

Feasibility of Incremental Translation

Sven Verdoolaege *Francky Catthoor*
Maurice Bruynooghe *Gerda Janssens*

Report CW 348, October 2002



Katholieke Universiteit Leuven
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Feasibility of Incremental Translation

Sven Verdoolaege *Francky Catthoor*
Maurice Bruynooghe *Gerda Janssens*

Report CW 348, October 2002

Department of Computer Science, K.U.Leuven

Abstract

Affine loop transformations, where the iteration domain of each statement is transformed by an affine function, have been used by several authors for program optimization. Some have proposed to apply an additional linear transformation on the combined iteration domains of all statements after the affine transformation, called the ordering phase. The affine transformation itself can, in principle, be decomposed into a linear transformation and a translation.

In this paper, we show that it is indeed possible to apply linear transformation, translation and ordering successively and that the translation step can even be performed incrementally. However, we also show that the complexity involved in ensuring a valid translation is much higher than in the absence of a subsequent ordering phase. In the latter case, translation is equivalent to loop fusion with loop shifting. We also discuss several cost functions to be used in a translation step in the presence of an ordering phase and some implications of loop fusion on code generation.

Keywords : polyhedral model, loop transformations, loop fusion

CR Subject Classification : D.3.4 Optimization

Contents

1	Introduction	3
2	One-dimensional example	4
3	Problem formulation	8
4	Feasibility of translation	16
5	Two-dimensional example	24
6	The ordering phase	28
6.1	The ordering of the phases	28
6.2	Loop fusion	30
6.3	Linear transformation	31
6.4	Multi-dimensional ordering	33
6.5	No ordering substep	34
7	Optimality	37
7.1	After ordering	37
7.2	Before ordering	40
7.2.1	Distance length	40
7.2.2	Deviation from dependence cone	40
7.2.3	Deviation from ideal distance vector	41
7.2.4	Conclusion	44
8	Strategy	44
9	Code Generation	45
10	Related Work	51
11	Conclusions and Future Work	52

List of Algorithms

1	Incremental translation	13
2	Combining two nodes	14

Listings

1	Target program	34
2	Program with bad locality	37
3	Program with good locality	38
4	Program with a loop-independent dependence	45
5	Non-overlapping iteration domains	46
6	Minimally overlapping iteration domains	47
7	An ALPHA program for Durbin's algorithm	48
8	Durbin after fusion without preprocessing	49
9	Durbin after fusion with preprocessing	50
10	Program resulting in overlapping domains in outer dimensions	50

List of Figures

1	Example program with dependences	5
2	Combining C and D	6
3	Combining C and D (with vertical displacement)	6
4	Combining A and B	7
5	Combining C and B	7
6	Initial dependence graph for the example in Figure 1	10
7	Decomposition of translated distance vectors	10
8	Dependence cone and valid ordering polyhedron	11
9	Comparison between a cone and a dependence cone.	12
10	One iteration of Algorithm 1	14
11	Invalid positions for $\vec{\alpha}_{C,D}$	17
12	Pairing off two circuits	18
13	Decomposition of a circuit containing a pseudo-edge	19
14	Illustration of the proof of lemma 6	22
15	Initial dependence graph G_0	25
16	Invalid positions for $\vec{\alpha}_{C,D}$	26
17	Graph with C and D combined	26
18	Invalid positions for $\vec{\alpha}_{A,B}$	26
19	Graph with A and B combined	27
20	Invalid positions for $\vec{\alpha}_{AB,CD}$	27
21	Graph with AB and CD combined	27
22	Complete translation	28
23	Comparison of translation before or after ordering	29
24	Locality in space vs. locality in time.	30
25	Equivalent placement for ordering vector $\vec{\pi}^T = [1 \ 0]$	32
26	Initial dependence graph with dependence polytopes on the left and with minimal distance vectors on the right (after reverse).	35
27	Intermediate dependence graphs	35
28	Translated dependence graph	36
29	Complete fusion	36
30	Optimizing locality	38
31	The function $ \cdot _+$	40
32	Locality cost function in each dimension for each dependence.	40
33	Deviation from the dependence cone	41
34	Dependence graph of the program in Listing 4	45
35	Non-overlapping iteration domains	46
36	Minimally overlapping iteration domains	46
37	Overlapping domains in outer dimensions	50

Feasibility of Incremental Translation

Sven Verdoolaege Francky Catthoor*
Maurice Bruynooghe Gerda Janssens

October 2002

1 Introduction

Loop transformations have been widely used to optimize the execution time and the memory usage of programs. In general, a loop transformation modifies the order in which loop iterations and statements within a loop body are executed. One way to perform this reordering is to define a schedule, assigning an execution time to each iteration of each statement in the program.

An example of this technique is affine-by-statement scheduling (Darte and Robert 1992; Feautrier 1992; Lim and Lam 1997), where each statement is scheduled by a (piece-wise) affine function θ that maps the iterations of that statement to time:

$$\theta_X : \vec{i} \mapsto \vec{c}_X^T \vec{i} + c_X, \quad (1)$$

where \vec{i} is a vector in the iterations space of statement X , i.e., it has the iterator values of the loops surrounding the statement as elements. Alternatively, van Swaaij et al. (1992) propose to split the scheduling operation into a placement step, mapping the sets of iteration vectors for all statements to a common iteration space and an ordering step, defining an *order*, which is a linear schedule in that space. In this way, an affine schedule is decomposed into an affine transformation

$$\vec{i} \mapsto A_X \vec{i} + \vec{a}_X \quad (2)$$

for each statement X and a common ordering vector $\vec{\pi}$. The resulting schedule for statement X is then

$$\theta_X : \vec{i} \mapsto \vec{\pi}^T A_X \vec{i} + \vec{\pi}^T \vec{a}_X, \quad (3)$$

i.e.,

$$\vec{c}_X^T = \vec{\pi}^T A_X \quad \text{and} \quad c_X = \vec{\pi}^T \vec{a}_X. \quad (4)$$

Although the latter technique requires the determination of a lot more coefficients, it may still be preferable if it allows for an incremental search for the different mappings.

The placement step of the second technique can be split up further into a linear transformation step, determining A_X , and a translation step determining \vec{a}_X . This split was already effectively made by van Swaaij (1992), and more explicitly so by Danckaert (2001). While techniques for finding good linear mappings are discussed by Verdoolaege et al. (2001), this document focuses on the translation step. More specifically, the main topic is the proof of the feasibility of incrementally finding valid translation vectors.

The main difficulty in finding valid translation vectors or a valid schedule in general is the existence of dependences between two iterations (\vec{i}_1 and \vec{i}_2)

*IMEC, also prof. at Katholieke Universiteit Leuven

of the same or different statements, since the depending iteration (\vec{i}_2) should be scheduled after the iteration on which it depends (\vec{i}_1). Such a (dynamic) dependence exists if the two access the same memory element, one through writing and the other through reading, and is generally written as $\vec{i}_1 \delta \vec{i}_2$. If a dynamic dependence exists between two iterations of two statements, we will say that the two statements exhibit a static dependence.

A dependence in which the first access is a write and the second a read is called a flow or true dependence. If the program is in single-assignment form this is the only kind of dependence that can exist. Although the theory equally applies to output and anti-dependences, which are write-write and read-write combinations respectively, since they induce the same ordering constraint, we prefer them to be removed, since this tends to lead to better solutions. Transforming general programs into single assignment code typically involves the introduction of some storage overhead, but this overhead can be effectively removed again after the loop transformations even with additional memory size gains (De Greef et al. 1997; Quilleré and Rajopadhye 2000; Tronçon et al. 2002).

In the next section, we apply affine-by-statement scheduling, i.e., without an ordering step, but with a split into a linear step and a translation step, on an example program, which shows that there are cases in which a translation step cannot find a valid solution. In Section 3 we define the concept of a valid translation in the context of an approach with an additional ordering step and we provide an incremental algorithm. The conditions under which a valid translation exists and the set of valid choices during each step of the algorithm are proven in Section 4. We subsequently apply the algorithm in Section 5 to the same example of Section 2. In Section 6 we then consider the optimal positioning of the ordering step with respect to the other two steps, showing that ensuring a valid translation is more cumbersome with a subsequent ordering step than it is with a preceding ordering step and that a slightly adapted version of translation after ordering is equivalent to loop fusion with loop shifting. In Section 7 we briefly discuss some possible locality related cost functions, both in case of a subsequent and a preceding ordering step and we consider ways of combining nodes in the incremental algorithm in Section 8. Section 9 is devoted to some implications of loop fusion on code generation. Finally, we compare with related work in Section 10 and conclude in Section 11.

2 One-dimensional example

By way of introduction, we will apply an incremental translation step on a one-dimensional example, as part of a technique similar to affine-by-statement scheduling, i.e., without an additional ordering step, but with a split into a linear part and a translation. During the construction of the schedules, we will also consider equivalent combinations of an affine mapping and an ordering vector, based on (4), because this is easier to visualize and because the combination illustrates the relation between the two approaches.

As explained in the previous section, the main constraint on a schedule is that dependences should be honored. If a dependence exists between an iteration \vec{i}_1 of a loop A (with schedule θ_A) and an iteration \vec{i}_2 of a loop B (with schedule θ_B), i.e., \vec{i}_1 writes an element read by \vec{i}_2 , then the latter has to be scheduled after the first, i.e., the time delay between the two should be at least one:

$$(\vec{c}_B^T \vec{i}_2 + c_B) - (\vec{c}_A^T \vec{i}_1 + c_A) \geq 1 \quad \text{if } \vec{i}_1 \delta \vec{i}_2.$$

Consider the program on the left side of Figure 1. It is written in an applicative language and cannot be executed in the order in which it is written down.

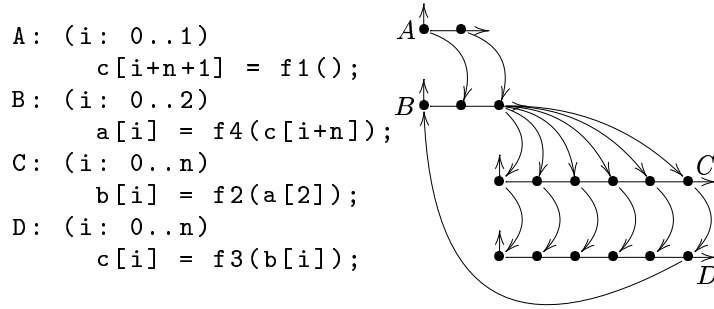


Figure 1: Example program with dependences

This is clear from the graph on the right of the figure, which shows the individual dependences between iterations: the first iteration of the B-loop depends on an iteration of the D-loop, which follows. In Figure 1 and the following figures $n = 5$ is assumed.

A one-dimensional affine schedule has the following form:

$$x \mapsto bx + c.$$

We first determine the linear part of the schedules, i.e., we select a value for b . For reasons of simplicity, we will use the same linear part for all the polytopes. It is clear that we cannot choose b to be positive, since the first iteration of the B-loop indirectly depends on the final iteration of the same loop. Therefore, $b = -k$, for some positive value k . We will use $k = 2$ in the example. An equivalent combination of affine mapping and ordering vector is k times the identity matrix as linear part of the affine transformation ($A = kI_{1,1}$), and -1 as the ordering vector ($\vec{\pi} = -1$). In the formulas in the remainder of this example, we will not substitute the value 2 for k , but rather further manipulate k symbolically, i.e., the schedules will be of the form

$$i \mapsto -ki + c_X.$$

The reason is that we want to demonstrate that not all choices for k will lead to a valid solution.

The example program exhibits four static dependences. The i -th iteration of the B-loop depends on iteration $i - 1$ of the A-loop, resulting in

$$-ki + c_B - (-k(i - 1) + c_A) \geq 1$$

or

$$c_B - c_A \geq k + 1. \quad (5)$$

Similarly for the dependences between C and D and D and B, we have:

$$-ki + c_D - (-ki + c_C) \geq 1$$

or

$$c_D - c_C \geq 1 \quad (6)$$

and

$$-k0 + c_B - (-kn + c_D) \geq 1$$

or

$$c_B - c_D \geq 1 - nk. \quad (7)$$

The dependence between B and C yields

$$-ki + c_C - (-k2 + c_B) \geq 1$$

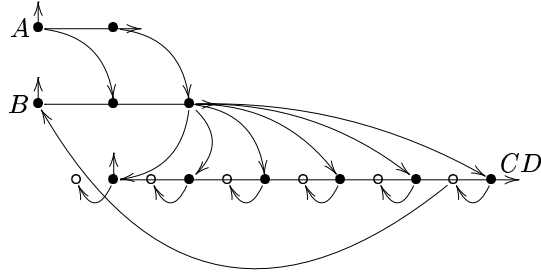


Figure 2: Combining C and D

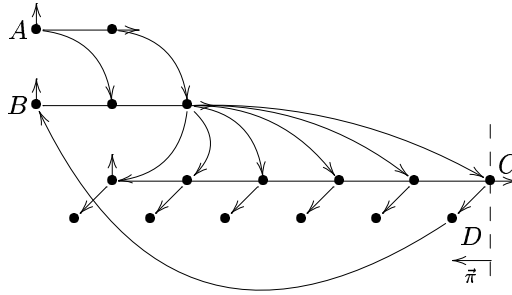


Figure 3: Combining C and D (with vertical displacement)

or

$$c_C - c_B \geq 1 + (i - 2)k,$$

which, evaluated for the extremes of the dependence domain ($i = 0$ and $i = n$), results in

$$c_C - c_B \geq 1 - 2k \quad (8)$$

and

$$c_C - c_B \geq 1 + (n - 2)k. \quad (9)$$

It is clear that as long as n is positive, (9) is more stringent than (8), so we do not need to consider the latter any further.

We can now proceed with an incremental translation step by selecting relative time delays one at a time. As our main focus is feasibility rather than optimality, we arbitrarily choose to consider first C and D and from (6) we conclude that we can select $c_D - c_C = 1$. Using (4), we then obtain $a_D - a_C = (-1)^{-1}1 = -1$ and the resulting combination of C and D is shown in Figure 2. Remember that we have scaled with a factor of $k = 2$ prior to translation.

Since this is a one-dimensional example, combining C and D results in a one-dimensional domain. To improve the visibility of the dependence distance vectors and the identification of the iterations of the different statements, we use a second dimension for the different statements, as shown in Figure 3. The figure also shows the ordering vector and one hyperplane perpendicular to the ordering vector. Each such hyperplane contains all iterations that are executed at the same time. A one dimensional hyperplane is a point, but it is drawn here as a line crossing the different statements. To avoid clutter, the hyperplane and ordering vector are only drawn in the iteration space containing C .

After eliminating c_D from (5), (9) and (7), we obtain

$$\begin{aligned} c_B - c_A &\geq k + 1 \\ c_C - c_B &\geq 1 + (n - 2)k \\ c_B - c_C &\geq 2 - nk. \end{aligned}$$

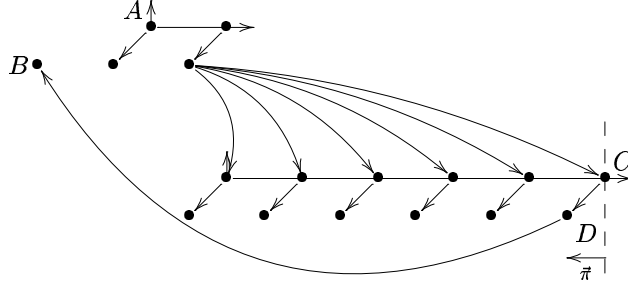


Figure 4: Combining A and B

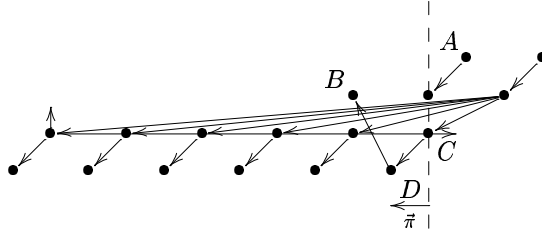


Figure 5: Combining C and B

A and B can be combined with a relative time delay $c_B - c_A = k + 1$. This is shown in Figure 4. Finally, C and A have to be combined and here a problem could arise. The two constraints on the relative time delays, yield:

$$2 - nk \leq c_B - c_C \leq (2 - n)k - 1$$

or

$$2 - nk \leq c_A - c_C + k + 1 \leq (2 - n)k - 1, \quad (10)$$

which requires

$$2 - nk \leq (2 - n)k - 1$$

or

$$k \geq \frac{3}{2},$$

but k was chosen prior to the translation step and does not necessarily satisfy this equation. Our choice of $k = 2$ does, however, and after substitution in (10) we obtain

$$-1 - 2n \leq c_A - c_C \leq -2n.$$

The final result, with $c_A - c_C = -2n - 1$, is shown in Figure 5.

In this section we have determined the relative offsets of a set of interdependent polytopes incrementally. By choosing an offset in the common iteration space for one of the polytopes, the other offsets can be derived from these relative offsets. The example showed that it may not always be possible to find a valid set of relative offsets in this way. This would have been the case had we chosen $k = 1$ during the linear transformation phase. In the remainder of this document we will show how this situation can be detected at the very beginning of the translation step and how it can be guaranteed that if a solution for the relative offsets exists, incremental translation will be able to find one.

3 Problem formulation

We will restrict our attention to programs with iteration domains, which is the set of all iteration vectors for which a given statement is executed, that can be represented as polytopes. This restriction requires that all loop bounds are manifest and expressed as affine combinations of outer loops. We will further assume that each iteration domain is defined in an iteration space of dimension N , where N is constant for a particular program and at least as large as the largest dimension of an iteration domain. This can be trivially accomplished by adding extra dimensions to lower-dimensional iteration spaces. The definitions of a polytope and of related concepts are included below. They are adapted from Wilde (1993), to which we refer for a more detailed discussion.

Definition 1 (Polyhedron) *A polyhedron P is a subspace of \mathbb{Q}^n bounded by a finite number of hyperplanes.*

$$P = \{ \vec{x} \mid A\vec{x} \geq \vec{b} \}$$

Theorem 1 *The set $P \in \mathbb{Q}^n$ is a polyhedron iff it can be written as*

$$P = \left\{ \sum_i \lambda_i \vec{\delta}_i + \sum_i \mu_i \vec{r}_i \mid \vec{\delta}_i \in V, \vec{r}_i \in R, \lambda_i, \mu_i \geq 0, \sum_i \lambda_i = 1 \right\}$$

with V and R finite subsets of \mathbb{Q}^n .

Theorem 1 is a simple consequence of well known theorems by Minkowski and Weyl. The notation in Definition 1 is sometimes referred to as the implicit notation, whereas the one in Theorem 1 is referred to as the explicit notation. We will call the sets V and R in the explicit notation the supporting points and rays respectively.

Definition 2 (Polytope) *A polytope is a bounded polyhedron.*

Example 1 The first loop of the program in Figure 1 on page 5 can be represented by the following polytope:

$$[0, 1] = \{ i \mid i \geq 0 \wedge i \leq 1 \} = \{ \lambda_0 0 + \lambda_1 1 \mid \lambda_0, \lambda_1 \geq 0, \lambda_0 + \lambda_1 = 1 \}.$$

The bounding hyperplanes are the points 0 and 1.

Definition 3 (Convex hull) *The convex hull of a set X is the set of all convex combinations of elements from X :*

$$\text{conv } X = \left\{ \sum_i \lambda_i \vec{x}_i \mid \vec{x}_i \in X, \lambda_i \geq 0, \sum_i \lambda_i = 1 \right\}.$$

Definition 4 (Cone) *The cone generated by a set X is the set of all positive combinations of elements from X :*

$$\text{cone } X = \left\{ \sum_i \lambda_i \vec{x}_i \mid \vec{x}_i \in X, \lambda_i \geq 0 \right\}.$$

Definition 5 (Ray) *A ray of a set \mathcal{K} is a vector \vec{r} such that $\vec{x} \in \mathcal{K}$ implies $(\vec{x} + \mu\vec{r}) \in \mathcal{K}$ for all $\mu \geq 0$.*

Definition 6 (Line) A line of a set \mathcal{K} is a vector \vec{l} such that $\vec{x} \in \mathcal{K}$ implies $(\vec{x} + \mu\vec{l}) \in \mathcal{K}$ for all μ .

As already indicated in Section 1, the validity of a translation is mainly determined by the dependences. The (dependence) distance vector, which measures the dependence distance in each dimension and which is defined next, is therefore a crucial part of the dependence graph, a structure used in constructing a valid translation. In the remainder of this section, we will define the concept of a valid translation in terms of other concepts related to distance vectors and we will provide an incremental algorithm for obtaining a translation. The main problem will then turn out to be the delineation of the set of valid choices to be made during the course of the algorithm.

