The Polytope Model for Optimizing Cache Locality

Benoît Meister, Vincent Loechner and Philippe Clauss
ICPS, Université Louis Pasteur, Strasbourg
Pôle API, Bd Sébastien Brant
67400 Illkirch FRANCE
e-mail: {meister,loechner,clauss}@icps.u-strasbg.fr

Abstract

An essential way to obtain high performance on todays computers consists in optimizing spatial and temporal locality of the data. Many scientific applications contain loop nests that operate on large multi-dimensional arrays whose sizes are often parameterized. In this work, temporal reuse occuring from several references in a same statement of a parameterized loop is modeled geometrically. More precisely, iterations are classified depending on the kind of temporal reuse associated to their references. This classification defines a partitioning of the iteration space as a union of disjoint parameterized convex polytopes. A convenient scanning of these polytopes allows to construct a new program that achieves optimal temporal locality for any values of the parameters. Moreover, optimal spatial locality is obtained by transforming the memory layout as presented in our preceding work.

Keywords Cache memory, spatial and temporal locality, loop nests, program performance optimization, optimizing compiler, parameterized polyhedron.

1 Introduction

As the disparity between processor cycle times and main memory access times grows, performances of programs are greatly dependent on the way the cache memory is effectively used. The unit of data transfer between main memory and cache is a block. If the same data in the cache is reused successively, this is called *temporal locality*. On the other hand, since the unit of data in cache is a block, once the block is brought into the cache, any access to the data elements in the block will be a cache hit. This is called *spatial locality*. Typically, a cache hit takes one processor cycle, while a cache miss takes 8 to 32 cycles. The ideal scheme for all memory references is to hit the cache, however, since cache size is much smaller than main memory, once new data need to be brought in from main memory, some data in the cache have to be replaced. There will be a cache miss when the replaced data is accessed again afterwards. Therefore, high performance requires programs to possess *cache locality*, i.e., to reuse the data in the cache before it is replaced.

Most of the scientific and engineering applications contain loop nests that operate on large multi-dimensional arrays. Unfortunately, most of these loops are written without any special attention paid to cache memory performance. It has already been observed that compiler techniques are useful for optimizing locality. Programmers and compiler writers often attempt to modify the access patterns of a program so that the majority of accesses are made to the nearby memory. Several efforts have been aimed at iteration space transformations and scheduling techniques to improve locality [6, 7, 12, 11], and also at data transformations to improve spatial locality [5, 3, 4, 8, 2].

We presented in a recent work [2] a method dedicated to spatial locality optimization based on the computation of a new reference evaluation function. This approach is equivalent to transforming the memory layout. But in this first proposal, the application field is limited to loops containing one unique reference to a given array. In this work, we consider more general loops containing several different references to a same array. At our knowledge, such cases were never considered in other works at this time. The main difficulty is related to the temporal reuse occuring from these several references.

We propose a geometrical model that consists in classifying iterations relatively to their reference to a same data. This classification results in a partitioning of the iteration space by disjoint polytopes. Temporal locality is then obtained by scanning some of these sets and by merging the statements of the other ones. Spatial locality is also obtained by extending our method presented in [2] : data referenced by the merged statements are interleaved in memory. The mathematical tools we use are operations on polyhedral sets (unions, intersections, differences, affine transformations) and implemented in the polyhedral library *PolyLib* (http://icps.u-strasbg.fr/PolyLib). The final scanning loops are automatically generated by using the Fourier-Motzkin algorithm also implemented in the polyhedral library *PolyLib* [9].

At this time, the algorithm presented in this paper has the following limitations :

- any data can be accessed at most once through a given reference occuring in the considered loop,
- no loop-carried data dependence is allowed.
- our geometrical approach involves some other restrictions on the affine references. These are precisely explained in the following section.

The objective of our next developments is to overcome these limitations. Anyway, this paper is dedicated to introduce a new geometrical model of memory accesses for which some natural directions of extensions are given in conclusion. Moreover, the kind of programs considered at this time covers a very large field of concrete cases.

The paper is organized as follows. In section 2, the considered loops nests structure is defined precisely. Our geometrical model of memory accesses is described in section 3, giving the classification criteria of the iterations. In section 4, we show how optimal temporal and spatial locality is obtained, by merging statements, transforming data, and generating the new program. Finally, conclusions and future directions are given in section 5.

2 Context

We consider a parameterized loop nest of depth n where more than one affine reference is made to a same array at each iteration. These references can also contain parameters in the constant part of the involved affine functions. The iteration space is an $n-$dimensional convex polyhedron D where each integer point is associated to an iteration of the loop nest. The references have different access matrices R_1, R_2,..., R_q and offset vectors $r_1, r_2, \ldots, r_q$, where q is the number of accesses made in the loop. Hence, each reference $\mathcal{R}_\alpha$ is characterized by the couple $\mathcal{R}_\alpha = (R_\alpha, r_\alpha)$.

As said in the introduction, any reference made by the loop has to induce one unique access to a given data. It means that no temporal reuse is allowed for each reference by itself and that any access matrix R_α is an invertible matrix. On the other hand, temporal reuse occuring from several different references to a same array is considered. Let I, J be two different iterations. If I references the same data through reference $\mathcal{R}_\alpha$ as J through reference $\mathcal{R}_\beta$, then $R_\alpha I + r_\alpha = R_\beta J + r_\beta$. That is to say :

$$R_\beta^{-1} R_\alpha I + R_\beta^{-1} r_\alpha - R_\beta^{-1} r_\beta = J \qquad \Longleftrightarrow \qquad T_{\alpha\beta} I + t_{\alpha\beta} = J$$

with $T_{\alpha\beta} = R_\beta^{-1} \cdot R_\alpha$ and $t_{\alpha\beta} = R_\beta^{-1} r_\alpha - R_\beta^{-1} r_\beta$. Thus, for each point I of D accessing an array element through $\mathcal{R}_\alpha$, we can determine which point J accesses the same data through $\mathcal{R}_\beta$ by $J = T_{\alpha\beta} I + t_{\alpha\beta}$.

Conversely, since R_α and R_β are invertible, $T_{\alpha\beta}$ is invertible too, and we can determine, for each point J of D accessing a data through $\mathcal{R}_\beta$ which point I accesses the same data through $\mathcal{R}_\alpha$ by $I = T_{\alpha\beta}^{-1} J - T_{\alpha\beta}^{-1} t_{\alpha\beta}$.

