Published in the International Journal of Parallel Programming, 26(6):671--696, 1998.

Reuse-Driven Tiling for Improving Data Locality

Jingling Xue† and Chua-Huang Huang‡

School of Mathematical and Computer Sciences
University of New England, Armidale, NSW 2351, Australia†
Department of Computer Science and Information Engineering,
National Dong Hwa University, Hualien, Taiwan, ROC‡

Abstract. This paper applies unimodular transformations and tiling to improve data locality of a loop nest. Due to data dependences and reuse information, not all dimensions of the iteration space will and can be tiled. By using cones to represent data dependences and vector spaces to quantify data reuse in the program, a reuse-driven transformational approach is presented, which aims at maximising the amount of data reuse carried in the tiled dimensions of the iteration space while keeping the number of tiled dimensions to a minimum (to reduce loop control overhead). In the special case of one single fully permutable loop nest, an algorithm is presented that tiles the program optimally so that all data reuse is carried in the tiled dimensions. In the general case of multiple fully permutable loop nests, data dependences can prevent all data reuse to be carried in the tiled dimensions. An algorithm is presented that aims at localising data reuse in the tiled dimensions so that the reuse space localised has the largest dimensionality possible.

Key words: Tiling, Loop Transformation, Data Locality, Nested Loops.

1 Introduction

This paper applies unimodular transformations and tiling to tile a perfect loop nest to improve data locality of the loop nest. Due to data dependences and reuse information, not all dimensions of the iteration space will and can be tiled. In general, the tiled program consists of a sequence of loops that iterate over the untiled dimensions followed by a sequence of loops that iterate over the tiled dimensions, improving untilisation of a single cache. The vector space spanned by the tiled dimensions is called the *tile space* of the program. Only the data reuse localised in the tile space can be exploited. The *data locality problem* addressed in this paper is to maximise the amount of data reuse localised in the tile space while minimising the dimensionality of the tile space. By minimising the dimensionality of the tile space, the number of tiled dimensions is kept to a minimum, and consequently, loop control overhead is reduced.

An example is used to illustrate the data locality problem addressed and the approach used.

EXAMPLE 1 Consider a triple loop nest:

```
do i = 1, N
   do j = 1, N
      do k = 1, N
         A(i - k, k) = A(i - k, k) + 1
```

Wolfe's Tiny [15] reports the following dependence matrix:

$$D = \begin{bmatrix} 0 \\ + \\ 0 \end{bmatrix}$$

To avoid arithmetic on direction values, D will be represented equivalently by cone$((0,1,0))$. Both representations describe the same set of distance vectors: $\{(0,1,0),(0,2,0),\cdots\}$.

How do we tile the program so that all data reuse is localised in the tile space? Since the entries in the dependence vector $(0,+,0)$ are all nonnegative, a straightforward solution is to tile all three dimensions of the iteration space. The tile space is three-dimensional and will include all data reuse in the program. However, an analysis of the data reuse in the program reveals that tiling two dimensions of the iteration space suffices to capture all data reuse in the cache optimally.

Suppose all arrays are stored in row-major order. Each reference $A(i-k,k)$ has self-temporal reuse in the space span$\{(0,1,0)\}$ because it accesses the same element for all iterations (i,j,k), where i and k are fixed and $1 \leqslant j \leqslant N$. In addition, $A(i-k,k)$ has self-spatial reuse in the space span$\{(0,1,0),(1,0,1)\}$ since each cache line is reused ℓ times, where ℓ is the cache line size. Observe, for example, that $A(2,3), A(2,4), A(2,5),\cdots$ are accessed at the iterations $(5,*,3),(6,*,4),(7,*,5),\cdots$, where the $*$'s denote any values in the range of the j loop. In this program, the data reuse is summarised by the following reuse matrix:

$$R = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The vector space spanned by the column vectors is called the *reuse space* of the program, which includes all data reuse — both temporal and spatial — of the program.

Based on the data dependences and reuse information of the program, we want to find a unimodular transformation to restructure the program so that in the transformed program:

1. The data dependences of the program are respected;
2. The inner two loops are fully permutable and can thus be tiled legally; and
3. As much data reuse as possible is carried in the inner two loops. (In this example, the reuse space coincides with the tile space.)

The following unimodular transformation satisfies all these three requirements:

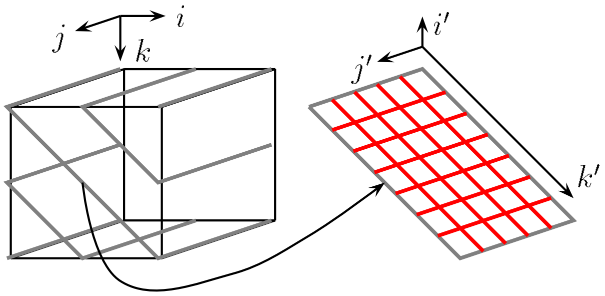
$$\begin{bmatrix} i' \\ j' \\ k' \end{bmatrix} = H \begin{bmatrix} i \\ j \\ k \end{bmatrix}, \quad \text{where } H = \begin{bmatrix} 1 & 0 & -1 \\ \hdashline 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

The tile space is the vector space spanned by the last two columns of H^{-1}, which coincides with the reuse space R of the program. The number of tiled dimensions is, thus, two.

The transformed program can be obtained as follows:

```
do i' = 1 - N, N - 1
   do j' = 1, N
      do k' = max(1, i' + 1), min(N, i' + N)
         A(i', k') = A(i', k') + 1
```

To exploit the two-dimensional data reuse, the inner two loops j' and k' can be tiled:



- **Tiling**: $H = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$
- **Reuse Space**: span$\{(0,1,0),(1,0,1)\}$
- **Tile Space**: span$\{(0,1,0),(1,0,1)\}$
- **Tile Shape**: the edges are $(0,1,0)$ and $(1,0,1)$
- **Tile Size:** $B_{j'} \times B_{k'}$

Figure 1: Iteration space tiling for Example 1.

do $i' = 1 - N,\ N - 1$
    do $j'j' = 1,\ N,\ B_{j'}$
        do $k'k' = \max(1, i'+1),\ \min(N, i'+N),\ B_{k'}$
            do $j' = j'j',\ \min(j'j' + B_{j'} - 1, N)$
                do $k' = k'k',\ \min(k'k' + B_{k'} - 1, N, i' + N)$
                    $A(i', k') = A(i', k') + 1$

The size of tiles is $B_{j'} \times B_{k'}$, which has to be fixed as machine-dependent parameters.

Figure 1 depicts the cubic iteration space of the original program, which are sliced into parallelograms parallel to the reuse space of the program. The largest slice is picked up to illustrate how a slice is tiled. Each slice is divided into tiles that are parallelograms whose edges are parallel to $(0,1,0)$ and $(1,0,1)$, i.e., the last two columns of H^{-1}. In the tiled program, the i' loop steps through all the slices and the $j'j'$ and $k'k'$ loops step between the tiles in each slice, and the j' and k' loops enumerate the points within a tile.

