Multi-dimensional Incremental Loop Fusion for Data Locality

Sven Verdoollaeghe, Maurice Bruynooghe, Gerda Janssens
Katholieke Universiteit Leuven Dept. Comp. Sc.
{sven,maurice,gerda}@cs.kuleuven.ac.be

Francky Catthoor
IMEC
Also prof. at K.U.Leuven
catthoor@imec.be

Abstract

Affine loop transformations have often been used for program optimization. Usually their focus lies on single loop nests. A few recent approaches also handle global programs with multiple loop nests but they are not really scalable towards realistic applications with dozens of nests. To reduce complexity, we split affine transformations into a linear transformation step and a translation step. This translation step can be used to perform general multi-dimensional loop fusion. We show that loop fusion can be performed incrementally and provide a greedy algorithm, which we illustrate on a simple example. Finally, we present a heuristic for data locality and provide some experimental results.

1: Introduction

As the memory subsystem typically accounts for over 50% of the power consumption [14, 15], optimizing the global memory accesses of an application is crucial for achieving low power realizations. An important factor in this optimization is the improvement of data locality through loop transformations. This improvement can both reduce the memory requirements by shortening the life-times of data elements and decrease the number of accesses to larger (and slower) memories by concentrating accesses to the same data element.

Many researchers have recognized the usefulness of data locality. However, in general, the algorithms they propose only perform a limited set of transformations, are of high complexity or make special assumptions on the initial code. Early research [26] focused on optimizing locality in single perfectly nested loops by applying a single linear transformation, ignoring the large buffers that exist between loop nests.

Loop fusion is a loop transformation that optimizes over multiple loop nests. In its purest form, it only decides which loop nests should be fused together without performing any other transformation. Finding the optimal fusion has been proven to be NP-complete, except for some special cases [4]. In general, these approaches require that all loops considered for fusing are compatible, i.e., that they have the same number of iterations.

Extending loop fusion with loop shifting allows more loops to be merged. The problem of maximizing the number of parallel loops that can be fused with loop shifting is NP-complete, even for acyclic dependence graphs with uniform dependences [2], although fusion of sequential loops is sometimes simpler (e.g., [4, 17]). Loop shifting for array contraction is also NP-complete for acyclic dependence graphs with uniform dependences [6]. Efficient
algorithms can be constructed by only considering acyclic dependence graphs with uniform dependences [20] or by using a cost function that allows linearization of the optimization problem for a single dimension assuming (near) uniform dependences and compatible loops [12, 23].

A very general way of performing loop transformation is to apply an affine transformation on each statement in a loop [7, 11, 18]. Although originally formulated for a single loop nest, it equally applies to multiple loop nests, but research has mainly focused on parallelization. Furthermore, exact dependences are typically used, which is relatively costly and requires exact dependence analysis to be available.

To reduce complexity, we split the search for affine transformations into two phases, one determining the linear parts and one determining the offsets. This second phase, which we call the translation phase, is a general form of loop fusion with loop shifting, as previously recognized by others (e.g., [6, 5, 8]), and is the focus of this paper. It solves the problem of the limited applicability of previous loop fusion approaches since it applies to general, possibly cyclic, dependence graphs of loop nests whose index sets can be represented or approximated by polytopes, i.e., it is not limited to perfectly nested loops and does not require the loop nests to be compatible. The translation phase can be used by itself or in conjunction with a preceding linear transformation phase [21].

Rather than using exact dependences, we use the distance vector abstraction, which contains sufficient information for the transformation. The use of this abstraction makes the transformation more efficient and allows the processing of programs that contain non-affine dependences, provided they can be approximated by distance vectors [13]. By using indirect distance vectors, we show that we are able to perform loop fusion incrementally using a simple greedy algorithm, which we prove to arrive at a feasible final solution. Finally, we propose a simple cost function for data locality. Note that the algorithm is not intended to find the optimal solution for either memory size or locality, but rather a good solution in a reasonable amount of time.

In Sect. 2 we will first discuss our program model. The main incremental loop fusion algorithm is presented in Sect. 3 along with a proof of its applicability and a generalization. We then propose a locality heuristic in Sect. 4, followed by the application of the loop fusion algorithm on a small example in Sect. 5. We continue with some experimental results in Sect. 6 and we conclude with a comparison to related work in Sect. 7.

2: Program model

The programs we consider consist of a number of statements, each enclosed in a (possibly zero-dimensional) loop nest and possibly guarded by conditionals (if-statements), with all the loop bounds and conditions affine expressions of outer loop iterators and structural parameters. The iteration domains, i.e., the sets of iterators for which each statement is executed, can then be represented by (unions of) polytopes (convex sets bounded by hyperplanes). In this representation, each dimension corresponds to a loop iterator. More general programs can still be modeled in this way by considering the smallest set of innermost loops containing non-affine or data-dependent conditions, possibly moved inwards from outer loops, as a single statement. For simplicity we embed all iteration domains in spaces of the same dimension, with a fixed coordinate value for the extra “dummy” dimensions. This is equivalent to (virtually) adding extra loops of trip count one.
Locality should be optimized for both flow and input dependences. As explained in the introduction, locality over flow dependences influences both the memory size and the number of accesses. Locality over input dependences, however, only influences the number of accesses. Although our model also handles output and anti-dependences (they merely impose an ordering constraint), better results can typically be obtained by removing them prior to loop fusion through a transformation to dynamic single assignment code.