We call $\mathcal{T}_{\alpha\beta} = (T_{\alpha\beta}, t_{\alpha\beta})$ the *temporal reuse relation* of references $\mathcal{R}_\alpha$ and $\mathcal{R}_\beta$.

In order to avoid some additional geometrical difficulties in this first proposal of the method, we have to ensure that $\mathcal{R}_\alpha$ and $\mathcal{R}_\beta$ span the same lattice of integer points. Hence, the access matrices must be such that $|det(R_\alpha)| = |det(R_\beta)|$ for any $\alpha, \beta \in [1..k]$, where $det(R)$ denotes the determinant of matrix R. Since $det(T_{\alpha\beta}) = det(R_\alpha)/det(R_\beta) = \pm 1$, $T_{\alpha\beta}$ is unimodular.

Loop-carried data-dependences are not allowed at this time by the method, since they would constrain the possible program transformations. This restriction, as the other ones stated above, is discussed in conclusion.

The paper is illustrated with the following example:

Example 1 Let us consider the following loop nest:

for $i = -N$ to N
  for $j = -N$ to N
    $X[i, j] = Y[i, i + j + 3] + Y[i + j, j]$

Two references are made to array Y:

$$\mathcal{R}_1 = R_1 I + r_1 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} I + \begin{pmatrix} 0 \\ 3 \end{pmatrix} \quad \mathcal{R}_2 = R_2 I + r_2 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} I + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

where $I = (i, j)^T$ is a point in the 2-dimensional iteration space $D = \{I \in \mathbb{Z}^2 | -N \leq i \leq N, -N \leq j \leq N\}$. The temporal reuse relation $\mathcal{T}_{12}$ is given by $T_{12} = R_2^{-1} \cdot R_1 = \begin{pmatrix} 0 & -1 \\ 1 & 1 \end{pmatrix}$ and $t_{12} = R_2^{-1} r_1 - R_2^{-1} r_2 = \begin{pmatrix} -3 \\ 3 \end{pmatrix}$.

For example, iteration $I = (8, -5)^T$ references the same array element through $Y[i, i+j+3]$ as iteration

$$J = \begin{pmatrix} 0 & -1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 8 \\ -5 \end{pmatrix} + \begin{pmatrix} -3 \\ 3 \end{pmatrix} = \begin{pmatrix} 2 \\ 6 \end{pmatrix}$$

through $Y[i + j, j]$.

$\square$

All the iterations will now be classified depending on the kind of temporal reuse that is induced by their references. This process is explained in the following section.

3 The polyhedral model of memory accesses

For the sake of clarity, we describe our model for only two references $\mathcal{R}_1$ and $\mathcal{R}_2$ to a same array. Then we explain how the model can be extended to any number of references. Since there is no possible confusion, $\mathcal{T}_{12}$, T_{12} and t_{12} will be denoted by $\mathcal{T}$, T and t in the following.

The iteration space is divided into four main iteration sets characterized by the following classification criteria :

- The iterations referencing through $\mathcal{R}_1$ a data also referenced by an other iteration through $\mathcal{R}_2$. Those are the iteration points having an image by $\mathcal{T}$ in the iteration space D. This iteration set is denoted by P_1. It defines a convex polytope:

$$P_1 = \{I \in D | TI + t \in D\}$$

- The iterations that are the images through $\mathcal{T}$ of the iterations in P_1. These are the iterations referencing through $\mathcal{R}_2$ a data also referenced by an other iteration through $\mathcal{R}_1$. This iteration set is denoted by P_2. It defines a convex polytope:

$$P_2 = \{I \in D | T^{-1}I - T^{-1}t \in D\}$$

- The iterations whose referenced data are never referenced by any other iteration. These are the iterations which are neither in P_1 nor in P_2. This set is denoted by L, and defines a union of convex polytopes $L = D - P_1 - P_2$.

We denote by C the set of iterations belonging simultaneously to P_1 and P_2, $C = P_1 \cap P_2$. We also define the set D_1 (respectively D_2) containing the points of P_1 (resp. P_2) that do not belong to P_2 (resp. P_1), $D_1 = P_1 - P_2 = P_1 - C$ (resp. $D_2 = P_2 - P_1 = P_2 - C$). D_1 (respectively D_2) is the set of iterations of P_1 (resp. P_2) referencing through $\mathcal{R}_2$ (resp. $\mathcal{R}_1$) a data that is never referenced by any other iteration of D.

Finally, iteration space D is partitioned into four disjoint sets : D_1, D_2, C and L. This partitioning process is schematized on figure 1.

Example 2 (continued)

The set P_1 is defined by:

$$\begin{aligned} P_1 &= \{(i,j)^T \in D | (-j-3, i+j)^T \in D\} \\ &= \{(i,j)^T \in \mathbb{Z}^2 | -N+3 \le i \le N, -N \le j \le N-6, -N \le i+j \le N\} \end{aligned}$$

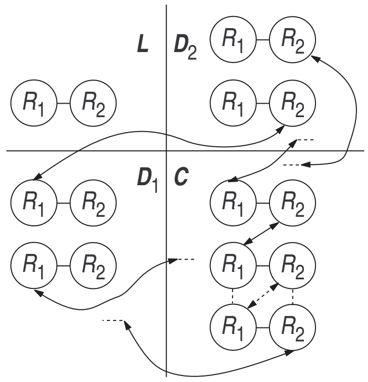

Figure 1: Schematization of the partitioning process where each line represents a reference to the same data.

Since $T^{-1} = \begin{pmatrix} 1 & 1 \\ -1 & 0 \end{pmatrix}$, P_2 is defined by :

$$
\begin{aligned}
P_2 &= \{(i,j)^T \in D | (i+j, -i-3)^T \in D\} \\
&= \{(i,j)^T \in \mathbb{Z}^2 | -N+3 \leq i \leq N-3, -N \leq j \leq N, -N+3 \leq i+j \leq N\}
\end{aligned}
$$