This example has outlined our approach to improving data locality of a program:

(a) Based on data dependences and reuse information, a (legal) unimodular transformation is found and applied to the program to maximise the amount of data reuse carried in the inner τ loops of the transformed program with τ made as small as possible. The inner τ loops of the transformed program — being fully permutable — are tiled so that all data reuse carried in these loops is exploited in the cache. Maximising the amount of data reuse tends to maximise the amount of data locality. Minimising τ minimises loop control overhead.

(b) The size of tiles is adjusted as a machine-specific optimisation.

This paper focuses on (a), improving and extending Wolf and Lam's earlier data locality work [12]. A detailed comparison with that and other related work is provided in Section 8. We use cones to represent the data dependences and vector spaces to quantify the data reuse in the program. This combination allows us to use matrix transformations to solve our data locality problem. In the special case of one single fully permutable loop nest, an algorithm is given that tiles the program optimally so that all data reuse is localised in the tile space and can thus be exploited in the cache. In the general case of multiple fully permutable loop nests, data dependences can prevent all data reuse to be localised in the tile space. The algorithm proposed in this general case aims at localising data reuse in the tile space so that the reuse space localised has the largest dimensionality possible.

The rest of this paper is organised as follows. Section 2 introduces the basic terminology and definitions. Section 3 describes some background information. In particular, the concept of time cone is introduced and an algorithm for solving the so-called time-cone problem is presented. This algorithm is the basis of several algorithms discussed in the paper. Section 4 presents an algorithm for creating the canonical form of fully permutable loop nests for a loop nest. Section 5 defines loop tiling considered in the paper. Section 6 reviews the reuse analysis framework due to Wolf and Lam [12]. Section 7 describes our approach to improving data locality of the program. Section 8 discusses the related work, and finally the paper is concluded in Section 9.

2 Notation and Terminology

$\mathbb{Z}$ and $\mathbb{Q}$ denote the set of integers and rationals, respectively. All relational operators, such as $\geqslant$, on two vectors are component-wise. The dimensions of vectors and whether they are row or column vectors are implied by the context in which they are used. We use $e_1, \cdots, e_n$ to represent the n *elementary vectors*, where e_k is the vector whose entries are all zeros except that the k-th entry is 1. A square integer matrix is *unimodular* if its determinant is ± 1. If A is a matrix, ker(A) stands for the set $\{x \mid Ax = 0\}$ and A^T its *transpose*. If $x_1, \cdots, x_m$ are vectors in $\mathbb{Z}^n$, span$\{x_1, \cdots, x_m\}$ is the linear space spanned by these vectors. If S is a set of vectors, span$\{S\}$ denotes the linear space spanned by all vectors in S. If x is an element of a set S, the notation $x \in S$ is used, and this notation is abused to indicate that a column vector x is a column of a matrix M, i.e., $x \in M$. If L is a linear space, $L^{\perp}$ represents its corresponding orthogonal linear space.

For the purposes of this paper, the concept of Hermite normal form is defined as follows.

DEFINITION 1 (HERMITE NORMAL FORM) Let $A \in \mathbb{Z}^{n\times n}$ be a nonsingular square matrix. Then there exists a unimodular matrix $U \in \mathbb{Z}^{n\times n}$ such that $UA = H$, where $H \in \mathbb{Z}^{n\times n}$ is a nonnegative nonsingular lower triangular matrix, called the *Hermite normal form*, of A.

Let S be a set of vectors. The notation matrix(S) stands for an arbitrary but fixed matrix formed with all vectors in S as its rows. For example, if $S = \{(1, 2, 0), (0, 1, 0)\}$, then matrix$(S)$ is either $\left[\begin{smallmatrix}1&2&0\\0&1&0\end{smallmatrix}\right]$ or $\left[\begin{smallmatrix}0&1&0\\1&2&0\end{smallmatrix}\right]$.

3 Background

Section 3.1 introduces some relevant results about perfectly nested loops. Section 3.2 explains how to use cones to represent data dependences. Section 3.3 discusses the time cone and contains an algorithm for solving the so-called time cone problem.

3.1 Perfectly Nested Loops

This paper considers perfectly nested loops with dependences represented as direction or distance vectors. A dependence vector for an n-deep loop nest is an n-vector $d = (d_1, \cdots, d_n)$, where the k-th entry d_k corresponds to the k-th loop (counting from the outermost to the innermost). Each entry d_k can be either an integer in $\mathbb{Z}$ or a *direction value* in $\{+, -, \pm\}$, where '$+$', '$-$' and '$\pm$' are Wolf and Lam's shorthands [12] for Wolfe's '$<$', '$>$' and '$*$', respectively [14]. A dependence vector is a *distance vector* if all its entries are integer values.

Let $D \in \mathbb{Z}^{n\times m}$ be the *dependence matrix* whose columns are the m dependence vectors of the program. Let $\mathcal{E}(d)$ be the set of all distance vectors represented by a dependence vector.

Algorithm toUimodular(T: matrix)

Let $P = \alpha T^{-1}$, where α is a positive integer such that $P \in \mathbb{Z}^{n\times n}$;
Reduce P to its Hermite normal form H such that $UP = H$;
return U;

Figure 2: Construction of a unimodular transformation.

All relational and lexicographic order operators such as $>$ and $\succ$ on dependence vectors have straightforward extensions. Let $\circ$ be one such operator. We define $d \circ 0$ if $\forall\, z \in \mathcal{E}(d) : z \circ 0$.

In a sequential program, all its dependence vectors are lexicographically positive.

ASSUMPTION 1 $D \succ 0$, i.e., $\forall\, d \in D : d \succ 0$.

DEFINITION 2 (LEGALITY OF TRANSFORMATION) A transformation $T \in \mathbb{Z}^{n\times n}$ is *legal* for an n-deep loop nest if T is nonsingular and $\forall\, d \in D : (\forall\, z \in \mathcal{E}(d) : Tz \succ 0)$.

DEFINITION 3 (FULLY PERMUTABLE LOOPS [13]) In a loop nest, the i-th through the j-th loop are *fully permutable* if $\forall\, d \in D : ((d_1, \cdots, d_{i-1}) \succ 0$ or $(d_i, \cdots, d_j) \geqslant 0)$.

DEFINITION 4 (CANONICAL FORM [13]) The *canonical form* of a loop nest is a supernest of multiple fully permutable nests, with each nest made as large as possible with respect to the outer nests.

DEFINITION 5 (CANONICAL TRANSFORMATION) A legal transformation for a loop nest is *canonical* if the transformed loop nest it creates is in the canonical form.

When searching for a unimodular transformation, we find it convenient to first construct a nonunimodular transformation and then use the algorithm in Figure 2 to obtain a unimodular one.

THEOREM 1 *Let $T \in \mathbb{Z}^{n\times n}$ be a legal nonunimodular transformation for a loop nest and U be returned from toUnimodular(T). Then:*

1. *U is legal.*

2. *If the i-th through the j-th loops are fully permutable in the transformed program by T, then the i-th through the j-th loops are fully permutable in the transformed program by U.*

3. *The last p columns of U span the same vector space as the last p columns of T.*

Proof. All three statements follow from the fact that $U = HP^{-1} = (1/\alpha)HT$, where α is positive and H is a nonnegative nonsingular lower triangular matrix (Definition 1). ■

ASSUMPTION 2 All arrays are assumed to be stored in row-major order.

