Compiler Cache Optimizations for Banded Matrix Problems

Wei Li *
Department of Computer Science
University of Rochester
Rochester, NY 14627
U.S.A.
wei@cs.rochester.edu

Abstract

Almost every modern processor is designed with a memory hierarchy organized into several levels, each of which is smaller, faster, and more expensive than the level below. High performance requires the effective use of the cached data, i.e. *cache locality*. Smart compiler transformations can relieve the programmer from hand-optimizing for the specific machine architectures.

Most of the existing compiler optimizations are developed for dense matrix programs. Irregular problems, on the other hand, have to rely on runtime optimizations, since the data access patterns are unknown at the compile-time. However, many scientific computing problems result in solving linear systems where the matrix of coefficients is *banded*, a structure known at the compile-time, but more complicated than the dense matrices. Banded matrix problems are interesting since substantial savings can be made by exploiting the mathematical properties of the bandedness. The complicated memory access patterns in the banded matrix programs make the existing compile-time optimizations impossible to use.

In this paper, we present a new compile-time technique for optimizing banded-matrix programs. We first develop a new data reuse model and an algorithm called *height reduction* to improve cache locality. Then with the height reduction algorithm, we extend loop tiling to exploit not only intra-tile data locality but also inter-tile data locality. We call the new tiling *affinity tiling*. We show that the algorithms also helps to eliminate or reduce *false sharing* in multiprocessor systems. With the height reduction algorithm and affinity tiling, significant performance improvement (speedups from 2.5 to 10) has been observed on HP workstations (over the original sequential code) and KSR1 multiprocessors (over the original parallel code).

*This work was supported in part by an NSF Research Initiation Award (CCR-9409120) and ARPA contract F19628-94-C-0057.

To appear in the Proceedings of the 1995 International Conference on Supercomputing.

1 Introduction

In a memory hierarchy, the cache is the memory level between CPU and the main memory. It is usually between 1KB to 256KB, and is divided into *blocks (lines)* of 4-128 bytes. The unit of data transfer between main memory and cache is a block. If the same data in the cache is reused, this is called *temporal locality*. Since the unit of data in cache is a block (line), once the block is brought into the cache, any access to the data elements in the block will be a cache hit. This is called *spatial locality*. Typically, a cache hit takes only one cycle, while a cache miss takes 8-32 cycles [20]. It is ideal for all memory references to hit in the cache, however, since cache size is much smaller than main memory, once new data needs to be brought in from main memory, some data in the cache has to be replaced. There will be a cache miss when the replaced data is accessed in the future. Therefore, high performance requires programs to possess *cache locality*, i.e. having data reuse of the data in the cache before it has been replaced.

In a multiprocessor system with multiple caches, data inconsistency may occur between memory and caches. For example, multiple caches may have inconsistent copies of the same cache block. The memory and caches may have inconsistent copies as well. This introduces the problem of *cache coherence*. Several cache coherence protocols have been developed to maintain data coherence for multiple processors. The *write invalidate* protocol causes all copies of the block in other caches to be invalidated before some processor writes to its local copy. A reading processor makes a local copy, and reads from the local copy. This scheme allows single writer and multiple readers. The *write update* protocol requires the writing processor to update all copies in other caches with the new value. A reading processor still makes a local copy. This scheme allows multiple writers and multiple readers. There are tradeoffs between these coherence protocols. Since multiple variables are located in the same block, even when different processors access different variables, it may cause the block to “pingpong” between processors under the write-invalidate protocol or to be updated unnecessarily under the write-update protocol. This problem is called *false sharing*, which has been identified by many researchers as a major obstacle to high performance [12, 4, 5, 10]. Therefore, in a multiprocessor system, we need to exploit cache locality and avoid false sharing.

In this paper, we make the following contributions:

- We develop a new data reuse model and a compiler algorithm called *height reduction* to improve cache locality. The advantage of this algorithm is that it can improve *banded matrix* programs as well as *dense matrix* programs.

- With the height reduction algorithm, we extend loop tiling to exploit not only intra-tile data locality but also inter-tile data locality. We call the new tiling *affinity tiling*.
- We show that the algorithm also helps to eliminate or reduce false sharing in multiprocessor systems.

These algorithms have been implemented in our compiler. We have observed significant performance improvement (from 250% to 1000%) of linear algebra routines on uniprocessor machines (HP workstation) and multiprocessors (KSR1).

The rest of the paper is organized as follows: In Section 2, we discuss the technical problems we are solving. This paper uses the linear loop transformation framework based on non-singular matrices and integer lattice theory [25] to optimize loop nests for better data locality and coherence. We review the loop transformation framework in Section 3. In Section 4, we define a new, more accurate cache reuse model. The optimization algorithm is described in Sections 5, and affinity tiling is presented in Section 6. False sharing is discussed in Section 7. Empirical evaluations on an HP workstation and KSR1 multiprocessor are reported in Section 8. Finally, we conclude in Section 9.

2 Motivation

2.1 Dense Matrix Computations

Most dense matrix problems have simple data access patterns. For example, usually the array references are like A[i, j] in a loop nest, where loop i is a outermost loop and loop j is the innermost loop. When the array A has column major physical layout, a cache line would be a segment of the column. Since the loop nest visits the array in row-major fashion, this loop nest will not reuse the data from the same cache line. By the time the program comes back to visit the next element in the same column after a (possible) long row, that cache line may have been evicted due to small cache size. However, a simple *loop interchange* [32] of loops i and j, if legal, will make the loop nest traverse the columns of the array, which matches the physical layout of the array. Data in caches are reused, since the next array element accessed by the program is the array element next to it in the same column and most likely in the same cache line.

2.2 Banded Matrix Computations

Many scientific computing problems result in solving linear systems where the matrix of coefficients is *banded* [18]. For example, when every variable x_i appears in a few "adjacent" equations of the ith equation, the coefficient matrix is a matrix, where the non-zeros exist in the neighborhood of the diagonal. A matrix A has *lower bandwidth* p if A[i,j] = 0 for $i > j + p$ and *upper bandwidth* q if A[i,j] = 0 for $j > i + q$.