Hence, L, C, D_1 and D_2 are deduced :

$$
\begin{aligned}
L &= D - P_1 - P_2 \\
&= \{(i,j)^T \in \mathbb{Z}^2 | i \geq -N, j \geq -N, i+j \leq -N-4\} \\
&\cup \{(i,j)^T \in \mathbb{Z}^2 | i \leq N, j \leq N, i+j \geq N+1\} \\
&\cup \{(i,j)^T \in \mathbb{Z}^2 | i \leq N, i+j \leq N, i+j \geq N-2, i \geq N-2\}
\end{aligned}
$$

$$
\begin{aligned}
C &= P_1 \cap P_2 \\
&= \{(i,j)^T \in \mathbb{Z}^2 | -N \leq i \leq N-3, -N \leq j \leq N-3, -N \leq i+j \leq N-3\}
\end{aligned}
$$

$$
\begin{aligned}
D_1 &= P_1 - P_2 \\
&= \{(i,j)^T \in \mathbb{Z}^2 | N-2 \leq i \leq N, j \geq -N, i+j \leq N-3\} \\
&\cup \{(i,j)^T \in \mathbb{Z}^2 | i \geq -N, j \geq -N, -N-3 \leq i+j \leq N+1\}
\end{aligned}
$$

$$
\begin{aligned}
D_2 &= P_2 - P_1 \\
&= \{(i,j)^T \in \mathbb{Z}^2 | i \geq -N, N-2 \leq j \leq N, i+j \leq N\} \\
&\cup \{(i,j)^T \in \mathbb{Z}^2 | i \leq N-3, j \leq N, N-2 \leq i+j \leq N\}
\end{aligned}
$$

D_1, D_2, L and C are represented on figure 2 for $N = 8$.

□

Iterations referencing the same data are linked by relation $\mathcal{T}$: iteration I_1 references through $\mathcal{R}_1$ the same data as iteration $TI_1 + t = I_2$ through $\mathcal{R}_2$, if and only if $I_1 \in D_1$ or $I_1 \in C$ according to the definitions of D_1 and C. Moreover in such a case, it comes that $I_2 \in C$ or $I_2 \in D_2$. If $I_2 \in C$, then I_2 references through $\mathcal{R}_1$ another data that is also referenced by iteration $TI_2 + t = I_3$ through $\mathcal{R}_2$.

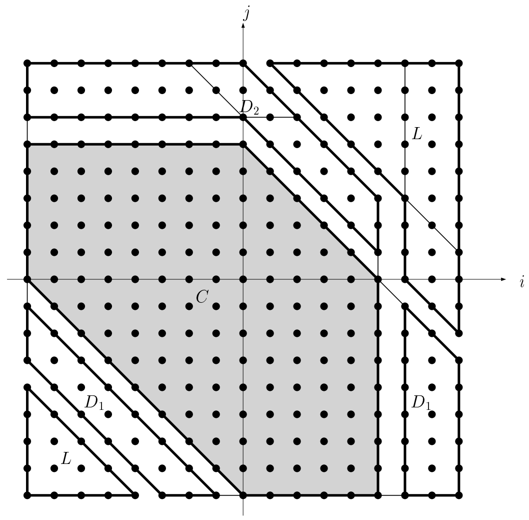

Figure 2: Partitioning the iteration space D by D_1, D_2, C and L.

In order to ensure temporal locality, iterations I_1, I_2 and I_3 should be computed consecutively. But the number of iterations that can be linked by $\mathcal{T}$ do vary. The next step consists now in defining sets of iterations characterized by the number of other iterations that are linked to it by $\mathcal{T}$. These sets are constructed as follows:

1) All the iterations of D_1 that are linked to iterations of D_2 by $\mathcal{T}$:

$$DC_1 = \{I \in D_1 | TI + t \in D_2\}$$

1') All the iterations of C that are linked to themselves by $\mathcal{T}$:

$$C_1 = \{I \in C | TI + t = I\}$$

2) All the iterations of D_1 that are linked to iterations of C, and these iterations are linked to iterations of D_2. In other words, these are the iterations of D_1 that are linked to iterations of D_2 by $\mathcal{T}^2$:

$$DC_2 = \{I \in D_1 | TI + t \in C, T^2 I + Tt + t \in D_2\}$$

2') All the iterations of C that are linked to other iterations of C, and these iterations are also linked to the first ones. Observe that the elements of C_1 have to be excluded. In other words, these are the iterations of C that are linked to themselves by $\mathcal{T}^2$:

$$C_2 = \{I \in C | TI + t \in C, T^2 I + Tt + t = I\} - C_1$$

...

k) All the iterations of D_1 that are linked to iterations of D_2 by $\mathcal{T}^k$:

$$DC_k = \{I \in D_1 | TI + t \in C, T^2I + Tt + t \in C, T^3I + T^2t + Tt + t \in C, \ldots, \\ T^kI + T^{k-1}t + \ldots Tt + t \in D_2\}$$

k') All the iterations of C that are linked to themselves by $\mathcal{T}^k$:

$$C_k = \{I \in C | TI+t \in C, T^2I+Tt+t \in C, \ldots, T^kI+T^{k-1}t+\ldots Tt+t = I\} - \bigcup_{1 \le q \le k-1} C_q$$

...

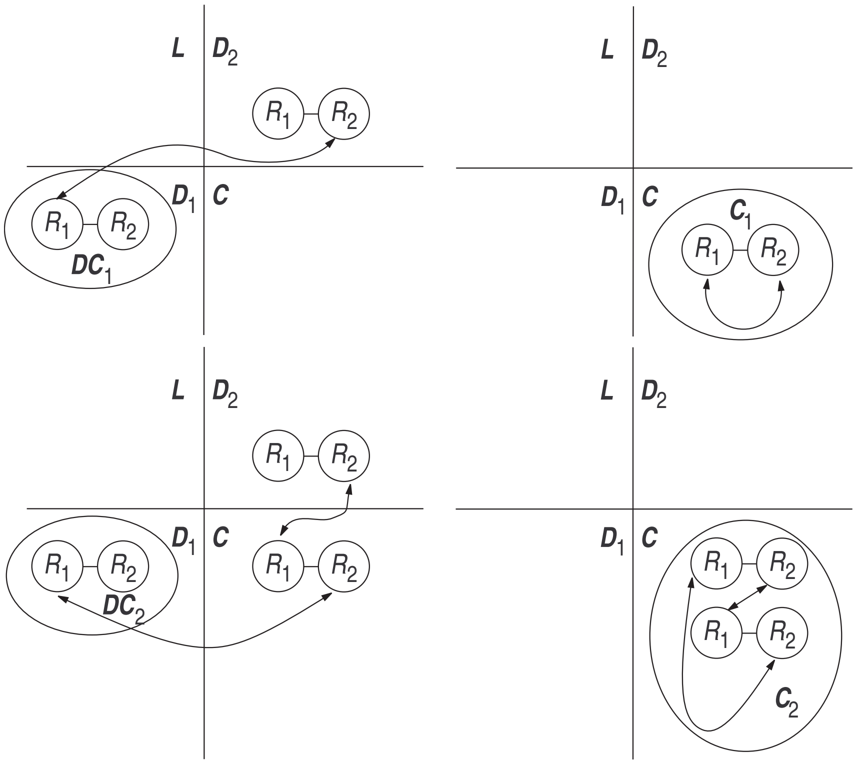

Figure 3: Definitions of sets DC_1, C_1, DC_2 and C_2.