Each dependence can be represented by what is known as a (dependence) distance vector \( \vec{d} \), which measures the dependence distance in each dimension and is calculated as the difference between the iterations involved in the dependence:

\[
\vec{d} = \vec{i}_2 - \vec{i}_1 \quad \text{if } \vec{i}_2 \text{ depends on } \vec{i}_1.
\] (1)

For example, in Fig. 1 (Sect. 5), the right hand side shows all the individual dependences of the program on the left. The distance “vector” for the dependence between iteration 1 of statement \( A \) and iteration 0 of statement \( B \) is \( 0 - 1 = -1 \). If all distance vectors generated by a reference are equal, the dependence is said to be uniformly generated or simply uniform. In the example, the dependence between \( A \) and \( B \) is uniform, whereas the one between \( B \) and \( C \) is not.

The convex hull of the set of all distance vectors representing dependences between different iterations of a pair of statements forms a polytope, known as the dependence polytope [27]. If the index expressions in both statements are affine and non data-dependent, then the dependence polytope can be easily obtained. Otherwise approximations may (have to) be used [13]. Originally, distance vectors were only defined between iterations in the same iteration space. We (and others) slightly relax this constraint by also allowing distance vectors between iterations of different iteration spaces, which always have the same dimension in our model. These generalized distance vectors will only be used in intermediate representations as the final transformed iteration domains will all be mapped onto the same common iteration space.

The dependence graph \( G \) (e.g., Fig. 3) collects all the required information. The nodes \( V_G \) of the dependence graph represent the statements and each link \( e \in E_G \) between two nodes represents the existence of a dependence between at least one pair of iterations of the two statements. Each link is further annotated by the corresponding dependence polytope \( P_e \), whereas the nodes are annotated by the iteration domain of the corresponding statement.

An affine transformation maps each iteration vector \( \vec{i} \) to a transformed iteration vector \( \vec{i}' \) in a common iteration space:

\[
\vec{i}' = A(\vec{i}) = A\vec{i} + \vec{a},
\] (2)

where \( A \) is a square integer non-singular matrix representing the linear part of the affine transformation and \( \vec{a} \) is an integer vector representing the translation part of the transformation. The iterations in the common iteration space are executed in lexicographical order, i.e., \( \vec{i} \) is executed before \( \vec{j} \) (\( \vec{i} \prec \vec{j} \)) iff \( \forall k < n : i_k = j_k \) and \( i_n < j_n \) for some \( n < N \). The code that effectuates this execution order is generated by scanning the transformed iteration domains in the common iteration space [22]. If the projections of two iteration domains on the first \( k \) dimensions overlap, then the effect will be that the outer \( k \) loops of the loop nests enclosing the corresponding statements are (partially) fused. This overlap and the relative shifts are governed by the translation offsets, whereas the linear transformation determines the shape and orientation of the iteration domains.
A pair of affine transformations \( A_p, A_q \) for a pair of statements \( p, q \) is legal iff all iterations are executed after the iterations on which they depend, in other words, iff all transformed distance vectors (except those for input dependence) are lexicographically strictly positive:

\[
d^\ell = A_q(\vec{r}_2) - A_p(\vec{r}_1) \succ 0 \quad \text{if } \vec{r}_2 \text{ depends on } \vec{r}_1.
\]

(3)

A set of affine transformations is legal iff (3) holds for all pairs of inter-dependent statements.

We propose to perform a general loop fusion by applying a transformation consisting solely of a translation vector to each of the iteration domains. This general loop fusion can be used by itself or it can be preceded by a linear transformation step. This linear transformation step should then optimize the locality over (indirect) self-dependences [16] and the regularity over all dependences [3]. In the next section we derive the conditions that must be satisfied by the result of the linear transformation step.

