Chapter 5

Transformations for Cache Locality

5.1 Introduction

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.

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. Typically, a cache hit takes only one cycle, while a cache miss takes 8-32 cycles [HP90]. 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. 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*.

A program may have data reuse, but might not be able to exploit cache locality due to the replacement of the data in the cache. Program transformations can improve the performance significantly. The idea is to change the data access sequence so that a data reuse can be translated into a cache hit than a cache miss. One important approach is to use *loop tiling*, invented by Wolfe [Wol89], to block the innermost loop nest so that data reuse within these smaller tiles can be exploited by the cache. Loop tiling can be improved by loop transformations such as loop interchange to bring some outer loops to the innermost position, and then apply loop tiling on the new innermost loops. Wolf and Lam have created a cache locality model, and applied unimodular transformations to choose the best innermost loop nest to tile [WL91a]. Exploiting cache locality has been an active research area. We will discuss more related work in Section 5.6.

Program transformations, such as tiling, have been proven to be very useful in improving cache locality. However, they have been usually applied in an ad hoc way. Wolf and Lam have proposed a cache model based on *reuse vector space*, where tiling can be applied to exploit cache locality. Unimodular transformations are employed to choose an innermost loop nest. They also have showed that it is difficult to choose a tile size for real machines, which usually have direct mapped caches or low associativity caches [LRW91]. The performance can be very erratic, if the block size is not chosen carefully.

In this chapter, we make the following contributions:

- We introduce a new simple cache model based on *reuse distances* that is more precise than the *reuse vector space* model.
- We develop a new loop transformation technique called *height reduction*, which optimizes directly on reuse distances, so that no exhaustive search is necessary.
- We also integrate height reduction with tiling. This algorithm is called *width reduction*, which combines both height reduction and tiling, and the benefit of tiling is retained. Height reduction helps reduce cache interferences when tiling for real machines.

The implementation of the algorithms is also quite simple when used with the *Lambda* loop transformation toolkit [LP93b]. We have observed speedups of 1.7 to 10 performance improvement on programs from linear algebra and NASA benchmark suite.

The rest of the chapter is organized as follows: in Section 5.2, we define a simple cache reuse model, and give a formulation of the problem. The height reduction algorithm is described in Sections 5.3, and the width reduction algorithm is presented in Section 5.4. Empirical evaluations were conducted on an HP 9000/720 workstation, and the results are reported in Section 5.5. Finally, we discuss related work in Section 5.6.

5.2 A Simple Cache Reuse Model

A program must have data reuse in order to exploit cache locality. For example, consider the program in Figure 5.1. The data element *A[i, j]* is used in both iterations (i, j) and $(i + 1, j - 1)$.

```
for i = 1, n
  for j = 1, n
     = A[i, j] + A[i-1, j+1]
  end for
end for
```

Figure 5.1: A Simple Example

5.2.1 Reuse Vectors

We can define data reuse concisely using an integer vector that represents two loop iterations that access the same data location. This vector is called *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.

Definition 5.2.1 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 in Figure 5.1 is $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$.

An important attribute of a reuse vector is the *height*.

Definition 5.2.2 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.

Definition 5.2.3 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 $n - i + 1$ [1].

The data reuse vector can be computed by solving a system of linear equations [GJG88,WL91a]. There is a reuse between $A[i, j]$ and $A[i-1, j+1]$, if for two iterations (i_1, j_1) and (i_2, j_2), $A[i_1, j_1]$ and $A[i_2 - 1, j_2 + 1]$ reference the same data. Therefore, the system of equations is as follows:

$$i_1 = i_2 - 1$$

[1] The outermost loop is loop 1.

$$j_1 = j_2 + 1$$

Let $r = \begin{pmatrix} i_2 \\ j_2 \end{pmatrix} - \begin{pmatrix} i_1 \\ j_1 \end{pmatrix}$ be the reuse vector. The solution is $r = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$. We can rewrite the equations using matrix notation.

$$A_1 K_1 = A_2 K_2 + c_2$$

where $A_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $A_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $c_2 = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$, $K_1 = \begin{pmatrix} i_1 \\ j_1 \end{pmatrix}$, and $K_2 = \begin{pmatrix} i_2 \\ j_2 \end{pmatrix}$.