The definitions of sets DC_1, C_1, DC_2 and C_2 are schematized on figure 3.

Sets DC_k and C_k are computed by applying the affine transformation defined by $\mathcal{T}$ to compositions of D_1, D_2 or C. Application of $\mathcal{T}$ to any set S is denoted by $\mathcal{T}(S)$ and to any point I by $\mathcal{T}(I)$. Hence, the sets defined above can also be defined as follows :

1) $DC_1 = D_1 \cap \mathcal{T}^{-1}(D_2)$

2) $DC_2 = D_1 \cap \mathcal{T}^{-1}(C) \cap \mathcal{T}^{-2}(D_2)$

2') $C_2 = C \cap \mathcal{T}^{-1}(C) \cap \{I \in C | T^2I + Tt + t = I\} - C_1$

...

k) $DC_k = D_1 \cap \mathcal{T}^{-1}(C) \cap \mathcal{T}^{-2}(C) \cap \ldots \cap \mathcal{T}^{-k}(C) \cap \mathcal{T}^{-k}(D_2)$

k') $C_k = C \cap \mathcal{T}^{-1}(C) \cap \mathcal{T}^{-2}(C) \cap \ldots \cap \mathcal{T}^{-k}(C)$
$\quad \cap \{I \in C | T^k I + T^{k-1}t + \ldots Tt + t = I\} - \bigcup_{1 \le q \le k-1} C_q$

...

Sets C_k define cycles of length k composed of points linked by $\mathcal{T}$. We partition any set C_k by sets $C_k^0, C_k^1, \ldots, C_k^{k-1}$ such that any set C_k^q contains one unique member of each cycle. The chosen partitioning criteria is the lexicographic order of the points in each cycle:

$$C_k^0 = \{I \in C_k | I \prec \mathcal{T}(I), I \prec \mathcal{T}^2(I), \ldots, I \prec \mathcal{T}^{k-1}(I)\}$$

where $\prec$ denotes the lexicographic order. Any set C_k^q, where $0 < q \le k-1$, is consequently determined by $C_k^q = \mathcal{T}^q(C_k^0)$.

Similarly, we define $DC_k^q = \mathcal{T}^q(DC_k)$ for any $0 \le q \le k$.

Example 3 (continued)

The iteration space D is fully partitioned for $k = 1$ to 6 by the following sets :

$$\begin{cases} DC_1^0 &= \{(i,j)^T \in \mathbb{Z}^2 | i+j \le N-3, i \le N, j \ge -N, i \ge N-2\} \\ DC_1^1 &= \{(i,j)^T \in \mathbb{Z}^2 | i+j \ge N-2, i \le N-3, i+j \le N, j \le N\} \end{cases}$$

$$\begin{cases} DC_3^0 &= \{(i,j)^T \in \mathbb{Z}^2 | i+j \ge -N-3, i+j \le -N-1, j \ge -N, j \le -N+2\} \\ DC_3^1 &= \{(i,j)^T \in \mathbb{Z}^2 | i \ge N-5, j \ge -N, i \le N-3, j \le -N+2\} \\ DC_3^2 &= \{(i,j)^T \in \mathbb{Z}^2 | i+j \ge N-5, i \le N-3, i+j \le N-3, i \ge +N-5\} \\ DC_3^3 &= \{(i,j)^T \in \mathbb{Z}^2 | j \ge N-2, i+j \le N-3, j \le N, i+j \ge N-5\} \end{cases}$$

$$\begin{cases} DC_4^0 &= \{(i,j)^T \in \mathbb{Z}^2 | i \ge -N, j \ge -N+3, i+j \ge -N-3, i+j \le -N-1\} \\ DC_4^1 &= \{(i,j)^T \in \mathbb{Z}^2 | i \le N-6, i+j \ge -N, j \le -N+2, j \ge -N\} \\ DC_4^2 &= \{(i,j)^T \in \mathbb{Z}^2 | i+j \le N-6, j \ge -N+3, i \ge N-5, i \le N-3\} \\ DC_4^3 &= \{(i,j)^T \in \mathbb{Z}^2 | j \le N-3, i \le N-6, i+j \ge N-5, i+j \le N-3\} \\ DC_4^4 &= \{(i,j)^T \in \mathbb{Z}^2 | i \ge -N, i+j \le N-6, j \ge N-2, j \le N\} \end{cases}$$

$$C_1^0 = \{(-3,0)^T\}$$

$$\begin{cases} C_6^0 &= \{(i,j)^T \in \mathbb{Z}^2 | i \ge -N, j \ge 0, i+j \le -4\} \\ C_6^1 &= \{(i,j)^T \in \mathbb{Z}^2 | i+j \ge -N, i \le -3, j \le -1\} \\ C_6^2 &= \{(i,j)^T \in \mathbb{Z}^2 | j \ge -N+3, i+j \le -3, i \ge -2\} \\ C_6^3 &= \{(i,j)^T \in \mathbb{Z}^2 | i \le N-6, j \le 0, i+j \ge -2\} \\ C_6^4 &= \{(i,j)^T \in \mathbb{Z}^2 | i+j \le N-6, i \ge -3, j \ge 1\} \\ C_6^5 &= \{(i,j)^T \in \mathbb{Z}^2 | j \le N-3, i+j \ge -3, i \le -4\} \end{cases}$$

$$\begin{aligned} L &= \{(i,j)^T \in \mathbb{Z}^2 | i \ge -N, j \ge -N, i+j \le -N-4\} \\ &\cup \{(i,j)^T \in \mathbb{Z}^2 | i \le N, j \le N, i+j \ge N+1\} \\ &\cup \{(i,j)^T \in \mathbb{Z}^2 | i \le N, i+j \le N, i+j \ge N-2, i \ge N-2\} \end{aligned}$$