Definition 7 (Dependence distance vector) For each dynamic dependence between two statement instances with iteration vectors \vec{i}_1 and \vec{i}_2 , the dependence distance vector $\vec{\delta}$ is the difference between the two iteration vectors.

$$\vec{\delta} = \vec{i}_2 - \vec{i}_1 \quad \text{if } \vec{i}_1 \delta \vec{i}_2 \quad (11)$$

Note that this is an extension of what is generally known as a distance vector, which is typically only defined between iterations of statements within the same loop (nest). Here, the two iterations may be elements of different iteration spaces of, by assumption, the same dimensionality. In the remainder of this text, we will usually omit the “dependence” qualification and simply refer to “distance vector”.

Using these concepts, a program can be represented by a dependence graph, which contains all the required information about dependences to perform the translation. The incremental translation algorithm will modify the dependence graph in each step until a valid translation (defined later) can trivially be determined.

Definition 8 (Dependence Graph) A dependence graph $G = \langle V, E, \mathcal{P}, \mathcal{D} \rangle$ consists of following elements:

- V is the set of nodes, each representing a set of statements in the original program.
- Each node p is adorned by a set \mathcal{P}_p of polytopes, each representing an iteration domain. \mathcal{P} is the set of all such \mathcal{P}_p
- E is the set of edges. An edge is a pair of nodes (p_1, p_2) . Each edge indicates the presence of a dependence between one of the iteration domains associated with p_1 and one of the iteration domains associated with p_2 .
- Each edge $e \in E$, $e = (p_1, p_2)$ is adorned by a polytope \mathcal{D}_e , which is the set of all the distance vectors generated by all dependences between an iteration domain in p_1 and an iteration domain in p_2 . \mathcal{D} is the set of all such \mathcal{D}_e

In the initial, pre-translation graph G_0 the nodes will typically be singletons and each \mathcal{P}_p a single polytope that has already been linearly transformed by techniques such as those discussed in Verdoolaege et al. (2001).

Example 2 The initial graph of the example in Section 2 (without linear transformation) is $G_0 = \langle V, E, \mathcal{P}, \mathcal{D} \rangle$, shown in Figure 6, with

- $V = \{\{A\}, \{B\}, \{C\}, \{D\}\}$

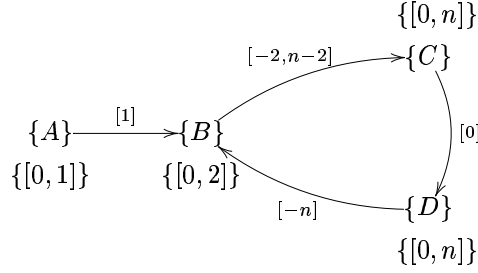


Figure 6: Initial dependence graph for the example in Figure 1

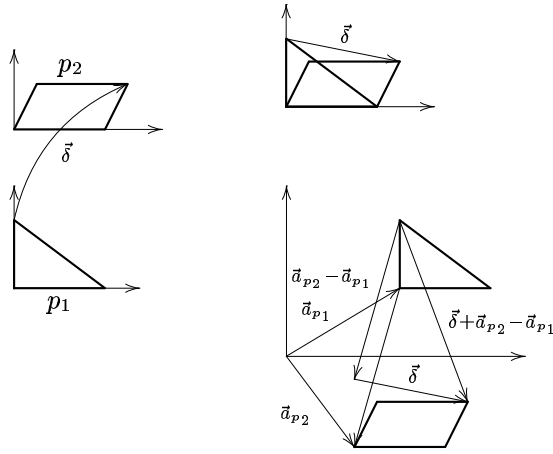


Figure 7: Decomposition of translated distance vectors. On the left, a dependence graph with a single dependence. On the right, the distance vector corresponding to the dependence with the top figure showing the canonical distance vector and the bottom figure showing the distance vector after translation of the polytopes involved in the dependence.

- $E = \{(\{A\}, \{B\}), (\{B\}, \{C\}), (\{C\}, \{D\}), (\{D\}, \{B\})\}$
- $\mathcal{P}_{\{A\}} = \{[0, 1]\}$
- $\mathcal{P}_{\{B\}} = \{[0, 2]\}$
- $\mathcal{P}_{\{C\}} = \{[0, n]\}$
- $\mathcal{P}_{\{D\}} = \{[0, n]\}$
- $\mathcal{D}_{(\{A\}, \{B\})} = \{1\}$
- $\mathcal{D}_{(\{B\}, \{C\})} = \{-2, -1, \dots, n-2\}$
- $\mathcal{D}_{(\{C\}, \{D\})} = \{0\}$
- $\mathcal{D}_{(\{D\}, \{B\})} = \{-n\}$

In subsequent examples, we will not use the set notation in subscripts but instead we will use the concatenation of the elements, i.e., we will write $\mathcal{D}_{(AB, CD)}$ instead of $\mathcal{D}_{(\{A, B\}, \{C, D\})}$.

We can now define what we mean with a translation.

Definition 9 (Translation) A translation T for a given dependence graph G is a function that assigns to each of the nodes p in V_G , an offset $\vec{a}_p \in \mathbb{Z}^n$ in the common iteration space.

In applying a translation to a dependence graph, each distance vector $\vec{\delta} = \vec{i}_2 - \vec{i}_1$ over $e = (p_1, p_2)$ is transformed into $(\vec{i}_2 + \vec{a}_{p_2}) - (\vec{i}_1 + \vec{a}_{p_1})$. As such, the

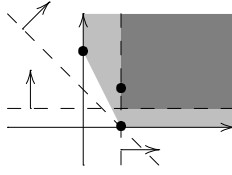


Figure 8: Dependence cone (■) and valid ordering polyhedron (■)

actual distance vectors generated by dependence $e = (p_1, p_2)$ will be given by the polytope $\mathcal{D}_e + \vec{a}_{p_2} - \vec{a}_{p_1}$, which we will call the *translated distance vectors*, denoted by \mathcal{D}_e^T . Figure 7 on the previous page illustrates the graphical decomposition of translated distance vectors.

In Section 2 we already showed that, in the absence of a subsequent ordering step, some translations do not lead to valid code. The presence of an ordering step makes that a larger set of translations can possibly lead to valid code as long as the ordering that is chosen after translation is valid for all the translated distance vectors. This means that the valid ordering polyhedron, as defined next, has to be non-empty for a valid translation to exist.

Definition 10 (Valid ordering polyhedron) *The valid ordering polyhedron (\mathcal{P}_O) is the set of all ordering vectors that are valid for a given set \mathcal{D} of dependence distance vectors.*

$$\mathcal{P}_O(\mathcal{D}) = \left\{ \vec{\pi} \mid \forall \vec{\delta} \in \mathcal{D} : \vec{\pi}^T \vec{\delta} \geq 1 \right\} \quad (12)$$

Example 3 Figure 8 shows the valid ordering polyhedron $\mathcal{P}_O(\mathcal{D})$ for the set of distance vectors $\mathcal{D} = \{(1, 0)^T, (1, 1)^T, (0, 2)^T\}$. The constraints in (12) are shown for each of the distance vectors. Note that the constraint generated by the distance vector $(1, 1)^T$ is redundant.

Rather than maintaining a set of all dependence distance vectors, we need only construct what is known as the dependence cone (Yang et al. 1994). The valid ordering polyhedron for a dependence cone generated by a set of dependence vectors is the same as that for the set itself.

Definition 11 (Dependence cone) *A dependence cone of X is the set of all (strictly) positive combinations of elements from X :*

$$\text{dcone } X = \left\{ \sum_i \lambda_i \vec{x}_i \mid \vec{x}_i \in X, \lambda_i \geq 0, \sum_i \lambda_i \geq 1 \right\},$$

where X is either a finite set or a polytope. In the latter case the dependence cone is equivalent to the dependence cone of the set of vertices of the polytope, which is a finite set.

Note that a dependence cone is strictly speaking not necessarily a cone, because of the additional requirement $\sum_i \lambda_i \geq 1$. For example, Figure 9 shows both the cone and the dependence cone generated by the set $\{(-3, 3)^T, (0, 2)^T, (1, 3)^T, (-1, 4)^T\}$.

Example 4 Figure 8 also shows the dependence cone $\text{dcone}(\mathcal{D})$ for the set of distance vectors $\mathcal{D} = \{(1, 0)^T, (1, 1)^T, (0, 2)^T\}$. The valid ordering polyhedron for $\text{dcone}(\mathcal{D})$ is the same as the one for \mathcal{D} . Notice that the point $(1, 1)^T$ lies in the (relative) interior of the dependence cone.

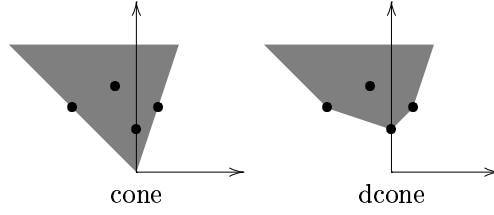


Figure 9: Comparison between a cone and a dependence cone.

Before defining the concept of a *valid* translation, we will first establish a simple criterion for the dependence cones with a non-empty valid ordering polyhedron. The proof requires two lemmas, including the following variant of Farkas' lemma.

Lemma 1 (Farkas') *The system $A\vec{x} \leq \vec{b}$ of linear inequalities has a solution \vec{x} iff for each $\vec{y} \geq \vec{0}$ with $\vec{y}^T A = 0$: $\vec{y}^T \vec{b} \geq \vec{0}$.*

$$(\exists \vec{x} : A\vec{x} \leq \vec{b}) \Leftrightarrow (\forall \vec{y} : \vec{y} \geq \vec{0}, \vec{y}^T A = 0 \Rightarrow \vec{y}^T \vec{b} \geq \vec{0})$$

For a proof see Schrijver (1986).

Lemma 2 *If a polyhedron does not contain the null-vector, then any positive linear combination of its supporting points and rays that results in the null-vector will have zero coefficients for all the supporting points.*

Proof Let V and R be matrices with the supporting points and rays of the polyhedron P as columns. Let \vec{x} and \vec{y} be any positive vectors such that

$$[V \quad R] \begin{bmatrix} \vec{x} \\ \vec{y} \end{bmatrix} = \vec{0}$$

and suppose that $\vec{x} \neq \vec{0}$, then the sum s of the elements of \vec{x} is a strictly positive number. Let $\vec{x}' = \vec{x}/s$ and $\vec{y}' = \vec{y}/s$, then, by Theorem 1,

$$\vec{0} = [V \quad R] \begin{bmatrix} \vec{x}' \\ \vec{y}' \end{bmatrix} \in P,$$

as $\sum_i x'_i = \sum_i (x_i / \sum_i x_i) = 1$, contradicting the assumption. \square

Theorem 2 *The valid ordering polyhedron for some distance vector polyhedron \mathcal{D} is non-empty iff \mathcal{D} does not contain the null-vector.*

Proof If \mathcal{D} contains the null-vector, then the valid ordering polyhedron is empty since $\vec{\pi}^T \vec{0} = 0$ for any $\vec{\pi}$. This proves that if \mathcal{D} 's valid ordering polyhedron is non-empty, then \mathcal{D} does not contain the null-vector.

Conversely, let V [R] be a matrix with the supporting points [rays] in the explicit notation of \mathcal{D} as columns. Then $V^T \vec{\pi} \geq \vec{1} \wedge R^T \vec{\pi} \geq \vec{0}$ ensures $\vec{\pi} \in \mathcal{P}_O$, i.e., $\vec{\pi}$ should be a solution of

$$-\begin{bmatrix} V^T \\ R^T \end{bmatrix} \vec{\pi} \leq \begin{bmatrix} -\vec{1} \\ \vec{0} \end{bmatrix}.$$

By Farkas' lemma (lemma 1), this set of equations has a solution iff

$$\forall \vec{y} : \vec{x}, \vec{y} \geq \vec{0}, -[V \quad R] \begin{bmatrix} \vec{x} \\ \vec{y} \end{bmatrix} = \vec{0} \Rightarrow [\vec{x}^T \quad \vec{y}^T] \begin{bmatrix} -\vec{1} \\ \vec{0} \end{bmatrix} \geq 0.$$

According to lemma 2, any pair of vectors satisfying the premise of the implication will have $\vec{x} = \vec{0}$, for which the consequent trivially holds. \square

We will call a dependence cone that contains the null-vector *degenerate*. According to Theorem 2 the valid ordering polyhedron is empty iff the dependence cone on which it is defined is degenerate. This dependence cone should include all the dependences over the whole program.

Definition 12 (Global dependence cone) *The global dependence cone $C_{G,T}$ for a given dependence graph G and a given translation T is the dependence cone generated by all the distance vectors.*

$$C_{G,T} = \text{dcone} \left(\bigcup_{(p_1,p_2) \in E_G} (\mathcal{D}_{(p_1,p_2)} + \vec{a}_{p_2} - \vec{a}_{p_1}) \right) \quad (13)$$

Definition 13 (Valid translation) *A valid translation is a translation that determines a non-degenerate global dependence cone.*

A valid translation will be determined incrementally, which means that in each step two nodes are combined with a certain relative offset until only a single node remains. This is shown in Algorithm 1. After applying this algorithm, a translation T^* for the original graph G_0 can be recovered as follows. Take the single node of the final graph G^* and assign it an arbitrary offset \vec{a} . Subsequently undo all the combinations in the reverse order, assigning one of the nodes the offset \vec{a} and the other $\vec{a} + \vec{a}$ as appropriate, until the original graph G_0 is reached and all its nodes have been assigned an offset.

The main problem in finding a valid translation has now been reduced to selecting good values for \vec{a} in step 4 of Algorithm 1. Furthermore, we wish to establish in advance whether it is possible to find such values. The solution to these two problems is the subject of the next section.

Algorithm 1 Incremental translation

1. Initialize G .
 2. If G contains a single node, stop.
 3. Select two nodes p_1 and p_2 in G .
 4. Select an offset \vec{a}_{p_1,p_2} of p_2 relative to p_1
 5. Replace G by $\text{combine}(G, p_1, p_2, \vec{a}_{p_1,p_2})$.
 6. Goto 2.
-

The combination of two nodes in step 5 of Algorithm 1 is detailed by Algorithm 2. It is based on the observation that each dependence that involves p_2 will become a dependence involving an extended p_1 . If p_2 is the producer, for example, a distance vector $\vec{\delta} + \vec{a}_{p_i} - \vec{a}_{p_2}$ will be rewritten as

$$\begin{aligned} \vec{\delta} + \vec{a}_{p_i} - \vec{a}_{p_2} &= (\vec{\delta} - \vec{a}_{p_1,p_2}) + \vec{a}_{p_i} - \vec{a}_{p_1} \\ &= \vec{\delta}^i + \vec{a}_{p_i} - \vec{a}_{p_1}, \end{aligned}$$

with

$$\vec{a}_{p_1,p_2} = \vec{a}_{p_2} - \vec{a}_{p_1}, \quad (14)$$

the relative offset of p_2 with respect to p_1 . In other words, the distance vector $\vec{\delta}^i$ between an iteration of p_2 when considered part of the new p_1 and an iteration

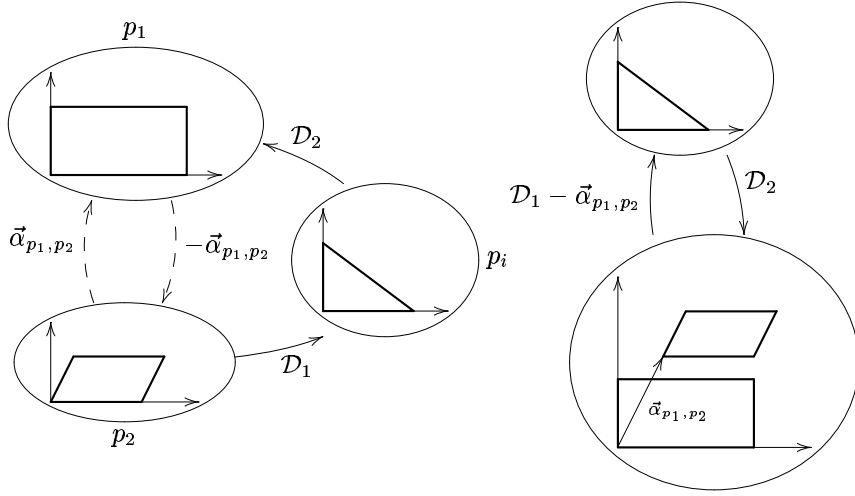


Figure 10: One iteration of Algorithm 1. The figure on the left shows the graph G prior to combination, with the two additional pseudo-edges. The figure on the right shows G' after combination.

of p_i is the sum of the original distance vector $\vec{\delta}$ between the iterations of p_2 and p_i and the relative offset of p_2 with respect to p_1 . We will sometimes add this relative offset to the graph by means of a pseudo-edge, representing a pseudo-dependence. Figure 10 shows how this combination works.

Algorithm 2 Combining two nodes

$G' = \langle V', E', \mathcal{P}', \mathcal{D}' \rangle = \text{combine}(G, p_1, p_2, \vec{\alpha})$, with

- $V' = V \setminus \{p_1, p_2\} \cup \{p_1 \cup p_2\}$
 - $\mathcal{P}'_p = \mathcal{P}_p \quad \forall p \in V \setminus \{p_1, p_2\}$
 - $\mathcal{P}'_{p_1 \cup p_2} = \mathcal{P}_{p_1} \cup (\mathcal{P}_{p_2} + \vec{\alpha})$
 - $E' = E \setminus \{(u, v) \in E \mid \{u, v\} \cap \{p_1, p_2\} \neq \emptyset\} \cup$
 $\{(p_1 \cup p_2, v) \mid (p_1, v) \in E \vee (p_2, v) \in E\} \cup$
 $\{(u, p_1 \cup p_2) \mid (u, p_1) \in E \vee (u, p_2) \in E\}$
 - $\mathcal{D}'_e = \mathcal{D}_e \quad \forall e \in E, e = (u, v), \{u, v\} \cap \{p_1, p_2\} = \emptyset$
 - $\mathcal{D}'_{(p_1 \cup p_2, v)} = (\mathcal{D}_{(p_1, v)} \cup (\mathcal{D}_{(p_2, v)} - \vec{\alpha}))$
 - $\mathcal{D}'_{(u, p_1 \cup p_2)} = (\mathcal{D}_{(u, p_1)} \cup (\mathcal{D}_{(u, p_2)} + \vec{\alpha}))$
 - $\mathcal{D}'_{(p_1 \cup p_2, p_1 \cup p_2)} = (\mathcal{D}_{(p_1, p_1)} \cup (\mathcal{D}_{(p_1, p_2)} + \vec{\alpha}) \cup (\mathcal{D}_{(p_2, p_1)} - \vec{\alpha}) \cup \mathcal{D}_{(p_2, p_2)})$
-

In the following section we will prove that Algorithm 1 works under certain conditions. The main idea behind the proof is that we only need to ensure that a valid ordering exists with respect to (indirect) dependences between a node in the dependence graph and itself. We will call these dependences *self-dependences* and the associated distance vectors *self-dependence distance vectors*. The following definitions define indirect dependences and the dependence cone generated by self-dependences.

Definition 14 (Fundamental path) *A fundamental path between a pair of*

nodes $l = (p_1, p_2)$ of a graph G is either a simple path from p_1 to p_2 or a simple circuit containing $p_1 = p_2$. A simple path [circuit] visits each node at most once.

$$\Pi_{G,l} = \left\{ \pi \mid \pi = (u_1, \dots, u_n), l = (u_1, u_n), |\{u_i\}| = n - \delta_{u_1, u_n}, \right. \\ \left. \forall 1 \leq i \leq n-1 : (u_i, u_{i+1}) \in E_G \right\}$$

with δ the Kronecker-delta, i.e., the number of different nodes in the path is either equal to or one less than the length of the path, depending on whether the first and last nodes of the path are the same.