Banded matrix problems are interesting since substantial savings can be made by exploiting the mathematical properties of the bandedness [1, 6, 9, 11, 16, 17, 26]. Savings can be made when solving banded triangular systems. Furthermore, many important numerical operations will preserve the bandedness. When solving banded systems, the triangular factor of LU decomposition, Cholesky decomposition and SVD decomposition will all be banded. Matrix multiplication, rank-k and rank-$2k$ updates of a symmetric matrix will result in a banded matrix as well when the input matrices are banded.

There are a few ways of representing a banded matrix. The most straightforward way is to use a dense matrix as shown in Figure 1(a). The advantage of this approach is that very little change is required of the dense matrix program. For example, as shown in Figure 1(b), the assignment of a banded matrix is a slight modification of the dense case, where the loop bounds are changed according to the band width. An alternative approach is to store the non-zero elements only as shown in Figure 1(c). The advantage of this approach is that memory space is saved. The disadvantage is that the program to access such a data structure has more complicated data access patterns that prevent existing compiler optimizations. An example of banded matrix assignment is shown in Figure 1(d).

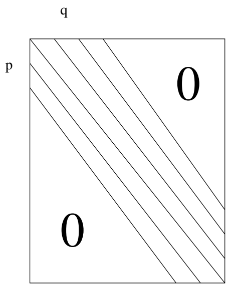
Following the convention of storing the banded matrix in the scientific computing community [18], we assume that all banded matrices are stored like Figure 1(c), i.e. only the non-zero elements are stored. This has increased the difficulty of exploiting data locality. In order to exploit cache locality, it is preferable that data accesses are within a small working set. Due to the complicated data access patterns, a reasonable implementation may not display data locality. For example, the program in Figure 1(d) traverses the array along the diagonals. The next element will definitely not be in the same cache line as the current one in either column-major (as in FORTRAN) or row-major (as in C) layout. The traditional compiler optimization techniques developed for dense matrix problems can not handle these complicated data access patterns.

In this paper, we present a new approach for optimizing banded matrix problems to improve data locality, where the banded matrices are stored with the "memory-saving" data structures as in Figure 1(c).

2.3 Related Work

There is an immense amount of literature in the scientific computing community on banded matrix computations [1, 6, 9, 11, 16, 17, 26]. A considerable amount of research deals with the reduction of the band width, and taking advantage of the mathematical properties of the banded matrices. For example, a nice property is that the triangular factors in LU are also banded when solving banded systems.

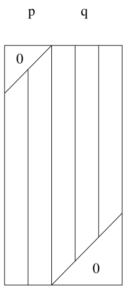
Previous work on compile-time algorithms for cache locality has been on *dense matrix* computations. Wolf and Lam [30] focus on loop tiling of the innermost loops as a means of achieving cache locality. They try all possible subset of the loops in the loop nest, and then try to bring that subset into the innermost position. The subset that can be brought into the innermost position, and has the best objective function from the *reuse vector space* model is chosen to be tiled. The techniques by Ferrante, Sarkar, and Thrash [14] estimate the number of distinct cache lines used by a given loop in a loop nest. Given this estimate, they compute the number of cache misses for a loop nest. Carr, McKinley and Tseng [7] developed a simple memory model that can be used for both perfectly and imperfectly nested loops. They used loop permutations, fusion and distribution for transformations. Gannon, Jalby and Gallivan [15] introduced the notion of *uniformly generated* data reuse, and proposed the notion of *windows* to capture the data reuse. Eisenbeis, Jalby, Windheiser and Bodin [13] used the windows to develop a strategy to explicitly manage the data transfers to local memory. Porterfield [27] studied the problem of estimating the number of cache lines for uniprocessor



(a) Dense Data Structures

```
for j = 1, n
   for i = max(1, j-q+1), min(j+p-1, n)
      A[i, j] =
   end
end
```

(b) Banded Matrix Algorithm



(c) Banded Data Structures

```
for j = 1, n
   for i = max(1, j-q+1), min(j+p-1, n)
      A[i, j-i+p+1] =
   end
end
```

(d) Banded Matrix Algorithm

Figure 1: Banded Matrix Computation

machines, when the cache line size is 1. Li and Pingali [24] showed that loop transformations can reduce non-local memory accesses on NUMA parallel machines. As a by-product, the technique can also improve cache locality by generating stride one accesses. A recent work by Cierniak and Li [8] unifies both data and control transformations for improving data locality.

The work by Eggers and Jeremiassen [12] and by Bianchini and LeBlanc [4] shows that for some programs, program restructuring and data restructuring can eliminate or reduce false sharing so that the performance can be improved. However, these transformation techniques are all performed by hand on specific application programs. No systematic compiler transformation techniques are available. Bolosky and Scott examine several candidate formalizations of false sharing, and provide additional evidence of its importance [5]. Dubois et al. describe a run-time mechanism to eliminate misses due to false sharing [10]. The recent work by Granston and Wijshoff proposes using loop blocking and scheduling of a single loop to avoid false sharing [19].

3 Linear Loop Transformations

In this paper, we use loop transformations to optimize the program for data locality. We use the linear transformation framework by Li and Pingali [25] to develop our new algorithms. This framework makes it possible to employ complex program transformations to improve data locality and eliminate false sharing for banded matrix programs. This transformation theory is based on the use of integer lattices as the model of loop nests and the use of non-singular matrices as the model of loop transformations.

The framework has been extended by Ayguade and Torres to allow the transformations of statements in the loop nest [2]. Linear loop transformation has been a very active area with a lot of interesting results, e.g. the work by Wolf and Lam [31], and Benerjee [3] on unimodular transformations; and other general frameworks by Sarkar and Thekkath [28], Kelly and Pugh [21], and Kulkarni, Stumm, Unrau and Li [22].

In this section, we give a brief overview of the framework from [25]. We reduce the problem of transforming a loop nest to transformation of its iteration space, where an iteration space is an integer lattice [29], and every dimension corresponds to one loop in the loop nest. For example, a loop nest with loop i and loop j (i the outer loop, and j the inner loop) is represented by a two dimensional iteration space. Loop iteration $(i = 2, j = 3)$ is represented by the point $(2, 3)$ in the iteration space.