This partitioning is shown on figure 4 for $N = 8$. □

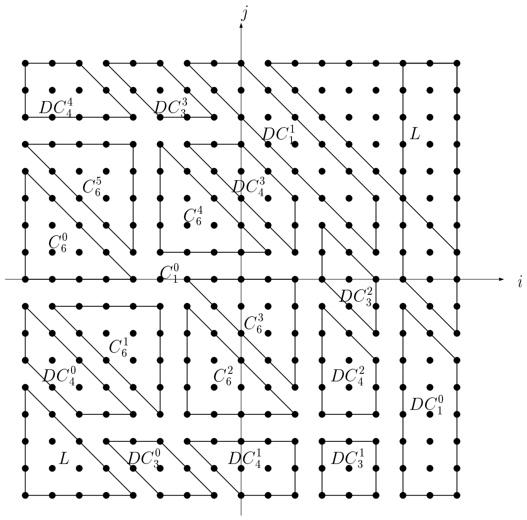

Figure 4: The final partitioning of D

The main difficulty of this partitioning process is to determine the maximum value of k so that D is completely partitioned. Two cases can occur depending on the existence of a lowest value $kmax > 0$ such that $\mathcal{T}^{kmax} = \mathbb{I}$, where $\mathbb{I}$ denotes the identity matrix:

1. if such a value exists, the set C_{kmax} will contain all the remaining points of C at the $kmax$-th step of the partitioning process. Hence D will be completely partitioned.

2. if such a value does not exist, we affect $kmax$ a high value. Our partitioning process stops as soon as C has been detected as being empty or $kmax$ has been reached. But since the iteration space D can be parameterized, the number of necessary partitioning steps can also be parameterized and there is no convenient value of $kmax$. In such a case, there will be some remaining iterations in D_1, D_2 and C that are not considered by our locality optimization process and $kmax$ can be seen as an optimization level.

We first have to determine if there exists a lowest value $kmax > 0$ such that $\mathcal{T}^{kmax} = \mathbb{I}$. Our method proceeds as follows.

Since $\mathcal{T}(I) = TI + t$, we write $\mathcal{T}$ in the homogeneous form of a single matrix:

$$\mathcal{T} = \begin{pmatrix} & T & & t \\ 0 & \dots & 0 & 1 \end{pmatrix}$$

Then, by computing the eigenvalues and eigenvectors of this matrix $\mathcal{T}$, we can write $\mathcal{T} = P^{-1} \cdot A \cdot P$, where the columns of P are the eigenvectors and A is the diagonal matrix of the eigenvalues. Since $\mathcal{T}^q = P^{-1} \cdot A^q \cdot P$ for any $q > 0$, we resolve the equation $A^{kmax} = \mathbb{I}$.

The eigenvalues v_q can be complex numbers or rational numbers. Hence, the rational eigenvalues must be equal to 1 or -1 in order to get a integer solution $kmax$. Moreover, if at least one eigenvalue is equal to -1 then $kmax$ must be even.

For the complex eigenvalues, we use their polar coordinates (ρ_q, θ_q): any complex number $a + bi$ can be written as $\rho \cos\theta + i \sin\theta = \rho\ e^{i\theta}$ with $\rho \geq 0$ and $-\pi < \theta < \pi$. Since $1 = 1 + 0i = 1\ e^{i2k\pi}$, $k \in \mathbb{Z}$, we have to resolve for each eigenvalue v_q:

$$(\rho_q\ e^{i\theta_q})^{kmax} = 1\ e^{i2k_q\pi} \Leftrightarrow \left\{ \begin{array}{l} \rho_q = 1 \\ i\theta_q kmax = i2k_q\pi \end{array} \right. \Leftrightarrow \left\{ \begin{array}{l} \rho_q = 1 \\ \theta_q kmax = 2k_q\pi \end{array} \right. \Leftrightarrow \left\{ \begin{array}{l} \rho_q = 1 \\ kmax = 2k_q\frac{\pi}{\theta_q} \end{array} \right.$$

Hence, there may be a solution if the polar coordinates of all the eigenvalues are such that $\rho_q = 1$. Moreover, there is at least one integer solution $kmax$ if any θ_q is of the form $\frac{\alpha_q}{\beta_q}\pi$, with $\alpha_p, \beta_q \in \mathbb{Z}$. If so, $kmax$ is defined as being equal to any $2k_q\frac{\beta_q}{\alpha_q}$. Since k_q has to be chosen such that $2k_q$ is a multiple of α_q to get an integer solution, the lowest solution $kmax$ is equal to the lower common multiple of the $\frac{2k_q}{\alpha_q}\beta_q$'s, $lcm_q(\frac{2k_q}{\alpha_q}\beta_q)$, where $k_q = \alpha_q$ if $\alpha_q\ mod\ 2 \neq 0$ and $k_q = \alpha_q/2$ if $\alpha_q\ mod\ 2 = 0$.

Example 4 (continued)

In the homogeneous form, $\mathcal{T}$ can be written as the matrix:

$$\mathcal{T} = \begin{pmatrix} 0 & -1 & -3 \\ 1 & 1 & 3 \\ 0 & 0 & 1 \end{pmatrix}$$

We want to resolve $\mathcal{T}^{kmax} = \mathbb{I}$. We first compute the eigenvalues of $\mathcal{T}$: $1, \frac{1}{2} + \frac{1}{2}i\sqrt{3}, \frac{1}{2} - \frac{1}{2}i\sqrt{3}$. The polar coordinates of the two complex ones are $(1, \pi/3)$, $(1, -\pi/3)$. Since any $\rho_q = 1$ and since any θ_q is in a convenient form, there exists a lowest value $kmax$ which is equal to the lower common multiple of $\frac{2k_1}{1}3$ and $\frac{2k_2}{-1}3$. Since $1\ mod\ 2 = 1$ and $-1\ mod\ 2 = 1$, we have $k_1 = 1$ and $k_2 = -1$. Finally, $kmax = lcm(6, 6) = 6$. One can verify that $\mathcal{T}^6 = \mathbb{I}$.

Hence, our partitioning process stops at the 6-th step where D is completely partitioned. □

Once all the sets DC_k^q and C_k^q have been defined, we generate the final program by scanning by loops two representative sets of each partitioning step k, and by merging into one iteration the statements that are linked through $\mathcal{T}$. This process is explained in detail in the following section.

