corresponding columns UA and RA have the same heights. This completes the proof.

B Computing an Index Transformation from the Access Matrix

Earlier in Section 3.2, we outline how to find an index transformation matrix to optimize for spatial locality given an access matrix A. The procedure consists of two steps. In the first step, we compute a nonsingular matrix that flattens A's columns as far as possible. In this section, we show an algorithm to find such a matrix.

The key to our algorithm is to find a series of elementary row operations (i.e., scaling a row, exchanging two rows, and adding a multiple of one row to another) that transforms A to the desired form. It is a well-known fact in linear algebra that any series of elementary row operations can be represented by a nonsingular matrix, say Q: the operations transform A to QA for any matrix A. Therefore, after transforming A appropriately with such operations, our algorithm returns the corresponding Q. Furthermore, we need not record the series of operations and then somehow compute Q. Instead, Q can be calculated efficiently by applying the operations to the identity matrix as they are selected. This yields the matrix Q in the end because the operations transform the identity matrix I to QI, which is, of course, equal to Q.

The main issue is selecting the appropriate elementary row operations. This is done by the algorithm shown in Figure 7. Each iteration performs two related tasks: choosing a target column to flatten and actually flattening it with elementary row operations. We first explain the first and second iterations of the algorithm and then generalize to other iterations.

In the first iteration (where $r = m$), we find the rightmost nonzero column — the column that we wish to flatten most, as discussed in Section 3.1.1 — and reduce its height to 1 with elementary row operations. Specifically, the algorithm picks the rightmost column that has nonzeros in any of the rows 1 through m. Suppose this is the c_1-th column in A. Then, our algorithm uses elementary row operations to place a nonzero in the bottommost (i.e., m-th) row and eliminate all other nonzeros above it in the target column, thus reducing the height to 1. The same operations are applied to R, which has been initialized as the identity matrix. R will become the transformation matrix in the end.

Thus, the elementary row operations chosen by the first iteration flatten column c_1 to the desired height, namely 1. From this point onward, the column remains unchanged even though subsequent iterations perform more elementary row operations. These operations involve only rows 1 through $m - 1$ since $r \leq m - 1$ in these iterations, and so have no effect on column c_1, which contains only zeros in those rows.

In the second iteration (where $r = m - 1$), we again select the column that we want to flatten most. However, we exclude columns with only zeros in rows 1 through $m - 1$ because, as just mentioned, these columns are not affected by whatever elementary row operations we might apply in the second and subsequent iterations. One of these excluded columns is column c_1, which we want to preserve rather than modify because it is already in a desired form. The others, if any, are scaled versions of column c_1. Their images under a certain linear transformation are determined by the image of column c_1, regardless of what the particular transformation might be. Since in the first iteration we have chosen the image of column c_1, the images of these other columns are already fixed, even when the transformation is still incomplete. Therefore, these columns, like column c_1 itself, are not considered any further.

Having chosen a target column (c_2), the second iteration uses elementary row operations to flatten it to a height of 2. A nonzero is placed in the $(m-1)$-th position and all nonzeros above are eliminated. It would be even better for locality if the height were reduced to 1, but this is impossible with an invertible linear transformation, as explained below.

The remaining iterations work similarly. In general, after k iterations have been completed, k target columns (c_1, c_2, ..., c_k) have been chosen, and column c_j has been flattened to a height of j. The columns in the *original* A fall into two categories: those that are linear combinations of the original columns c_1, ...,

c_k and those that are linearly independently of them. (Before the first iteration, all columns belong to the latter.) These two sets of columns are treated differently.

The $(k+1)$-th iteration excludes the first set — columns that are linear combinations of c_1 through c_k. This is because the images of the linear combinations are already fixed once we have chosen the images of the columns c_1 through c_k, even though the transformation is still incomplete. There is no point in any attempt to transform the former differently. After the k-th iteration, all these columns have nonzeros only in the last k positions. By this property the $(k+1)$-th iteration identifies and excludes them.

Among the columns in the second set (which have at least one nonzero in the first r positions), the $(k+1)$-th iteration picks a target column c_{k+1} and reduces its height to $k+1$ using elementary row operations. $k+1$ is the lowest height we can achieve with an invertible linear transformation. For the sake of argument, suppose we could in fact go further. Then, the images of the target columns c_1 through c_{k+1} would all have heights k or less. These $k+1$ vectors span a vector space of k or fewer dimensions because they contain nonzeros only in the last k positions. They are linearly independent because the corresponding columns in the original A are linearly independent and the transformation is invertible. However, it is impossible to find more than k linearly independent vectors in a k-dimensional vector space, let alone one with fewer dimensions.