Example 5 The initial graph of the example in Section 2 has the following fundamental paths:

$$\begin{aligned} \Pi_{G_0,(A,B)} &= \{(A, B)\} \\ \Pi_{G_0,(A,C)} &= \{(A, B, C)\} \\ \Pi_{G_0,(A,D)} &= \{(A, B, C, D)\} \\ \Pi_{G_0,(B,B)} &= \{(B, C, D, B)\} \\ \Pi_{G_0,(B,C)} &= \{(B, C)\} \\ \Pi_{G_0,(B,D)} &= \{(B, C, D)\} \\ \Pi_{G_0,(C,B)} &= \{(C, D, B)\} \\ \Pi_{G_0,(C,C)} &= \{(C, D, B, C)\} \\ \Pi_{G_0,(C,D)} &= \{(C, D)\} \\ \Pi_{G_0,(D,B)} &= \{(D, B)\} \\ \Pi_{G_0,(D,C)} &= \{(D, B, C)\} \\ \Pi_{G_0,(D,D)} &= \{(D, B, C, D)\} \end{aligned}$$

Definition 15 (Indirect distance vector polytope) The indirect distance vector polytope $\mathcal{V}_{G,l}$ with $l = (p_1, p_2)$ is the convex hull of all indirect distance vectors over all fundamental paths in G between p_1 and p_2 . An indirect distance vector over a path is the sum of distance vectors for each segment of the path.

$$\mathcal{V}_{G,l} = \text{conv} \left\{ \vec{\delta} \mid \exists \pi \in \Pi_{G,l} : \forall 1 \leq i \leq n-1 : \exists \vec{\delta}_i \in \mathcal{D}_{G,(u_i, u_{i+1})}, \vec{\delta} = \sum_{i=1}^{n-1} \vec{\delta}_i \right\} \quad (15)$$

Example 6 The initial graph of the example in Section 2 (without linear transformation) yields the following indirect distance vector polytopes:

$$\begin{aligned} \mathcal{V}_{G_0,(A,A)} &= \emptyset \\ \mathcal{V}_{G_0,(A,B)} &= [1] \\ \mathcal{V}_{G_0,(A,C)} &= [-1, n-1] \\ \mathcal{V}_{G_0,(A,D)} &= [-1, n-1] \\ \mathcal{V}_{G_0,(B,B)} &= [-2-n, -2] \\ \mathcal{V}_{G_0,(B,C)} &= [-2, n-2] \\ \mathcal{V}_{G_0,(B,D)} &= [-2, n-2] \\ \mathcal{V}_{G_0,(C,B)} &= [-n] \\ \mathcal{V}_{G_0,(C,C)} &= [-2-n, -2] \\ \mathcal{V}_{G_0,(C,D)} &= [0] \\ \mathcal{V}_{G_0,(D,B)} &= [-n] \\ \mathcal{V}_{G_0,(D,C)} &= [-2-n, -2] \\ \mathcal{V}_{G_0,(D,D)} &= [-2-n, -2] \end{aligned}$$

Definition 16 (Self-dependence (full) cone) *The self-dependence cone for a given node p of a graph G is the dependence cone generated by all self-dependence distance vectors.*

$$R_{G,p} = \text{dcone } \mathcal{V}_{G,(p,p)} \quad (16)$$

The self-dependence cone for the whole graph is the convex (hull of the) union of the self-dependence cones of all nodes in the graph.

$$R_G = \text{conv } \bigcup_{p \in V_G} R_{G,p} \quad (17)$$

The self-dependence full cone is then the cone generated by the self-dependence cone.

$$\overline{R}_G = \text{cone } R_G$$

Example 7 For the initial graph of the example in Section 2 (without linear transformation), the self-dependence cone is:

$$R_{G_0} = R_{G_0,A} = R_{G_0,B} = R_{G_0,C} = (-\infty, -2]$$

and the self-dependence full cone is:

$$\overline{R}_{G_0} = (-\infty, 0].$$

4 Feasibility of translation

This section consists of two theorems that determine the necessary conditions for a valid translation to exist and the set of valid choices for the relative offsets between two nodes in case the conditions hold. We first establish the second result. As auxiliary lemmas, we first prove that if a valid translation exists, then the initial self-dependence cone is not degenerate. Then we prove that we can maintain this property for intermediate self-dependence cones by making appropriate choices for the relative offsets and finally we show that the final self-dependence cone is equal to the global dependence cone, meaning that it is indeed sufficient to only consider self-dependences. Readers who wish to skip the proofs can proceed on page 20.

Lemma 3 *The self-dependence cone for the initial dependence graph R_{G_0} is a subset of the global dependence cone $C_{G,T}$ for any translation T .*

$$\forall T : R_{G_0} \subset C_{G,T}$$

Proof Let \vec{r}_p be an element of the self-dependence cone of some node $p \in V_{G_0}$: $\vec{r}_p \in R_{G_0,p}$. Then from (16), $\vec{r}_p = \sum_{j=1}^m \lambda_j \vec{\delta}_j$ for some positive λ_j s, $\sum_j \lambda_j \geq 1$ and some $\vec{\delta}_j$ s from $\mathcal{V}_{G_0,(p,p)}$. For each $\vec{\delta}_j$,

$$\begin{aligned} \vec{\delta}_j &= \sum_{i=1}^{n_j-1} \vec{\delta}_{j,i} \\ &= \sum_{i=1}^{n_j-1} \vec{\delta}_{j,i} + \sum_{i=2}^{n_j} \vec{a}_{j,i} - \sum_{i=1}^{n_j-1} \vec{a}_{j,i} \\ &= \sum_{i=1}^{n_j-1} \vec{\delta}_{j,i}^t, \end{aligned}$$

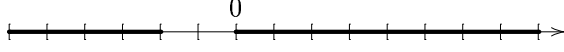


Figure 11: Invalid positions for $\vec{\alpha}_{C,D}$, the offset of the reference point of D relative to C .

with $\vec{\delta}'_{j,i} = \vec{\delta}_{j,i} + \vec{a}_{i+1} - \vec{a}_i \in C_{G_0,T}$ and where the second equality holds because $\vec{a}_1 = \vec{a}_{n_j}$. Therefore $\vec{r}_p = \sum_{k=1}^n \mu_k \vec{\delta}'_k$, with $\vec{\delta}'_k = \vec{\delta}'_{j,i}$ for some i and j and each μ_k the sum of some λ_j s. Since each λ_j appears in the sum of at least one μ_k , we have $\sum_k \mu_k \geq 1$ and so $\vec{r}_p \in C_{G_0,T}$. Finally, any $\vec{r} \in R_{G_0}$ is a convex combination of such \vec{r}_p s, so we can conclude that it is also an element of $C_{G_0,T}$. \square

Corollary 1 *If a valid translation exists, then R_{G_0} is not degenerate.*

Proof The proof follows trivially from lemma 3. \square

Lemma 4 *If R_G is non-degenerate then $R_{G'}$ with $G' = \text{combine}(G, p_1, p_2, \vec{\alpha}_{p_1, p_2})$ is also non-degenerate iff the offset $\vec{\alpha}_{p_1, p_2}$ between the two nodes p_1 and p_2 is such that it satisfies the following condition:*

$$\vec{\alpha}_{p_1, p_2} \notin (\overline{R}_G + \mathcal{V}_{G, (p_2, p_1)}) \cup (- (\overline{R}_G + \mathcal{V}_{G, (p_1, p_2)})) \quad (18)$$

Example 8 Referring to examples 6 and 7, the set of invalid relative offsets $\vec{\alpha}_{C,D}$ for the initial graph of the example in Section 2 (without linear transformation), is given by

$$((-\infty, 0] + [-2, -n - 2]) \cup (- ((-\infty, 0] + [0])),$$

i.e., the only valid offset is $\vec{\alpha}_{C,D} = -1$, as illustrated in Figure 11.

Proof We prove that $R_{G'}$ is degenerate iff $\vec{\alpha} = \vec{\alpha}_{p_1, p_2}$ is a member of the set in the right hand side of (18). First note that any circuit in G' and its corresponding indirect distance vectors corresponds to a circuit in G'' , where G'' is G with two additional pseudo-edges: one between p_1 and p_2 with $\mathcal{D}_{(p_1, p_2)} = -\vec{\alpha}$ and one between p_2 and p_1 with $\mathcal{D}_{(p_2, p_1)} = \vec{\alpha}$. The graph G'' has one additional circuit, viz. the one composed of the two pseudo-edges.

If $R_{G'}$ is degenerate then there exists an $\vec{r} \in R_{G'}$ such that $\vec{r} = \vec{0}$. According to (17) and (16), any element from $R_{G''}$ is the sum of indirect distance vectors over a circuit, i.e.,

$$\vec{r} = \sum_k \lambda_k \vec{\delta}_k \quad \lambda_k > 0, \sum_k \lambda_k \geq 1$$

where each $\vec{\delta}_j$ is an element of $\mathcal{V}_{G'', (p, p)}$ for some $p \in V_G$, excluding the case of the circuit composed of the two pseudo-edges, and where we have left out the terms for which $\lambda_k = 0$ as they do not have any influence. Since the path over which $\vec{\delta}_j$ is defined is a simple circuit, it will contain at most one of the two pseudo-edges. We can then group the circuits into three groups: one group $\lambda_{1,j} \vec{\delta}_{1,j}$ containing the circuits that do not contain any of the pseudo-edges, one group $\lambda_{2,j} \vec{\delta}_{(p_1 \rightarrow p_2), j}$ for those that contain the (p_1, p_2) link and one group $\lambda_{3,j} \vec{\delta}_{(p_2 \rightarrow p_1), j}$ for those that contain the (p_2, p_1) link. Let $\mu_1 := \sum_j \lambda_{1,j}$ and $\lambda'_{1,j} := \lambda_{1,j} / \mu_1$ and similarly for the other μ s and λ' s, then:

$$\begin{aligned} \vec{r} &= \sum_j \lambda_{1,j} \vec{\delta}_{1,j} + \sum_j \lambda_{2,j} \vec{\delta}_{(p_1 \rightarrow p_2), j} + \sum_j \lambda_{3,j} \vec{\delta}_{(p_2 \rightarrow p_1), j} \\ &= \mu_1 \sum_j \lambda'_{1,j} \vec{\delta}_{1,j} + \mu_2 \sum_j \lambda'_{2,j} \vec{\delta}_{(p_1 \rightarrow p_2), j} + \mu_3 \sum_j \lambda'_{3,j} \vec{\delta}_{(p_2 \rightarrow p_1), j} \end{aligned} \quad (19)$$

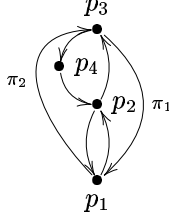


Figure 12: Pairing off two circuits

with

$$\mu_i \geq 0, \quad \lambda'_{i,j} > 0, \quad \sum_i \mu_i \geq 1 \quad \text{and} \quad \sum_j \lambda'_{i,j} = 1.$$

We now consider three cases based on the relative order of μ_2 and μ_3 .

- $\mu_2 = \mu_3$

In this case we can successively pair off circuits containing one of the pseudo-edges to circuits containing the other pseudo-edge, constructing cycles with the two pseudo-edges removed until $\mu_2 = \mu_3 = 0$. That is, in the first iteration, assume (without loss of generality) that $\lambda_{2,1} \leq \lambda_{3,1}$ and rewrite

$$\lambda_{2,1} \vec{\delta}_{(p_1 \rightarrow p_2),1} + \lambda_{3,1} \vec{\delta}_{(p_2 \rightarrow p_1),1}$$

in (19) as

$$\lambda''_1 \vec{\delta}'_1 + (\lambda_{3,1} - \lambda_{2,1}) \vec{\delta}_{(p_2 \rightarrow p_1),1},$$

with $\lambda''_1 = \lambda_{2,1}$ and $\vec{\delta}'_1$ a distance vector over the circuit constructed from the circuits π_1 and π_2 that $\vec{\delta}_{(p_1 \rightarrow p_2),1}$ and $\vec{\delta}_{(p_2 \rightarrow p_1),1}$ are defined over, by first traversing π_1 from p_2 to p_1 and then π_2 from p_1 back to p_2 . For example, the circuits $\pi_1 = (p_1, p_2, p_3, p_1)$ and $\pi_2 = (p_1, p_3, p_4, p_2, p_1)$, shown in Figure 12, would be combined into the circuit $(p_2, p_3, p_1, p_3, p_4, p_2)$. In each iteration of this rewriting process, μ_2 and μ_3 are decreased by λ''_k and at least one circuit containing a pseudo-edge is removed. After a number of iterations, $\mu_2 = \mu_3 = 0$ and all circuits containing a pseudo-edge are exhausted and transformed into circuits not containing a pseudo-edge. Equation 19 then becomes

$$\vec{r} = \sum_j \lambda_{1,j} \vec{\delta}_{1,j} + \sum_k \lambda''_k \vec{\delta}'_k,$$

with

$$\sum_i \lambda_{1,j} + \sum_k \lambda''_k = \mu_1 + \mu_2 > 0$$

and each $\vec{\delta}'_k$ a distance vector over a circuit and therefore in R_G as it is the sum of distance vectors over simple circuits. We are then left with a strictly positive combination of elements from the original R_G . Since by assumption $\vec{0} \notin R_G$, no strictly positive combination of elements of R_G can result in the null-vector, so in this case \vec{r} cannot be the null-vector.

- $\mu_2 < \mu_3$

As in the previous case, we can pair off some of the circuits, resulting in an element of \overline{R}_G . This will exhaust the circuits that contain the pseudo-edge (p_1, p_2) , but will leave $\mu_3 - \mu_2$ circuits that contain the other pseudo-edge.

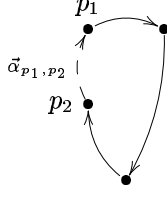


Figure 13: Decomposition of a circuit containing a pseudo-edge

An indirect distance vector over such a circuit can be decomposed into $\vec{\alpha}$ and some element from $\mathcal{V}_{G,(p_1,p_2)}$. As a result we have:

$$\vec{r} = \vec{r}' + (\mu_3 - \mu_2)\vec{\alpha} + (\mu_3 - \mu_2) \sum_j \lambda'_{3,j} \vec{\delta}'_j$$

with $\vec{r}' \in \overline{R}_G$ and each $\vec{\delta}'_j \in \mathcal{V}_{G,(p_1,p_2)}$. If $\vec{r} = \vec{0}$ then, since $\mu_3 - \mu_2$ is strictly positive:

$$\vec{\alpha} = -\frac{1}{\mu_3 - \mu_2} \vec{r}' - \sum_j \lambda'_{3,j} \vec{\delta}'_j$$

or

$$\vec{\alpha} \in -(\overline{R}_G + \mathcal{V}_{G,(p_1,p_2)}). \quad (20)$$

- $\mu_2 > \mu_3$

This case is completely analogous to the previous case and yields:

$$\vec{\alpha} \in \overline{R}_G + \mathcal{V}_{G,(p_2,p_1)}. \quad (21)$$

Conversely, let $\vec{\alpha}$ be an element of either of the sets in equations (20) and (21), then $R_{G'}$ is degenerate. To see this, take (21) and let

$$\vec{\alpha} = \vec{r} + \sum_j \lambda_j \vec{\delta}_j \quad (22)$$

with $\vec{r} \in \overline{R}_G$, $\sum_j \lambda_j = 1$ and for each $\vec{\delta}_j: \vec{\delta}_j \in \mathcal{V}_{G,(p_2,p_1)}$. Obviously $\vec{\alpha} = \sum_j \lambda_j \vec{\alpha}$, so we can combine each $\vec{\delta}_j$ with $-\vec{\alpha}$ and rewrite (22) as

$$\vec{0} = \vec{r} + \sum_j \lambda_j \vec{\delta}'_j$$

with $\vec{\delta}'_j = \vec{\delta}_j - \vec{\alpha} \in \mathcal{V}_{G',(p_1,p_1)}$ and so $\vec{r}' = \sum_j \lambda_j \vec{\delta}'_j \in R_{G'}$. If $\vec{r} = \vec{0}$ then we have found an element in $R_{G'}$ equal to the null-vector. If \vec{r} is in R_G then it is also in $R_{G'}$ and again we find an element $\vec{r} + \vec{r}'$ in $R_{G'}$ equal to the null-vector. In the final case $\vec{r} = \lambda \vec{r}''$ with $\vec{r}'' \in R_{G'}$ and $0 < \lambda < 1$, and then $\vec{r}'' + \lambda^{-1} \vec{r}' = \vec{0} \in R_{G'}$. \square

Lemma 5 *The final self-dependence cone is equal to the global dependence cone.*

$$R_{G^*} = C_{T^*}$$

Proof We prove that each set is a subset of the other.

- $R_{G^*} \subset C_{T^*}$

Each element \vec{r} in R_{G^*} can be written as a positive combination of indirect distance vectors over a circuit $\vec{r} = \sum_j \lambda_j \vec{r}_j$, with $\sum_j \lambda_j \geq 1$ from the

single node to itself. By construction, each such distance vector will be composed of distance vectors corresponding to “real” edges in the original graph G_0 and relative offsets corresponding to pseudo-edges in the original graph. Together they will form a circuit in the original graph. Replace all adjacent pseudo-edges in this circuit by a single pseudo-edge connecting the starting node of the first pseudo-edge to the ending node of the last pseudo-edge. Insert a pseudo-edge from a node to itself between each pair of adjacent real edges. The resulting circuit alternates between real edges and pseudo-edges and we can rewrite the indirect distance vector as follows:

$$\begin{aligned}\vec{r}_j &= \sum_k v_{u_{2k}, u_{2k+1}} - \vec{a}_{u_{2k+1}, u_{2k+2}} \\ &= \sum_k v_{u_{2k}, u_{2k+1}} + \vec{a}_{u_{2k+1}} - \vec{a}_{u_{2k+2}} \\ &= \sum_k v_{u_{2k}, u_{2k+1}} + \vec{a}_{u_{2k+1}} - \vec{a}_{u_{2k}}\end{aligned}$$

The last equality holds because the first and final nodes are the same and hence have the same offset. The last identity shows that $\vec{r}_j \in C_{T^*}$ and thus \vec{r} is in C_{T^*} as well.

- $R_{G^*} \supset C_{T^*}$

Each element $\vec{\delta}$ in C_{T^*} can be written as a positive combination of distance vectors with $\sum_j \lambda_j \geq 1$. Any such distance vector is the sum of a distance vector over an edge (p_1, p_2) in G_0 and the relative offset \vec{a}_{p_2, p_1} between the end points and is therefore an element of R_{G^*} . As a result, $\vec{\delta}$ is in R_{G^*} as well.

□

Theorem 3 *If a valid translation exists, then any translation is valid iff it can be constructed by Algorithm 1 with choices for \vec{a} according to lemma 4.*

Proof The theorem follows almost directly from lemma 5. If a translation exists, then by corollary 1 any translation we construct using \vec{a} 's based on lemma 4 will have a non-degenerate R_{G^*} and will thus be valid. Conversely, for any valid translation, we can in each step choose a relative offset that corresponds to this translation. The self-dependence cone $R_{G'}$ that results from the combination in each step is a superset of the one R_G prior to combination. I.e., we have

$$R_{G_0} \subset \dots \subset R_G \subset R_{G'} \subset R_{G''} \dots \subset R_{G^*}.$$

Since the final R_{G^*} is non-degenerate, so will all the others and therefore those choices will correspond to valid ones according to lemma 4. □

Note that Theorem 3 does not imply that the application of Algorithm 1 will always yield a valid translation. If no valid translation exists, then obviously the algorithm will not be able to construct one. Moreover, the theorem does not exclude the possibility that certain choices for \vec{a} lead to a situation where condition (18) of lemma 4 has no (integer) solutions. The following theorem details the conditions under which a valid translation is guaranteed to exist, but we first give an example of where a valid translation does not exist.

Example 9 Consider again the example in Section 2, but this time with the linear transformation applied. This means that all the difference “vectors”

from the examples in the previous section have to be multiplied by 2 as well. In particular, we have:

$$\begin{aligned}
\mathcal{V}_{G_0,(B,C)} &= [-4, 2n - 4] \\
\mathcal{V}_{G_0,(D,B)} &= [-2n] \\
\mathcal{V}_{G_0,(C,D)} &= [0] \\
\mathcal{V}_{G_0,(D,C)} &= [-4 - 2n, -4] \\
R_{G_0} &= (-\infty, -4] \\
\overline{R}_{G_0} &= (-\infty, 0]
\end{aligned}$$

Condition (18) then becomes

$$\tilde{\alpha}_{C,D} \notin (-\infty, -4] \cup [0, \infty).$$

We choose $\alpha_{C,D} = -3$ and we obtain

$$\begin{aligned}
\mathcal{V}_{G_1,(B,CD)} &= [-4, 2n - 4] \\
\mathcal{V}_{G_1,(CD,B)} &= [-2n + 3] \\
R_{G_1} &= (-\infty, -1] \\
\overline{R}_{G_1} &= (-\infty, 0]
\end{aligned}$$

Condition (18) now becomes

$$\tilde{\alpha}_{B,CD} \notin (-\infty, -2n + 3] \cup [-2n + 4, \infty)$$

which has no integer solution. This means that the algorithm can still make globally invalid choices even though a valid translation exists as was demonstrated in Section 2.

Theorem 4 *If $N \geq 2$, with N the dimension of the problem, and if the initial self-dependence cone allows for a valid ordering, then a valid translation exists.*

We will provide two proofs for Theorem 4. The first directly proves that the constraint (18) always has an integer solution if the conditions stated in Theorem 4 hold. The second is based on the construction of a set of ordering vectors for which a compatible valid translation exists. Note that the dimension of the problem space can arbitrarily be increased by adding extra dimensions, so the requirement $N \geq 2$ is not a restriction.

Lemma 6 *If $N \geq 2$ and if R_G is non-degenerate, then the constraint (18):*

$$\tilde{\alpha}_{p_1,p_2} \notin (\overline{R}_G + \mathcal{V}_{G,(p_2,p_1)}) \cup (-\overline{R}_G + \mathcal{V}_{G,(p_1,p_2)})$$

has an (infinite number of) integer solution(s).