3.2 Dependence Cone and Integer Dependence Matrix

By exploiting the transitivity of dependence relations, cones can be used to represent the data dependences in the program. This abstraction dispenses with direction values, making it possible to use integer arithmetic to check the legality of a transformation.

As is customary, we define $\text{cone}(b_1, \cdots, b_p) = \{\lambda_1 b_1 + \cdots + \lambda_p b_p \mid \lambda_1, \cdots, \lambda_p \geqslant 0\}$ [11].

In [6], Irigoin discussed to represent a dependence vector d as:

$$\mathcal{P}(d) \quad = \quad o + \text{cone}(r_1, \cdots, r_p) \tag{1}$$

where o and $S = \{r_1, \cdots, r_p\}$ are constructed as follows:

1. Initially, $S := \{\}$.
2. For every entry d_k of d, where $1 \leqslant k \leqslant n$, repeat the following steps:
   (a) If $d_k \in \mathbb{Z}$ is an integer, then $o_k := d_k$.
   (b) If $d_k =$ '$+$', then $o_k := 1$ and $S := S + \{e_k\}$.
   (c) If $d_k =$ '$-$', then $o_k := -1$ and $S := S + \{-e_k\}$.
   (d) If $d_k =$ '$\pm$', then $o_k := 0$ and $S := S + \{e_k, -e_k\}$.

By construction, $\mathcal{P}(d) = \mathcal{E}(d)$. If d is a distance vector, then $S = \{\}$, implying that $\mathcal{P}(d) = \{d\}$.

$\mathcal{P}(d)$ is sometimes referred to as the *dependence cone* for the single dependence vector d. Precisely, $\mathcal{P}(d)$ is a polyhedron generated by the single vertex o and the extremal rays $r_1, \ldots, r_p$. $\mathcal{P}(d)$ can be considered as the translation of the cone $\text{cone}(r_1, \cdots, r_p)$ by the distance o.

Given a dependence cone $\mathcal{P}(d)$ of the form (1), we define $\mathcal{O}(d) = o$ and $\mathcal{R}(d) = [r_1, \cdots, r_p]$. In the case when $\mathcal{P}(d) = \{d\}$, $\mathcal{R}(d) = [\,]$, indicating an *empty matrix*.

The *integer dependence matrix*, D^I, is defined as follows:

$$D^I \quad = \quad [\mathcal{O}(d_1), \mathcal{R}(d_1), \cdots, \mathcal{O}(d_m), \mathcal{R}(d_m)] \tag{2}$$

where $d_1, \cdots, d_m$ are the m dependence vectors in D.

EXAMPLE 2 Consider a triple loop nest with the dependence matrix:

$$D \quad = \quad [d_1, d_2] \quad = \quad \begin{bmatrix} 1 & 0 \\ \pm & 1 \\ 0 & 0 \end{bmatrix}$$

The dependence cones for the two dependence vectors are:

$$\begin{aligned} \mathcal{P}(d_1) &= (1,0,0) + \text{cone}((0,1,0),(0,-1,0)) \\ \mathcal{P}(d_2) &= (0,1,0) \end{aligned}$$

Thus, $\mathcal{O}(d_1) = (1,0,0)$, $\mathcal{R}(d_1) = \left[\begin{smallmatrix} 0 & 0 \\ 1 & -1 \\ 0 & 0 \end{smallmatrix}\right]$, $\mathcal{O}(d_2) = (0,1,0)$ and $\mathcal{R}(d_2) = [\,]$. This leads to the following integer dependence matrix:

$$D^I \quad = \quad [\mathcal{O}(d_1), \mathcal{R}(d_1), \mathcal{O}(d_2), \mathcal{R}(d_2)] \quad = \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

3.3 Time Cone and Bases

In this paper, a *basis* of a cone $P = \{x \mid xA \geqslant 0\}$ is defined as a set of maximally linearly independent vectors contained in the cone. P is *pointed* if $\ker(A^T) = 0$.

The following cone is known as the *time cone*:

$$C(D) \quad = \quad \{t \mid tD \geqslant 0\}$$

Because $\mathcal{P}(d) = \mathcal{E}(d)$ for every $d \in D$, we have $C(D) = C(D^I)$.

The *time cone* is related to the canonical form of a loop nest. The first $\dim(C(D))$ rows of any canonical transformation form a basis of $C(D)$. This implies that the number of loops in the outermost fully permutable nest of the canonical form is exactly $\dim(C(D))$.

Because $D \succ 0$, we always have $1 \leqslant \dim(C(D)) \leqslant n$. If D contains distance vectors only, then $\dim(C(D)) = n$. In this special case, the canonical form consists of one single fully permutable loop nest. In the general case when D contains '$-$' or '$\pm$', we usually have $\dim(C(D)) < n$. For example, the time cone in Example 2 is two-dimensional: $\dim(C(D)) = 2 < n = 3$.

Several algorithms proposed in this paper require us to find a basis for the time cone $C(D)$.

Because $C(D) = C(D^I)$, the problem of constructing a basis for $C(D)$ is reduced to one of constructing a basis for $C(D^I)$. Several methods based on the simplex method can be used to construct a basis for $C(D^I)$ [5, 8]. In practice, the number and magnitudes of the vectors in D^I are small. The algorithm ConeBasis, given in Figure 3, is proposed to find a basis for a cone based on a classic decomposition of the cone into a linear space and a pointed cone [11, p. 100]. PointedConeBasis, which finds a basis for a pointed cone, is not explained here because it is a slight modification of an algorithm fully discussed in [16, Figure 10].

Let us apply the time cone algorithm ConeBasis to Example 2 to find a basis for $C(D)$. The cone $C(D^I)$ in the example is not pointed: $\ker((D^I)^T) = \{(0,0,1)\}$. So $K = [0\ 0\ 1]$. PointedConeBasis($[-K, K, D^I]$) is called to find a basis for the pointed cone $C([-K, K, D^I])$, which is $\{(1,0,0)\}$. Hence, $\{(1,0,0),(0,0,1)\}$ is found to be a basis for $C(D)$.

4 Canonical Transformations

When constructing a legal transformation, we often need to know at which loop of the transformed loop nest, a dependence vector is carried (in its entirety). The usual concept of dependence-carrying loop [15] is extended for general dependence vectors.

DEFINITION 6 (DEPENDENCE-CARRYING LOOP) Let T be a legal transformation and t_i be its i-th row. A dependence vector $d \in D$ is carried at the k-th loop in the transformed loop nest if $\forall\, z \in \mathcal{E}(d) : (t_1 z, \cdots, t_k z) \succ 0$ and $\exists\, z \in \mathcal{E}(d) : t_1 z = \cdots = t_{k-1} z = 0$.

The polyhedron cone $\mathcal{P}(d)$ is useful in identifying the dependence-carrying loop for d.

LEMMA 1 *Let T be a legal transformation and t_i be its i-th row. In the transformed loop nest, a dependence vector $d \in D$ is carried at the k-th loop if* $\forall\, 1 \leqslant i < k : t_i\mathcal{O}(d) = 0$ *and* $t_k\mathcal{O}(d) > 0$.