References

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

[2] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In *Proceedings of Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, October 1994.

[3] Michal Cierniak and Wei Li. Unifying data and control transformations for distributed shared memory machines. In *Proceedings of ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.

[4] Digital Equipment Corporation, Maynard, MA. *DEC 3000 Model 400/400S AXP Technical Summary*, November 1992.

[5] J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, pages 1–17, March 1990.

[6] Todd A. Dutton, Daniel Eiref, Hugh R. Kurth, James J. Reisert, and Robin L. Stewart. The design of the DEC 3000 AXP systems, two high-performance workstations. *Digital Technical Journal*, 4(4), 1992.

[7] Susan J. Eggers and Tor E. Jeremiassen. Eliminating false sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, August 1991.

[8] Tor E. Jeremiassen and Susan J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

[9] Y.-J. Ju and H. Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformation. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 344–358, August 1991.

[10] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the Sixth International Workshop on Languages and Compilers for Parallel Computing*, August 1993.

[11] Ken Kennedy and Kathryn S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of 1992 International Conference on Supercomputing*, 1992.

[12] Shun-Tak Leung and John Zahorjan. Restructuring arrays for efficient parallel loop execution. Technical Report 94-02-01, Department of Computer Science and Engineering, University of Washington, 1994.

[13] Wei Li. *Compiling for NUMA Parallel Machines.* PhD thesis, Cornell University, Ithaca, NY, August 1993.

[14] Wei Li. Compiler cache optimizations for banded matrix problems. In *Proceedings of 1995 International Conference on Supercomputing*, 1995.

[15] Wei Li and Keshav Pingali. A singular loop transformation framework based on non-singular matrices. In *Proceedings of the Fifth International Workshop on Languages and Compilers for Parallel Computing*, August 1992.

[16] Alexander Schrijver. *Theory of Linear and Integer Programming.* Series in Discrete Mathematics. Wiley, 1986.

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

[18] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[19] Michael Edward Wolf. *Improving Locality and Parallelism in Nested Loops.* PhD thesis, Department of Computer Science, Stanford University, August 1992.