Proof If \overline{R}_G is contained in a hyperplane H , then the set $(\overline{R}_G + \mathcal{V}_{G,(p_2,p_1)}) \cup (-\overline{R}_G + \mathcal{V}_{G,(p_1,p_2)})$ is contained in the same hyperplane. Otherwise we could take $\vec{\delta}_1 \in \mathcal{V}_{G,(p_2,p_1)}$ and $\vec{\delta}_2 \in -\mathcal{V}_{G,(p_1,p_2)}$ with $\vec{\delta}_2 \notin \vec{\delta}_1 + H$. But then $\overline{R}_G \ni \vec{\delta}_1 - \vec{\delta}_2 \notin H$, which contradicts the assumption. Let \vec{g} be any integer vector, different from the null vector, orthogonal to H and $\vec{\delta}_1 \in \mathcal{V}_{G,(p_2,p_1)}$, then $\vec{\delta}_1 + \vec{g}$ is not in H and therefore a solution to (18).

If \overline{R}_G is not contained in a single hyperplane, then let H_1 and H_2 be two distinct supporting hyperplanes of \overline{R}_G :

$$H_1 = \{\vec{x} | \vec{g}_1^T \vec{x} = 0\} \quad \text{and} \quad H_2 = \{\vec{x} | \vec{g}_2^T \vec{x} = 0\},$$

with \vec{g}_1 and \vec{g}_2 pointing ‘‘outward’’ (see Figure 14), i.e.,

$$\overline{R}_G \subset \{\vec{x} | \vec{g}_1^T \vec{x} \leq 0\} \cap \{\vec{x} | \vec{g}_2^T \vec{x} \leq 0\}. \quad (23)$$

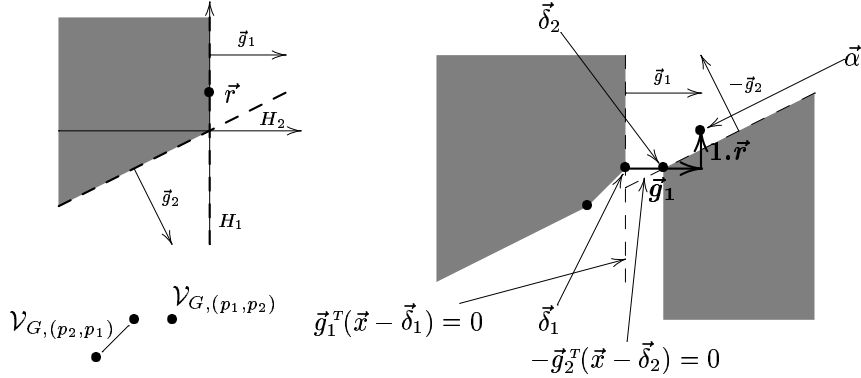


Figure 14: Illustration of the proof of lemma 6: on the top left the self-dependence full cone, on the bottom left the indirect distance vector polytopes and on the right the illegal regions for the relative offset

Furthermore, let $\vec{\delta}_1$ and $\vec{\delta}_2$ be defined as follows:

$$\vec{\delta}_1 = \operatorname{argmax}_{\vec{x} \in \mathcal{V}_{G,(p_2,p_1)}} \vec{g}_1^T \vec{x} \quad \vec{\delta}_2 = \operatorname{argmax}_{\vec{x} \in -\mathcal{V}_{G,(p_1,p_2)}} -\vec{g}_2^T \vec{x},$$

then and because of (23):

$$\forall \vec{x} \in \overline{\mathcal{R}}_G + \mathcal{V}_{G,(p_2,p_1)} : \vec{g}_1^T (\vec{x} - \vec{\delta}_1) \leq 0$$

and similarly

$$\forall \vec{x} \in -(\overline{\mathcal{R}}_G + \mathcal{V}_{G,(p_2,p_1)}) : -\vec{g}_2^T (\vec{x} - \vec{\delta}_2) \leq 0.$$

Let \vec{r} be an arbitrary integer point in $H_1 \setminus H_2$, i.e., $\vec{g}_1^T \vec{r} = 0$ and $\vec{g}_2^T \vec{r} < 0$, then $\vec{\alpha} = \vec{\delta}_1 + \vec{g}_1 + n\vec{r}$ with

$$n \geq \frac{1 + \vec{g}_2^T (\vec{\delta}_1 - \vec{\delta}_2 + \vec{g}_1)}{-\vec{g}_2^T \vec{r}}$$

is a solution to (18) since

$$\vec{g}_1^T (\vec{\delta}_1 + \vec{g}_1 + n\vec{r} - \vec{\delta}_1) = \|\vec{g}_1\|^2 + n \cdot 0 > 0$$

and

$$-\vec{g}_2^T (\vec{\delta}_1 + \vec{g}_1 + n\vec{r} - \vec{\delta}_2) \geq 1.$$

□

Proof (First proof of Theorem 4) The proof follows trivially from lemma 6. □

The second proof of lemma 4 is slightly more involved, but will yield additional insight in the relation between ordering and translation which we will exploit in Section 6. To establish the proof, we need the concept of an *averaged* self-dependence cone and three lemmas. The remainder of this section is devoted to the proofs of these lemmas.

Definition 17 (Averaged self-dependence cone) *The dependence cone \mathcal{R}_G generated by all indirect distance vectors over all simple circuits divided by the length of the respective circuit is called the averaged self-dependence cone.*

$$\mathcal{R}_{G,p} = \operatorname{dcone} \left\{ \vec{\delta} \mid \exists \pi \in \Pi_{G,(p,p)} : \forall 1 \leq i \leq n-1 : \vec{\delta}_i \in \mathcal{D}_{G,(u_i,u_{i+1})}, \vec{\delta} = \frac{1}{|\pi|} \sum_{i=1}^{n-1} \vec{\delta}_i \right\} \quad (24)$$

$$\mathcal{R}_G = \text{conv} \bigcup_{p \in V_G} \mathcal{R}_{G,p} \quad (25)$$

Example 10 The initial graph of the example in Section 2 (without linear transformation) contains a single circuit (see Figure 6 on page 10) with distance vector polytope $[-2 - n, -2]$ (see example 6). The averaged self-dependence cone is therefore $\mathcal{R}_{G_0} = (-\infty, -2/3]$.

Lemma 7 *For any ordering vector that is valid for the initial averaged self-dependence cone and for which the gcd (greatest common divisor) of its elements is 1, a valid translation can be found such that the ordering vector is valid for the corresponding final dependence cone.*

Proof We will prove the lemma via induction. The lemma holds if the initial graph contains a single node, since in this case all simple circuits are of length one and the averaged self-dependence cone equals the dependence cone.

If the initial graph contains more than one node, we perform one iteration of the loop in Algorithm 1, reducing the number of nodes by one. Suppose $\vec{\pi}$ is valid for the averaged self-dependence cone of graph G , then for all indirect distance vectors $\vec{\delta}$ over a simple circuit of length l :

$$\vec{\pi}^T \vec{\delta} \geq l. \quad (26)$$

From the construction of G' (Algorithm 2), it follows that any new indirect distance vector over a simple circuit, i.e., one that does not already exist in G , contains exactly one $\vec{\delta}_i$ in (15) of the form either $\vec{\alpha}$ or $-\vec{\alpha}$. In order for (26) to hold in G' , the indirect distance vectors containing $\vec{\alpha}$ yield a constraint of the form

$$\vec{\pi}^T \vec{\alpha} \geq l_1 - \vec{\pi}^T \vec{\delta}_1, \quad (27)$$

with $\vec{\delta}_1$ some indirect distance vector over a simple path from p_1 to p_2 in G . Similarly, those containing $-\vec{\alpha}$ yield a constraint of the form

$$\vec{\pi}^T \vec{\alpha} \leq \vec{\pi}^T \vec{\delta}_2 - l_2, \quad (28)$$

with $\vec{\delta}_2$ some indirect distance vector over a simple path from p_2 to p_1 in G .

If all constraints are either all of the form (27) or all of the form (28), then a suitable choice for $\vec{\alpha}$ can easily be found. Otherwise, suppose there is pair of constraints such that no $\vec{\alpha}$ exists satisfying

$$l_1 - \vec{\pi}^T \vec{\delta}_1 \leq \vec{\pi}^T \vec{\alpha} \leq \vec{\pi}^T \vec{\delta}_2 - l_2, \quad (29)$$

i.e., $l_1 - \vec{\pi}^T \vec{\delta}_1 > \vec{\pi}^T \vec{\delta}_2 - l_2$ or

$$\vec{\pi}^T (\vec{\delta}_1 + \vec{\delta}_2) < l_1 + l_2 \quad (30)$$

for some $\vec{\delta}_1$ and $\vec{\delta}_2$ as described above. But then, by linking $\vec{\delta}_1$ and $\vec{\delta}_2$ a cycle is obtained in G of length $l_1 + l_2$, which can be decomposed into one or more simple circuits with combined length $l_1 + l_2$ and with $\vec{\delta}_1 + \vec{\delta}_2$ as sum of the indirect distance vectors. Since, by assumption, (26) holds in G , this yields a contradiction with (30). Therefore (26) also holds in G' .

The gcd of $\vec{\pi}$ being 1 is needed to ensure that (29) not only has a solution for $\vec{\alpha}$ but also an integer solution, which is then guaranteed by the Extended Euclidean Algorithm (Knuth 1968). This completes the proof. \square

Corollary 2 *The ordering vector can be chosen prior to translation.*

Proof The proof follows trivially from lemma 7. \square

Lemma 8 *If $N \geq 2$ and the valid ordering polyhedron $\mathcal{P}_O(R)$ for some dependence cone R is non-empty, then the valid ordering polyhedron contains a vector with 1 as the gcd of its elements.*

Proof Select two linearly independent vectors $\vec{\pi}_1$ and $\vec{\pi}_2$ from $\mathcal{P}_O(R) \cap \mathbb{Z}^n$. If no such two vectors exists, then $\mathcal{P}_O(R)$ is not full-dimensional. But then R contains a line which is impossible since R is finitely generated and does not contain the null vector. Consider the set of vectors $\vec{\pi}^k = \vec{\pi}_1 + k\vec{\pi}_2$ with $k \geq 0$ and divide each of these vectors by the gcd of its elements to obtain $\vec{\pi}^{k'} = \lambda_{k,1}\vec{\pi}_1 + \lambda_{k,2}\vec{\pi}_2$ for some $\lambda_{k,1}$ and $\lambda_{k,2}$. No two $\vec{\pi}^{k'}$ can be linearly dependent since otherwise the corresponding $\vec{\pi}^k$ would be linearly dependent, which is impossible because $\vec{\pi}_1$ and $\vec{\pi}_2$ are linearly independent. In particular, no two $\vec{\pi}^{k'}$ can be equal. On the other hand, if $\vec{\pi}^{k'}$ is not an element of $\mathcal{P}_O(R)$, then $\lambda_{k,1} + \lambda_{k,2} < 1$. Since only a finite number of $\vec{\pi}^{k'}$ can satisfy this condition, there must be at least one $\vec{\pi}^{k'}$ in $\mathcal{P}_O(R)$. \square

Lemma 9 *If R_G is non-degenerate then so is \mathcal{R}_G .*

Proof Suppose \mathcal{R}_G is degenerate. Then there exist μ_i , $\lambda_{i,j}$, $\pi_{i,j}$ and $\vec{\delta}_{i,j,k}$ with

$$\sum_i \mu_i \sum_j \lambda_{i,j} \left(\frac{1}{|\pi_{i,j}|} \sum_{k=1}^{n_{i,j}-1} \vec{\delta}_{i,j,k} \right) = \vec{0} \quad (31)$$

with

$$\sum_i \mu_i = 1 \quad \text{and} \quad \sum_j \lambda_{i,j} \geq 1$$

and $\pi_{i,j}$ and $\vec{\delta}_{i,j,k}$ as in (24). By multiplying (31) by $\prod_{l,m} |\pi_{l,m}|$, we obtain

$$\sum_i \mu_i \sum_j \lambda_{i,j} \prod_{(l,m) \neq (i,j)} |\pi_{l,m}| \left(\sum_{k=1}^{n_{i,j}-1} \vec{\delta}_{i,j,k} \right) = \vec{0}$$

with

$$\sum_i \mu_i = 1 \quad \text{and} \quad \sum_j \lambda_{i,j} \prod_{(l,m) \neq (i,j)} |\pi_{l,m}| \geq 1$$

and then R_G is degenerate as well. \square

Proof (Second proof of Theorem 4) Since R_G is not degenerate, then by lemma 9 so is \mathcal{R}_G , which means that $\mathcal{P}_O(\mathcal{R}_G)$ is non-empty. Combined with $N \geq 2$, lemma 8 yields that $\mathcal{P}_O(\mathcal{R}_G)$ contains a vector with elements that have a gcd of 1. Lemma 7 then shows that a valid translation exists. \square

Corollary 3 *If $N \geq 2$ then a valid choice for $\vec{\alpha}_{p_1,p_2}$ in lemma 4 always exists.*

Proof By Theorem 4, R_G being non-degenerate is sufficient to ensure the existence of a valid translation, which by Theorem 3 can be constructed using lemma 4. \square

5 Two-dimensional example

In this section, a valid translation for the example of Section 2 will be constructed using Algorithm 1. This time, however, the common iteration space is

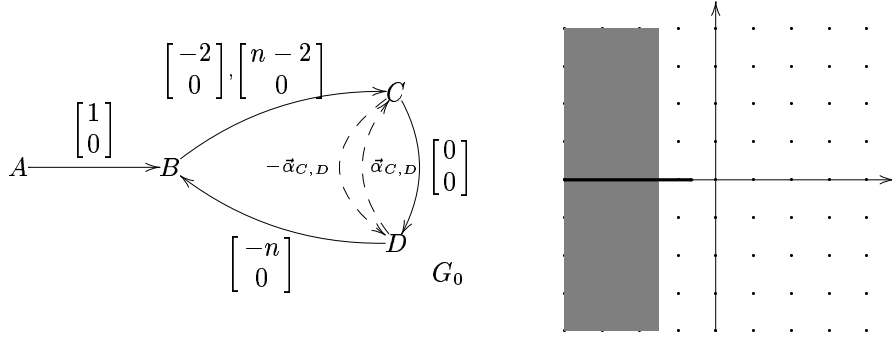


Figure 15: Initial dependence graph G_0 on the left. On the right, \mathcal{R}_{G_0} in thick lines and $\mathcal{P}_O(\mathcal{R}_{G_0})$ as a shaded area.

two-dimensional and we do not perform any linear transformation. Figure 15 shows the initial graph. Each set \mathcal{D}_e is represented by its vertices.

It is clear that the graph contains a single cycle of length three and that the extreme distance vectors of the cycle are $[-n-2, 0]^T$ and $[-2, 0]^T$. The initial self-dependence cone is then

$$R_{G_0} = \left\{ \lambda \begin{bmatrix} -2 \\ 0 \end{bmatrix} \mid \lambda \geq 1 \right\},$$

which is non-degenerate. Combined with the fact that $N = 2 \geq 2$, Theorem 4 ensures that a valid translation exists. As guaranteed by lemma 9, the averaged self-dependence cone is also non-degenerate:

$$\mathcal{R}_{G_0} = \left\{ \frac{\lambda}{3} \begin{bmatrix} -2 \\ 0 \end{bmatrix} \mid \lambda \geq 1 \right\}.$$

In a first step, we will combine C and D . The self-dependence full cone is

$$\overline{R}_G = \text{cone } R_{G_0} = \text{cone } \mathcal{R}_{G_0} = \left\{ \lambda \begin{bmatrix} -1 \\ 0 \end{bmatrix} \mid \lambda \geq 0 \right\} = (-\infty, 0]$$

and the relevant indirect distance vector polytopes are

$$\mathcal{V}_{(C,D)} = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right\}$$

and

$$\mathcal{V}_{(D,C)} = \text{conv} \left\{ \begin{bmatrix} -n-2 \\ 0 \end{bmatrix}, \begin{bmatrix} -2 \\ 0 \end{bmatrix} \right\}.$$

Figure 16 shows the resulting invalid positions for $\alpha_{C,D}$ based on condition (18) as well as the selected value $\vec{\alpha}_{C,D} = [0 \ 1]^T$ (marked by \times). The result of the combination of C and D is shown in Figure 17. Note that since the distance vector from CD to itself is a distance vector from a node to itself, it corresponds to a translated distance vector in the final translation and it has also been drawn in Figure 17.

Graph G_1 contains two cycles, one of length one with distance vector $[0 \ 1]^T$ and one of length two with extreme distance vectors $[-n-2, -1]^T$ and $[-2, -1]^T$. The averaged self-dependence cone is therefore

$$\mathcal{R}_{G_1} = \text{dcone} \left\{ \begin{bmatrix} -1 \\ -\frac{1}{2} \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}$$

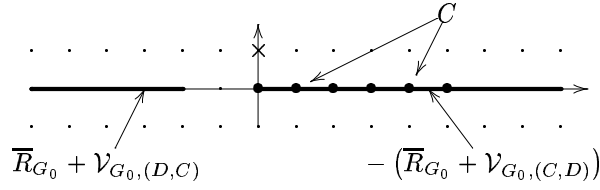


Figure 16: Invalid positions for $\vec{\alpha}_{C,D}$, the offset of the reference point of D relative to C .

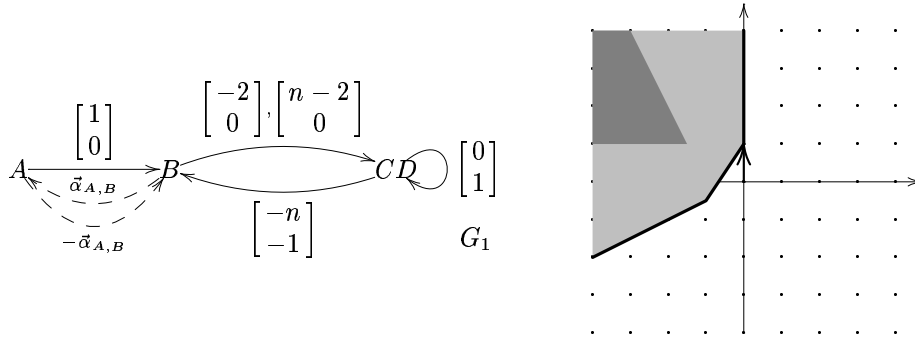


Figure 17: Graph with C and D combined. Cf. Figure 15

and the full cone is

$$\text{cone } \mathcal{R}_{G_1} = \text{cone} \left\{ \begin{bmatrix} -2 \\ -1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}.$$

Let us now consider the combination of A and B . The only relevant indirect distance vector polytope is

$$\mathcal{V}_{(A,B)} = \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\}.$$

Figure 18 shows the resulting invalid positions for $\alpha_{A,B}$ based on condition (18) as well as the selected value $\vec{\alpha}_{A,B} = [-1 \ 1]^T$. The result of the combination of A and B is shown in Figure 19. Since the only additional cycle (from AB to itself) is identical to the cycle from CD to itself, $\mathcal{R}_{G_2} = \mathcal{R}_{G_1}$.

In the final step, AB and CD are combined. We have

$$\mathcal{V}_{(AB,CD)} = \text{conv} \left\{ \begin{bmatrix} -1 \\ -1 \end{bmatrix}, \begin{bmatrix} n-1 \\ -1 \end{bmatrix} \right\}$$

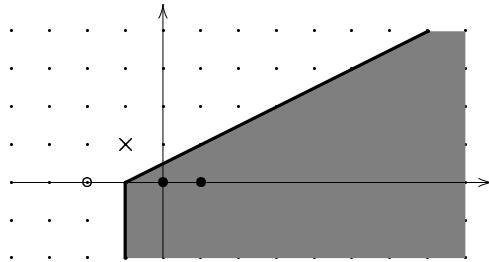


Figure 18: Invalid positions for $\vec{\alpha}_{A,B}$, the offset of the reference point of B relative to A .

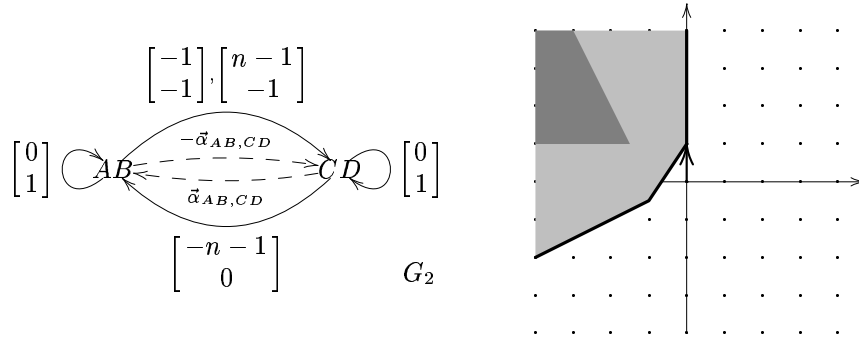


Figure 19: Graph with A and B combined. Cf. Figure 15

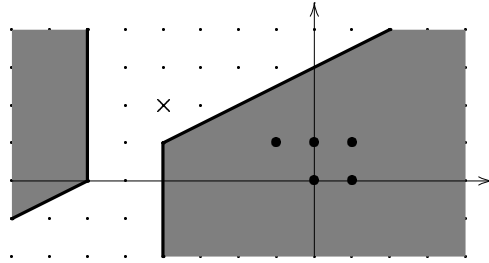


Figure 20: Invalid positions for $\vec{\alpha}_{AB,CD}$, the offset of the reference point of CD relative to AB .

and

$$\mathcal{V}_{(CD,AB)} = \left\{ \left[\begin{array}{c} -n-1 \\ 0 \end{array} \right] \right\}.$$

The final application of (18) is shown in Figure 20.