The framework is quite general, and allows all linear, one-to-one mappings from the iteration space of the source program to the iteration space of the target program. Linear, one-to-one mappings between iteration spaces can be modeled using integer, non-singular matrices. Performing a sequence of transformations corresponds to composing the mappings between iteration spaces, which, in turn, can be modeled as the product of the matrices representing the individual transformations. Since the product of any number of integer, non-singular matrices is integer and non-singular, it follows that the set of linear transformations is closed under composition.

When we construct a transformation, we need to make sure the transformation does not violate the data dependences, since not every linear transformation is legal with respect to the data dependences in the loop nest. The resulting data dependence vector is computed by applying the transformation to the original data dependence vector. A transformation is legal iff the every resulting dependence vector by applying the transformation to the original dependence vector is lexicographically positive.

When we have applied a legal transformation, we must generate DO-loops that scan the points of the target iteration space in lexicographic order, and replace occurrences in the loop body of the old loop indices with the new loop indices. Some of these problems are non-trivial. More details can be found in [25].

To see how this framework is useful for optimizing banded matrix programs, we consider the program in Figure 1(d) again. The loop nest (j, i) is represented by the iteration space (j, i). Let (u, v) be the iteration space of the transformed loop nest. We can use the following non-singular matrix as the transformation matrix.

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} -1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} j \\ i \end{pmatrix}$$

From this matrix, we can generate the new loop nest using the algorithms in [25]. The new loop nest is shown below. This program is good for cache locality, since it traverses the array in column-major fashion.

```
for u = max(1-q, 1-n), min(-1+p, -1+n)
  for v = max(1, u+1), min(n, u+n)
    A(v, -u+p+1) = 0
  end
end
```

The remaining question is how to construct the transformation matrix automatically at compile-time. Obviously, the transformation matrix has to be non-singular, and does not violate data dependences [25]. In the next section, we propose a cache reuse model, and in Sections 5 and 6 we present the algorithms for constructing transformations to improve data locality.

4 A Cache Reuse Model

In this section, we present a new data reuse model. A program must have data reuse in order to exploit cache locality. For example, consider the program where the loops i and j have two references *A[i, j]* and *A[i-1, j+1]* in the loop body. The data element *A[i, j]* is accessed in both iteration (i, j) via the reference *A[i, j]* and iteration $(i+1, j-1)$ via the reference *A[i-1, j+1]*. Therefore, there is reuse carried by the reference *A[i, j]*.

4.1 Reuse Vectors

We define data reuse concisely using an integer vector that represents two loop iterations that access the same data location. This vector is called the *reuse vector*. Consider a loop nest of n loops. An iteration of the loop nest can be represented by an integer vector of dimension n. Let $\vec{j}_1$ and $\vec{j}_2$ be two iterations that access the same data location. $\vec{r} = \vec{j}_2 - \vec{j}_1$ is called a *reuse vector*. A reuse vector is an n-dimensional vector in a loop nest of depth n. The reuse vector for the example above is $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$.

An important attribute of a reuse vector is the *height*. The *height* of a reuse vector is the number of dimensions from the first non-zero entry to the last entry. For example, the height of the reuse vector $\vec{r} = (0, 2, 0, -1)^T$ is 3, since the first non-zero is at the second dimension. Another important term is the notion of a reuse vector *carried* by some loop. If the ith dimension has the leading non-zero in the reuse vector r, then r is said to be *carried* by the loop i. For example, the reuse vector above is carried by the second loop. If a reuse vector is carried by the loop i, then the height of the reuse vector is $n - i + 1$.

The data reuse vector can be computed by solving a system of linear equations [15, 30]. In general, for an m-dimensional array R in a loop nest of depth n, $R[a_{11}i_1 + ... + a_{1n}i_n + a_{10}, ..., a_{m1}i_1 + ... + a_{mn}i_n + a_{m0}]$ can be written as $R[A\vec{i} + \vec{a}]$, where $A = (a_{ij})$ and $\vec{a}^T = (a_{10}, ..., a_{m0})$. There is *temporal reuse* between references $A\vec{i} + \vec{a}_1$ and $A\vec{i} + \vec{a}_2$ iff there exist iterations $\vec{i}_1$ and $\vec{i}_2$ such that $A\vec{i}_1 + \vec{a}_1 = A\vec{i}_2 + \vec{a}_2$. Then, the reuse vector $\vec{r} = \vec{i}_2 - \vec{i}_1$ is the solution to the following equation: $A\vec{r} = \vec{b}$, where $\vec{b} = \vec{a}_1 - \vec{a}_2$. Any reuse vector is a linear combination of a basis of the null space of A plus a special solution to the system: $\vec{r} = c_1\vec{r}_1 + ... + c_k\vec{r}_k + \vec{r}_0$. The set $\{\vec{r}_1, ..., \vec{r}_k\}$ is a basis of the null space of A. $\vec{r}_0$ is special solution to the system. For *spatial reuse*, data reuse depends on the memory layout of arrays. In FORTRAN, where arrays are column major, reuse vectors are computed by solving the following system: $\overline{A}\vec{r} = \overline{b}$ and $|\vec{\alpha}_1^T\vec{r} - b_1| < s$, where $\overline{A}$ is A with first row deleted (or first few rows depending on the size of these array dimensions), $\overline{b}$ is $\vec{b}$ with the first element deleted, $\vec{\alpha}_1^T$ is the first row of A, b_1 is the first element of $\vec{b}$, and s is the size of a cache line.

Definition 4.1 *Reuse distance* is defined to be the number of loop iterations between two reuse iterations.

Given a reuse vector $\vec{r}^T = (r_1, ..., r_n)$, the reuse distance d can be computed easily: $d = r_1S_1 + ... + r_nS_n$, where $S_i = \prod_{k=i+1}^{n} w_k$, and w_k is the trip count of loop k.

There is a useful fact we will use for optimizing the reuse distance.

Lemma 4.1 *If every loop has more than one iteration, then* $\forall_i S_i > S_{(i+1)}$.

To increase the probability of cache locality, we need to shorten the reuse distance.

Theorem 4.1 *If the number of leading zeros in the reuse vector are increased, the reuse distance is shortened.*

Proof: Clearly, since $\{ S_1, ..., S_n \}$ is a strictly decreasing sequence.