The first row of A_1 is the coefficients of the subscript $1 * i + 0 * j$ in the first dimension of $A[i, j]$, the second row of A_1 is from the second subscript. A_2 is computed in a similar way as A_1. c_2 is the constant vector, where the first element is from the constant in the first dimension of $A[i + 1, j - 1]$, and the second element is from the second dimension.

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 $R[A\vec{i} + a]$, where $A = (a_{ij})$. There is temporal reuse between references $A\vec{i} + a_1$ and $A\vec{i} + a_2$ iff there exist iterations $\vec{i}_1$ and $\vec{i}_2$ such that

$$A\vec{i}_1 + a_1 = A\vec{i}_2 + a_2.$$

Then, the reuse vector $\vec{r} = \vec{i}_2 - \vec{i}_1$ is the solution to the following equation.

$$A\vec{r} = a_1 - a_2 \tag{5.1}$$

Any reuse vector is a linear combination of a basis of the null space of A plus a special solution to the system.

$$\vec{r} = a_1\vec{r}_1 + ... + a_k\vec{r}_k + \vec{r}_0 \tag{5.2}$$

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 5.1.

For spatial locality, 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.

```
for i = 1, n                    for i = 1, n
  for j = 1, n                    for j = 1, 5
    = A[j]                          = A[j]
  end for                         end for
end for                         end for
     (a)                             (b)
```

Figure 5.2: Reuse Distances

$$\begin{aligned} \overline{A}\vec{d} &= \overline{b} \\ |r_1^A\vec{d} - b_1| &< s \end{aligned}$$

where $\overline{A}$ is A with first row deleted, $\overline{b}$ is b with the first element deleted, r_1^A is the first row of A, b_1 is the first element of b, and s is the size of a cache line.

5.2.2 Reuse Distance

Reuse vectors are simple, but they are not sufficient for predicting cache locality. Consider the examples in Figure 5.2. Both have the same reuse vector $(1, 0)^T$. If n is much larger than the cache size, program (a) will have less chance to reuse the data in the cache. On the other hand, program (b) will almost certainly have cache reuse.

Therefore, the trip count of loops should be included in our model. Formally, we define the *width* of a loop.

Definition 5.2.4 Let *width* of loop k be the trip count of loop k.

Definition 5.2.5 The *reuse distance* is defined to be the number of loop iterations between two reuse iterations.

Given r the reuse vector, the reuse distance d can be computed easily.

$$d = r_1 S_1 + ... + r_n S_n$$

where $S_i = \prod_{k=i+1}^{n} w_k$, and w_k is the width of loop k.

For example, the reuse distance for program (a) in Figure 5.2 is n, and for program (b) the distance is 5.

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

Lemma 5.2.1 *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. From the definition of reuse distance and Lemma 5.2.1, we see there are two ways of achieving that. First, we can introduce more leading zeros in the reuse vector, since $\{ S_1, ..., S_n \}$ is a strictly decreasing sequence. Second, we can reduce the width of a loop. The first optimization should be more powerful, since S_i is usually much larger than $S_{(i+1)}$.

Then the optimization problem for cache locality can be decomposed into two subproblems.

- Height Reduction: lowering the loop that carries the reuse.
- Width Reduction: decreasing loop trip counts.

5.2.3 Data Reuse Matrix

To make the optimization on reuse vectors more convenient, we introduce another important data structure called the *data reuse matrix*, similar to the data access 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 in equation 5.2. A data reuse matrix is a matrix that has the basis vectors and the special solution as its columns. Note that all reuses, *self, group, temporal* and *spatial*, defined in [WL91a] are included in the reuse matrix. In other words, the reuse matrix for the reuse computed from equation 5.2 is as follows:

$$R = (\vec{r}_1, ..., \vec{r}_k, \vec{r}_0)$$

In general, there are many individual reuse matrice (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 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.

In the rest of the chapter, we show loop transformations can be used to achieve height reduction and width reduction. Since our loop transformations are linear, the transformed reuse vector can be easily computed.