Figure 21 shows the final graph with a single node, where $\vec{\alpha}_{AB,CD} = [-n+1 \ 2]^T$ was used, as shown in Figure 20. The final single indirect distance vector polytope contains

$$\mathcal{V}_{G_2,(AB,CD)} + \vec{\alpha}_{AB,CD} = \text{conv} \left\{ \left[\begin{array}{c} -n \\ 1 \end{array} \right], \left[\begin{array}{c} 0 \\ 1 \end{array} \right] \right\}$$

and

$$\mathcal{V}_{G_2,(CD,AB)} - \vec{\alpha}_{AB,CD} = \left\{ \left[\begin{array}{c} -2 \\ -2 \end{array} \right] \right\}.$$

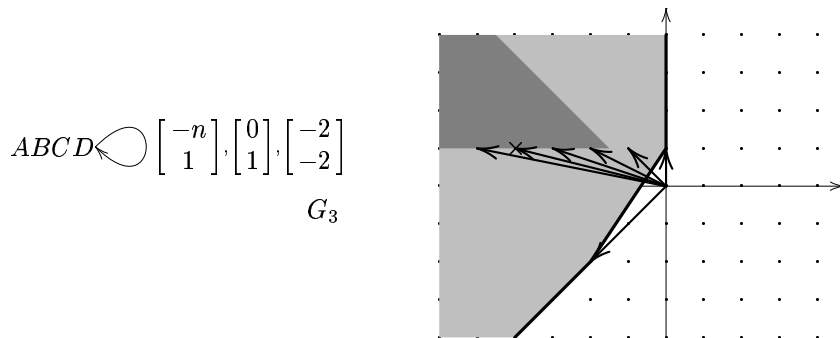


Figure 21: Graph with AB and CD combined. Cf. Figure 15

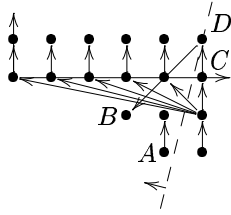


Figure 22: Complete translation

The dependence space figure now contains all the distance vectors of the final translation, which is shown in Figure 22. The ordering vector in this figure is $\vec{\pi}^T = [-4 \ 1]$, which is part of $\mathcal{P}_O(\mathcal{R}_{G_3})$ as shown in Figure 21 on the previous page.

6 The ordering phase

6.1 The ordering of the phases

In Section 4, we proved that incremental translation is feasible when performed in between a linear transformation and an ordering phase. Corollary 2, however, shows that incremental translation can also be performed *after* the ordering phase. In this section we will discuss the relative merits of these two approaches.

Incremental translation after ordering can be performed using the same algorithms (i.e., algorithms 1 and 2) as translation before ordering. Rather than using (18) for choosing a valid relative offset, however, (29) should be used. Also, it is not necessary to maintain the complete distance vector polytope \mathcal{D}_e for each edge in G , but rather a single scalar is needed. This significantly simplifies the operations in Algorithm 2. To see that only a single scalar is needed for each edge, consider the set of constraints of the form (27) and (28). In both cases, only the constraint based on the indirect distance vector $\vec{\delta}$ with minimal time delay ($\vec{\pi}^T \vec{\delta}$) is needed, since it is more restrictive than the constraints of the same set for other indirect distance vectors. The indirect distance vector with minimal time delay can only be the sum of direct distance vectors with minimal time delays and hence only these direct distance vectors have to be remembered. The downside is that we have to remember the indirect distance vector with minimal time delay not only for each pair of nodes connected by a path, but for each pair of nodes connected by a path and each possible length of such a path, as constraints (27) and (28) depend on this length.

Figure 23 presents an overview of the differences between ordering-first and translation-first (the final column is explained later). The feasibility criterion for ordering-first is slightly more complicated, but, by lemmas 8 and 9, feasibility for translation-first implies feasibility for ordering-first. The differences in (indirect) distance vectors has been discussed above. The feasible region for (the time delay of) the relative offset is very simple for ordering-first: it is a single interval (where one of the ends of the interval may extend to infinity). For translation-first, the feasible region is a non-convex multi-dimensional region, which requires the calculation of self-dependence cones which is not needed for ordering-first. The self-dependence cone for the *initial* graph is still needed to determine feasibility. Note that it may not be required to calculate the feasible region for translation-first: an arbitrary valid relative offset can be obtained via the technique in the proof of lemma 6. Also, a suitable cost function may a priori eliminate a large part of the search space.

	Translation first	Ordering first	Loop fusion
Feasibility	$N \geq 2 \wedge \vec{0} \notin R_{G_0}$ (Theorem 4)	$\exists \vec{\pi} \in \mathcal{P}_O(\mathcal{R}_{G_0}) : \gcd \vec{\pi} = 1$ (Lemma 7)	$\vec{0} \notin R_{G_0}$
Distance vectors	$\mathcal{D}_{G,e}$	$\min \vec{\pi}^T \mathcal{D}_{G,e}$	$\min \vec{\pi}^T \mathcal{D}_{G,e}$
Indirect distance vectors	$\mathcal{V}_{G,(p_1,p_2)}$	$\min \vec{\pi}^T \mathcal{V}_{G,(p_1,p_2),l}$	$\min \vec{\pi}^T \mathcal{V}_{G,(p_1,p_2)}$
Self-dependence cone	required	not required	not required
Legal relative offsets	$\vec{\alpha}_{p_1,p_2} \notin (\overline{R}_G + \mathcal{V}_{G,(p_2,p_1)}) \cup (-(\overline{R}_G + \mathcal{V}_{G,(p_1,p_2)}))$ (Equation (18))	$l_1 - \vec{\pi}^T \vec{\delta}_1 \leq \vec{\pi}^T \vec{\alpha} \leq \vec{\pi}^T \vec{\delta}_2 - l_2$ (Equation (29))	$-\vec{\pi}^T \vec{\delta}_1 \leq \vec{\pi}^T \vec{\alpha} \leq \vec{\pi}^T \vec{\delta}_2$ (Equation (34))
Cost functions	Only geometrical	Geometrical + time related	Geometrical + time related

Figure 23: Comparison of translation before or after ordering



Figure 24: Locality in space vs. locality in time.

As to cost functions, Danckaert et al. (2000) note that prior to ordering, only geometrical features can be used in cost functions. This restriction makes it hard to choose good relative offsets. For example, suppose we want to place the target of a dependence as close as possible to the source and suppose that we have to choose between positions 1 and 2 in Figure 24. When only considering geometrical features, 2 is the better choice since it is closer in space to the source of the dependence. If, however, the ordering phase would choose the ordering vector shown in the figure, then 1 is a lot closer in *time* than 2.

In general, if the translation step makes its choices oblivious of the following ordering step, then this ordering step will have a hard time optimizing all the dependences since it only has the freedom of selecting a single ordering vector for all the statements in the common iteration space. In order to make any time-related decisions, the translation step would have to estimate the position of the ordering vector or at least some limited range for this position. By placing the ordering step before the translation step, timing information is directly available. The effect on the ordering step is that the ordering vector has to be chosen based not on all the dependences, but only on the self-dependences.

Example 11 For an example of ordering before translation, we need only refer to Section 2 as in building the equivalent affine mapping and ordering vector combination, we chose the ordering vector before determining the translation. As shown in example 10, the averaged self-dependence cone without linear transformation is $\mathcal{R}_{G_0} = (-\infty, -2/3]$. The corresponding valid ordering polyhedron $\mathcal{P}_O(\mathcal{R}_{G_0}) = (-\infty, -3/2]$ does not contain any element with $\text{gcd } 1$. This explains why we had to scale the iteration domains by a factor k at least 2. With $k = 2$, we have $\mathcal{R}_{G'_0} = (-\infty, -4/3]$ and $-1 \in \mathcal{P}_O(\mathcal{R}_{G'_0}) = (-\infty, -3/4]$, which satisfies the requirements of lemma 7.

Note that the feasibility criterion mentioned in Figure 23 on the previous page is a criterion for the feasibility of the combination of translation and ordering. If the ordering is performed prior to the translation, it must still ensure that a valid translation is possible. According to lemma 7, the ordering vector has to be chosen from the valid ordering polyhedron for the averaged self-dependence cone for the initial graph and its gcd has to be 1.

6.2 Loop fusion

As mentioned above, a drawback of ordering-first is that an indirect distance vector has to be stored for each possible length of path between two nodes in the dependence graph. This can be alleviated by relaxing constraint (26):

$$\vec{\pi}^T \vec{\delta} \geq l$$

to

$$\vec{\pi}^T \vec{\delta} \geq 1 \tag{32}$$

for self-dependence distance vectors in the initial graph and to

$$\vec{\pi}^T \vec{\delta} \geq 0 \tag{33}$$

for additional self-dependence distance vectors in the derived dependence graphs. As a result, the constraint on valid relative offsets (29) is relaxed to:

$$-\vec{\pi}^T \vec{\delta}_1 \leq \vec{\pi}^T \vec{\alpha} \leq \vec{\pi}^T \vec{\delta}_2. \quad (34)$$

Due to this simplification, though, some self-dependence distance vectors in the final graph, which correspond to translated distance vectors in the original graph, may have a time delay of 0, which is not allowed. To solve this, we need to determine an ordering for depending operations that are scheduled at the same time. Consider the graph consisting of all the edge in the dependence graph with translated distance vectors of zero time delay. Each component in this graph is a DAG (Directed Acyclic Graph), since the time delay over any circuit in the dependence graph is at least 1. A valid execution order can therefore be determined by a topological sort of the components of the graph. This is in fact how loop fusion with loop bumping (Manjikian and Abdelrahman 1995; Fraboulet et al. 1999; Song et al. 2001) works, although they typically do not explicitly address this issue.

One way of modeling the above construction is to extend the dimension of the ordering and to apply an additional translation in the additional dimension. If we determine a one-dimensional ordering $\vec{\pi}$ in the ordering step, as we have done up to this point in this document, then the final (two-dimensional) schedule becomes

$$\theta_X : \vec{i} \mapsto \begin{bmatrix} \vec{\pi}^T \\ \vec{0}^T \end{bmatrix} A_X \vec{i} + \begin{bmatrix} \vec{\pi}^T \\ \vec{0}^T \end{bmatrix} \vec{a}_X + \begin{bmatrix} 0 \\ a_{X,N+1} \end{bmatrix} \quad (35)$$

(compare with (3)). As usual with multi-dimensional schedules, the iterations are executed in lexicographical order.

Due to the relaxed constraint on self-dependences (32), it is sufficient that an ordering vector with gcd 1 exist in the valid ordering polyhedron for the self-dependence cone (rather than the averaged self-dependence cone). It is therefore sufficient that the valid ordering polyhedron be non-empty. The case for $N \geq 2$ derives from lemma 8 and for $N = 1$, we know that either 1 or -1 is a member of $\mathcal{P}_O(R_{G_0})$ if it non-empty since every distance vector is integer. This explains the final column of Figure 23 on page 29.

The feasibility criterion for the translation phase itself is in this case that the ordering vector be chosen from the valid ordering polyhedron for the (non-averaged) self-dependence cone. This simply means that (32) should hold for all self-dependences.

6.3 Linear transformation

In Section 6.1 we argued that it is preferable to perform the ordering before the translation, because it is possible, simpler and allows for more accurate cost functions. Similar reasoning holds for the linear transformation phase. As we show in the following lemma, the same effect of a change in ordering can be affected by linearly transforming the common iteration space as a whole. As with the translation phase, a pre-ordering linear transformation phase can only use geometrical cost functions such as *regularity* (Verdoolaege et al. 2001), whereas a post-ordering linear transformation phase can also use time-related cost functions such as locality (in time).

Lemma 10 *Applying any ordering vector $\vec{\pi}^T$ with gcd of its elements 1 is equivalent to applying any other ordering vector $\vec{\pi}'^T$ with gcd of its elements 1 to a unimodularly transformed common iteration space.*

$$\vec{\pi}^T = \vec{\pi}'^T U \quad (36)$$

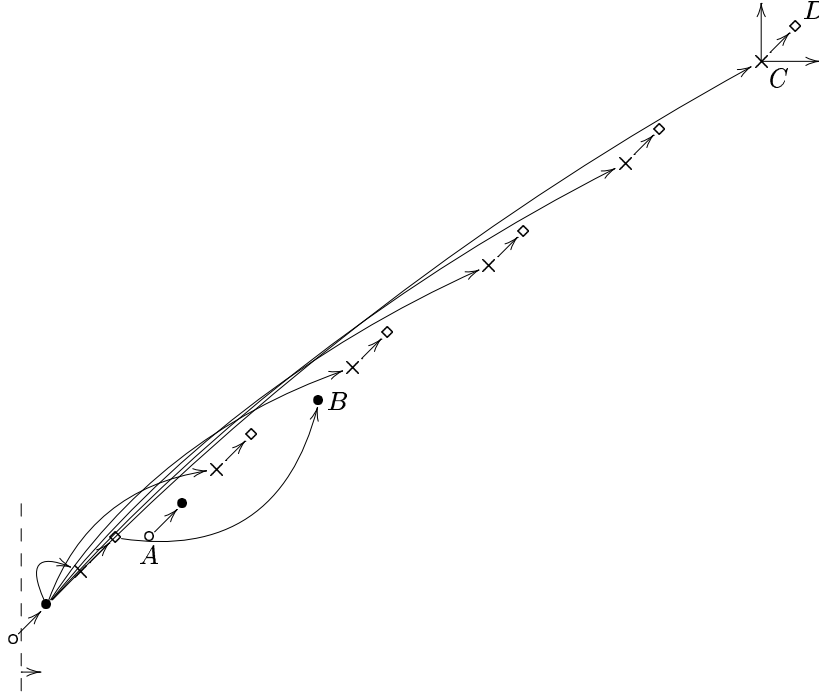


Figure 25: Equivalent placement for ordering vector $\vec{\pi}^T = [1 \ 0]$

Proof Since the gcd of their elements is 1, both ordering vectors can be extended to unimodular matrices K and K' (Bik 1996). $K'^{-1}K$ can be used as unimodular U in (36). \square

In lemma 10, we only consider ordering vectors with gcd of the elements 1 since an ordering vector with a different gcd determines the same relative ordering of the statement iterations as the ordering vector divided by its gcd. If the gcd of the two ordering vectors are g and g' respectively, then we can also apply the transformation $g/g'U$ to the common iteration space, but this may map iteration points to non integer values.

Example 12 In section 5 we obtained the mapping shown in Figure 22 on page 28 and we choose the ordering vector $\vec{\pi}^T = [-4 \ 1]$, also shown in the same figure. Suppose we want an equivalent placement/ordering combination where the ordering is given by $\vec{\pi}'^T = [1 \ 0]$. Following the proof of Lemma 10, we unimodularly extend both ordering vectors and obtain

$$K = \begin{bmatrix} -4 & 1 \\ -3 & 1 \end{bmatrix} \quad \text{and} \quad K' = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},$$

with the determinants of K and K' respectively -1 and 1 . Since K' is the identity matrix, $U = K'^{-1}K = K$, which is the matrix with which we have to transform all iteration domains in the common iteration space to obtain an equivalent placement/ordering combination. For example, the iteration domain of statement B in Figure 22 contains the three points $[3 \ -1]^T$, $[4 \ -1]^T$ and $[5 \ -1]^T$, which are transformed to $[-13 \ -10]^T$, $[-17 \ -13]^T$ and $[-21 \ -16]^T$ respectively. This alternative placement/ordering is shown in Figure 25. Note that, as guaranteed by Lemma 10, the effective execution order of the statement instances is unaltered.

6.4 Multi-dimensional ordering

So far, we have only considered one-dimensional orderings, which is what van Swaaij (1992) uses. However, in determining a one-dimensional ordering, he first builds a multi-dimensional ordering, which he transforms into a one-dimensional ordering through loop coalescing. In a multi-dimensional ordering, the schedule “time” is a vector and the execution order is determined by the lexicographical order of these time vectors. Loop coalescing maintains this execution order.

Danckaert et al. (2000) propose to apply conventional scheduling or space-time mapping techniques (Feautrier 1992; Lamport 1974; Lengauer 1993) on the common iteration space in the ordering phase. Two of the referred to techniques consist of a one-dimensional schedule, the other of a multi-dimensional schedule. Note that Feautrier (1992) constructs a (piece-wise) affine schedule *per statement*. Presumably what is meant is that the whole of all the iteration domains in the common iteration space is seen as a single (compound) statement, i.e., that the same affine transformation is applied to all iteration domains. Otherwise, the ordering phase would not use any of the information constructed in the previous two phases, rendering them quite useless.

We are therefore justified in also considering multi-dimensional orderings. In fact, as we show in the next lemma, the problems of finding a one-dimensional or a multi-dimensional ordering are equivalent (for non-parametrized problems) in the sense that any valid ordering vector (with gcd of its elements 1) can be transformed into a valid (unimodular) multi-dimensional ordering and vice versa. Again, it is sufficient to only consider ordering vectors with gcd 1 and unimodular multi-dimensional orderings as scaling does not affect the relative execution order.

Lemma 11 *Any valid ordering vector with gcd 1 is equivalent to a valid unimodular multi-dimensional ordering and vice versa.*

Proof The unimodular extension (Bik 1996) of the ordering vector is a valid multi-dimensional ordering since its first row separates all the dependences. Conversely, a unimodular ordering can be converted into an ordering vector through loop coalescing. That is, let K be the multi-dimensional ordering and let G^* be the graph corresponding to the common iteration space. Let P^* be the union of all iteration domains in G^* after ordering, i.e.,

$$P^* = K \bigcup_{P \in \mathcal{P}_p} P \quad \text{with} \quad V_{G^*} = \{p\}.$$

Let s_i be the extent of the bounding box of P^* in dimension i , i.e.,

$$s_i = \max \vec{e}_i^T P^* - \min \vec{e}_i^T P^* + 1,$$

then $\vec{\pi}^T = \vec{\beta}^T K$ with

$$\beta_i = \prod_{j=i+1}^N s_j$$

is a valid ordering vector, since it preserves the execution order of K . The gcd of $\vec{\pi}$ is 1 because the gcd of $\vec{\beta}$ is 1 ($\beta_N = 1$) and because K is unimodular. \square

With a multi-dimensional ordering, we can identify a loop with each dimension of the ordering to generate source code containing loop nests and corresponding to the transformed common iteration space (Quilleré et al. 2000; Bastoul 2002).

Example 13 The ordering vector shown in Figure 22 on page 28 is $\vec{\pi}^T = [-4 \ 1]$. An equivalent multi-dimensional ordering is

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

```

for (i = 0; i <= n ; ++i) {
  if (i <= 1)
    c[n+2-i] = f1(); /* A */
  if (i <= 2)
    a[2-i] = f4(c[n+2-i]); /* B */
  b[n-i] = f2(a[2]); /* C */
  c[n-i] = f3(b[n-i]); /* D */
}

```

Listing 1: Target program

i.e., the outer loop corresponds to the first dimension and the inner loop to the second dimension, with the outer loop executed in reversed order. The generated code is shown in Listing 1. Note that the j -loop is not explicit since no statement has iterations with different j -values.