The algorithm developed in Section 5 uses loop transformations to reduce the height of the reuse vectors.

4.2 Data Reuse Matrix

To make the optimization on reuse vectors more convenient, we introduce another important data structure called the *data reuse matrix*. The set of reuse vectors is a linear combination of a basis for the null space of A plus a special solution as shown above. A data reuse matrix is a matrix that has the basis vectors and the special solution as its columns. In other words, the reuse matrix for the reuse is as follows: $R = (\vec{r}_1, ..., \vec{r}_k, \vec{r}_0)$. Note that all reuses, *self, group, temporal* and *spatial*, defined in [30] are included in the reuse matrix.

In general, there are many individual reuse matrices (for example, every pair of array references that have reuse generate a reuse matrix.). The *global* reuse matrix is the union of all these individual

The outermost loop is loop 1.

reuse matrices. It is important to have priority on the reuse vectors. A simple heuristic is to construct the global reuse matrix as follows. Let u be the number of individual reuse matrices. We choose the highest priority for the vector space intersection of all individual reuse matrices (each matrix represents a vector space). We compute a basis for that intersection, and make the set of basis vectors as the first columns of the global reuse matrix. The second step is to compute a basis for the intersection of any $u-1$ individual reuse matrices, and add the basis vectors as the columns to the right of the global reuse matrix. This set of vectors has the second priority. Then we compute the intersections of any $u-2$ reuses and so on. The basis vectors from individual reuses have the lowest priority. Therefore, the matrix is ordered from the left to the right with the decreasing priority.

Since our loop transformations are linear, the transformed reuse vector can be easily computed.

Lemma 4.2 *If T is a linear transformation and $\vec{r}$ is a reuse vector, then $T\vec{r}$ is the new reuse vector.*

Theorem 4.2 *If T is a non-singular transformation and R is a reuse matrix, then the new reuse matrix is TR.*

5 Height Reduction

From the discussion in Section 4 (Theorem 4.1), we know that given a data reuse matrix, the goal of the height reduction algorithm will be to find the minimum s such that rows 1 to $n-s$ of the data reuse matrix are all zeros. As we discussed in Section 3, when we construct a legal transformation, we need to make sure that the transformation is non-singular, and does not violate data dependences.

We use the following notation. Let T be the transformation matrix being constructed, D be the dependence matrix (every column is a dependence vector), and R be the reuse matrix (every column is a reuse vector).

We construct the matrix row by row, i.e. we construct row 1 first, then row 2, row 3, ... and so on. Therefore, T is the matrix whose rows are what we have constructed so far. We need to add enough rows to make a non-singular matrix T. To construct the first row of the transformation so that the first row of the new reuse matrix is 0, from Theorem 4.2, we need to find a vector in the null space of the old reuse matrix. Suppose we have formed a matrix T of s rows that satisfies the condition. To extend the matrix, we must find a vector v such that the following three conditions are satisfied:

1. v is linearly independent of the rows of T,
2. v is in the null space of the columns of R, and
3. v does not violate D.

The first condition guarantees that the resulting transformation matrix is non-singular, and the second condition guarantees that data reuse is not carried by loop $s+1$. For example, consider the reuse matrix $R=(1,-1,0)^T$. To compute the first row of the transformation matrix, we can use the vector $(0,0,1)$, which is an element of the null space of R. Now the partial transformation T is $(0,0,1)$. To find the second row, we need to find a vector v that satisfies both the above conditions. In this case, $(1,1,0)$ would be a solution.

While both non-singularity and dependence conditions must be satisfied simultaneously, for presentation purpose we first show how the condition of non-singularity can be satisfied, then we show how the dependence condition can be satisfied, and at last we present the complete algorithm.

Non-singularity The non-singularity condition can be satisfied using an efficient algorithm that finds an orthogonal matrix (therefore, non-singular) instead of any non-singular matrix. For example, instead of finding a linear independent vector v, we can find a vector from the null space of T^T. It is easy to compute a vector from the null space of both T^T and R. Let Q be the column union of T^T and R. If Q does not span the whole vector space, any vector in the null space is a linear combination of a set of basis vectors: $v=a_1x_1+\ldots+a_tx_t$. If Q spans the whole space, then some reuse vectors must be deleted before any row can be added to T^T. The vectors deleted are the ones with the lowest priority. Recall that the reuse matrix is ordered from the left to the right, with the highest priority column as the first column, and the lowest priority column as the last column.

It may appear that finding an orthogonal matrix restricts the solution space. The good news is that we can show that finding a vector from the null space of T^T is *not* less general than finding a vector that is linearly independent of T^T.

Theorem 5.1 *If there exists a linearly independent vector v_1 of T^T that is in the null space of R and does not violate D, then there is an orthogonal vector v of T^T that is in the null space of R and does not violate D.*

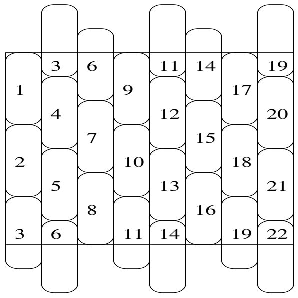
Proof: Let v be the resulting vector of v_1 projected to the orthogonal space of T^T by the projector J (e.g. $J=I-T(T^TT)^{-1}T^T$). Since $v_1^TR=0$, $v^TR=(Jv_1)^TR=v_1^TJ^TR$. Notice that J is a symmetric matrix, and R is already in the orthogonal space of T^T. Therefore, $J^TR=JR=R$. Hence $v^TR=v_1^TR=0$.

Similarly, $v^TD=v_1^TJ^TD=v_1^TD\geq 0$, if $v_1^TD\geq 0$. □

Dependences Now, we consider data dependences. In addition to the non-singularity condition, any row added must not violate data dependences. Any dependence carried by the existing rows can be deleted from the dependence matrix, since any added row will not produce illegal dependences [25]. Since $v=a_1x_1+\ldots+a_tx_t$, we need to choose a set of values for $\{a_1,\ldots,a_t\}$ such that $v^TD\geq 0$. We first consider the case where there is only one data dependence vector. Let d be the dependence vector, then $v^Td=a_1x_1^Td+\ldots+a_tx_t^Td$. We choose the values of each a_i to make sure that v^Td is legal, i.e. it is lexicographically positive. In the general case where there are more than one dependence vector, the value of a_i must be consistent with every dependence vector. If this fails, some reuse vectors must be deleted.