This geometrical model is extensible to more than two references. Since we consider a very general case, the sets where temporal reuse happens are becoming exponentially more highly structured as the number of references is increasing. Hence, trying to fully optimize such a system may become very complex for great values of $kmax$. For example, if we consider a loop with three references $Y[\mathcal{R}_1(I)], Y[\mathcal{R}_2(I)]$ and $Y[\mathcal{R}_3(I)]$, these three references will define three temporal reuse relations $\mathcal{T}_{12}, \mathcal{T}_{13}$ and $\mathcal{T}_{23}$. The result of our partitioning process would consist in sets S_k defined by a non-oriented and connex graph G_k of k nodes, whose degrees d are such that $1 \leq d \leq 3$ (except for the set L where any $d = 0$). Each edge is associated to one of the three temporal reuse relations $\mathcal{T}_{12}$, $\mathcal{T}_{13}$ and $\mathcal{T}_{23}$ and each node is associated to iterations I. Since several sets of iterations would match such a graph, any set S_k^q is defined as the set of all iterations that match the q-th node.

4 Generation of the final program

4.1 Optimizing temporal locality

For any partitioning step k, we consider the sets DC_k^0, DC_k^1, ..., DC_k^k and C_k^0, C_k^1, ..., C_k^{k-1}. By chosing for example DC_k^0 (resp. C_k^0) as the representative set, we generate a scanning loop for this set by using the Fourier-Motzkin algorithm implemented in the polyhedral library *PolyLib*. The instructions put in the resulting loop are those associated to all the sets DC_k^q (resp. C_k^q). A given iteration I of this loop computes the statements associated to points I, $\mathcal{T}(I)$, $\mathcal{T}^2(I)$, ... and temporal locality occurs between any two successive statements.

Example 5 (continued)

By generating scanning loops for the sets DC_1^0, DC_3^0, DC_4^0, C_1^0, C_6^0 and L shown on figure 5 for $N = 8$, and by merging the convenient statements, we obtain the program shown on figure 6. Observe that the same references occur between successive statements.

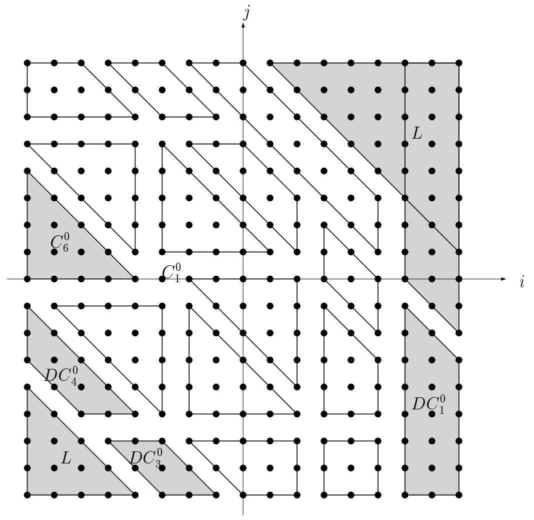

Figure 5: The scanned sets DC_1^0, DC_3^0, DC_4^0, C_1^0, C_6^0 and L.

□

Let us see how the final program could be generated in the case of more than two references in the original loop. Starting from the partitioning result described at the end of section 3, the final program would be constructed by scanning sets S_k^0 by loops and by merging all iterations defined by the sets S_k. This merging process would be more complicated: since the iterations are associated to different paths of G_k, they should be interleaved cleverly so that temporal locality would be ensured.

4.2 Optimizing spatial locality

In order to achieve spatial locality for all these generated loops, we extend our method presented in [2]. Its principle is to provide a new array reference evaluation function such that the order of the data in memory matches exactly the order in which they are referenced by the loop. Hence, any block loaded in the cache is completely accessed until it is replaced and the corresponding data will no more be reused in the future.

This function is determined by counting the number of iterations occuring before a given reference. The result is an Ehrhart polynomial [1] whose variables are the indices of the considered array. Hence, this polynomial defines a new reference function and the considered array is transformed into a one-dimensional array.

The best use of this technique is to implement it in a compiler in order to define a new array reference evaluation process. However, it can also be written directly in the source code: during the execution of the loop, any successive references are made to sucessively stored data, the reference function simply consists in adding one to the previous reference index value, starting from a given base address for the considered array.

In order to be used in the source code resulting from our temporal locality optimization, this method has to be extended: since several statements occurs in the loops, a polynomial must be computed for each different reference. Moreover, these polynomials have to be transformed such that the referenced data are interleaved in memory:

- if the loop results from sets $C_k^{\prime q}$, iteration I is of the form:

  $Y[\mathcal{R}_0(I)] \ldots Y[\mathcal{R}_{k-1}(I)]$

  $Y[\mathcal{R}_1(I)] \ldots Y[\mathcal{R}_0(I)]$

  $Y[\mathcal{R}_2(I)] \ldots Y[\mathcal{R}_1(I)]$

  ...

  $Y[\mathcal{R}_{k-1}(I)] \ldots Y[\mathcal{R}_{k-2}(I)]$

  A polynomial is computed for each reference $\mathcal{R}_0, \ldots, \mathcal{R}_{k-1}$. The results are polynomials $EP_0, \ldots, EP_{k-1}$. At this step and for a given I, the evaluation of all the polynomials will result in the same value. In order to obtain k sucessive values, we rather consider the polynomials $kEP_0 - k + 1, kEP_1 - k + 2, \ldots, kEP_{k-1}$. Iteration I can then be rewritten as:

  $Y[r] \ldots Y[r + k - 1]$

  $Y[r + 1] \ldots Y[r]$

  $Y[r + 2] \ldots Y[r + 1]$

  ...

  $Y[r + k - 1] \ldots Y[r + k - 2]$

  $r = r + k$

  where r is initialized at the base address of array Y.

- if the loop results from sets $DC_k^{\prime q}$, Iteration I is of the form:

  $Y[\mathcal{R}_1(I)] \ldots Y[\mathcal{R}_0(I)]$

  $Y[\mathcal{R}_2(I)] \ldots Y[\mathcal{R}_1(I)]$

  $Y[\mathcal{R}_3(I)] \ldots Y[\mathcal{R}_2(I)]$

...

$Y[\mathcal{R}_k(I)]\ldots Y[\mathcal{R}_{k-1}(I)]$

No temporal reuse occurs for reference $Y[\mathcal{R}_0(I)]$. Hence, a polynomial has also to be computed for this reference. We consider the polynomials $(k+1)EP_0 - k$, $(k+1)EP_1 - k + 1, \ldots, (k+1)EP_k$. Iteration I can then be rewritten as:

$Y[r+1]\ldots Y[r]$

$Y[r+2]\ldots Y[r+1]$

$Y[r+3]\ldots Y[r+2]$

...