The use of multi-dimensional orderings also makes the equivalence between ordering and linear transformation of the common iteration space more explicit. The final schedule is

$$\theta_X : \vec{i} \mapsto \Pi A_X \vec{i} + \Pi \vec{a}_X, \quad (37)$$

which clearly shows that replacing the multi-dimensional ordering Π with ΠU is equivalent to transforming the common iteration space with U^T . Combined with the technique (35) from Section 6.2, adding an extra dimension to eliminate the need to remember the circuit lengths, we obtain:

$$\theta_X : \vec{i} \mapsto \begin{bmatrix} \Pi \\ \vec{0}^T \end{bmatrix} A_X \vec{i} + \begin{bmatrix} \Pi \\ \vec{0}^T \end{bmatrix} \vec{a}_X + \begin{bmatrix} 0 \\ a_{X,N+1} \end{bmatrix},$$

or equivalently

$$\theta_X : \vec{i} \mapsto \tilde{\Pi} \tilde{A}_X \vec{i}' + \tilde{\Pi} \vec{a}'_X,$$

with

$$\tilde{X} = \begin{bmatrix} X & \vec{0} \\ \vec{0}^T & 1 \end{bmatrix}, \quad \vec{i}' = \begin{bmatrix} \vec{i} \\ 0 \end{bmatrix} \quad \text{and} \quad \vec{a}'_X = \begin{bmatrix} \vec{a}_X \\ a_{X,N+1} \end{bmatrix}.$$

6.5 No ordering substep

If we perform the ordering phase prior to linear transformation and translation, as argued for in sections 6.1 and 6.3, we can simply choose the default ordering, i.e., with the identity matrix as multi-dimensional ordering, essentially eliminating the ordering phase. This default ordering corresponds to a lexicographical ordering on the iteration vectors in the common iteration space. Eliminating Π from (37), we obtain

$$\theta_X : \vec{i} \mapsto A_X \vec{i} + \vec{a}_X,$$

which is exactly (multi-dimensional) affine-by-statement scheduling. Adding the extra dimension, we obtain

$$\theta_X : \vec{i} \mapsto \tilde{A}_X \vec{i}' + \vec{a}'_X.$$

The feasibility criterion for the translation step (32) in this case is that all self-dependence distance vectors should be lexicographically strictly positive:

$$\vec{\delta} \succ \vec{0}.$$

Since the lexicographically smallest strictly positive value is \vec{e}_N , this is equivalent to

$$\vec{\delta} \succcurlyeq \vec{e}_N. \quad (38)$$

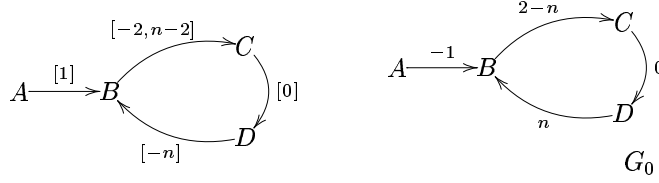


Figure 26: Initial dependence graph with dependence polytopes on the left and with minimal distance vectors on the right (after reverse).

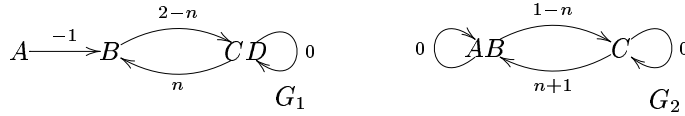


Figure 27: Intermediate dependence graphs

Likewise, during loop fusion, additional self-dependence distance vectors should be lexicographically positive

$$\vec{\delta} \succcurlyeq \vec{0},$$

and the valid relative offsets constraint (34) becomes

$$-\vec{\delta}_1 \preccurlyeq \vec{\alpha} \preccurlyeq \vec{\delta}_2. \quad (39)$$

Note that in all these constraints, it is the *minimal* distance vector, i.e., the lexicographically minimal element of \mathcal{V}_e for a given edge e , that determines whether it holds. As such, there is no need to actually store \mathcal{V}_e , or even \mathcal{D}_e . Rather, we only need to store the minimal element.

Example 14 Once more we consider the example from Section 2, but this time using the loop fusion technique from Section 6.2 and without an ordering step. Figure 26 shows the initial dependence graph on the left. The graph contains a single cycle, with indirect distance vectors in the range $[-n-2, -2]$. Since these are negative, (38) does not hold and a linear transformation needs to be applied. In this case we can simply reverse all the iteration domains. The resulting minimal distance “vectors” are shown in the same figure on the right. The single minimal self-dependence distance vector is 2 and satisfies (38). Note, though, that it does not satisfy (26), as the length of the cycle is 3, so this program cannot be fused without the relaxation discussed in Section 6.2.

We combine the nodes in the same order as in sections 2 and 5. When combining C and D , the constraint on the relative offset (39) is

$$-0 \preccurlyeq \vec{\alpha}_{C,D} \preccurlyeq 2$$

and we choose $\vec{\alpha}_{C,D} = 0$. The resulting dependence graph G_1 is shown in Figure 27. Next, A and B are combined at an offset of $\vec{\alpha}_{A,B} = 1$, resulting in G_2 and finally AB and CD are combined at an offset of $\vec{\alpha}_{AB,CD} = n-1$, satisfying the constraint on the relative offset $-(1-n) \preccurlyeq \vec{\alpha}_{AB,CD} \preccurlyeq n+1$. We arbitrarily choose $\vec{a}_A = \vec{a}_{ABCD} = 0$ and obtain $\vec{a}_B = 1$ and $\vec{a}_C = \vec{a}_D = n-1$.

Figure 28 shows the dependence graph after translation. Some of the translated minimal distance vectors are zero and so a topological sort to determine the offset in the innermost dimension is required, which is also shown in the same figure. The resulting fused program is shown in Figure 29, with the common iteration space on the left and generated code on the right. In the iteration space, the horizontal axis represents the single dimension of the problem and the vertical axis represents the additional dimension that orders the

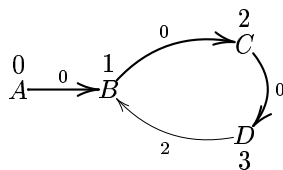


Figure 28: Translated dependence graph

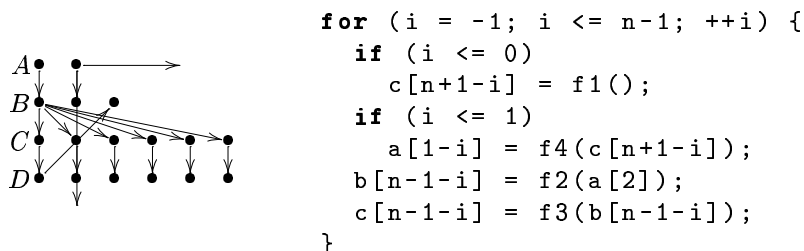


Figure 29: Complete fusion

statements inside the inner loop. The ordering of the iterations is equivalent to the one shown in Figure 22 on page 28.

To close this section, we wish to reconsider the motivation for using an ordering phase. First note that both approaches (with or without ordering phase) have the same set of solutions with respect to the final execution order of the statement instances. They only differ in *how* they obtain such a solution. In comparing his methodology to affine-by-statement scheduling (Feautrier 1992; Kelly and Pugh 1993; Feautrier 1996), Danckaert (2001) mentions the following advantages of his approach:

- The complexity is reduced. In each phase, a cost function can be used which is more simple than a combined cost function.
- It is a new approach compared to existing work. The latter has not led to an automated loop transformation methodology.

He also mentions the following disadvantages:

- All transformations can be performed in many different ways, by different combinations of placement and ordering. Future work should investigate whether at least part of this redundancy can be removed.
- Only heuristic (*i.e.*, *non time-related*) cost functions are possible during the placement, such that potentially a placement could be chosen which does not allow a good ordering (and final solution) anymore.

The second advantage only means that it may be fruitful to consider a different approach. It does not mean that this different approach has immediately led to an automated loop transformation methodology. In fact, this document is part of an attempt to attain this goal. As to the claim that complexity is reduced, it is based on the assumption that the ordering phase is orthogonal to the placement phase (the combination of linear transformation and translation) and that simpler cost functions can be used in both phases. However, no formal

```

for (i = 0; i <= n ; ++i)
    a[i] = fa(a[i-1]);
for (i = 0; i <= n ; ++i)
    b[i] = fb(a[i]);
for (i = 0; i <= n ; ++i)
    c[i] = fc(c[i-1], b[i], b[n-i]);
for (i = 0; i <= n ; ++i)
    d[i] = fd(a[i]);
for (i = 0; i <= n ; ++i)
    e[i] = fe(e[i-1], d[i], d[n-i]);
for (i = 0; i <= n ; ++i)
    f[i] = ff(a[i]);
for (i = 0; i <= n ; ++i)
    g[i] = fg(c[i-1], f[i], f[n-i]);
result = c[n] + e[n] + g[n];

```

Listing 2: Program with bad locality

proofs of these assumptions are given and the results from Section 6.1 rather indicate the opposite, as we show that assuming a fixed ordering simplifies the problem of finding a valid translation.

7 Optimality

Although the main focus of this document is the *feasibility* of (incremental) translation, we will also consider some cost functions for *optimality*, mainly locality, in this section. Throughout this section we assume that \mathcal{D}_e is the *multi*-set of all distance vectors over an edge e , rather than just the set, i.e., the number of occurrences of each distance vector is of importance.

Optimizing locality between two accesses to the same memory element can have a positive effect on the number of accesses. Optimizing locality over (flow) dependences, i.e., between a write and a read, reduces the life-times of array elements, which can have an additional effect on the memory size. By reducing the distance between successive accesses to the same memory element, the likelihood of that element residing in a register or a cache increases, reducing the number of accesses to slower memories in the memory hierarchy. By reducing the maximal distance between a write and a read access, the array element needs to be stored for a shorter amount of time, freeing up memory for other elements of the same or other arrays, which in general reduces the total memory requirements.

It should be noted that locality is not *always* good for memory size. Compare, for example, the programs in listings 2 and 3. The first has bad locality, since the elements of a , calculated in the first loop, are used in the loop that calculates f , but 4 loops separate these two loops. The program in Listing 3 is maximally merged and values only need to be kept over a single loop. However, the number of array elements that need to be stored is larger in the second program, since three arrays: b , d and f are alive between the two loops, whereas in the first program at most two arrays are simultaneously alive.

7.1 After ordering

If we assume a fixed ordering during translation, we can optimize locality in the outer-most dimension first and then continue to inner dimensions. Consider the

```

for (i = 0; i <= n ; ++i) {
    a[i] = fa(a[i-1]);
    b[i] = fb(a[i]);
    d[i] = fd(a[i]);
    f[i] = ff(a[i]);
}
for (i = 0; i <= n ; ++i) {
    c[i] = fc(c[i-1], b[i], b[n-i]);
    e[i] = fe(e[i-1], d[i], d[n-i]);
    g[i] = fg(c[i-1], f[i], f[n-i]);
}
result = c[n] + e[n] + g[n];

```

Listing 3: Program with good locality

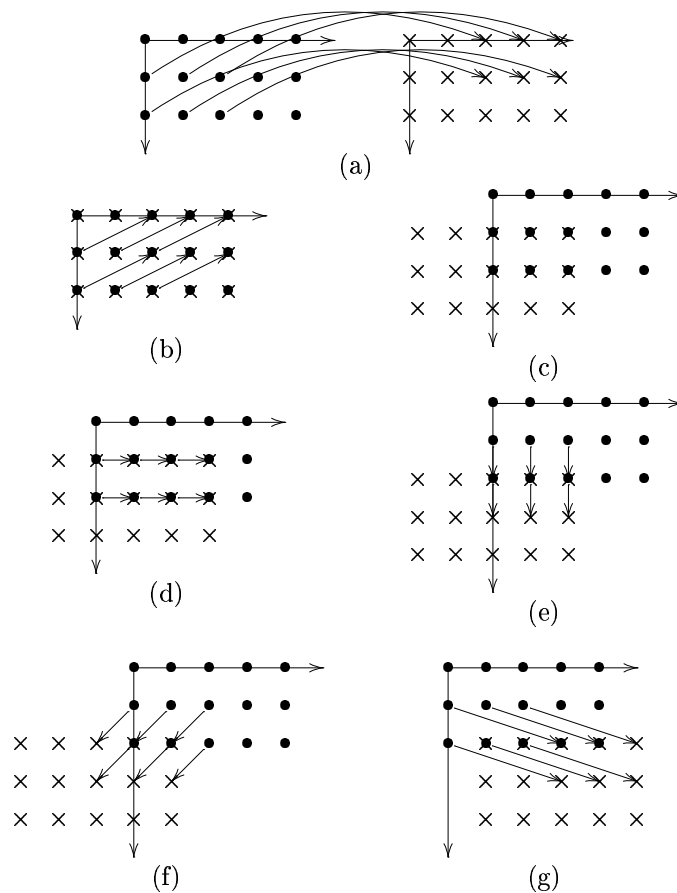


Figure 30: Optimizing locality. (a) a dependence. (b) an illegal translation. (c) optimal translation. (d) suboptimal in inner dimension. (e) suboptimal in outer dimension. (f) minimal shift. (g) maximal shift

example dependence in Figure 30a. The two polytopes are $m \times n$ and each of the $(m-1)(n-2)$ distance vectors is equal to $\vec{\delta} = (-1, 2)^T$, as demonstrated in the (illegal) translation in (b). The outer loop is drawn top to bottom and the inner loop left to right. Figure 30c shows the optimal translation for this pair of polytopes, with a relative offset of $\vec{\alpha} = (1, -2)^T$. To see that is important to optimize the locality in the outer dimension first, compare (d) and (e). In (d) the consumption polytope is shifted one to the right (the inner dimension) compared to (c), which increases the number of live elements by 1, whereas in (e) the consumption polytope is shifted one down (the outer dimension) compared to (c), which increases the number of live elements by $n-2$.

In the case of a single uniform dependence, as in Figure 30, the optimal translation is obtained by ensuring that the translated distance vector is equal to the null vector, assuming that we use the technique in Section 6.2. In any case the translated distance vectors will have to be lexicographically positive (33). If, in optimizing the offset in a certain dimension, all coefficients of some translated distance vector are already made zero in the outer dimensions, the coefficient for the current dimension will be constrained and we can simply try to minimize it in order to optimize locality. However, if all translated distance vectors have a non-zero coefficient in one outer dimension, then there may be no such constraint. Minimizing the coefficient would lead to a value of $-\infty$. Figure 30f shows that we need not consider such extreme values. The number of live elements is the same as in (e) and further shifting to the left does not change this value. Likewise, (g) shows the maximal shift in inner dimension for the same shift in outer dimension that changes the number of live elements, and any additional shift to the right should not incur any locality cost.

Another option, and the one taken by Fraboulet et al. (1999), is to only consider (uniform) dependences in inner dimensions with translated distance vectors of all zeros in the outer dimensions. However, as can be seen from the difference between figures 30e and 30g, it may still be useful to optimize the inner dimensions in any case, even for uniform dependences, to which Fraboulet et al. (1999) limit themselves. On the other hand, if the translated distance is non-zero in an outer dimension, then either there is a constraint due to another dependence in the same direction preventing the distance to become zero or there is a dependence in the other direction, pulling the relative offset in the opposite direction. In both cases, the dependence under consideration is not (as) relevant and so probably should be removed. In Section 9 we will briefly return to this issue.

Summarizing this section, in determining a relative offset in step 4 of Algorithm 1, we work from the outermost dimension inward. For each dimension, the coefficient of the relative offset is optimized by minimizing the sum of terms for each dependence, where each term has the following form:

$$w (|x - \beta_i|_+ - |x - \gamma_i|_+). \quad (40)$$

In this term, w is the weight of the dependence, x is the coefficient to be determined and β_i and γ_i can be derived from the production and consumption polytopes and the dependence function and may also depend on the coefficients of the relative offset in outer dimensions for non-rectangular domains; $|\cdot|_+$ is the function that maps a value to itself if it is positive and to zero otherwise (Cf. Figure 31). The function in (40) is shown in Figure 32.

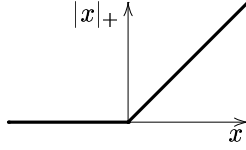


Figure 31: The function $|\cdot|_+$

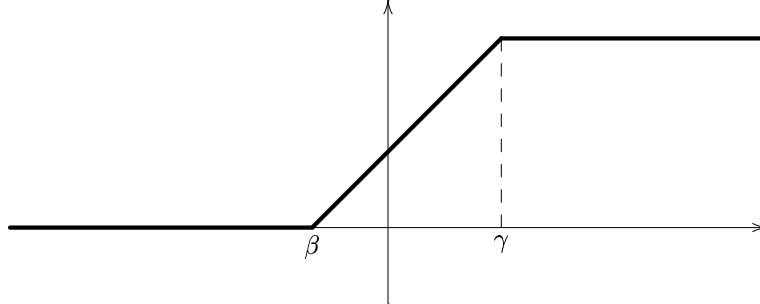


Figure 32: Locality cost function in each dimension for each dependence.

7.2 Before ordering

7.2.1 Distance length

If the ordering is not known prior to translation, then optimizing locality is more difficult. A straightforward cost function is the sum of the lengths of all translated distance vectors, i.e.,

$$\sum_{\vec{\delta} \in \mathcal{D}_{(p_1, p_2)}} \|\vec{\delta} + \vec{\alpha}_{p_1, p_2}\| + \sum_{\vec{\delta} \in \mathcal{D}_{(p_2, p_1)}} \|\vec{\delta} - \vec{\alpha}_{p_1, p_2}\|,$$

or equivalently

$$\sum_{\vec{\delta} \in \mathcal{D}_{(p_1, p_2)}^T \cup \mathcal{D}_{(p_2, p_1)}^T} \|\vec{\delta}\|. \quad (41)$$

To derive a linear cost function from (41), the 1-norm can be used, also known as the Manhattan distance. This cost function was already used by van Swaij (1992), but it suffers from the problems explained in Section 6.1.

Example 15 Consider the choice for the relative offset between A and B in Section 5 after C and D have already been joined. The choice $\vec{\alpha}_{A, B} = [-1 \ 1]^T$ leads to two translated distance vectors $[0 \ 1]^T$, with combined length 2, which is minimal. Another choice is $\vec{\alpha}'_{A, B} = [-2 \ 0]^T$, indicated in Figure 18 on page 26 by a circle, which leads to two translated distance vectors $[-1 \ 0]^T$, also with combined length 2. The first option is preferable however, since the distance vectors obtained are identical to those generated by the dependence between C and D . Therefore, an ordering vector that is optimal for the latter is also optimal for the former. For the second option, this does not hold. Cost function (41) does not capture this preference.

7.2.2 Deviation from dependence cone

We can improve on this cost function by including the dependence cone. If the dependence cone increases during a translation step, the valid ordering polytope decreases and, in general, this will lead to worse solutions. We therefore want

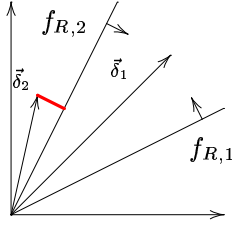


Figure 33: Deviation from the dependence cone

to minimize the increase of the dependence cone. Suppose the dependence full cone is given by the following equation:

$$\bar{R}_G = \left\{ \vec{x} \in \mathbb{Z}^n \mid F_R \vec{x} \geq \vec{0} \right\}.$$

The following cost function can then be used to measure the deviation from the dependence cone:

$$\sum_{\vec{\delta} \in \mathcal{D}_{(p_1, p_2)}^T \cup \mathcal{D}_{(p_2, p_1)}^T} \sum_s | -\vec{f}_{R,s}^T \vec{\delta} |_+, \quad (42)$$

where $\vec{f}_{R,s}^T$ are the constraints defining the dependence full cone, i.e., the rows of F_R . The effect of this cost function is illustrated in Figure 33. The distance vector $\vec{\delta}_1$ lies within the current dependence full cone and does not incur any cost. The distance vector $\vec{\delta}_2$, however, violates constraint $f_{R,2}$ and therefore does incur a cost, as indicated on the figure. Note that (42) should be used in conjunction with (41). A disadvantage of this cost function is that as soon as the self-dependence cone is of “full” size, it has no effect.

Example 16 As in the previous example, we consider the choice $\vec{\alpha}_{A,B}$ in Section 5. The dependence full cone at this stage is

$$\bar{R}_{G_1} = \left\{ \vec{x} \in \mathbb{Z}^n \mid \begin{bmatrix} -1 & 2 \\ -1 & 0 \end{bmatrix} \vec{x} \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right\} \quad (43)$$

Since the resulting translated distance vectors for both choices, $[0 \ 1]^T$ and $[-1 \ 0]^T$, both satisfy the constraints, cost function (42) does not help us in this case. Suppose, however, that the dependence between D and B did not exist and that there hence would not be any self-dependence. In this case, the dependence full cone would be

$$\bar{R}_{G_1} = \text{cone} \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\} = \left\{ \vec{x} \in \mathbb{Z}^n \mid \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \end{bmatrix} \vec{x} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \right\}$$

and cost function (42) would be able to differentiate between $[0 \ 1]^T$ and $[-1 \ 0]^T$, since the latter does not satisfy the first constraint, whereas cost function (41) would still not be able to make any distinction.