Algorithm From the above discussions, we are able to sketch the algorithm as follows. Let T be the transformation matrix being constructed, D be the dependence matrix (every column is a dependence vector), and R be the reuse matrix (every column is a reuse vector). Initially, there are no rows in T; D has all of the original dependence vectors; and R has all of the original reuse vectors. The algorithm constructs T row by row. At every step, one row is added to T; some dependence vectors are deleted from D if they are carried by the rows in T; and some reuse vectors may be deleted from R. The following invariants are maintained at each stage of the construction: $TD=0$ and $TR=0$.

At every step, a new row matrix Q is created by simply com-



(a) Array A

```
for ti = 1, n, b
  for tj = 1, n, b
    for i = ti, min(n, ti+b-1)
      for j = tj, min(n, tj+b-1)
        A[i, j]
```

(b) simple tiling

```
for tj = 1, n, b
  for ti = 1, n, b
    for j = tj, min(n, tj+b-1)
      for i = ti, min(n, ti+b-1)
        A[i, j]
```

(c) affinity tiling

Figure 2: Inter-tile Locality

bining the rows of T and the columns of R. If Q spans the whole space, then its null space is empty. Some lower priority reuse vectors have to be deleted until Q does not span the whole space. Once a solution, which consists of a set of all valid vectors, for the null space of Q is found, we need to choose a vector that satisfies the data dependences as discussed above. More details can be found in [23].

6 Affinity Tiling

The goal of loop tiling [32, 30] is to exploit the data locality within the tile. We propose a new tiling called *affinity tiling*, which makes use of height reduction, to exploit the data locality not only within a tile but also among tiles. The experiments in Section 8 show that an important advantage of affinity tiling is that it is less sensitive to the tile size.

6.1 Inter-tile Locality

Inter-tile locality can be described using the example in Figure 2. A is an 8x8 array, which has the column major physical layout. Each cache line has three words. The first column takes the first two cache lines, and a part of the third cache line; the second column occupies the rest of the third cache line, the fourth and fifth cache lines, and a part of the sixth cache line; and so on. When we tile the loops with tile size b that equals the cache line size (the choice of b is irrelevant), the first tile (ti=1 and tj=1) has the first cache line, a part of the third, the fourth, the sixth and the seventh cache lines. The tile (ti=4, tj=1) also has a part of the fourth and seventh cache lines. Therefore, there is reuse between these two tiles. The tiled version in Figure 2(b) may not exploit the reuse, when the array is much larger than the cache. On the other hand, the tiled version in Figure 2(c) will always turn the reuse into locality. The algorithm presented in Section 6.2 would generate the tiled version in Figure 2(c).

6.2 Affinity Tiling Algorithm

Now we describe the affinity tiling algorithm, which is an extension to the height reduction algorithm. The intuition is that after height reduction is applied, the loop nest is in the right "order". Therefore loop tiling of m loops will result in both the "tile" loop nest (the outer m loops) and the "in-tile" loop nest (the inner m loops) in the right order to exploit both intra-tile and inter-tile locality.

Loop tiling is not always legal [32, 30]. If the dependence vector is $(1,-1)^T$, it will prevent tiling. The reader can easily check that tiling would violate data dependences. However, loop transformations may be applied to make a loop nest tilable. A loop reversal on the second loop will change the dependence vector to $(1,1)^T$, and make tiling legal. Before we construct such transformation, we need to know the dependence condition for tiling. For a loop nest $i_1,..,i_n$, loops $i_k,...,i_n$ can be tiled if any dependence carried by loops $i_k,...,i_n$ has no negative entries. Any negative entry in the dependence vectors carried by the loops $i_1,...i_{(k-1)}$ will not prevent the tiling of loops $i_k,...,i_n$.

The affinity tiling algorithm constructs a transformation matrix that reduces the height of the reuse vectors, and creates opportunities for tiling. It will find the largest innermost loop nest by minimizing the *blocking level* k. A row s is calculated to reduce the height of the reuse vectors, and to make the loop i_s tilable. In the algorithm, a dependence vector will "flow" from D, the dependence matrix carried by loops $i_k,...i_n$, to D_o, the dependence matrix carried by loops $i_k,..,i_{(s-1)}$, and then be deleted if it is carried by the loop $i_1,...,i_{(k-1)}$. When the vector from the null space of T and R is computed, we need to make sure that it does not violate the data dependences in D. If this fails, then no such row s exists. To guarantee that the nest $i_k,...,i_s$ can be tiled, we will not generate negative entry in any dependence vector in D_o. If a negative entry is generated for a dependence vector d, then the highest tilable level, k, is the loop below the loop carrying d. Therefore, if we have to generate negative entries, it is better to give up the dependence vectors carried by the outer loops in loops $i_k,..,i_{(s-1)}$.

7 Eliminating False Sharing

As we mentioned earlier, false sharing occurs when two or more processes access non-overlapping portions of the same coherence block (at least one of them with writes). True sharing means that a data element is defined by one process, and used by another. True sharing implies communication or synchronization. To achieve more parallelism, and less communication, both true sharing and false sharing should be eliminated. If the program is an explicit parallel program, then true sharing can not be avoided. Only false sharing may be eliminated. However, when we start from a sequential code, we have the flexibility to parallelize the program so that both true sharing and false sharing can be eliminated or reduced.

```
for i = 1, n
  for j = i, min(i+2*b-2, n)
    for k = max(i-b+1, j-b+1, 1),
            min(i+b-1, j+b-1, n)
      Z(i, j-i+1) = Z(i, j-i+1)
        +X(k, i-k+b)*Y(k, j-k+b)
        +X(k, j-k+b)*Y(k, i-k+b)
```