Lemma 5.2.2 *If T is a linear transformation and $\vec{r}$ is a reuse vector, then $T\vec{r}$ is the new reuse vector.*

Theorem 5.2.1 *If T is a non-singular transformation and R is a reuse matrix, then the new reuse matrix is TR.*

Proof:

Consider any reuse vector r.

$$\vec{r} = a_1\vec{r}_1 + ... + a_k\vec{r}_k + \vec{r}_0$$

Then

$$T\vec{r} = a_1T\vec{r}_1 + ... + a_kT\vec{r}_k + T\vec{r}_0.$$

Since T is non-singular, $T\vec{r}_1, ..., T\vec{r}_k$ are linearly independent of each other.
□

5.2.4 Discussion

To see the difference with the *reuse vector space* model, we consider the example in Figure 5.3. In version (a), the reuse vector for the reference $A[i,j]$ is $(1,0)^T$ (spatial reuse), the reuse vectors for $A[i,k]$ are $(0,1)^T$ (temporal reuse) and $(1,0)^T$ (spatial reuse), and the reuse vectors for $A[j,k]$ are $(1,0)^T$ (temporal reuse) and $(0,1)^T$ (spatial reuse). The Wolf/Lam model would conclude that the reuse vector space is the whole space ij, since both i-loop and j-loop carry reuse. Similarly, the reuse vector space for version (b) is also the whole space ij. Therefore, the Wolf/Lam model does not distinguish between these two versions. In summary, the cost function computed using the Wolf/Lam model is a function of the *dimension* of the reuse vector space, i.e. the *number* of loops that carry reuse. Therefore, the order of loops is irrelevant as long as the loops carry reuse. If data dependences prevent a subset to become the innermost loop nest, unimodular transformations are applied so that the innermost loop nest may be tilable.

As our experimental results in Section 5.6 show, version (b) performs much better than version (a). In our *reuse distance* model, the reuse vector $(1,0)^T$ is regarded as the important vector to be optimized, since it occurs much more frequently than the other one, and is the basis vector for the intersection of all reuses. Version (b) will be the optimized version generated by the algorithms presented in the rest of the chapter.

We describe the height reduction algorithm in Section 5.3, and the width reduction algorithm in Section 5.4.

```
for i = k+1, n
  for j = k+1, i
    A(i,j) = A(i,j) - A(i,k) * A(j,k)
  end for
end for
```

(a)

```
for j = k+1, n
  for i = max(j, k+1), n
    A(i,j) = A(i,j) - A(i,k) * A(j,k)
  end for
end for
```

(b)

Figure 5.3: Comparison of Models

5.3 Height Reduction

Given a data reuse matrix, the goal of height reduction is to find the minimum s such that rows 1 to $n-s$ of the data reuse matrix are all zeros. When we construct a legal transformation, we need to make sure that the transformation is non-singular, and does not violate data dependences.

Let P^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).

5.3.1 Non-singularity

First, we satisfy the condition of non-singularity. To construct the first row of the transformation so that the first row of the new reuse matrix is 0, from Theorem 5.2.1, we need to find a vector in the null space of the old reuse matrix. Suppose we have formed a matrix P^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 P^T, and
2. v is in the null space of R.
3. v does not violate D, the data dependence matrix.

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 $R = \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix}$. To compute the first row of the transformation matrix, we can use the vector $(0,0,1)^T$, which is an element of the null space of R. Now the partial transformation P^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.

These three conditions can be satisfied using more efficient algorithms. For example, instead of finding a linear independent vector v, we can find a vector from the null space of P^T. It is easy to compute a vector from the null space of both P^T and R. Let Q be the column union of P^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: $v = a_1x_1 + ... + a_tx_t$.

If Q spans the whole space, then some reuse vectors must be deleted before any row can be added to P^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.

Now we show that finding a vector from the null space of P^T is *not* less general than finding a vector that is linearly independent of P^T.

Theorem 5.3.1 *If there exists a linearly independent vector v_1 of P^T that is in the null space of R and does not violate D, then there is an orthogonal vector v of P^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 P^T by the projector J (e.g. $J = I - P(P^TP)^{-1}P^T$). Since $v_1^TR = 0$, $v^TR = (Jv_1)^TR = v_1^TJ^TR$. Notice that J is a symetric matrix, and R is already in the orthogonal space of P^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 $v_1^TD \geq 0$. □

5.3.2 Dependences

Now, we consider data dependences. Any row added must not violate data dependences. Any dependence carried by the existing rows can be deleted from the dependence matrix. Since $v = a_1x_1 + ... + a_tx_t$, we need to choose a set of values for $\{a_1, ..., a_t\}$ 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

$$v^Td = a_1x_1^Td + ... + a_tx_t^Td.$$

To decide the values of each a_i, we use the following rules. Let *dir* be x_i^Td computed using the dependence algebra defined in Section 3.5.1.