7.2.3 Deviation from ideal distance vector

The final cost function we discuss attempts to estimate the ordering vector that will be produced by the ordering phase and then to optimize the new translated distance vectors with respect to this ordering vector. The cost function used in determining the optimal ordering vector is assumed to be locality as well, i.e., the ordering vector $\vec{\pi}$ is selected that minimizes

$$\sum_{\vec{\delta} \in \mathcal{D}_{(p,p)}} \vec{\pi}^T \vec{\delta} \quad \text{with} \quad V_{G^*} = \{p\},$$

subject to

$$\vec{\pi}^T \vec{\delta} \geq 1 \quad \forall \vec{\delta} \in \mathcal{D}_{(p,p)}.$$

For estimating the optimal ordering vector at a certain stage in the translation phase, we can use the translated distance vectors that are already known at that stage, i.e., the direct self-dependence distance vector. The ordering vector minimizing the scalar product with the sum of those distance vectors, is the same as the one minimizing the scalar product with their average:

$$\vec{\delta} = \left(\frac{1}{n}\right) \sum_{p \in V_G} \sum_{\vec{\delta} \in \mathcal{D}_{(p,p)}} \vec{\delta},$$

with n the number of distance vectors. Although $\vec{\pi}$ is optimal with respect to the average distance vector, the average distance vector may not be optimal with respect to $\vec{\pi}$ since $\vec{\pi}$ is constrained and may not reach its unconstrained optimum. We therefore proceed to construct an “ideal distance vector” and the cost function will measure the deviation from this ideal distance vector.

The ideal distance vector should be part of the current self-dependence cone, but it should be as close as possible to its border, since it is the border of the self-dependence cone that determines the constraints on the ordering vector. More precisely, the extremal rays of the self-dependence cone determine these constraints, since any element of the self-dependence cone is a convex combination of its extremal rays and therefore so are the constraints. Our choice for ideal distance vector is the extremal ray of the self-dependence cone that is “closest” to average distance vector.

Let the dependence full cone be given by

$$\overline{R}_G = \left\{ \vec{x} \in \mathbb{Z}^n \mid G_R \vec{x} = \vec{0}, F_R \vec{x} \geq \vec{0} \right\},$$

with $F_R \vec{x} \geq \vec{0}$ an independent set of t inequalities that does not entail any implicit equalities. Furthermore, we assume, without loss of generality, that the inequalities are ordered as follows:

$$\vec{f}_{R,i}^T \vec{\delta} \leq \vec{f}_{R,j}^T \vec{\delta} \quad \forall i < j.$$

Let s be the number of rows in G_R , i.e., the number of equalities defining the self-dependence full cone, then by intersecting \overline{R}_G with $N-s-1$ of its supporting hyperplanes defined by its inequalities, we obtain a one-dimensional face of \overline{R}_G , i.e., one of its extremal rays. Let the ideal distance vector (direction) be this extremal ray:

$$\begin{aligned} & \overline{R}_G \cap \left\{ \vec{x} \in \mathbb{Z}^n \mid \vec{f}_{R,i}^T \vec{x} = 0, \forall 1 \leq i < N-s \right\} \\ &= \left\{ \vec{x} \in \mathbb{Z}^n \mid G_R \vec{x} = \vec{0}, \vec{f}_{R,i}^T \vec{x} = 0, \forall 1 \leq i < N-s, \vec{f}_{R,k}^T \vec{x} \geq 0 \right\}, \end{aligned}$$

where $\vec{f}_{R,k}^T \vec{x} \geq 0$ is one of the remaining inequalities, i.e., $k \geq N-s$.

We want each of the new translated distance vectors to be as close as possible to this ideal distance vector, i.e., for each of the $N-s-1$ $\vec{f}_{R,i}^T$, we want $\vec{f}_{R,i}^T \vec{\delta}$ to be as close to zero as possible. In the case of non-uniform or multiple dependences, it will not be possible to make this scalar product zero for all distance vectors, but if we cannot make it zero, we want to err on the positive side, since then, at least, the distance vector will not extend the self-dependence cone to the wrong side. That is, for each $\vec{f}_{R,i}^T$ with $i < N-s$, we want

$$\vec{f}_{R,i}^T (\vec{\delta} + \vec{\alpha}_{p_1,p_2}) \geq 0 \quad \forall \vec{\delta} \in \mathcal{D}_{(p_1,p_2)}$$

and

$$\vec{f}_{R,i}^T(\vec{\delta} - \vec{\alpha}_{p_1,p_2}) \geq 0 \quad \forall \vec{\delta} \in \mathcal{D}_{(p_2,p_1)}$$

i.e.,

$$f_{R,i}^l := \max_{\vec{\delta} \in \mathcal{D}_{(p_1,p_2)}} -\vec{f}_{R,i}^T \vec{\delta} \leq \vec{f}_{R,i}^T \vec{\alpha}_{p_1,p_2} \leq \min_{\vec{\delta} \in \mathcal{D}_{(p_2,p_1)}} \vec{f}_{R,i}^T \vec{\delta} =: f_{R,i}^u. \quad (44)$$

For any $\vec{f}_{R,i}$, (44) will always have a solution for $\vec{\alpha}_{p_1,p_2}$. Otherwise, we can find $\vec{\delta}_1 \in \mathcal{D}_{(p_1,p_2)}$ and $\vec{\delta}_2 \in \mathcal{D}_{(p_2,p_1)}$ such that

$$-\vec{f}_{R,i}^T \vec{\delta}_1 \geq \vec{f}_{R,i}^T \vec{\delta}_2,$$

but then

$$\vec{f}_{R,i}^T \vec{\delta}_1 + \vec{f}_{R,i}^T \vec{\delta}_2 \leq 0,$$

which contradicts the fact that $\vec{\delta}_1 + \vec{\delta}_2$ is a member of $\mathcal{V}_{G,(p_1,p_1)}$, which is a subset of \overline{R}_G .

The cost function that measures the deviation from the ideal distance vector then consists of the following terms for each of the $f_{R,i}$ with $i < N - s$:

$$w_a |f_{R,i}^l - \vec{f}_{R,j}^T \vec{\alpha}_{p_1,p_2}|_+ + w_b |\vec{f}_{R,j}^T \vec{\alpha}_{p_1,p_2} - f_{R,i}^u|_+ \quad (45)$$

and/or

$$w_a |\vec{f}_{R,j}^T \vec{\alpha}_{p_1,p_2} - f_{R,i}^u|_+ + w_b |f_{R,i}^u - \vec{f}_{R,j}^T \vec{\alpha}_{p_1,p_2}|_+, \quad (46)$$

with $w_a \gg w_b$, since a negative deviation for the scalar product is worse than a positive deviation. Even if both \mathcal{D}_{p_1,p_2} and \mathcal{D}_{p_2,p_1} are non-empty, we may elect to only use either (45) or (46), possibly based on the number of elements in those two multi-sets, since the two terms will pull $\vec{\alpha}_{p_1,p_2}$ in opposite directions. For each of the equalities of \overline{R}_G , a term of the form (45) or (46) is added as well, but then with $w_a = w_b$. Finally, for $\vec{f}_{R,k}$, the scalar product need not approach zero, but should preferably be positive, so an additional term is required:

$$w_c \sum_{\vec{\delta} \in \mathcal{D}_{(p_1,p_2)}^T \cup \mathcal{D}_{(p_2,p_1)}^T} |-\vec{f}_{R,k}^T \vec{\delta}|_+. \quad (47)$$

Example 17 Consider again the choice for the relative offset between A and B in Section 5 after C and D have already been joined. Each of the direct self-dependence distance vectors is equal to $[0 \ 1]^T$, and therefore so is the average distance vector. From (43) in example 16, and because

$$[-1 \ 0] \vec{\delta} < [-1 \ 2] \vec{\delta},$$

we know $\vec{f}_{R,1}^T = [-1 \ 0]$ and $\vec{f}_{R,k}^T = \vec{f}_{R,2}^T = [-1 \ 2]$. The ideal distance vector direction is:

$$\begin{aligned} & \left\{ \vec{x} \in \mathbb{Z}^n \mid \begin{bmatrix} -1 & 2 \\ -1 & 0 \end{bmatrix} \vec{x} \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right\} \cap \{ \vec{x} \in \mathbb{Z}^n \mid [-1 \ 0] \vec{x} = 0 \} \\ & = \{ \vec{x} \in \mathbb{Z}^n \mid [-1 \ 0] \vec{x} = 0, [-1 \ 2] \vec{x} \geq 0 \}. \end{aligned}$$

Further,

$$f_{R,1}^l = \max_{\vec{\delta} \in \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\}} -[-1 \ 0] \vec{\delta} = 1$$

and $f_{R,1}^u$ is undefined. The cost function of the form (45) is

$$w_a |1 - [-1 \ 0] \vec{\alpha}_{p_1,p_2}|_+ + w_b |[-1 \ 0] \vec{\alpha}_{p_1,p_2} - 1|_+,$$

which evaluates to 0 for $\vec{\alpha}_{p_1,p_2} = [-1 \ 1]^T$ and to w_b for $\vec{\alpha}_{p_1,p_2} = [-2 \ 0]^T$, which are the two candidate relative offsets from example 15. The cost function

of the form (47) evaluates to zero for both of these choices. In other words, this cost (set of) cost function(s) correctly prefers $[-1 \ 1]^T$.

Finally, we note that in any of the cost functions described in section 7.2, \mathcal{D}_e can be replaced by \mathcal{V}_e to obtain alternative cost functions. As with \mathcal{D} , \mathcal{V} should be a multi-set and it should only contain the distance vectors themselves rather than their convex hull, i.e., conv should be dropped from (15). Naturally, an actual implementation need not necessarily maintain this set of distance vectors, but can use some approximation.

7.2.4 Conclusion

In section 7.2, we have discussed three (or six if you count the variants based on the indirect distance vectors) locality cost functions. The first only considers the dependence distances and may result in distance vectors pointing in all different directions (with the only restriction that a valid ordering should still exist), leaving little room for the ordering phase to do any optimization. The second tries to minimize the growth of the dependence cone in order to maximize the freedom for the ordering phase. However, it does not discriminate between different solutions that may lead to very different locality. The most promising cost function seems to be the third proposed cost function, which tries to estimate the ordering vector that will be produced by the subsequent ordering phase and optimizes the distance vectors with respect to this estimated ordering vector. In sum, it appears that not only for feasibility, but also for optimality it is advantageous to know the ordering vector prior to the translation phase.

8 Strategy

The algorithm for incremental translation (Algorithm 1) leaves two issues unspecified: which two nodes to select in step 3 and which relative offset to select in step 4. The second issue has been partly clarified in Section 7, but there are some additional considerations. We will not attempt to solve all of them here, but merely mention some of them.

When selecting a valid relative offset, we have so far only considered restrictions that ensure that a valid ordering still exists. Additional restrictions could be imposed, for example, ensuring that no two polytopes overlap in the common iteration space, as done by van Swaaij (1992). Allowing overlapping polytopes slightly distorts the locality cost functions, since a dependence may span more computations than indicated by the distance. On the other hand, the locality cost functions are in themselves only heuristics, so this distortion may not be that important. Additionally, overlapping polytopes may lead to generated code that is more complicated than for non-overlapping polytopes. In Section 9 we will consider this issue in more detail for the approach without an ordering step.

Not allowing overlapping polytopes leads to additional complexity during the translation phase, both for calculating the extra constraint and because the resulting search space will be more irregular and will in general lead to more subspaces if divided in convex spaces, which may be required to solve linear programs. Furthermore, the results from Section 4 would have to be extended to ensure that a valid solution always exists.

Another difference with the translation phase of van Swaaij (1992), is that he divides the search space, based on the non-overlappability, into convex regions that fit the entire polytope to be placed. In contrast, in this document, we have formulated the constraints in terms of the relative offsets of *reference points* of (sets of) polytopes.

```

for (i = 1; i <= n; ++i) {
    a[i] = b[i-1]; /* A */
    b[i] = a[i];   /* B */
}

```

Listing 4: Program with a loop-independent dependence



Figure 34: Dependence graph of the program in Listing 4

As to selecting two nodes to be combined, there are many possibilities. Arguably the simplest is to select a single node in the original graph and to successively add other nodes to this growing node, but multiple “nuclei” are equally possible. Although it is possible to combine two nodes that do not share a direct dependence, it is better to only combine nodes that do share a direct dependence, since more useful cost functions can be used in this case. Among those pairs of nodes that share a direct dependence, the pair with the dependence that is most “important” should be selected first, where importance can, for example, be measured by the number of array elements involved. Finally, we note that the nodes in the original graph need not represent single statements but can be clusters with internal relative offsets that have been determined more exhaustively.

9 Code Generation

In Section 6.2, we motivated the relaxation of constraint (26) to (32) and the accompanying addition of an extra dimension by referring to the fact that fewer indirect distance vectors need to be stored and updated. In the absence of an ordering step, an additional advantage of the technique in Section 6.2 is that the class of programs that can be handled is extended. (In the presence of an ordering step, the same can be said if the problem dimension is one and for other problem dimensions an ordering vector would have to be chosen that may lead to very bad code.) The set of programs that can be handled additionally is a subset of those that contain indirect self-dependences that are composed of one or more loop-independent dependences in the inner loop. An example member of this set is the program in Listing 4, with corresponding dependence graph shown in Figure 34. The single indirect distance vector equals 1, which satisfies (38), but does not satisfy the corresponding constraint without relaxation:

$$\vec{\delta} \not\geq l\vec{e}_N, \quad (48)$$

since the length of the cycle l is 2 in this case. The cause of this non-compliance is the existence of a loop-independent dependence between A and B in the inner loop. Note that a different approach to handle these cases is to only add an extra innermost dimension, since there will be no loop-independent dependences in this newly created dimension. In other words, the indirect distance vectors over a cycle that satisfy (38) will also satisfy (48) in the enlarged space, since $\vec{e}_N \geq l\vec{e}_{N+1}$ for any l .

A potential disadvantage of the technique in Section 6.2 is that it may introduce additional overlapping polytopes in the common iteration space. During

```

for (i = 1; i <= n; ++i)
    a[i] = f1(a[i-1]);
for (i = n+1; i <= 2*n; ++i)
    a[i] = f2(a[i-1]);

```

Listing 5: Non-overlapping iteration domains



Figure 35: Non-overlapping iteration domains

code generation, this in turn will lead to extra conditionals or extra iterations that are peeled off, depending on the code generation mode (Quilleré, Rajopadhye, and Wilde 2000). Consider, for example, the program in Listing 5 and the compatible mapping to the common iteration space shown in Figure 35. The first iteration of the second loop depends on the final iteration of the first loop. If, during translation, we adhere to the strict ordering constraint, we simply obtain the mapping shown in Figure 35. If, on the other hand, we allow distance vectors to become zero and apply a topological sort in an extra dimension afterwards, as proposed in Section 6.2, we obtain the mapping shown in Figure 36. The corresponding generated code is shown in Listing 6. Although we have “increased locality” by reducing the dependence distance from 1 to 0 (in the dimension under consideration), the resulting code is in no ways better than the original. Rather, it is worse since it is more irregular, making it more difficult for subsequent optimization steps to perform their analysis and optimization.

Although in principal a more sophisticated code generation algorithm could produce the code in Listing 5 from the mapping in the common iteration space in Figure 36, this will become more difficult if multiple polytopes are involved, especially if the dependence graph contains cycles. The problem of unwanted overlapping iteration domains can more easily be handled during the translation itself.

Before attempting to solve the problem, we need to identify the cases in which it occurs. The unwanted overlap is overlap that does not exist if we insist on a dependence distance of at least one in the inner dimension but does exist if we allow a dependence distance of zero. This means that the width of the overlap in the inner dimension will be exactly one. The overlap can be avoided decrementing the minimal distance vector between the two nodes under consideration by $\vec{s} = \vec{e}_N$ during a preprocessing step.

More specifically, assume that all \mathcal{P}_p s are singletons and let P_p be the single element, i.e., P_p is the iteration space of the statement represented by p . Let $\vec{\delta}_{p,q}$ be the minimal (indirect) distance vector over the edge (p, q) , i.e., $\vec{\delta}_{p,q}$ is the lexicographically minimal element of $\mathcal{V}_{(p,q)}$. We examine each pair of interdependent nodes (p, q) in the original dependence graph in turn. If the final translation is able to fully optimize locality over this edge, then the overlap

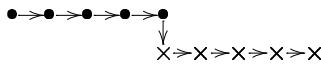


Figure 36: Minimally overlapping iteration domains

```

for (i = 1; i <= n-1; ++i)
    a[i] = f1(a[i-1]);
a[n] = f1(a[n-1]);
a[n+1] = f2(a[n]);
for (i = n+1; i <= 2*n-1; ++i)
    a[i+1] = f2(a[i]);

```

Listing 6: Minimally overlapping iteration domains

between the corresponding iteration domains will be given by

$$P_p + \vec{\delta}_{p,q} \cap P_q. \quad (49)$$

If this overlap has width one in the inner dimension, we ensure that the translated minimal distance vector is at least $\vec{s} = \vec{e}_N$ by decrementing $\vec{\delta}_{p,q}$ by \vec{s} (and adjusting all other minimal distance vectors accordingly) thus avoiding the overlap. In the special case where the iteration domains completely overlap, i.e., $P_p + \vec{\delta}_{p,q} = P_q$, we do not change the minimal distance vector since this kind of overlap does not carry any of the disadvantages mentioned above.

Note that we can only decrement the minimal distance vector (and thereby avoid the overlap) if the following condition holds:

$$\vec{d}_{p,q}^* + \vec{d}_{q,p}^* \succcurlyeq \vec{e}_N + \vec{s}. \quad (50)$$

Otherwise, (38) will be violated in the initial graph after the preprocessing step. This does not pose a problem, since, as explained before, (50) will only fail to hold if the original program contains loop-independent dependences in the inner loop. In these cases, the overlap will typically not be of width one in the inner dimension.

Example 18 Listings 8 and 9 show the code generated from two applications of the loop fusion technique, as discussed in Section 6.2 and without a prior linear transformation, on a Durbin algorithm, taken from Quilleré and Rajopadhye (2000) and reproduced in Listing 7. The first code was obtained without the preprocessing step of this section and the second code was obtained with the preprocessing step. The code without preprocessing is much more irregular than the code with preprocessing; it is longer and contains more and longer loops.

The initial dependence graph contains six pairs of statements that have their minimal distance vector changed by the preprocessing step because they could, and in fact do, lead to unwanted overlap. Loop fusion after preprocessing results in a common iteration space where only two iteration domains overlap, viz. the two that correspond to the final loop in both listings. This is a desired overlap, since the width in the inner dimension is larger than one. Moving the iteration domains apart would effectively decrease locality and cause more iterations to be peeled off.

Although the technique in this section originated as a solution to a problem in the innermost dimension, introduced by relaxing the time-delay constraint, it may also be used to solve a similar problem in outer dimensions. Consider, for example, the program in Listing 10 on page 50. Figure 37 on page 50 shows the relative positioning of the two iteration domains in the program using loop fusion and the technique to avoid unwanted overlap in the innermost dimension. Without this avoidance technique, the second iteration domain would be placed as indicated by the double line. Although we have removed the overlap in the innermost dimension, the placement is still not optimal from the point of view of code generation since there is still a minimal overlap in the outer dimension.

```

system Durbin : { N | N >= 2 }
                ( r : { i | 1<=i<=N } of real )
                returns ( y : { i | 1<=i<=N } of real );
var
  acc : { k,i | (i+1,2)<=k<=N; 0<=i} of real;
  Y   : { k,i | i<=k<=N; 1<=i} of real;
  B   : { k   | i<=k<=N} of real;
let
  y[i] = Y[N,i];
  acc[k,i] = case
    { | 2<=k<=N; i=0 } : 0[];
    { | i+1<=k<=N; 1<=i } :
      acc[k,i-1] + r[i] * Y[k-1,k-i];
  esac;
  Y[k,i] = case
    { | k=1, i=1 } : -r[1];
    { | (2,i+1)<=k } : Y[k-1,i] + Y[k,k] * Y[k-1,k-i];
    { | k=i; 2<=i<=N } : (-r[k] - acc[k,k-1]) / B[k];
  esac;
  B[k] = case
    { | k=1 } : 1[];
    { | 2<=k } :
      B[k-1] * (1[] - Y[k-1,k-1] * Y[k-1][k-1]);
  esac;
tel;

```

Listing 7: An ALPHA program for Durbin's algorithm

That is, there is a single value for the outer dimension that appears as the first coordinate for iterations of both statements. This overlap will result in extra conditionals or loop peelings in the outer dimension during code generation.

The detection of this situation is similar to the case for the innermost dimension: a shift in the k th dimension may be beneficial if the projection on the outermost k dimensions of the overlap (49) has width 1 in the k th dimension, provided the projections of the iteration domains themselves on the same space have a width larger than 1 in the k th dimension. This kind of overlap typically occurs with iteration domains that are connected through a non-uniform dependence. This does not mean that the loops involved cannot be fused at all, since the dependence may be (sufficiently) uniform in the outermost dimensions and then these dimensions can still be merged.