(a) Source

```
for u = max(2-2*b, 2-n-b, 0),
        min(-2+2*b, -2+n+b, -2+2*n, -1+n)
  for v = u+2*max(-u+1-b, 1-b, 1-n, -u+1-n),
          u+2*min(-u-1+b, -1+b, -u-1+n), 2
    for w = max(1, (u+v)/2+1),
            min(n, (-u+v)/2+n, (u+v)/2+n)
      Z((-u-v)/2+w, u+1) = Z((-u-v)/2+w, u+1)
        +X(w, (-u-v)/2+b)*Y(w, (u-v)/2+b)
        +X(w, (u-v)/2+b)*Y(w, (-u-v)/2+b)
```

(b) Transformed

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

(c) Reuse Matrix

$$\begin{pmatrix} -1 & 1 & 0 \\ -1 & -1 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$

(d) Transformation

Versions	Source	Transformation	Strength Reduction
Times (sec.)	101.3	14.36	14.26
Speedup		7.05	7.10

(e) Execution Times

Figure 3: Banded SYR2K

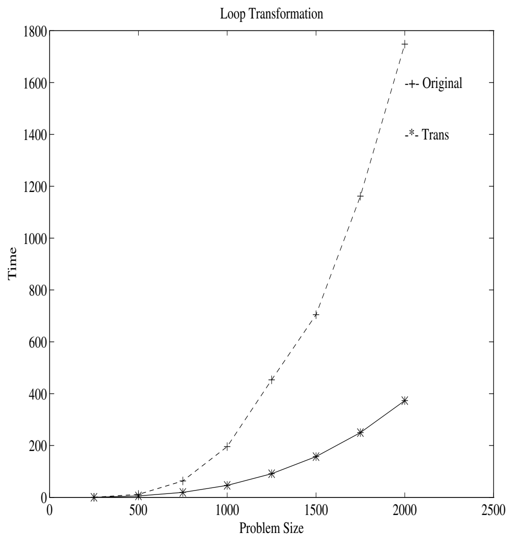


Figure 4: Cholesky Decomposition

We will show that the algorithm developed in Section 5 also helps to eliminate or reduce false sharing. Consider the simple example in Figure 1(d). If we parallelize loop j, then each process works on one diagonal (from the top-right corner to the bottom-left corner) of the matrix. The adjacent diagonals share the same block , which causes false sharing. On the other hand, if we parallelize loop i, then each process works on one row of the array. Once again, the adjacent processes will cause false sharing by sharing the same block. However, if we transform the program to the one as shown at the end of Section 3, and parallelize loop u, then there is no false sharing, since every process is working on one column of the array.

The example in Figure 1(d) displays only spatial reuse, which is carried by loop j. When loop j is parallelized, the spatial reuse turns into false sharing. If there were temporal reuses carried by loop j, then a write/read and write/write reuse would be true sharing that requires synchronization.

A block can be a page in a DSM system, or a cache line. More precisely, it is the unit of data coherence.

We sketch the algorithm for eliminating or reducing false sharing. First the height reduction algorithm is applied to the loop nest. The data reuses, including spatial reuses that cause false sharing, are "pushed" into the innermost loops as much as possible. The outermost loops will be the loops that do not carry reuses. If a loop that carries no spatial reuse is parallelized, then there is no false sharing. If a loop that carries only spatial reuse is parallelized, then false sharing can be eliminated or reduced by blocking according to the block/page/line size at the cost of parallelism.

8 Experiments

8.1 Methodology

We conducted experiments on two machines. One is a uniprocessor (HP 9000/720); the other is a multiprocessor system(KSR1). HP 9000/720 has 256KB data cache, and 32MB main memory. A word has 4 bytes. The cache latency is 1 cycle, and the memory latency is 15 cycles. Benchmark programs are first optimized by our optimizing compiler, and then compiled with the native FORTRAN compiler with the -O option. KSR1 is a 64-bit shared memory multiprocessor. Each KSR1 processor is connected to a *subcache*, which is the first level cache. The subcache has 0.5MB divided into a 0.25 MB instruction subcache and a 0.25MB data subcache. The second level cache, called the *local cache*, has 32MB. As expected, it is much faster to access the subcache than local cache, which is faster to access than the local cache of some other processor. For example, the ratio of a ring1 memory access latency and a subcache access latency is 300. Our compiler takes programs in FORTRAN, and generates programs in FORTRAN with KSR parallel constructs. The native KSR FORTRAN compiler with the Presto runtime system is used to generate executable code. The native scalar optimizations are used. The timing is recorded using the KSR Performance Monitor, Pmon.

We present the results for two applications, banded rank-$2k$ update and Cholesky decomposition. The banded rank-$2k$ update

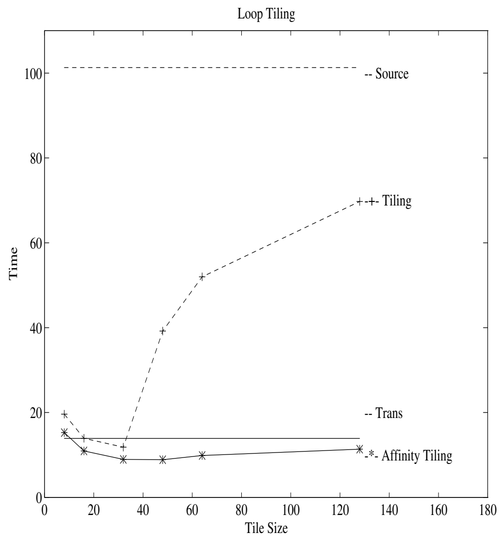


Figure 5: Affinity Tiling - banded rank-2k Update

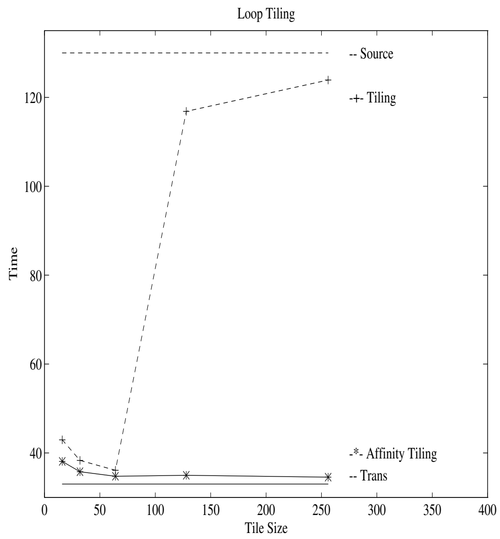


Figure 6: Affinity Tiling - Cholesky Decomposition

program computes the rank-2k update for a banded symmetric matrix. Cholesky computes the factorization of a positive definite matrix.