$Y[r+k]\ldots Y[r+k-1]$

$r = r + k + 1$

- if the loop results from the set L, Iteration I is of the form:

  $Y[\mathcal{R}_1(I)]\ldots Y[\mathcal{R}_2(I)]$

  A polynomial is computed for each reference ; $2EP_1 - 1$ and $2EP_2$ are considered. Iteration I can then be rewritten as:

  $Y[r]\ldots Y[r+1]$

  $r = r + 2$

Example 6 (continued)

For each loop shown on figure 6, the convenient polynomials are computed. Consequently, Y is now a one-dimensional array whose elements are referenced in the order of their storage by the final program shown on figures 7 and 8.

We now present performance results to demonstrate the impact of our locality optimization method. Our experimental platform is an 32-processor SGI Origin 2000 at the University Louis Pasteur in Strasbourg France. This machine uses 195 MHz R10000 processors, each with a 32 KB L1 data cache and a 4 MB L2 unified cache. The processors can fetch and decode four instructions per cycle and can run them on five pipelined functionnal units. Both caches are two-way associative and nonblocking. Up to four outstanding misses from the combined two-levels of cache are supported. The R10000 processor dynamically schedules instructions whose operands are available in order to hide the latency of cache misses. For the L1 cache hits, the latency is 2 to 3 cycles; and for the L1 misses that hit in the L2 cache, the latency is 8 to 10 cycles. The main memory accesses take 62 cycles.

We compare the execution times on one processor between the initial program and our final program by making N vary from 300 to 4000, corresponding to matrices of sizes 600×600 to 8000×8000 of floating point numbers. Results are shown on figure 9, showing an execution time of 58.60 seconds for the original program and a execution time of 11.55 seconds for our final program with $N = 4000$. The speedup tends to be greater than 5 between both sequential programs.

□

5 Conclusion

We proposed in this work a new model to fully optimize both spatial and temporal cache memory locality of parameterized loop nests with affine references to arrays. Our algorithm is based on a geometric representation of the program, and manipulates parameterized polytopes. Intersections, unions, differences, affine transformations, and loop generation are performed using the *Polylib* (http://icps.u-strasbg.fr/PolyLib). We have developed an implementation of the domain splitting algorithm presented in section 3, which can be downloaded at http://icps.u-strasbg.fr/PolyLib/Split. The figures presented in this paper are screenshots of the resulting domains, using the *PolyLib* tool *VisuDomain*.

The restrictions of the current algorithm are the following. Firstly, the access functions of the data have to be affine and bijective functions of the loop indices and the parameters. This restriction could be removed in the future by considering sets of points referencing a same data through a given reference, and by linking such sets by a temporal reuse relation. Secondly, the temporal reuse relation $\mathcal{T}$ has to be unimodular in order for the linked iterations to touch the same lattice of integer points in our geometrical model. This constraint will eventually be removed once the geometrical tools to manipulate $\mathbb{Z}$-polyhedra are available. Union and difference are non-trivial operations, but some works have already been started in this direction [10].

Moreover at this time, we did not take into account the data dependencies induced by the loop nest in order to get an unconstrained code generation: if there are loop-carried data dependences, the resulting code has to verify the causal constraints in order to be valid. This would constrain the partitioning, the scanning directions, and the loop schedule. But it also may yield impossible loop generation as presented in section 4, because of the unrolling/merging step of our algorithm. In this case, we can still transform the program for spatial optimization, and perform temporal locality wherever possible.

We have shown that our method allows to greatly improve the performance of even simple programs. In the proposed example, we generate a sequential program which is longer than the original one, but running five times faster. This shows that many codes may be appreciably optimized for no cost at all compared to investing in expensive parallel calculators. Moreover, our locality optimization framework can also be used for parallelizing programs, since it yields a minimum number of communications between processors.

References

[1] Ph. Clauss and V. Loechner. Parametric Analysis of Polyhedral Iteration Spaces. *Journal of VLSI Signal Processing*, Vol. 19, No. 2, Kluwer Academic Pub., July 1998.

[2] Ph. Clauss et B. Meister. Automatic Memory Layout Transformations to Optimize Spatial Locality in Parameterized Loop Nests. *4th Annual Workshop on Interaction between Compilers and Computer Architectures, INTERACT-4*, Toulouse, France, january 2000.

[3] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Optimizing Spatial Locality in Loop Nests using Linear Algebra. *Proc. 7th Int. Workshop on Compilers for Parallel Computers*, Sweden, June 1998.

[4] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A Hyperplane Based Approach for Optimizing Spatial Locality in Loop Nests. *Proc. 11th ACM Int. Conf. on Supercomputing*, Melbourne, Australia, July 1998.

[5] S.-T. Leung et J. Zahorjan. Optimizing data locality by array restructuring. University of Washington, Department of Computer Science and Engineering. Technical Report 95-09-01, 1995.

[6] W. Li. *Compiling for NUMA parallel machines*. Ph.D. dissertation, Dept. Computer Science, Cornell University, Ithaca, NY, 1993.

[7] K. McKinley, S. Carr and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 1996.

[8] M.F.P. O'Boyle et P.M.W. Knijnenburg. Nonsingular Data Transformations: Definition, Validity, and Applications. *Int. J. of Parallel Programming*, Vol. 27, No. 3, pp. 131-159, juin 1999.

[9] F. Quilleré, S. Rajopadhye and D. Wilde. Generation Of Efficient Nested Loops From Polyhedra. To appear in *Int. J. of Parallel Programming*, 2000.

[10] P. Quinton, S. Rajopadhye and Tanguy Risset. On Manipulating Z-polyhedra using a Canonical Representation. *Parallel Processing Letters*, 1997.

[11] M. Wolf and M. Lam. A data locality optimizing algorithm. *Proc. ACM SIGPLAN 91 Conf. Programming Language Design and Implementation*, pages 30-44, Toronto, Ont., June 1991.