Proof. Based on the dependence cone $\mathcal{P}(d)$ given in (1), all distance vectors $z \in \mathcal{E}(d)$ can be expressed as $z = \mathcal{O}(d) + \lambda_1 r_1 + \cdots + \lambda_p r_p$, where $\lambda, \cdots, \lambda_p$ are any arbitrary nonnegative integers and $r_1, \cdots, r_p$ are the rays in $\mathcal{R}(d)$. This means that $z' = \mathcal{O}(d) + \lambda_j r_j$ is also a distance vector.

```
Algorithm PointedConeBasis (D^I: matrix in Z^{n×m})

S := {};

for every set of n − 1 columns x_1, ··· , x_{n−1} in D^I do
    Let A = [x_1, ··· , x_{n−1}] be an n × (n − 1) matrix;
    Reduce A to row echelon form B, i.e, find a unimodular matrix
    U ∈ Z^{n×n} such that UA = B, where B is in row echelon form [15, p. 115];
    if the last row of B is the only row of B that is entirely zero then
        /* x_1, ··· , x_{n−1} are linearly independent, and u_n is the n-th row of U */
        if u_n D^I ⩾ 0 then       /* u_n = x_1 × ··· × x_{n−1} up to scaling */
            S := S + {u_n};     /* u_n is an extremal ray */
        else if u_n D^I ⩽ 0 then
            S := S + {−u_n};     /* −u_n is an extremal ray */
        endif
    endif
    if |S| = n then break; /* only n rays to be found when dim(C(D^I)) = n */
endfor

return S;
```

```
Algorithm ConeBasis (D: matrix in Z^{n×m})

Construct D_1^I from D according to (2);
/* The basic idea of finding a basis for C(D^I) is to decompose C(D^I) as:
   C(D^I) = L + Q, where L = ker((D^I)^T) and Q = L^⊥ ∩ C(D^I)
   Here, L is known as the linearity space of C(D^I) and Q is pointed [11]. */
Let [b_1, ··· , b_ℓ] be the column vectors forming a basis of ker(D^I);
Express Q as: Q = {t | t[−K, K, D^I] ⩾ 0}, where K = [b_1, ··· , b_ℓ];
return {b_1, ··· , b_ℓ} + PointedConeBasis([−K, K, D^I]);
```

Figure 3: Construction of a basis for a cone.

Since T is legal, $Tz' = T\mathcal{O}(d) + T\lambda_j r_j \succ 0$. We are given the fact that $\forall\, 1 \leqslant i < k : t_i\mathcal{O}(d) = 0$ and $t_k\mathcal{O}(d) > 0$. So we must have $(t_1 r_j, \cdots, t_k r_j) \succeq 0$. (If $(t_1 r_j, \cdots, t_{k-1} r_j) = 0$, then $t_k r_j > 0$ must be true. Otherwise, $Tz' \succ 0$ cannot hold when λ_j is made arbitrarily large.) This implies that $(t_1 z', \cdots, t_k z') \succ 0$, and consequently, $(t_1 z, \cdots, t_k z) \succ 0$ for every distance vector $z \in \mathcal{E}(d)$. $\mathcal{O}(d)$ is also a distance vector, which satisfies $t_1\mathcal{O}(d) = \cdots, t_{k-1}\mathcal{O}(d) = 0$ by the hypothesis. Hence, the lemma is true by Definition 6. ■

Darte and Vivien's algorithm [2] for finding a canonical transformation is refined and depicted in Figure 4. Note that Wolf and Lam's heuristics-based algorithm [12] does not guarantee to succeed in all cases [2]. CanonicalTrans first finds a nonsingular canonical transformation T and then uses the algorithm in Figure 2 to derive from T a unimodular canonical transformation U. Let B be the number of times FPNest is called. B represents the number of fully permutable nests in the canonical form. By calling our time cone algorithm ConeBasis recursively, CanonicalTrans

Algorithm FPNest(D : matrix, T: matrix, k : integer)

S_k := ConeBasis($[-T^T, T^T, D]$); /∗ *build the k-th FPNest* ∗/
T_k := matrix(S_k);
$T := \left[\begin{smallmatrix} T \\ T_k \end{smallmatrix}\right]$;
if T is of size $n \times n$ **then return**;
$D_{k+1} := \{d \mid \forall\, d \in D : T_k\mathcal{O}(d) = 0\}$;
/∗ *D_{k+1} contains dependence vectors not carried so far (Lemma 1)* ∗/
FPNest($D_{k+1}, T, k+1$);

Algorithm CanonicalTrans(L : loop nest (of depth n))

$D_1 := D$;
$T := [\,]$; /∗ an empty matrix for notational convenience ∗/
FPNest($D_1, T, 1$);
U := toUnimodular(T);
return U;

Figure 4: Construction of a unimodular canonical transformation.

builds recursively the following nonsingular transformation:

$$T = \begin{bmatrix} T_1 \\ \vdots \\ T_B \end{bmatrix}$$

where T_k is obtained when FPNest is called the k-th time and is used to create the k-th fully permutable nest in the canonical form.

EXAMPLE 3 Let us apply CanoninalTrans to Example 2. In the first call FPNest($D_1, T, 1$), where $D_1 = D$ and $T = [\,]$, ConeBasis finds, say, $S_1 = \{(1,0,0),(0,0,1)\}$ as a basis of $C(D_1)$. Let $T = T_1 = \left[\begin{smallmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{smallmatrix}\right]$. The dependence vector $(1, \pm, 0)$ is already carried in the outermost loop in the transformed loop nest. We have $D_2 = [0\ 1\ 0]$. In the second call FPNest($D_2, T, 2$), ConeBasis will return $S_2 = \{(0,1,0)\}$ as a basis for $C([-T_1^T, T_1^T, D_2])$. Let $T_2 = [0\ 1\ 0]$, we obtain:

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

T is already unimodular: U = toUnimodular(T) $= T$. The canonical form has two fully permutable nests: the outer nest consists of the original outermost and innermost loops and the inner nest consists of the original middle loop.

THEOREM 2 *U returned from Figure 4 is a canonical unimodular transformation.*

Proof. In the k-th call to FPNest, let $D_k = D$. $C(D_k)$ must be at least one-dimensional larger than $C(D_{k-1})$, implying the existence of T_k and the eventual termination of the algorithm. By construction, we have (a) ConeBasis returns a basis of $C([-T^T, T^T, D_k])$, which is used to create

(a) LOOP NEST

do $x_1 = a_1,\ b_1$
  $\cdots$
    do $x_n = a_n,\ b_n$

(b) TRANSFORMED LOOP NEST

do $y_1 = c_1,\ d_1$
  $\cdots$
    do $y_{n-\tau} = c_{n-\tau},\ d_{n-\tau}$
      do $z_1 = f_1,\ g_1$
        $\cdots$
          do $z_\tau = f_\tau,\ g_\tau$

(c) TILED LOOP NEST

do $y_1 = c_1,\ d_1$
  $\cdots$
    do $y_{n-\tau} = c_{n-\tau},\ d_{n-\tau}$
      do $z_1z_1 = f'_1,\ g'_1,\ B_1$
        $\cdots$
          do $z_\tau z_\tau = f'_\tau,\ g'_\tau,\ B_\tau$
            do $z_1 = \max(z_1z_1, f_1),\ \min(z_1z_1 + B_1 - 1, g_1)$
              $\cdots$
                do $z_\tau = \max(z_\tau z_\tau, f_\tau),\ \min(z_\tau z_\tau + B_\tau - 1, g_\tau)$

Figure 5: Tiling of a loop nest.

the largest k-th fully permutable nest with respect to the outer $k-1$ fully permutable nests, and (b) T is legal. Thus, T is a canonical transformation. According to both (a) and (b) in Theorem 1, U is a canonical unimodular transformation. ∎

5 Tiling Transformations

As mentioned in Section 1, our approach to improving data locality of a loop nest proceeds as follows. Firstly, a unimodular transformation is found and applied to the program to maximise the amount of data reuse carried in the inner τ loops of the transformed program with τ made as small as possible. Secondly, the inner τ loops of the transformed program are tiled so that all data reuse carried in these loops can be exploited in the cache. In this section, loop tiling used in this paper is defined and some related concepts ate made precise.