- if dir = "<" then $a_i = 1$

- if dir = ">" then $a_i = -1$
- if dir = "=" then a_i = any value
- if dir = "<=" then $a_i = 1$
- if dir = ">=" then $a_i = -1$
- if dir = "<>" then $a_i = 0$
- if dir = "$*$" then $a_i = 0$

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. For the special cases where the dependence vector is a distance vector, these rules can be further optimized.

5.3.3 Algorithm

The algorithm is shown in Figure 5.4. Let P^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). The following invariants are maintained during the construction: $P^T D = 0$ and $P^T R = 0$.

We start from an empty transformation matrix. The notation P^T denotes a row matrix. A new matrix Q^T is created by simply combining the rows of P^T and R^T. If Q^T spans the whole space, then its null space is empty. Some lower priority reuse vectors have to be deleted until Q^T does not span the whole space. Once a solution, which consists of a set of all valid vectors, for the null space of Q^T is found, we need to choose a vector that satisfies the data dependences as discussed in the previous section.

For example, we consider the loop nest in Figure 5.3(a) again. The reuse matrix is $R = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. The first column $(1, 0)^T$ is the intersection of all three reuse vecter sets computed from references A[i,j], A[i,k] and A[j,k] as shown in Section 5.2.4. Therefore the second column is dropped while computing the first row of the transformation. The partial transformation is $P^T = (0, 1)$. The completion algorithm is then called to complete the partial transformation to get the full transformation $T = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. The transformed version is in Figure 5.3(b).

Input: *A reuse matrix R, a dependence matrix D and dimension n.*
Output: *A legal transformation matrix T that maximizes cache reuse.*
Algorithm HeightReduction(R, D, n) : Matrix
begin
  $P^T = \emptyset$; $s = 0$;
  While (($s < n$) *and* (*R not empty*) *do*
    $Q^T = union(P^T, R^T)$;
    While ($rank(Q) = n$) *do*
      delete the lowest priority reuse vector from R
    solve $Q^T v = 0$, *let* $v = a_1 x_1 + ... a_t x_t$
    choose $a_1, ..., a_t$ *such that* $v^T D = a_1 x_1^T D + .. + a_t x_t^T D \geq 0$.
    if success then
      $D = D - d$, *if d is a dependence vector and* $v^T d > 0$
      $P_s^T = v^T$; $s = s + 1$;
    else
      delete the reuse vector with the lowest priority from R
    end if
  End-While
  return(P^T);
end

Figure 5.4: Height Reduction

```
for i = 1, n                    for t_i = 0, nb-1
  for j = 1, n                    for t_i = 0, nb-1
    A[i, j] = A[i-1, j-1]           for i = t_i * b + 1, min(n, (t_i + 1) * b - 1)
                                      for j = t_i*b + 1, min(n, (t_i + 1) * b - 1)
                                        A[i, j] = A[i-1, j-1]
    (a) Source Code                        (b) Tiled
```

Figure 5.5: Loop Tiling

5.4 Width Reduction

Loop width can be reduced using *loop tiling* [Wol89]. For example, the reuse vectors for the loop nest in Figure 5.5 are of the form of $a * \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. The reuse matrix spans the whole space. The loop nest in (b) is the tiled version of the loop nest in (a), where b is the block size, and nb is the number of blocks computed from b and n. Note that extra "tile" loops have been introduced. This adds some overhead to the execution time.

Loop tiling is not always legal. If we replace the statement in Figure 5.5(a) by *A[i,j] = A[i-1,j+1]*, then the dependence vector is $(1, -1)^T$, which prevents 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. This is the simplified case of the *fully permutable* loop nest defined by Wolf and Lam [WL91a]. 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 algorithm in Figure 5.6 constructs a transformation matrix that reduces the height of the reuse vectors, and creates opportunity 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 P 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

Input: *A reuse matrix R, a dependence matrix D and dimension n.*
Output: *A legal transformation matrix T that maximizes data reuse.*
Algorithm WidthReduction(R, D, n) : Matrix
begin
  $k = 1$; $P^T = \emptyset$; $s = 0$;
  While $(s < n)$ *do*
    $Q^T = union(P^T, R^T)$;
    While $(rank(Q) = n)$ *delete the lowest priority reuse vector from R*
    solve $Q^T v = 0$, *let* $v = a_1 x_1 + ... a_t x_t$
    choose $a_1, ..., a_t$ *such that* $v^T D = a_1 x_1^T D + .. + a_t x_t^T D \geq 0$,
      and try to optimize for $v^T D_o \geq 0$.
    if success then
      For $((d \in D_o)$ *and* $(v^T d < 0))$ $k = max(k, r+1)$, *if loop* i_r *carries d.*
      For $((d \in D)$ *and* $(v^T d > 0))$ $D = D - d$; $D_o = D_o + d$
      For $((d \in D_o)$ *and d carried by loops* $i_1, .., i_{(k-1)})$ $D_o = D_o - d$
      $P_s^T = v^T$; $s = s + 1$;
    else delete the reuse vector with the lowest priority from R
  End-While
  return(P^T);
end