8.2 Height Reduction

We use banded SYR2K to show that our techniques can handle quite general data access patterns. Again, only non-zero elements are stored as in Figure 1(c). We include the original program in Figure 3(a) for convenience. First, we compute the reuse matrix, shown in Figure 3(c), using the algorithm in Section 4. The reuse vectors computed from the references *Z(i, j-i+1)* are the second and third columns in the reuse matrix, which include both temporal and spatial reuses (Note that the temporal reuses are a subset of the spatial reuse. Therefore, only spatial reuse vectors are computed.); the reuse vectors (both temporal and spatial) for *X(k, i-k+1)* (also *Y(k, i-k+b)*) are the fourth and fifth columns of the reuse matrix; and the last two columns are the temporal and spatial reuse vectors for *Y(k, j-k+b)* (also *X(k, j-k+b)*). The first column is the basis vector of the intersection of all reuses, i.e. the most important reuse dimension. We then call the height reduction algorithm to produce the transformation matrix, shown in Figure 3(d), from that reuse matrix. Given the transformation matrix, we use the algorithms in [25] to generate the new loop nest, shown in Figure 3(b). The generation of the new loop nest requires handling general loop bounds with *max*, *min*, and variables, which is provided by our loop transformation framework. In the transformed version, height reduction exploits the spatial locality of the reference C, A, and B, with respect to the innermost loop. In the final stage, the loop nest is tiled to improve the temporal locality.

We would like to emphasize that the multiplications and divisions generated in the transformed code can *always* be strength-reduced to additions and subtractions [25]. In fact, we will present the results with and without strength reduction.

The execution times for the different versions on the HP workstation are shown in Figure 3(e). The problem size n is 500. The time for the original version is 101.3 seconds. After height reduction transformations, the time is reduced to 14.36 seconds, which has the speedup of 7.05. The division operations do not seem to have any negative performance impact, since strength reduction of the divisions in the program does not change the performance.

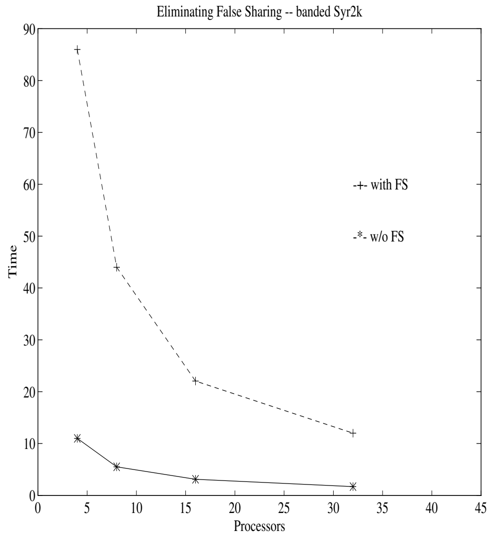
Another example is Cholesky decomposition written in the form of three nested loops. We would like to study data sizes ranging from 250 to 2000. Since the cache size on the HP 9000/720 is 64K words, a two dimensional array of 250 $\times$ 250 is about the largest that can fit in the cache. The original code has poor locality, since the inner two loops traverse the array in a row-wise fashion, while FORTRAN uses column major form for array layout. The cache lines (column segments) may be displaced before the other elements in the same cache line can be accessed. The transformation is a loop interchange of the innermost two loops.

The results are listed in Figure 4. When the problem size is 250, there is almost no improvement. This is expected, since the array is small enough to fit in the cache. When the size is 500, we see a moderate improvement of factor 2. The improvement is significant when the size is larger than 1000. We get the speedup of 5 over the original version.

8.3 Affinity Tiling

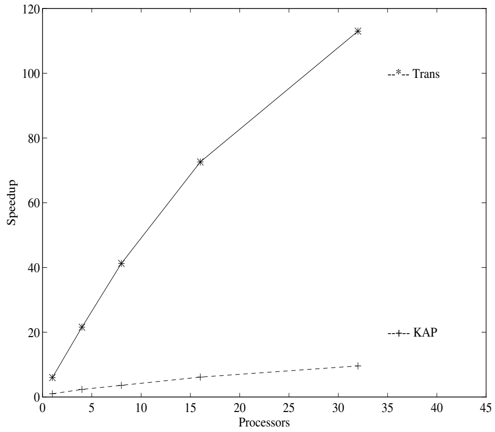
We compare our results using the affinity tiling with the simple tiled version. The results on the HP workstation are in Figure 5 and Figure 6 in which *Source* represents the original version; *Tiling* represents the simple tiling version; *Trans* represents the version with height reduction; and *Affinity Tiling* represents the version with affinity tiling. The performance of simple tiled version varies dramatically, when the tile size changes, while the performance of our version is quite stable, and always outpaces the simple tiling version.

Loop tiling should be applied with care. One problem is that loop tiling is not always profitable. As we see in Figure 6, the trans-



Processors	4	8	16
Original Version (sec.)	86	44	22.06
False sharing elim (sec.)	22.9	11.6	5.96
Speedup	3.76	3.79	3.7
Strength Reduction (sec.)	11.4	5.7	3.0
Speedup	7.54	7.71	7.35

Figure 7: Eliminating False Sharing – banded rank-$2k$ update



Prog/Nodes	4	8	16	32
KAP	249.45	162.54	95.41	60.32
Speedup	2.34	3.6	6.14	9.6
Trans	27.09	14.17	8.05	5.11
Speedup1	21.58	41.27	72.6	113
Speedup2	3.63	6.95	10.88	19.27

Figure 8: Cholesky Decomposition on KSR Multiprocessors

formed version using height reduction has the best performance. Loop tiling actually added some overhead. The second problem is that in the absence of an accurate model to decide the tile size, loop tiling may not give a better performance as expected, and may even give much worse performance. Affinity tiling helps to ease the tile size sensitivity problem for these two examples in Figure 5 and 6.