DEFINITION 7 (TILING AND RELATED CONCEPTS) Consider an n-deep loop nest of the form given in Figure 5(a).

1. A *tiling* for the loop nest is a legal unimodular transformation $H_\tau \in \mathbb{Z}^{n\times n}$ as follows:

$$\begin{bmatrix} y_1 \\ \vdots \\ y_{n-\tau} \\ z_1 \\ \vdots \\ z_\tau \end{bmatrix} = H_\tau \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

2. The *transformed loop nest*, depicted in Figure 5(b), is the loop nest restructured by H_τ. The first $n-\tau$ loops are referred to as the y-*loops*. The inner τ loops are fully permutable and are referred to as the z-*loops*. In the original loop nest, the k-th loop enumerates the iteration space along the direction e_k. In the transformed loop nest, the k-th loop enumerates the

iteration space along the direction given by the k-th column of H_τ^{-1}. This leads directly to the following formal definition of the tile space.

3. The *tile space*, $\mathcal{T}(H_\tau)$, is the linear space spanned by the last τ columns of H_τ^{-1}.

4. The *tiled loop nest*, given in Figure 5(c), is an $(n + \tau)$-deep loop nest obtained from the transformed loop nest with the inner τ loops tiled with the tile *size* of $B_1 \times \cdots \times B_\tau$. Techniques for generating tiled code are discussed in [7, 17]. Geometrically, by tiling the loop nest, we cut the iteration space into slices parallel to the tile space $\mathcal{T}(H_\tau)$ and then tile each slice with rectangular tiles of size $B_1 \times \cdots \times B_\tau$. In the tiled loop nest, the $y_1, \cdots, y_{n-\tau}$ loops enumerate all slices, the $z_1z_1, \cdots, z_\tau z_\tau$ loops step between tiles, and the $z_1, \cdots, z_\tau$ loops execute the points within a tile.

5. The *tile shape* is determined by its edges (adjoining at a common vertex), which are parallel to the last τ columns of H_τ^{-1} and the *tile size* is $B_1 \times \cdots \times B_\tau$.

All concepts defined above are introduced in Example 1 and illustrated in Figure 1.

6 Reuse Analysis

For completeness, we review Wolf and Lam's reuse framework on uniformly generated references [12]. The basic idea is to use vector spaces to represent the directions in which reuse is found. These are the directions to be included in the tile space.

Let $\{A(Mx + c_1), \cdots, A(Mx + c_g)\}$ be the set of all uniformly generated references for an array A in the loop nest. Let M_S be the matrix M with its last row removed. Let $c_{S,i}$ ($c_{S,j}$, resp.) be the vector c_i (c_j, resp.) with its last entry removed. The four types of reuse within this uniformly generated set are quantified as follows:

1. The *self-temporal reuse space* for a single reference is ker(M). Two iterations x_1 and x_2 access the same element of A if and only if $x_1 - x_2 \in$ ker(M).

2. The *self-spatial reuse space* for a single reference is ker(M_S). Two iterations x_1 and x_2 may access the same cache line only if $x_1 - x_2 \in$ ker(M_S).

3. The *group-temporal reuse space* for the set is ker(M) + span$\{r\}$, where $r = \{r_{i,j} \mid \forall\, 1 \leqslant i, j \leqslant g : r_{i,j}$ is a particular solution to $Mx + c_i = Mx + c_j\}$.

4. The *group-spatial reuse space* for the set is ker(M_S) + span$\{r_S\}$, where $r_S = \{r_{i,j} \mid \forall\, 1 \leqslant i, j \leqslant g : r_{i,j}$ is a particular solution to $M_S x + c_{S,i} = M_S x + c_{S,j}\}$.

5. The *reuse space* for the set is the linear space spanning all four individual spaces, which is always equal to the group-spatial reuse space because it contains all the other three.

DEFINITION 8 (REUSE SPACE AND REUSE VECTORS) The *reuse space*, R, for a loop nest is the linear space spanning the reuse spaces for all uniformly generated sets in the loop nest. A vector in R is called a *reuse vector* or *direction*.

7 Reuse-Driven Tiling for Locality

The notion of reuse-carrying loop is formally defined below.

DEFINITION 9 (REUSE-CARRYING LOOP) Let T be a legal transformation and t_i be its i-th row. The k-th loop of the transformed loop nest is said to carry a reuse vector $r \in R$ if $\forall\, 1 \leqslant i < k$: $t_i r = 0$ and $t_k r \neq 0$, and in this case, the loop is said to carry reuse.

According to this definition, the $k \times n$ top submatrix of a transformation T completely determines the reuse carried in the innermost $(n-k)$ loops in the transformed loop nest. This property is stated as a lemma below and will be used to construct a tiling transformation incrementally.

LEMMA 2 *Let T be the $k \times n$ top submatrix of a legal transformation. The reuse space carried in the innermost $n-k$ loops of the transformed loop nest, called the localised reuse space, is* $\ker(T) \cap R$.

Proof. A reuse vector $r \in R$ is carried in the innermost $n-k$ loops if and only if $\forall\, 1 \leqslant i \leqslant k$: $t_i r = 0$, i.e., if and only if $r \in \ker(T)$, where t_i is the i-th row of T. Hence, $\ker(T) \cap R$ is the reuse space localised in the innermost $n-k$ loops in the transformed loop nest. ■

Based on this lemma, the reuse space localised in the tile space is given by $\mathcal{T}(H_\tau) \cap R$.

DEFINITION 10 (OPTIMAL TILING) A tiling H_τ is *optimal* if for every tiling $H'_{\tau'}$, we have:

- $\dim(\mathcal{T}(H_\tau) \cap R) \geqslant \dim(\mathcal{T}(H'_{\tau'}) \cap R)$, and
- $\tau \leqslant \tau'$ when $\dim(\mathcal{T}(H_\tau) \cap R) = \dim(\mathcal{T}(H'_{\tau'}) \cap R)$.

An optimal tiling H_τ is said to be *locality-optimal* if $\mathcal{T}(H_\tau) \supseteq R$ because all data reuse is localised in the tile space and can thus be exploited in the cache.