Figure 5.6: Width Reduction

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 $i_k, .., i_{(s-1)}$.

5.4.1 Tile Size and Cache Interferences

Another important issue is the block size. A naive approach is to choose a block size large enough that the block of data fits in the cache. This has been shown to cause very poor performance on real machines, which usually have direct mapped cache, or have caches with very small set associativity (2 or 4) [LRW91]. Performance can be erratic when tile size changes, and it is difficult to predict the optimal tile size. The main reason for performance degradation is cache interference, i.e. two elements are mapped to the same cache location. Width reduction also helps eliminate the erratic cache behavior because height reduction is performed as a part of width reduction. Experimental results in

Section 5.5.5 show that performance changes dramatically if tiling is directly applied to Cholesky decomposition, while tiling produces good performance independent of tile size after height reduction.

5.5 Experiments on Uniprocessor with Caches: HP 9000

5.5.1 The Machine Architecture

We conducted our experiments on an HP 9000/720, which 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 *Pnuma*, and then compiled with the native FORTRAN compiler with the -O option.

We first show the results of banded SYR2K and the NASA benchmark from the SPEC benchmark suite, and then study the effect of data sets and loop tiling.

5.5.2 Banded SYR2K Results

We use banded SYR2K to show that our techniques can handle quite general data access patterns. We include the original program in Figure 5.7(a) for convenience. First, we compute the reuse matrix, shown in Figure 5.7(c), using the algorithm in Section 5.2. 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 5.7(d), from that reuse matrix. Given the transformation matrix, we use the completion algorithm to generate the new loop nest, shown in Figure 5.7(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.