[12] M. Wolf. *Improving locality and parallelism in nested loops*. Ph.D. dissertation, Dept. Computer Science, Stanford University, CSL-TR-92-538, 1992.

```
/* DC_1 */
for i = N − 2 to N
    for j = −N to −i + N − 3
        X[i, j] = Y[i, i + j + 3] + Y[i + j, j]
        X[−j − 3, i + j + 3] = Y[−j + 3, i + 3] + Y[i, i + j + 3]

/* C_1 */
X[−3, 0] = Y[−3, 0] + Y[−3, 0]

/* DC_3 */
for i = −5 to −1
    for j = max(−i − N − 3, −N) to min(−N + 2, −i − N − 1)
        X[i, j] = Y[i, i + j + 3] + Y[i + j, j]
        X[−j − 3, i + j + 3] = Y[−j − 3, i + 3] + Y[i, i + j + 3]
        X[−i − j − 6, i + 3] = Y[−i − j − 6, −j] + Y[−j − 3, i + 3]

/* DC_4 */
for i = −N to −4
    for j = max(−N + 3, −i − N − 3) to −i − N − 1
        X[i, j] = Y[i, i + j + 3] + Y[i + j, j]
        X[−j − 3, i + j + 3] = Y[−j − 3, i + 3] + Y[i, i + j + 3]
        X[−i − j − 6, i + 3] = Y[−i − j − 6, −j] + Y[−j − 3, i + 3]
        X[−i − 6, −j] = Y[−i − 6, −i − j − 3] + Y[−i − j − 6, −j]

/* C_6 */
for i = −N to −4
    for j = 0 to −i − 4
        X[i, j] = Y[i, i + j + 3] + Y[i + j, j]
        X[−j − 3, i + j + 3] = Y[−j − 3, i + 3] + Y[i, i + j + 3]
        X[−i − j − 6, i + 3] = Y[−i − j − 6, −j] + Y[−j − 3, i + 3]
        X[−i − 6, −j] = Y[−i − 6, −i − j − 3] + Y[−i − j − 6, −j]
        X[j − 3, −i − j − 3] = Y[j − 3, −i − 3] + Y[−i − 6, −i − j − 3]
        X[i + j, −i − 3] = Y[i + j, j] + Y[j − 3, −i − 3]

/* L (three loops) */
for i = −N to −4
    for j = −N to −i − N − 4
        X[i, j] = Y[i, i + j + 3] + Y[i + j, j]
for i = 1 to N
    for j = −i + N + 1 to N
        X[i, j] = Y[i, i + j + 3] + Y[i + j, j]
for i = N − 2 to N
    for j = −i + N − 2 to −i + N
        X[i, j] = Y[i, i + j + 3] + Y[i + j, j]
```

Figure 6: The program after temporal locality optimization.

```
rY = 1
rX = 1
/* DC_1 */
for i = N − 2 to N
    for j = −N to −i + N − 3
        X[rX] = Y[rY + 1] + Y[rY]
        X[rX + 1] = Y[rY + 2] + Y[rY + 1]
        rX = rX + 2
        rY = rY + 3

/* C_1 */
X[rX] = Y[rY] + Y[rY]
rX = rX + 1
rY = rY + 1

/* DC_3 */
for i = −5 to −1
    for j = max(−i − N − 3, −N) to min(−N + 2, −i − N − 1)
        X[rX] = Y[rY + 1] + Y[rY]
        X[rX + 1] = Y[rY + 2] + Y[rY + 1]
        X[rX + 2] = Y[rY + 3] + Y[rY + 2]
        rX = rX + 3
        rY = rY + 4

/* DC_4 */
for i = −N to −4
    for j = max(−N + 3, −i − N − 3) to −i − N − 1
        X[rX] = Y[rY + 1] + Y[rY]
        X[rX + 1] = Y[rY + 2] + Y[rY + 1]
        X[rX + 2] = Y[rY + 3] + Y[rY + 2]
        X[rX + 3] = Y[rY + 4] + Y[rY + 3]
        rX = rX + 4
        rY = rY + 5

/* C_6 */
for i = −N to −4
    for j = 0 to −i − 4
        X[rX] = Y[rY] + Y[rY + 5]
        X[rX + 1] = Y[rY + 1] + Y[rY]
        X[rX + 2] = Y[rY + 2] + Y[rY + 1]
        X[rX + 3] = Y[rY + 3] + Y[rY + 2]
        X[rX + 4] = Y[rY + 4] + Y[rY + 3]
        X[rX + 5] = Y[rY + 5] + Y[rY + 4]
        rX = rX + 6
        rY = rY + 6
```

Figure 7: The program after temporal and spatial locality optimizations.

$/*\ L$ (three loops) $*/$
for $i = -N$ to -4
    for $j = -N$ to $-i - N - 4$
        $X[rX] = Y[rY] + Y[rY + 1]$
        $rX = rX + 1$
        $rY = rY + 2$
for i $= 1$ to N
    for $j = -i + N + 1$ to N
        $X[rX] = Y[rY] + Y[rY + 1]$
        $rX = rX + 1$
        $rY = rY + 2$
for $i = N - 2$ to N
    for $j = -i + N - 2$ to $-i + N$
        $X[rX] = Y[rY] + Y[rY + 1]$
        $rX = rX + 1$
        $rY = rY + 2$

Figure 8: Figure 7 continued.

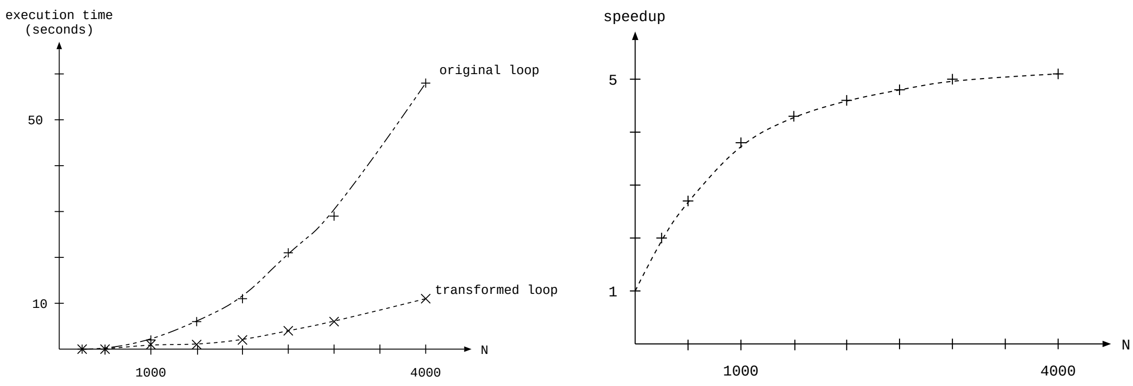

Figure 9: Execution times for the initial and final programs and the resulting speedup.