We make use of the canonical form of fully permutable loop nests and distinguish two cases. In the special case of one single fully permutable loop nest, the algorithm presented always finds a locality-optimal tiling. In the general case of multiple fully permutable nests, the algorithm presented aims at localising data reuse in the tile space so that the reuse space contained in the tile space has the largest dimensionality possible. This algorithm finds a locality-optimal tiling in the special case considered in Theorem 5.

7.1 One Single Fully Permutable Loop Nest

In this special case, we can always find a locality-optimal tiling H_τ such that $\mathcal{T}(H_\tau) \supseteq R$ and τ is the smallest possible. Due to data dependences, $\mathcal{T}(H_\tau)$ may contain R as a strict subspace. This means that the number of tiled dimensions may be larger than the dimensionality of the reuse space. If dim$(R) = n$, all dimensions of the iteration space must be tiled, in which case, any canonical transformation is locality-optimal. In general, an optimal solution H_τ is found using the algorithm in Figure 6, which calls our time cone algorithm twice, once to construct its first $n-\tau$ rows and once to construct its last τ rows.

THEOREM 3 *H_τ returned by OptTiling given in Figure 6 is locality-optimal.*

Algorithm OptTiling(L: loop nest)

Step 1. /∗ *Construct a basis of* $C_1 = \{t \mid tR = 0, tD \geqslant 0\}$ ∗/
S_1 := ConeBasis($[-R, R, D]$);
T := matrix(S_1);
Step 2. /∗ *Construct a basis of* $C_2 = \{t \mid tT^T = 0, tD_2 \geqslant 0\}$ ∗/
$D_2 := \{d \mid \forall\, d \in D : T_k\mathcal{O}(d) = 0\}$;
S_2 := ConeBasis($[-T^T, T^T, D_2]$);
Step 3. $T := \begin{bmatrix} T \\ \text{matrix}(S_2) \end{bmatrix}$;
H := toUnimodular(T);
$\tau := |S_2|$; /∗ *Theorem 3* ∗/
return H_τ;

Figure 6: Construction of a locality-optimal tiling for one fully permutable loop nest.

Proof. By Definition 10, we show that $\mathcal{T}(H_\tau) \supseteq R$ and τ is the smallest possible.

Existence. We show that OptTiling returns a unimodular transformation H_τ. By examining the three steps of the algorithm, it suffices to show that $|S_2| = \tau$, i.e., dim$(C_2) = \tau$. The canonical form for the loop nest under consideration has one fully permutable nest. Thus, $\dim(C(D)) = n$. Since all columns of D_2 are taken from D. we must have $\dim(C(D_2)) = n$. After Step 1 is completed, rank$(T) = n - \tau$ (i.e., $\dim(C_1) = n - \tau$) because the $n - \tau$ rows of T are a basis of C_1. Thus, dim$(\{t \mid tT^T = 0\}) = \tau$, and consequently, dim$(C_2) = \tau$. So in Step 3, a basis of τ vectors in C_2 can be found to complete T as a nonsingular matrix. Finally, according to Theorem 1, H_τ is unimodular.

Legality. H_τ is legal due to the fact that T is legal by construction and Theorem 1.

Optimality. By construction, $\mathcal{T}(H_\tau) \supseteq R$. If H_τ is not optimal, there must exist a tiling $H'_{\tau'}$ such that $\mathcal{T}(H'_{\tau'}) \supseteq R$ and $\tau' < \tau$. This implies that the first $n - \tau'$ rows of $H'_{\tau'}$ are a basis of C_1, which is impossible since $n - \tau' > n - \tau =$ dim(C_1).

Hence, H_τ is locality-optimal by Definition 10. ∎

EXAMPLE 4 Let us trace the algorithm of Figure 6 to construct the optimal tiling transformation discussed in Example 1. In Step 1, ConeBasis is called to find $S_1 = \{(1, 0, -1\}$ as a basis for the cone C_1. Let $T = [1\ 0\ -1]$. The outermost loop in the transformed loop nest thus created does not carry any dependences. Thus, $D_2 = D = [0\ +\ 0]$. Calling ConeBasis on C_2, we obtain $S_2 = \{(0, 1, 0), (1, 0, 1)\}$ a basis of C_2. In Step 3, we obtain:

$$T = \begin{bmatrix} 1 & 0 & -1 \\ \hdashline 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

T is not unimodular because $|\det(T)| = 2$. So the algorithm finally returns the optimal tiling:

$$H_2 \quad = \quad \text{toUnimodular}(T) \quad = \quad \left[\begin{array}{ccc} 1 & 0 & -1 \\ \hdashline 0 & 1 & 0 \\ 1 & 0 & 0 \end{array}\right]$$

Since $\mathcal{T}(H_2) = R$, two dimensions of the iteration space are tiled to exploit two degrees of reuse.

Sometimes, in order to localise all reuse in the tile space, the number of tiled dimensions has to be larger than the dimensionality of the reuse space.

EXAMPLE 5 Consider the following loop nest:

$$D \quad = \quad \begin{bmatrix} 1 & 1 & 0 \\ + & 0 & 0 \\ 0 & 0 & 2 \\ 0 & 0 & -1 \end{bmatrix}, \quad R \; = \; \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}$$

Because C_1 is one-dimensional, in Step 1, $\{(1,0,0,0)\}$ is found as a basis of C_1. We have $T = [1\ 0\ 0\ 0]$. In Step 2, the first two dependence columns of D are found to be carried in the outermost loop in the transformed loop nest. So $D_2 = [0, 0, 2, -1]$. Because C_2 is three-dimensional, $\{(0,1,0,0), (0,0,1,1), (0,0,1,2)\}$ is a basis of C_2. In Step 3, we obtain:

$$T \quad = \quad \left[\begin{array}{cccc} 1 & 0 & 0 & 0 \\ \hdashline 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 3 & 1 & 2 \end{array}\right]$$

Since T is unimodular, $H_3 = \text{toUnimodular}(T) = T$. The tile space $\mathcal{T}(H_3)$ strictly contains the reuse space R: dim$(\mathcal{T}(H_3)) = 3$ and dim$(R) = 2$. To exploit two degrees of reuse in the program, three dimensions of the iteration space have to be tiled.

7.2 Multiple Fully Permutable Loop Nests

When the canonical form of a loop nest consists of several fully permutable nests, it is only profitable to tile the innermost permutable nest. Two complications must be recognised.

- Firstly, the approach of first transforming the loop nest into the canonical form and then tiling the innermost permutable nest to optimise reuse cannot always exploit all reuse available. This is because the reuse information is not used in the construction of the canonical form. Any reuse that is carried in outer permutable nests cannot be exploited, even though an appropriate transformation may move the reuse into the innermost permutable nest. For example, the following loop nest

$$D \quad = \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ - & 0 & 1 \end{bmatrix}, \quad R \; = \; \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \end{bmatrix} \qquad (3)$$

is already in the canonical form of two fully permutable nests with the first two loops in the outer nest and the last loop in the inner nest. Tiling the last loop alone does not exploit any reuse more than the original program does. Using the algorithm in Figure 7, the following tiling transformation is found to exploit all reuse of the program:

$$H_2 = \begin{bmatrix} 1 & 1 & 0 \\ \cdots & \cdots & \cdots \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4}$$

- Secondly, it is no longer always possible to exploit all reuse in the program. Due to dependence constraints, some data reuse cannot be localised in the innermost permutable nest. In the extreme case, a program is simply *untilable* because all data reuse will be carried in outer fully permutable nests. This is illustrated by the following example:

$$D = \begin{bmatrix} 1 & 0 & 0 & 0 \\ - & + & 0 & 0 \\ 0 & - & 1 & 0 \\ 0 & - & 0 & 1 \end{bmatrix}, \quad R = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

  The first loop must be in a fully permutable nest by itself, and likewise for the second loop. The last two loops can be placed in the same fully permutable nest. Therefore, the reuse is all carried in the first two loops. Tiling the last two loops does not exploit any reuse in the program. Thus, this program does not have any locality.