The execution times for the different versions are shown in Figure 5.7(e). The problem size n is 500. The time for the original version is 101.3 seconds. After

```
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((-i-j)/2+k, i+1) = Z((-i-j)/2+k, i+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 5.7: Banded SYR2K

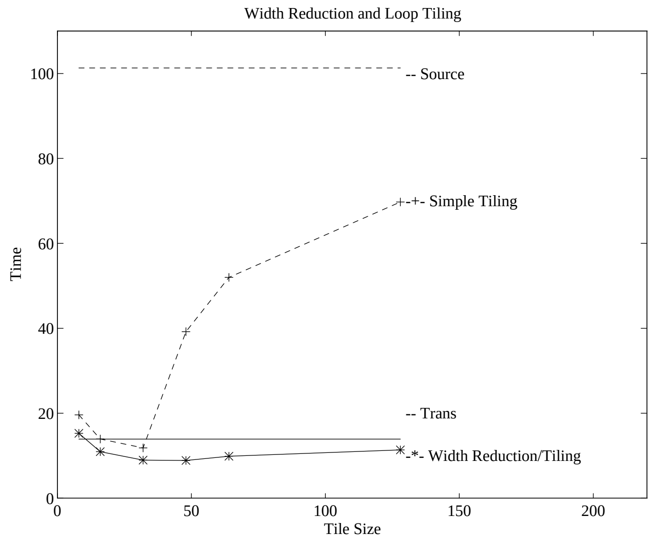

Figure 5.8: Width Reduction- banded rank-$2k$ Update

height reduction transformations, the time is reduced to 14.36 seconds, which has the speedup of 7.05. Strength reduction of the divisions in the program has not changed the performance.

We compare our results using the width reduction with the simple tiled version. The results are in Figure 5.8 in which *Source* represents the original version; *Simple Tiling* represents the simple tiling version; *Trans* represents the version with height reduction; and *Width Reduction/Tiling* represents the tiling version with width reduction. 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.

5.5.3 NASA Benchmark Results

The NASA benchmark contains 7 floating point intensive programs written in Fortran 77. For each kernel, the program generates its own input data, performs the kernel and compares the result against an expected result.

A brief description of the programs and major data structures is list below.

- MXM: matrix multiply.

  The major data structures are three 2-dimensional arrays of sizes 256×128, 128×64, and 256×64.

- FFT: complex radix 2 FFT on 2D array.

  The major data structure is a 2-dimensional array of size 128×256.

- CHOLSKY: Cholesky decomposition on a set of input matrices. It also performs backward substitution.

  The major data structures are two 3-dimensional arrays of sizes $250 \times 4 \times 40$ and $3 \times 250 \times 40$.

- BTRIX: Block tridiagonal matrix solution along one dimension of a four dimensional array.

  The major data structures are a 4-dimensional array of sizes $30 \times 30 \times 30 \times 5$ and three 4-dimensional arrays of size $5 \times 5 \times 30 \times 30$.

- GMTRY: Sets up arrays for a vortex method solution and performs Gaussian elimination on the resulting arrays.

  The major data structures are one 2-dimensional array of size 500×500, and three 2-dimensional arrays of size 100×5.

- EMIT: Creates new vortices according to certain boundary conditions.

  The major data structures are one 2-dimensional array of size 500×500, and four 2-dimensional arrays of size 100×5.

- VPENTA: inverts 3 pentadiagonal matrices.

  The major data structures are two 3-dimensional arrays of size $128 \times 128 \times 3$, and seven 2-dimensional arrays of size 128×128.

We applied the height reduction algorithm to these programs, and Table 5.1 contains the performance results. No loop transformations have been applied to FFT, GMTRY and EMIT. There are a few reasons for this: there are data dependences that prevent loop transformation, the loops are imperfectly nested, or the loops are already in the right form for cache locality.

For MXM and CHOLSKY, transformations are applied, but no improvement are seen. This is because that the test data set is so small that they almost fit in the caches [2], and furthermore MXM has been hand-optimized for locality by unrolling the outermost loop 4 times. We will study the effect of the size of the data sets in Section 5.5.4.

For BTRIX and VPENTA, since both have arrays that are larger than the data cache, we see improvement from our transformations. If the data sets were even larger, we would expect to see much better improvement.

5.5.4 The Effect of Data Sets

In this section, we study matrix multiplication and Cholesky decomposition with larger data sets.

[2] Recall that the size of the data cache on the HP 9000/720 is 64K words.

Table 5.1: NASA benchmark on HP 9000/720

Prog	MXM	FFT	CHO	BTRIX	GMTRY	EMIT	VPENTA
Source	14.11	93.05	40.61	55.78	111.34	15.80	113.50
New	14.24	N/A	40.70	34.37	N/A	N/A	99.52
Speedup	1		1	1.64			1.12

A natural way of writing matrix multiplication is the simple *ijk* form in Figure 5.9(a). However, this version is not suitable for cache locality, because, for example, the spatial locality of neither C nor A is exploited, and there is only temporal locality of C and spatial locality of B. The NASA kernel chooses the *kji* form, and then unrolls the k-loop 4 times. Figure 5.9 gives the execution times for the problem size of 750.

Our height reduction algorithm will transform both versions to the new version in Figure 5.9. In the transformed version, the spatial locality of C is exploited by the innermost loop, the temporal locality is carried by the second loop. References A and B have spatial locality and temporal locality respectively. As expected, the new version performs significantly better, requiring only 68.65 seconds. Compared to the *ijk* version, the speedup is 5.88; compared to the *kji* version, the speedup is 10.

Another example is the Cholesky Decomposition, shown in Figure 5.10(a). 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 250×250 is about the largest that can fit in the cache. The original code has poor locality, since the inner loop walks across the columns, 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, and the new version is shown in Figure 5.10(b).

The results are listed in Figure 5.10(c). 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 speedup of 2. The improvement is significant when the size is larger than 1000. We get speedup of 5 over the original version.

5.5.5 The Effect of Tiling

We use Cholesky decomposition as the example to illustrate the effect of tiling. If we apply tiling directly on the ij-nest in Figure 5.10(a), we get the results in Table 5.2 for a problem of size 1000.(The original version took 196.05 seconds.) The performance varies greatly, when the block size changes. Although tiling is an important optimization, it is very difficult to predict the optimal block

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

(a) *ijk*

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

(b) *kji*

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

(c) new

Versions	*ijk*	*kji*
Source	379.69	708.37
Transformed	68.65	68.65
Speedup	5.88	10

(d) Execution Times (Size = 750)

Figure 5.9: Matrix Multiplication