To enforce a shift in the k th dimension, we cannot simply use $\vec{s} = \vec{e}_k$ to change the minimal distance vector, since this choice will not only constrain the relative offset in k th dimension, but also, and arbitrarily so, in the inner dimensions. For example, we would not be able to select the position as indicated by the dashed frame in Figure 37 on page 50 with this choice. Formally, we can use

$$\vec{s} = \vec{e}_k - \infty \sum_{i=k+1}^N \vec{e}_i$$

and adapt the loop fusion algorithm to deal with this situation specifically. In fact, this is a situation where the relative offset of an outer dimension is constrained without a constraint on the inner dimension(s), as alluded to in the discussion of Figure 30 on page 38 in Section 7. We can then use the minimal shift in the inner dimensions.

```

Y[0][0] = -r[0][0];
B[0][0] = 1;
B[1][0] = B[0][0] * (1 - Y[0][0] * Y[0][0]);
acc[0][0] = 0;
acc[0][1] = acc[0][0] + r[0][0] * Y[0][0];
Y[1][1] = (-r[1][0] - acc[0][1]) / B[1][0];
B[2][0] = B[1][0] * (1 - Y[1][1] * Y[1][1]);
Y[1][0] = Y[0][0] + Y[1][1] * Y[0][0];
acc[1][0] = 0;
acc[1][1] = acc[1][0] + r[0][0] * Y[1][1];
acc[1][2] = acc[1][1] + r[1][0] * Y[1][1];
Y[2][2] = (-r[2][0] - acc[1][2]) / B[2][0];
B[3][0] = B[2][0] * (1 - Y[2][2] * Y[2][2]);
for (j = 1; j <= 2; j++) {
    Y[2][j-1] = Y[1][j-1] + Y[2][2] * Y[1][-j+2];
}
for (i = 4; i <= 8; i++) {
    acc[i-2][0] = 0;
    acc[i-2][1] = acc[i-2][0] + r[0][0] * Y[i-2][i-2];
    for (j = -6; j <= i-10; j++) {
        acc[i-2][j+8] = acc[i-2][j+7] + r[j+7][0] * Y[i-2][i-2];
    }
    acc[i-2][i-1] = acc[i-2][i-2] + r[i-2][0] * Y[i-2][i-2];
    Y[i-1][i-1] = (-r[i-1][0] - acc[i-2][i-1]) / B[i-1][0];
    B[i][0] = B[i-1][0] * (1 - Y[i-1][i-1] * Y[i-1][i-1]);
    for (j = 1; j <= i-1; j++) {
        Y[i-1][j-1] = Y[i-2][j-1] + Y[i-1][i-1] * Y[i-2][i-j-1];
    }
}
acc[7][0] = 0;
acc[7][1] = acc[7][0] + r[0][0] * Y[7][7];
for (j = -6; j <= -1; j++) {
    acc[7][j+8] = acc[7][j+7] + r[j+7][0] * Y[7][7];
}
acc[7][8] = acc[7][7] + r[7][0] * Y[7][7];
Y[8][8] = (-r[8][0] - acc[7][8]) / B[8][0];
B[9][0] = B[8][0] * (1 - Y[8][8] * Y[8][8]);
for (j = 1; j <= 8; j++) {
    Y[8][j-1] = Y[7][j-1] + Y[8][8] * Y[7][-j+8];
}
acc[8][0] = 0;
acc[8][1] = acc[8][0] + r[0][0] * Y[8][8];
for (j = -6; j <= 0; j++) {
    acc[8][j+8] = acc[8][j+7] + r[j+7][0] * Y[8][8];
}
acc[8][9] = acc[8][8] + r[8][0] * Y[8][8];
Y[9][9] = (-r[9][0] - acc[8][9]) / B[9][0];
Y[9][0] = Y[8][0] + Y[9][9] * Y[8][8];
y[0][0] = Y[9][0];
for (j = 2; j <= 9; j++) {
    Y[9][j-1] = Y[8][j-1] + Y[9][9] * Y[8][-j+9];
    y[0][j-1] = Y[9][j-1];
}
y[0][9] = Y[9][9];

```

Listing 8: Durbin after fusion without preprocessing

```

Y[0][0] = -r[0][0];
B[0][0] = 1;
B[1][0] = B[0][0] * (1 - Y[0][0] * Y[0][0]);
for (i = 2; i <= 9; i++) {
    acc[i-2][0] = 0;
    for (j = -8; j <= i-10; j++) {
        acc[i-2][j+9] = acc[i-2][j+8] + r[j+8][0] * Y[i-2][i-2];
    }
    Y[i-1][i-1] = (-r[i-1][0] - acc[i-2][i-1]) / B[i-1][0];
    B[i][0] = B[i-1][0] * (1 - Y[i-1][i-1] * Y[i-1][i-1]);
    for (j = 2; j <= i; j++) {
        Y[i-1][j-2] = Y[i-2][j-2] + Y[i-1][i-1] * Y[i-2][i-j];
    }
}
acc[8][0] = 0;
for (j = -8; j <= 0; j++) {
    acc[8][j+9] = acc[8][j+8] + r[j+8][0] * Y[8][8];
}
Y[9][9] = (-r[9][0] - acc[8][9]) / B[9][0];
for (j = 2; j <= 10; j++) {
    Y[9][j-2] = Y[8][j-2] + Y[9][9] * Y[8][-j+10];
    y[0][j-2] = Y[9][j-2];
}
y[0][9] = Y[9][9];

```

Listing 9: Durbin after fusion with preprocessing

```

for (i = 0; i <= n; ++i)
    for (j = 0; j <= n; ++j)
        a[i][j] = f(i, j);
for (i = 0; i <= n; ++i)
    b[i] = a[n-i][n];

```

Listing 10: Program resulting in overlapping domains in outer dimensions

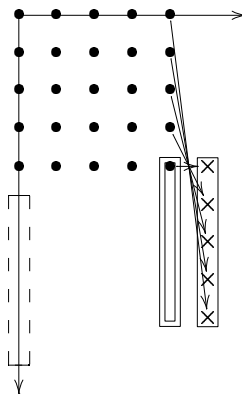


Figure 37: Overlapping domains in outer dimensions

Note that there may be a trade-off between the generated code and locality. The shifts for code generation do not change the relative ordering of the iterations of the two statements involved, but as soon as a third iteration domain overlaps (possibly only in outer dimensions) with either of the two inter-dependent iteration domains, then the shift will change the relative ordering of the iterations of either or both of the two statements with respect to the third. Therefore, it may also have an influence on locality, which is why we did not just use $\vec{s} = \vec{e}_k$. If we only shift in the inner dimension, then the influence on locality is minimal, since the increase or decrease in the number of iteration that are executed between two iterations involved in a dependence is limited by the number of iteration domains with which the two inter-dependent domains overlap. If we also shift in outer dimensions, then the effect on locality can be more profound. It may therefore be more prudent to consider these outer overlaps during loop fusion itself rather than during a preprocessing step, although it does mean that we may not be able to avoid as much overlap as we could with this preprocessing step.

10 Related Work

Loop transformations have been investigated by many researchers. The general case of affine transformations has also received much attention, but has mostly been oriented toward parallelization. A notable exception to this is the research by Lim et al. (2001), although it is based on their earlier parallelization research. They first identify all available parallelism using the techniques of Lim and Lam (1997), then they greedily combine components if they share reuse, after which they apply array contraction and finally generalized blocking to further improve locality.

The approach consisting of affine transformations and an additional ordering phase was used by both van Swaaij (1992) and Danckaert (2001). The former provides algorithms for all three phases, whereas the latter concentrates on the first, linear transformation phase. However, they do not provide any evidence that their algorithms will always produce valid solutions.

Similar to affine transformations, loop fusion has also been the subject of much research, although most of it is focused on “pure” loop fusion, where sets of loops are identified that can be fused directly, without bumping any of the loops. Darte (1999) provides an overview of this kind of loop fusion and some complexity results. A few researchers have also considered loop fusion combined with loop bumping, which is identical to our translation step after ordering, notably Manjikian and Abdelrahman (1995), Fraboulet et al. (1999) and Song et al. (2001).

Manjikian and Abdelrahman (1995) only shift to allow loop fusion and not to optimize locality, i.e., they only shift if the dependence distance would be negative otherwise. Furthermore, they only consider uniform dependences and acyclic dependence graphs. Bouchebaba (2002) optimizes for locality, but requires perfectly nested loops of the same depth, uniform dependences, rectangular iteration domains and a dependence graphs in the form of a chain. Determining the optimal relative offsets is trivial in this case and can also be obtained using our general loop fusion algorithm and a simple locality cost function.

Fraboulet et al. (1999) perform loop fusion with loop shifting to optimize a particular cost function which allows them to transform the problem into a minimal-cut, maximal-flow algorithm, which can be efficiently solved. The cost function minimizes the maximal dependence distance for all flow dependences emanating from the same array element, i.e., they minimize the life-times of array elements. They only consider uniform dependences.

Song et al. (2001) extend the work of Fraboulet et al. (1999). Although they do not explicitly impose a restriction to uniform dependences, their assumption on the maximal dependence distances only allows for some exceptional cases of non-uniform dependences. Although the authors claim that their technique applies to imperfectly nested loops, they in fact rely on a preprocessing step including loop peeling and loop partitioning to identify and massage the loop nests that are to be fused. By contrast, our approach takes any program as input, provided sufficient dependence analysis can be performed. Loop peeling is the *result* of (the code generation after) the fusion, rather than a step that must be performed to enable loop fusion.

Furthermore, although the authors claim their cost function minimizes the temporary array storage, this only applies to the individual arrays. The total storage space required at a given point in the program may actually increase due to the fact that more temporary arrays are simultaneously alive, as shown in Listing 3. The cost functions we propose in Section 7 suffer from the same short-sightedness, but they can more easily be extended to include such effects since the algorithm does not depend on the cost function.

All the loop fusion with loop bumping techniques discussed in this section, except for the work of Bouchebaba (2002), only apply to one dimension, which, in the case of (Song et al. 2001), can be the result of coalescing multiple dimensions, and a heuristic is used for multiple dimensions.

11 Conclusions and Future Work

In the context of a three-phase approach for loop transformations, consisting of a linear transformation, a translation and an ordering, we have derived a simple criterion for the linear transformation steps that ensures that the following steps can find a valid solution. Furthermore, we have proved that if this criterion is satisfied, translation can be performed incrementally. However, the proofs show that it is relatively complicated to ensure a valid translation in this context, which has prompted us to reevaluate the appropriateness of an extra ordering step. Without an ordering step and by adding an extra innermost dimension, we obtain a translation step that performs general loop fusion with loop shifting.

In Section 7, we have considered several cost functions for optimizing locality: one for an approach without an ordering step, which is a simple extension of a well known one-dimensional cost function to multiple dimensions, and three new cost function for an approach with an ordering step. Finally, we have investigated some of the implications of loop fusion on code generation and have presented a method for removing unwanted overlap of iteration domains.

Although we have discussed several cost functions, we have mainly focused on the feasibility of the translation step. Further research is required to evaluate the relative merits of different cost functions. This evaluation should also include other, possibly non locality-related, cost functions. When using multiple cost functions, it could be interesting to have the loop transformation step produce not a single solution but a set of solutions which show the trade-off between the different cost functions. If, on the other hand, the cost functions indicate that there is little variance over a range of solutions, we may want to produce partial scheduling information.

Since an approach without an ordering steps seems to be the most promising, additional research is also required on the linear transformation step to take this new context into account. The linear transformation step will also have to ensure the feasibility criterion for the subsequent translation step.

List of Symbols

$ \cdot _+$	The function that maps a value to itself if it is positive and to zero otherwise, page 39
\vec{a}_X	The offset of an affine transformation for statement X , see equation (2), page 3
A_X	The linear part of an affine transformation for statement X , see equation (2), page 3
\vec{a}_{p_1, p_2}	The relative offset of p_2 with respect to p_1 , see equation (14), page 13
$C_{G, T}$	The global dependence cone for dependence graph G and translation T , see equation (13), page 13
\mathcal{D}_e	The set of distance vectors over an edge e , page 9
\mathcal{D}_e^T	The set of distance vectors over an edge e translated by translation T , page 11
$\delta_{i, j}$	The Kronecker delta: $\delta_{i, j}$ equals 1 if $i = j$ and 0 otherwise
$\vec{i}_1 \delta \vec{i}_2$	Indicates a dependence. We say that \vec{i}_2 depends on \vec{i}_1 or that there is a dependence between \vec{i}_1 and \vec{i}_2 , page 4
$\vec{\delta}$	A dependence distance vector, see equation (11), page 9
\vec{e}_i	A unit vector with $e_{ij} = 0$ for $i \neq j$ and $e_{ii} = 1$
$I_{n, n}$	The $n \times n$ identity matrix
N	The dimension of the problem space, page 8
p	A node in the dependence graph, page 9
\mathcal{P}_p	The set of iteration domains corresponding to node p , page 9
π	A path
$\vec{\pi}$	An ordering vector, see equation (3), page 3
$\mathcal{P}_O(\mathcal{D})$	The valid ordering polyhedron for the set of dependence vectors \mathcal{D} , see equation (12), page 11
\overline{R}_G	The self-dependence full cone of G , see equation (17), page 16
R_G	The self-dependence cone of G , see equation (16), page 16
\mathcal{R}_G	The averaged self-dependence cone of G , see equation (25), page 23
T	A translation, page 10
θ_X	A schedule for statement X , see equation (1), page 3
$\mathcal{V}_{G, l}$	The indirect distance vector polytope defined over the paths between p_1 and p_2 in G , see equation (15), page 15

References

- Bastoul, C. (2002). Generating loops for scanning polyhedra. Technical Report 2002/23, Versailles University.
- Bik, A. J. C. (1996). *Compiler Support for Sparse Matrix Computations*. Ph. D. thesis, University of Leiden, The Netherlands.
- Bouchebaba, Y. (2002). *Optimisation des transferts de données pour le traitement du signal: pavage, fusion et réallocation des tableaux*. Ph. D. thesis, Ecole des Mines de Paris (France).
- Danckaert, K. (2001). *Loop transformations for data transfer and storage reduction on multiprocessor systems*. Ph. D. thesis, KU Leuven.
- Danckaert, K., F. Catthoor, and H. De Man (2000, November). A preprocessing step for global loop transformations for data transfer and storage optimization. In *Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, San Jose, CA.
- Darte, A. (1999). On the complexity of loop fusion. In *IEEE PACT*, pp. 149–157.
- Darte, A. and Y. Robert (1992, April). Affine-by-statement scheduling of uniform loop nests over parametric domains. Technical Report 92-16, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon.
- De Greef, E., F. Catthoor, and H. De Man (1997). Memory size reduction through storage order optimization for embedded parallel multimedia applications. *Parallel Computing* 23(12), 1811–1837.
- Fautrier, P. (1992, October). Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *International Journal of Parallel Programming* 21(5), 313–348.
- Fautrier, P. (1996). Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, pp. 79–103.
- Fraboulet, A., G. Huard, and A. Mignotte (1999, November). Loop Alignment for Memory Accesses Optimization. In *Twelfth International Symposium on System Synthesis Proceedings (ISSS'99)*, pp. 71–77. IEEE Computer Society Press.
- Kelly, W. and W. Pugh (1993, April). A framework for unifying reordering transformations. Technical Report CS-TR-3193, Dept. of CS, Univ. of Maryland, College Park.
- Knuth, D. E. (1968). *The Art of Computer Programming*, Volume 1: Fundamental Algorithms. Reading, Massachusetts: Addison-Wesley.
- Lamport, L. (1974, February). The parallel execution of DO loops. *Communications of the ACM* 17(2), 83–93.
- Lengauer, C. (1993). Loop parallelization in the polytope model. In *International Conference on Concurrency Theory*, pp. 398–416.
- Lim, A. W. and M. S. Lam (1997). Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the Twenty-fourth Annual ACM Symposium on the Principles of Programming Languages*, Paris, France.
- Lim, A. W., S.-W. Liao, and M. S. Lam (2001). Blocking and array contraction across arbitrarily nested loops using affine partitioning. *ACM SIGPLAN Notices* 36(7), 103–112.

- Manjikian, N. and T. Abdelrahman (1995, February). Fusion of loops for parallelism and locality. Technical Report CSRI-315, Computer Systems Research Institute, University of Toronto, Canada.
- Quilléré, F. and S. Rajopadhye (2000, September). Optimizing memory usage in the polyhedral model. In *ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 22, Issue 5*, pp. 773–815.
- Quilléré, F., S. Rajopadhye, and D. Wilde (2000, October). Generation of efficient nested loops from polyhedra. In *International Journal of Parallel Programming, vol 28, no 5*.
- Schrijver, A. (1986). *Theory of linear and integer programming*. Jonh Wiley & Sons.
- Song, Y., R. Xu, C. Wang, and Z. Li (2001). Data locality enhancement by memory reduction. In *International Conference on Supercomputing*, pp. 50–64.
- Tronçon, R., M. Bruynooghe, G. Janssens, and F. Catthoor (2002). Storage size reduction by in-place mapping of arrays. In A. Cortesi (Ed.), *Verification, Model Checking and Abstract Interpretation, Third Int. Workshop, VMCAI 2002, Revised Papers*, Volume 2294 of *LNCS*, pp. 167–181. Springer-Verlag.
- van Swaaij, M. (1992). *Data flow geometry: exploiting regularity in system-level synthesis*. Ph. D. thesis, KU Leuven.
- van Swaaij, M., F. Franssen, F. Catthoor, and H. D. Man (1992, March). Modelling data and control flow for high-level memory management. In *Proc. 3rd ACM/IEEE Eur. Design Automation Conf.*, Brussels, Belgium, pp. 8–13.
- Verdoolaege, S., F. Catthoor, M. Bruynooghe, and G. Janssens (2001, November). A heuristic for improving the regularity of accesses by global loop transformations in the polyhedral model. Report CW 325, Department of Computer Science, K.U.Leuven, Leuven, Belgium.
- Wilde, D. K. (1993). A library for doing polyhedral operations. Technical Report 785, IRISA, Rennes, France.
<http://www.irisa.fr/EXTERNE/bibli/pi/pi785.html>.
- Yang, Y.-Q., C. Ancourt, and F. Irigoin (1994). Minimal data dependence abstractions for loop transformations. In *Languages and Compilers for Parallel Computing*, pp. 201–216.

Index

- Abdelrahman, 31, 51
- Bastoul, 33
- Bik, 32, 33
- Bouchebaba, 51, 52
- cone, **8**
- averaged self-dependence, **22**, **23**, 25, 30
 - dependence, **11**
 - global dependence, **13**, 13, 16, 19
 - self-dependence, **16**
 - self-dependence full, **16**
- convex hull, **8**
- cost functions, 37–44
- Danckaert, 3, 30, 33, 36, 51
- Darte, 3, 51
- De Greef, 4
- degenerate, **13**, 13, 17, 19–21, 24, 25
- dependence
- anti, 4
 - between two iterations, 3
 - dynamic, 3
 - flow, 4
 - output, 4
 - pseudo, **14**, 14
 - self, **14**
 - static, 3
 - true, 4
- dependence cone, *see* cone, dependence
- dependence graph, **9**
- deviation
- from dependence cone, 41
 - from ideal distance vector, 42
- Directed Acyclic Graph (DAG), 31
- distance vector, **9**
- average, 42
 - ideal, 42
 - indirect, **15**
 - minimal, 35
 - self-dependence, **14**
 - translated, **10**, 10, 25, 39, 40
- Durbin, 47
- edge
- pseudo, **14**, 17–20
- explicit notation, **8**
- Feautrier, 3, 33, 36
- Fraboulet, 31, 39, 51, 52
- greatest common divisor (gcd), 23, 24, 30–33
- implicit notation, **8**
- iteration domain, **8**
- iteration vector, 3
- Kelly, 36
- Knuth, 23
- Lam, 3, 51
- Lamport, 33
- Lengauer, 33
- Lim, 3, 51
- line, **8**, 24
- linear transformation, 3, 4, 9, 24, 28, 31, 34
- locality, 37–44
- loop bumping, 31
- loop coalescing, 33
- loop fusion, 29–31
- Manjikian, 31, 51
- multi-set, 37, 44
- optimality, 37–44
- ordering, 28–37
- multi-dimensional, 33–34
 - valid, 11
- overlap
- unwanted, 45
- path
- fundamental, **14**, 15
- polyhedron, **8**
- valid ordering, **11**
- polytope, **8**
- indirect distance vector, **15**
- Pugh, 36
- Quilleré, 4, 33, 46, 47
- Rajopadhye, 4, 46, 47
- ray, **8**, 8
- regularity, 31
- Robert, 3
- schedule, **3**, 34
- scheduling, 33
- affine-by-statement, 3
 - multi-dimensional, 34

Schrijver, 12
Song, 31, 51, 52
sort
 topological, 31, 35
supporting point, 8

translation, 10
 incremental, 13
 valid, 9, 13, 17, 20, 21, 24
Tronçon, 4

valid ordering polyhedron, *see* poly-
 hedron, valid ordering
valid translation, *see* translation, valid
van Swaaij, 3, 33, 40, 44, 51
Verdoolaege, 3, 9, 31

Wilde, 8, 46

Yang, 11