Our data locality algorithm depicted in Figure 7 evolves naturally from Figure 4. The two algorithms are identical except two differences: (a) FPNest is renamed to LoopNest with a fourth parameter added to export the number of loops to be tiled, and (b) a number of statements, inside the box highlighted, are added for the purposes to be explained below.

Our data locality algorithm creates the transformed loop best recursively and greedily, loop by loop, starting from the outermost loop. In line (a), we create as many y-loops as possible that do not carry any reuse. If this fails, we check in line (b) to see if it is possible to generate all remaining loops as the z-loops. That being the case, line (c) will be executed and the recursion will terminate at the current recursive call. Otherwise, line (d) will be executed, creating one y-loop using the row vector obtained as a sum of all vectors in S_k. This step has a well-founded explanation. Consider the case where all vectors of S_k are used to create a total of $|S_k|$ loops. We can always apply the wavefront transformation of appropriate size:

$$\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

to these $|S_k|$ loops so that after transformation, the first transformed loop — the y-loop created in line (d) — carries all dependences that are carried by all these $|S_k|$ loops and the other $|S_k| - 1$ loops can be moved into the innermost permutable nest.

```
Algorithm LoopNest(D : matrix, T: matrix, k : integer, τ : integer)

        Calculate the reuse space R_k localised in ker(T) by Lemma 2;
        if R_k = [ ] then
            Print "No locality can be exploited";
            Stop;
        endif
(a)     S_k := ConeBasis([−R_k, R_k, −T^T, T^T, D]);
        if |S_k| = 0 then    /* C([−R_k, R_k, −T^T, T^T, D]) = {0} */
(b)         S_k := ConeBasis(−T^T, T^T, D);
            if |S_k| = n − rank(T) then   /* T has full-row rank */
(c)             τ := |S_k|;    /* the inner τ loops to be tiled */
            else
               /* this dimension of reuse not localised in the tile space */
(d)             Replace all vectors x_1, x_2, ··· in S_k by x_1 + x_2 + ···;
            endif
        endif
T_k := matrix(S_k);
T := [T ; T_k];
if T is of size n × n then return;
D_{k+1} := {d | ∀ d ∈ D : T_k O(d) = 0};
/* D_{k+1} contains dependence vectors not carried so far (Lemma 1) */
LoopNest(D_{k+1}, T, k + 1, τ);

Algorithm Tiling(L : loop nest (of depth n))

D_1 := D;
T := [ ];
LoopNest(D_1, T, 1, τ);
H := toUnimodular(T);
return H_τ;
```

Figure 7: Construction of a tiling in the general case.

We repeat the same process recursively until either all data reuse has been carried in the y-loops constructed so far, in which case, the program will not be tiled, or a tiling transformation has been found when line (c) has been finally reached.

THEOREM 4 *H_τ found in Figure 7 is a legal tiling transformation.*

Proof. The algorithm in Figure 7 is adapted from the algorithm in Figure 4 for constructing a canonical transformation. If $S_k = \{\}$ for the S_k constructed in line (a), we ignore the reuse space R and call S_k := BasisCone($[-T^T, T^T, D]$) in line (b). We must have $S_k \neq \{\}$ for the same reason why $S_k \neq \{\}$ holds for the same statement in Figure 4 (Theorem 2). Thus, if LoopNest returns normally to the caller Tiling, T must be a legal transformation. According to Theorem 1, H_τ must be a legal tiling transformation. ■

THEOREM 5 *H_τ found in Figure 7 is locality-optimal if line (d) is not executed.*

Proof. If the construction of H_τ never involves line (d) executed, then all reuse of the loop nest must be carried in the inner τ loops of the transformed loop nest, i.e., $\mathcal{T}(H_\tau) \supseteq R$. According to Definition 10, it suffices to show that τ is the smallest possible. Assume, to the contrary, that there exists a tiling transformation $H'_{\tau'}$ such that $\mathcal{T}(H'_{\tau'}) \supseteq R$ and $\tau' < \tau$. We show that this contradicts to the greedy nature of the algorithm in Figure 7. Let h'_k be the first row of $H'_{\tau'}$ such that $h'_k \notin \text{span}\{h_1, \cdots, h_{n-\tau}\}$; such a row always exists because $\tau' < \tau$. Since $\text{span}\{h'_1, \cdots, h'_{k-1}\} \subseteq \text{span}\{h_1, \cdots, h_{n-\tau}\}$, the dependence matrix D' carried in the inner $n-k+1$ loops of the transformed loop nest by $H'_{\tau'}$ must contain as a submatrix the dependence matrix D carried in the inner τ loops of the transformed loop nest by H_τ. This means that $h'_k D \geqslant 0$ because $h'_k D' \geqslant 0$. We also know that $h'_k R = 0$ because $\mathcal{T}(H'_{\tau'}) \supseteq R$. Therefore, before constructing the last τ rows of H_τ in line (c), the algorithm in Figure 7 would have found h'_k as a row vector for creating one more y-loop in the transformed loop nest. This contradicts to the fact H_τ generates only $n - \tau$ y-loops. ■

EXAMPLE 6 Consider the following quadruple loop nest:

$$D = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & + & 0 & 0 \\ - & 0 & 1 & 0 \\ - & 0 & - & 1 \end{bmatrix}, \quad R = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ -1 & 1 \end{bmatrix}$$

The canonical form consists of two permutable nests with first two loops in the outer nest and the last two loops in the inner nest. The following transformation is found to be locality-optimal because line (d) is not executed (Theorem 5):

$$H_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hdashline 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

LoopNest is called three times. The first two rows of H_2 are found in line (a) in the first two calls, respectively, and the last two rows of H_2 are found in line (b) in the third call. In this example, tiling the innermost two loops exploits all two degrees of reuse available.

EXAMPLE 7 Consider the last example in the paper:

```
do i = 1, N
   do j = 1, N
      do k = 1, N
         do m = 1, N
            A(i, j, 2 * k) = A(i - 1, j - 1, k + m) + C(k, m, i)
```

Wolfe's Tiny reports the following dependence matrix (with simplifications):

$$D = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ \pm & 0 \\ \pm & + \end{bmatrix}$$

The reuse space for the loop nest can be computed as follows:

$$R = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \end{bmatrix}$$

In the first call to LoopNest, $S_1 = \{\}$ in line (a) because $\dim(R) = 4$ and $|S_1| = 2$ in line (b) because $\dim(C(D)) = 2$. S_1 contains a basis for the time cone $C(D)$. Assuming that $S_1 = \{(1,0,0,0),(0,1,0,0)\}$, we obtain $T = T_1 = [1\ 1\ 0\ 0]$.