8.4 Eliminating False Sharing

In rank-$2k$ update, no false sharing exists after the loop transformation, since each process works on different columns of the matrix Z. The improvement over the original parallel version is quite significant. Results from KSR (time unit = second) are shown in Figure 7. The problem size is 500. The speedups of the transformed version over the untransformed version are between 7-8 from 4 nodes to 16 nodes. Notice that strength reduction of the divisions is important.

We studied Cholesky decomposition and compared our result with KAP, a commercial parallelizing compiler on the KSR1. The problem size used is 1000. The original sequential version took 584.81 seconds. Figure 8 has the execution times and speedups of both KAP and our versions. Our version had a speedup of 21.58 on 4 nodes over the original single processor version! The row *Speedup1* in the table contains the speedups over the original single processor version. The reason for the extraordinary speedup is that loop transformations helped exploit data reuse in subcaches and local caches on a single processor as well. The access latency of remote accesses is even longer with network/bus contention caused by false sharing. To see the cache reuse effect on the sequential version, we also ran the transformed version on one node. The execution time of the transformed version is 98.5 seconds, a speedup of about 6 over the original version due to the improved data locality. Then we compute the speedups over the transformed one node version to get the row *Speedup2*.

9 Conclusion

In this paper, we have developed a new data reuse model and an algorithm called *height reduction* to improve cache locality. The advantage of this algorithm is that it can improve banded matrix programs as well as dense matrix programs. With the height reduction algorithm, we have extended loop tiling to exploit not only intra-tile data locality but also inter-tile data locality. We call the new tiling *affinity tiling*. We have showed that the algorithm also helps to eliminate or reduce false sharing in multiprocessor systems. With the height reduction algorithm and affinity tiling, significant performance improvement (speedups from 2.5 to 10) has been observed on HP workstations and KSR1 multiprocessors.

10 Acknowledgment

I would like to thank Keshav Pingali, Tom LeBlanc, Michal Cierniak, Richard Schooler, Bob Gottlieb and the anonymous ICS referees for their helpful comments and insights on the previous versions of this paper.

References

[1] E.L. Allgower. Exact inverse of certain band matrices. *Numer. Math.*, 21:279–84, 1973.

[2] E. Ayguade and J. Torres. Partitioning the statement per iteration space using non-singular matrices. In *Proceedings of The 1993 ACM International Conference on Supercomputing*, July 1993.

[3] U. Banerjee. Unimodular transformations of double loops. In *Proceedings of the Workshop on Advances in Languages and Compilers for Parallel Processing*, pages 192–219, August 1990.

[4] R. Bianchini and T. J. LeBlanc. Software caching on cache-coherent multiprocessors. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, pages 521–526, December 1992.

[5] William J. Bolosky and Michael L. Scott. False sharing and its effect on shared memory performance. In *Proceedings of the Fourth Usenix Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 57–71, San Diego, CA, September 1993. Also available as MSR-TR-93-1, Microsoft Research Laboratory, September 1993.

[6] Z. Bolte. Bounds for rounding errors in the Gaussian elimination for band systems. *J. Inst. Math. Applic.*, 16:133–42, 1975.

[7] S. Carr, K. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, October 1994.

[8] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *Proceedings of SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, California, June 1995.

[9] E. Cuthill. Several strategies for reducing the bandwidth of matrices. In D.J. Rose and R.A. Willoughby, editors, *Sparse Matrices and Their Applications*. Plenum Press, New York, 1972.

[10] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The detection and elimination of useless misses in multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, pages 88–97, San Diego, CA, May 1993.

[11] I.S. Duff. A survey of sparse matrix research. In *Proc. IEEE*, volume 65, pages 500–535, 1977.

[12] S. J. Eggers and T. E. Jeremiassen. Eliminating false sharing. In *Proc. 1991 International Conference on Parallel Processing*, 1991.

[13] C. Eisenbeis, W. Jalby, D. Windheiser, and F. Bodin. A strategy for array management in local memory. In *Proc. 3th Annual Workshop on Languages and Compilers*, August 1990.

[14] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and exchange cache effectiveness. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, August 1991.

[15] 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.

[16] N.E. Gibbs, W.G. Poole, and P.K. Stockmeyer. An algorithm for reducing the bandwidth and profile of a sparse matrix. *SIAM J. Num. Anal.*, 13:236–50, 1976.

[17] N.E. Gibbs, W.G. Poole, and P.K. Stockmeyer. A comparison of several bandwidth and profile reduction algorithms. In *ACM Trans. Math Soft.*, volume 2, pages 322–30, 1976.

[18] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 1989.

[19] E. D. Granston and H. A. G. Wijshoff. Managing pages in shared virtual memory systems: Getting the compiler into the game. In *Proceedings of the 1993 International Conference on Supercomputing*, July 1993.

[20] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[21] W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, Dept. of Computer Science, University of Maryland, College Park, April 1993.

[22] D. Kulkarni, M. Stumm, R. Unrau, and W. Li. A generalized theory of loop transformations. Technical Report CSRI-317, Computer Systems Research Institute, Department of Computer Science, Department of Electrical and Computer Engineering, University of Toronto, December 1994.

[23] W. Li. Compiler optimizations for cache locality and coherence. Technical Report 504, Department of Computer Science, University of Rochester, April 1994.

[24] W. Li and K. Pingali. Access Normalization: Loop restructuring for NUMA compilers. *ACM Transactions on Computer Systems*, 11(4), November 1993. An earlier version appeared in Proc. 5th International Conference on Architectural Support for Programming Languages and Operating Systems, October, 1992.

[25] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *International Journal of Parallel Programming*, 22(2), April 1994. An earlier version appeared in the Proc. 5th Annual Workshop on Languages and Compilers for Parallelism, August, 1992.

[26] R.S. Martin and J.H. Wilkinson. Solution of symmetric and unsymmetric band equations and the calculation of eigenvalues of band matrices. *Numer. Math.*, 9:279–301, 1967.

[27] A. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, May 1989.

[28] V. Sarkar and R. Thekkath. A general framework for iteration-reordering loop transformations. In *SIGPLAN'92 Programming Language and Implementation Conference*, June 1992.

[29] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

[30] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[31] M. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, October 1991.

[32] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman Publishing, London, 1989.