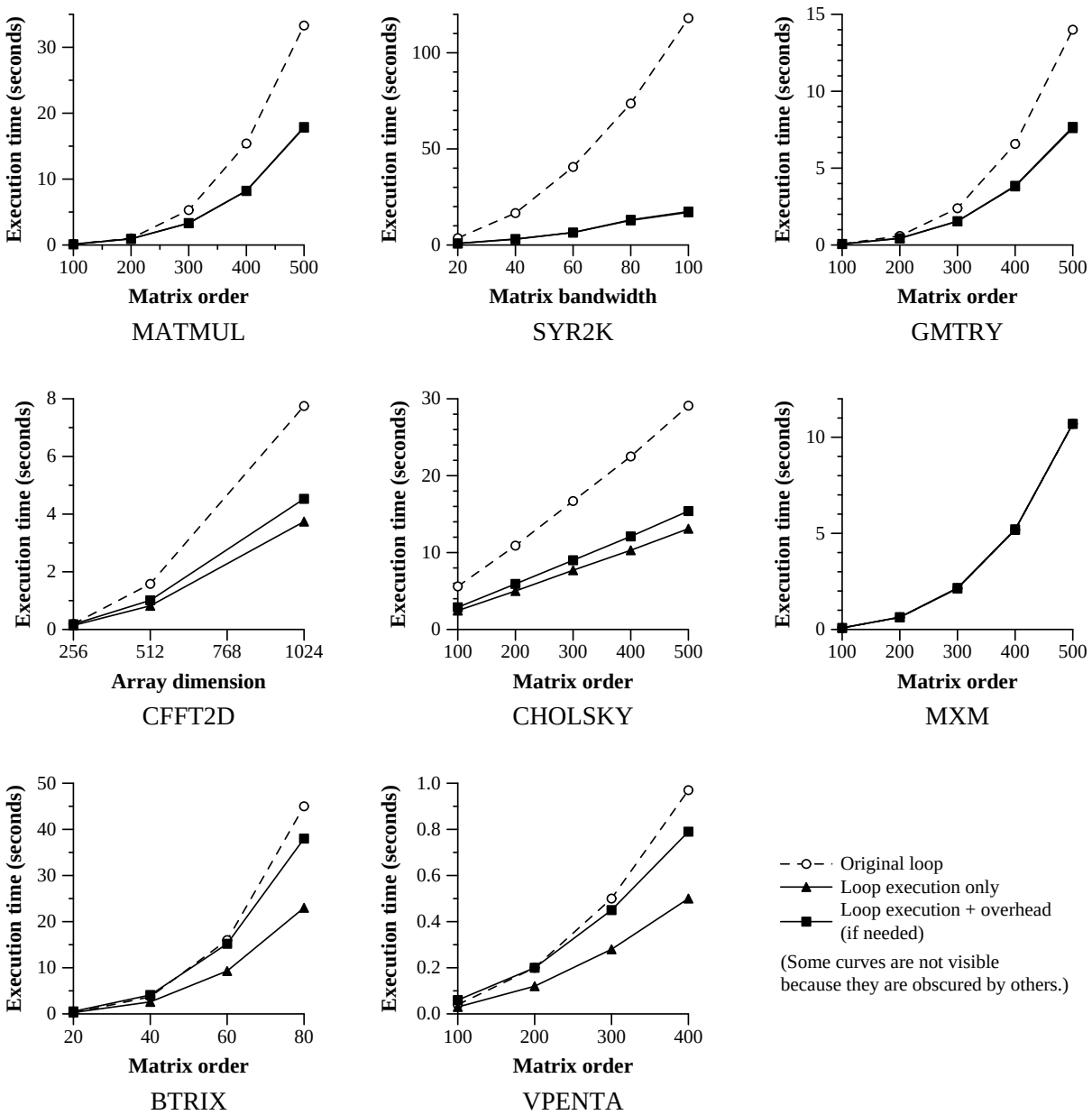

Figure 4: Performance with and without Array Restructuring

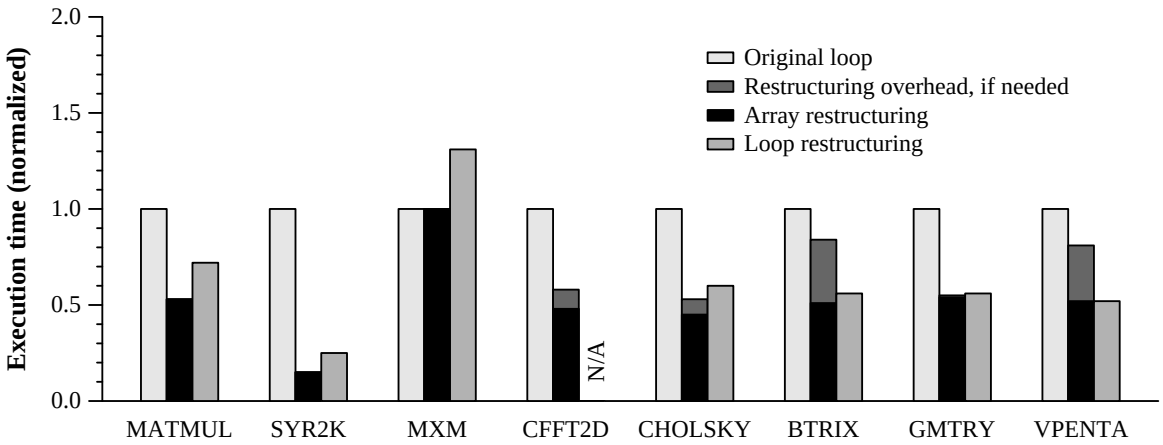

Figure 5: Comparing Array and Loop Restructuring

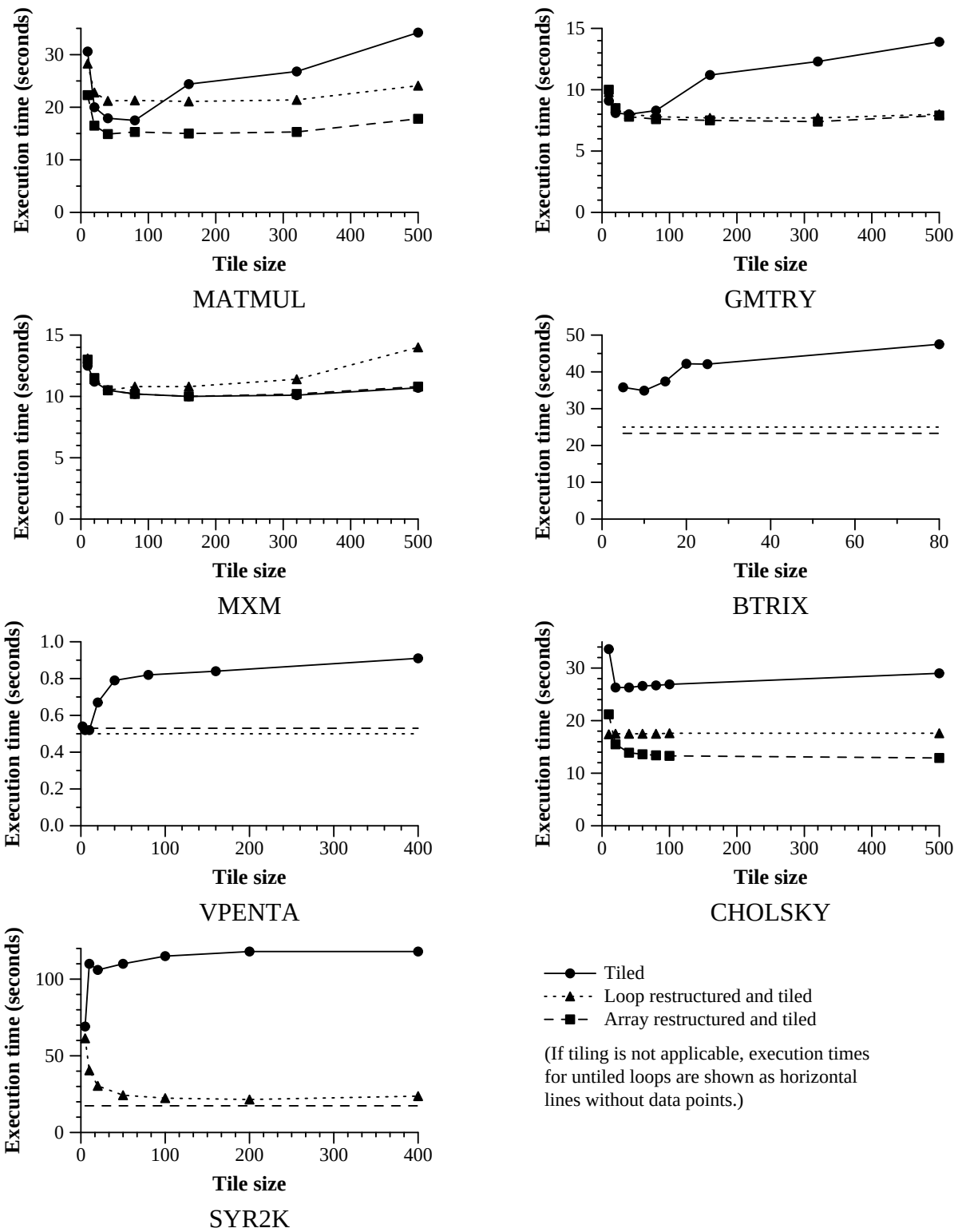

Figure 6: Array Restructuring and Tiling

```
Input: A = access matrix
Output: R = Nonsingular matrix that flattens A's columns

m = number of rows in A
R = m-dimensional identity matrix

for r = m downto 1 do

  /* Choose target column to flatten. */
  Find rightmost column of A with nonzero(s) between rows 1 and r.
  if no such column then
    return
  else
    Let column c be the column found.
  endif

  /* Flatten target column.  Top nonzero will be in row r. */
  /* 'Pivot' to make A[r,c] nonzero.  Update R accordingly. */
  if (A[r,c] = 0) then
    Find a nonzero element among A[1..r,c].  Suppose it is A[s,c].
    Exchange row r and row s in matrix A.
    Exchange row r and row s in matrix R.
  endif
  /* Eliminate nonzeros in column c above row r.  Update R accordingly. */
  for t = 1 to r-1 do
    Multiply row r in A by A[t,c]/A[r,c] and subtract from row t in A.
    Multiply row r in R by A[t,c]/A[r,c] and subtract from row t in R.
  endfor

endfor
```

Figure 7: Computing an Index Transformation from the Access Matrix