The dependence matrix and the reuse matrix for the three remaining loops are as follows:

$$D_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ + \end{bmatrix}, \quad R_2 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & -1 & 0 \end{bmatrix}$$

In the second call to LoopNest, $S_2 = \{\}$ in line (a) because $\dim(R_2) = 3$. In line (b), we find that $S_2 = \{(0,1,0,0),(0,0,1,0),(0,0,0,1)\}$. Since $\tau = n - \text{rank}(T) = 3$, we obtain:

$$T = \begin{bmatrix} 1 & 1 & 0 & 0 \\ \hdashline 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5}$$

T is unimodular, so $H_3 = \text{toUnimodular}(T) = T$. In the transformed loop nest nest, the innermost three loops will be tiled to exploit three out of four degrees of reuse in the program. H_3 is optimal by Definition 10 because one degree of reuse has to be carried at the outermost loop in the transformed loop nest due to data dependences.

8 Related Work

This work is directly related to [2, 6, 12, 16]. Based on Irigoin's dependence cone work [6], the search space for legal transformations is defined by the time cone. An algorithm presented in our previous work [16] is modified to find a basis for the time cone. Based on this algorithm, Darte and Vivien's algorithm for finding canonical transformations is implemented. The reuse framework based on vector spaces is due to Wolf and Lam [12].

The relevance of cones to solving compiler problems is being increasingly recognised. Successful applications are dependence abstraction [4, 18], loop scheduling [3], and tiling for parallelism [1, 3, 10, 16]. One more application considered in this paper is tiling for data locality.

The concept of fully permutable loop nests introduced in [13] has emerged to be a useful. Initially in [13], the concept is related to the maximal degree of doall parallelism inherent in the program. In [3, 9] and this paper, the concept is also exploited in loop tiling.

This work improves and extends Wolf and Lam's earlier data locality work [12]. There are three main differences. Firstly, we use arbitrary vectors to represent the reuse directions while

Wolf and Lam use only elementary vectors. In the case of Example 1, Wolf and Lam will split the reuse direction $(1, 0, 1)$ into $(1, 0, 0)$ and $(0, 0, 1)$ and approximate the reuse space as:

$$R = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

As a result, Wolf and Lam's algorithm will tile all three loops, which is not optimal.

Secondly, we consider a much larger search space than Wolf and Lam. Their search space consists of a total of 2^n different ways of dividing the transformed loop nest into y-loops and z-loops. In the case of the example given in (3), one optimal solution we find is given in (4). In Wolf and Lam's approach, the reuse space for the loop nest will be approximated as:

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

A total of 2^3 possible ways to tile the loop nest will be considered: $\{\{i\}, \{j, k\}\}$, $\{\{j\}, \{i, k\}\}$, $\{\{k\}, \{i, j\}\}$, $\{\{i, j\}, \{k\}\}$, $\{\{i, k\}, \{j\}\}$, and $\{\{j, k\}, \{i\}\}$. In each case, the first subset contains the loops that are left untiled and the second subset contains the loops to be tiled innermost. The final transformation found cannot include the reuse direction $(1, -1, 0)$ in the tile space and is thus not optimal.

Thirdly, Wolf and Lam use a simple data locality model to select optimal solutions. Their model includes the cache line size as the only machine-specific parameter. In this paper, we show that, in many important cases, such as the case when the canonical form has one single permutable nest (Theorem 3) and the special cases covered in Theorem 5, the optimal solutions that exploit all reuse can be found based on the reuse information only. In the general case when $|S_k| > 1$ in line (d) of Figure 7, accurate locality estimates can help to choose alternative solutions, which is the future work. Our algorithm in Figure 7 reduces the number of times locality estimates are calculated and contains a placeholder in line (d), where different locality models can be plugged in and experimented with.

9 Conclusion

We have provided algorithms for improving data locality of a perfect loop nest. In the case of one single fully permutable nest, the program is tiled optimally so that all reuse, and consequently, all locality is exploited. In this special case, optimising reuse always optimises data locality. In the general case, the program is tiled in order to localise as much data reuse as possible in the tile space. In the general case, optimising reuse optimises potentially data locality of the program. Presently, our algorithm in Figure 7 relies on the reuse information only to construct a transformation matrix. For each row of the matrix created in line (d) of Figure 7, the corresponding loop in the transformed loop nest carries one dimension of reuse that cannot be exploited because the loop is not in the innermost permutable nest. A data locality model can help to make alternative choices in line (d).

10 Acknowledgements

The first author would like to thank Michael E. Wolf for discussions about his data locality algorithm and he is supported by an Australian Research Council Grant A49600987.

References

[1] P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling. *Integration, the VLSI Journal*, 17:33–51, 1994.

[2] A. Darte and F. Vivien. A comparison of nested loops parallelization algorithms. Technical Report 95–11, Ecole Normale Supérieure de Lyon, May. 1995.

[3] A. Darte and F. Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. Technical Report 96–34, Ecole Normale Supérieure de Lyon, November 1996.

[4] A. Darte and F. Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. In *Proc. of the 1996 International Conference on Parallel Architectures and Compilation Techniques*, pages 281–291, Boston, MA,, 1996.

[5] M. E. Dyer and L. G. Proll. An algorithm for determining all extreme points of a convex polytope. *Mathematical Programming*, 12:81–96, 1977.

[6] F. Irigoin. Loop reordering with dependence direction vectors. Technical Report EMP-CAI-I A/184, Ecole Nationale Superieure des Mines de Paris, Nov. 1988.

[7] F. Irigoin and R. Triolet. Supernode partitioning. In *Proc. of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 319–329, San Diego, California., Jan. 1988.

[8] H. Le Verge. A note on chernikova's algorithm. Technical Report 635, IRISA (INRIA-Rennes), Feb. 1992.

[9] H. Ohta, Y. Saito, M. Kainaga, and H. Ono. Optimal tile size adjustment in compiling for general DOACROSS loop nests. In *1995 ACM International Conference on Supercomputing*, pages 270–279. ACM Press, 1995.

[10] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *J. of Parallel and Distributed Computing*, 16(2):108–230, Oct. 1992.

[11] A. Schrijver. *Theory of Linear and Integer Programming*. Series in Discrete Mathematics. John Wiley & Sons, 1986.

[12] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proc. of the ACM SIGPLAN'91 Conf. on Programming Language Design and Implementation*. ACM, Jun. 1991.

[13] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, Oct. 1991.

[14] M. J. Wolfe. More iteration space tiling. In *Supercomputing '88*, pages 655–664, Nov. 1989.

[15] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[16] J. Xue. Communication-minimal tiling of uniform dependence loops. *Journal of Parallel and Distributed Computing*, 42(1):42–59, 1997.

[17] J. Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.

[18] Y. Q. Yang, C. Ancourt, and F. Irigoin. Minimal data dependence abstractions for loop transformations. In *Proc. of the 7th Workshop on Languages and Compilers for Parallel Computing*, Ithaca, Aug 1994.