```
for k = 1, n
  A(k,k) = sqrt(A(k,k))
  for i = k+1, n
    A(i,k) = A(i,k)/A(k,k)
  end for
  for i = k+1, n
    for j = k+1, i
      A(i,j) = A(i,j)-A(i,k)*A(j,k)
    end for
  end for
end for
```

(a) Source

```
for k = 1, n
  A(k,k) = sqrt(A(k,k))
  for i = k+1, n
    A(i,k) = A(i,k)/A(k,k)
  end for
  for u = k+1, n
    for v = max(u, k+1), n
      A(v,u) = A(v,u)-A(v,k)*A(u,k)
    end for
  end for
end for
```

(b) Transformed

Size	250	500	750	1000	1250	1500	1750	2000
Source	0.61	11.71	63.41	196.05	453.25	704.65	1161.74	1748.19
Trans	0.59	5.74	19.17	46.49	91.12	157.29	249.65	373.15
Speedup	1.03	2	3.33	4.35	5	4.55	4.55	4.55

(c) Execution Times

Figure 5.10: Cholesky Decomposition

Table 5.2: Loop Tiling (Size 1000)

Block Size	16	32	64	100	200
Times (sec.)	60.77	56.15	55.74	175.08	184.85

Table 5.3: Loop Tiling after Height Reduction (Size 1000)

Block Size	16	32	64	100	200
Times (sec.)	52.78	50.12	48.16	49.09	47.89

size [LRW91]. If a random block size is chosen, the performance may be very poor.

In our approach, we apply height reduction first. The reuse vector that has the most number of occurrences is $(1, 0)^T$, which is computed from all of the four references. The execution time of the untiled transformed version is 46.49 seconds (Figure 5.10(c)), about more than 20% improvement over the tiled untransformed version. The actual percentage depends on the choice of the block size. If we apply tiling after height reduction, the block size is not a dramatic factor causing big performance differences. Therefore, this optimization is more stable, since it does not depend on the cache mapping, cache size and problem size. There is some performance degradation due to the extra overhead of tiling. The results are shown in Table 5.3.

5.6 Discussion and Related Work

In this section, we discuss related work.

Gannon, Jalby and Gallivan [GJG88] introduced the notion of *uniformly generated* data reuse. Porterfield [Por89] studied the problem of estimating the number of cache lines for uniprocessor machines, when the cache line size is 1. The techniques by Ferrante, Sarkar, and Thrash [FST91] estimate the 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.

Wolf and Lam [WL91a] 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 cost function computed using the reuse vector space model is a function of the *dimension* of the reuse vector space, i.e. *number* of loops that carry reuse.

Therefore, the order of loops is irrelevant as long as the loops carry reuse. If data dependences prevent a subset to become the innermost loop nest, unimodular transformations are applied so that the inner most loop nest may be tilable. For example, the reuse vector space for the ij-nest in the Cholesky decomposition in Figure 5.10(a) is the whole space that include both i and j dimensions, which has the maximum reuse (the maximum reuse dimension is 2, in the 2-dimensional space), according to the reuse vector space model, and the subset that includes loop i and loop j is tilable. Therefore, loop tiling is performed on the loop nest ij. The experimental results are reported in Table 5.2.

Our approach is based on reuse vector and reuse distance (we refer to this as the *reuse distance model*), rather than the dimension of the reuse space. Therefore, the order of loops does make a difference in our model. It takes the reduction of reuse vector height as the primary optimization. In the Cholesky example, by reducing the reuse vector $(1, 0)^T$, which is the basis vector of the intersection of all reuses, we get the new version in Figure 5.10(b), which would have the same cost function as the original version under the reuse space cost model. Therefore, our *reuse distance model* with the reuse vectors and reuse distances improves upon the reuse vector space model, and is a more precise than the reuse space model. This allows us to develop algorithms to get better performance. The experimental results are shown in Table 5.3, which shows that our approach generates the better results. The second advantage is that since the transformations are operated directly on the reuse vectors, no exhaustive search is necessary. The third advantage is that the performance of the generated code is much less dependent on the tile size. Another advantage of the reuse distance model is that we can represent more complicated data reuses such as the data access patterns in the banded SYR2K example in Figure 5.7. For the reuse vector $(1, 1, 0)$, the reuse space model will make an approximation by having both loops carry reuse, but we can optimize on it directly without losing the precision.