### 3: Incremental loop fusion

We will first show the implications of naively applying the translation step directly. With the linear part equal to the identity matrix, the affine transformation (2) simplifies to \( \vec{r}^\ell = T(\vec{r}) = \vec{r} + \vec{a} \) and, using (1), the ordering constraint (3) becomes

\[
d^\ell = T_q(\vec{r}_2) - T_p(\vec{r}_1) = \vec{d} + \vec{a}_q - \vec{a}_p \succ 0 \quad \text{if } \vec{r}_2 \text{ depends on } \vec{r}_1.
\]

(4)

Since the translation and iteration vectors are integer, the smallest strictly positive sum is \( \vec{e}_N \), the unit vector with 1 in the \( N \)th position. Furthermore, since \( \vec{a}_q - \vec{a}_p \) is constant for the pair of statements \( (p, q) \), it is sufficient to check (4) for the lexicographically minimal distance vector \( \vec{d}_{p,q} \) between \( p \) and \( q \):

\[
d^\ell = \vec{d}_{p,q} + \vec{a}_q - \vec{a}_p \succ \vec{e}_N \quad \text{if } (p, q) \in E_G.
\]

(5)

In the special case of a self-dependence, i.e., \( p = q \), (5) further simplifies to

\[
d^\ell = \vec{d}_{p,p} \succ \vec{e}_N,
\]

(6)

which must be ensured by the linear transformation step since it is independent of the translation step. If the translation is performed without a prior linear transformation step, then (6) must hold in the original program, which will automatically be the case if it has been written in an imperative language.

Not only direct self-dependences, but also indirect self-dependences impose constraints on the linear transformation phase. Consider a cycle \( \pi \) of length \( l \) in the dependence graph \( G \): \( \pi = (p_0, p_1, \ldots, p_{l-1}, p_l), p_0 = p_l, \forall i < l : (p_i, p_{i+1}) \in E_G \). Each of the links in the cycle \( \pi \) needs to satisfy a constraint of the form (5), i.e., \( \sum_{i=0}^{l-1}(\vec{d}_{i,i+1} + \vec{a}_{i+1} - \vec{a}_i) \succ \sum_{i=0}^{l-1} \vec{e}_N \).

Since \( \vec{a}_l = \vec{a}_0 \), we obtain \( \sum_{i=0}^{l-1} \vec{d}_{i,i+1} \succ l \vec{e}_N \). Let \( E^l_G \) be the set of indirect links of length \( l \) in the graph, i.e., \( E^l_G = E_G \) and

\[
E^l_G = \left\{ (x, y) \mid \exists z : (x, z) \in E^{l-1}_G, (z, y) \in E_G \right\} \quad \text{for } l > 1
\]

and let \( \vec{d}_{p_1,p_2} \) be the lexicographically minimal distance vector over the indirect link \( (p_1, p_2) \in E^l_G \), then the linear transformation needs to ensure:

\[
\forall l, \forall (p, p) \in E^l_G : \vec{d}_{(p,p)} \succ l \vec{e}_N,
\]

(7)
i.e., each indirect self-dependence distance vector needs to be larger than the length of the cycle over which it is defined. This yields the following proposition:

**Proposition 1** If a legal fusion exists, then (7) must hold.

As we will show next, this is also a sufficient condition for a legal fusion to exist and is therefore the only constraint on the linear transformation phase.

The basic technique for incremental loop fusion works as shown on the right: in each step two nodes \( p_1 \) and \( p_2 \) are selected and fused together at a relative offset \( \vec{a}_{p_1,p_2} \) until a graph is obtained with only a single node. The combine operation in step 5 replaces the two selected nodes \( p_1 \) and \( p_2 \) by a single fused node \( p' \), which combines the iteration domain of \( p_1 \) at offset \( \vec{0} \) and the iteration domain of \( p_2 \) at \( \vec{a}_{p_1,p_2} = \vec{a}_{p_2} - \vec{a}_{p_1} \), and adjusts the minimal distance vectors. For example, for any \( p_i \), \( d_{p_i,p_j}^l \) is the minimum of \( d_{p_1,p_i}^l \) and \( d_{p_2,p_i}^l - \vec{a}_{p_1,p_2} \), where the distance vectors are shifted by \( -\vec{0} \) and \( -\vec{a}_{p_1,p_2} \) respectively due to (1). Similarly, for any \( d_{p_i,p'}^l \), the shifts are \( \vec{0} \) and \( \vec{a}_{p_1,p_2} \). Specifically, for \( d_{p_i,p'}^l \), we have

\[
d_{p_i,p'}^l = \min \left\{ d_{p_1,p_i}^l, d_{p_2,p_i}^l, d_{p_1,p_2}^l + \vec{a}_{p_1,p_2}^l, d_{p_2,p_1}^l - \vec{a}_{p_1,p_2}^l \right\}. \tag{8}
\]

To obtain the translation offsets for the nodes in the original graph, the single node of the final graph is assigned an arbitrary offset. Then, for each combination of \( p_1 \) and \( p_2 \) into \( p' \), \( p_1 \) is assigned the same offset as \( p' \) and \( p_2 \) is assigned the offset of \( p' \) incremented with \( \vec{a}_{p_1,p_2} \). This process of decombination is continued until the original graph is reached.

The choice of a relative offset in step 4 of Algorithm 1 is bounded by the following proposition.

**Proposition 2** If (7) holds in the original graph, then the incremental loop fusion algorithm yields a legal loop fusion provided that in each iteration, a relative offset is chosen that satisfies

\[-\left( \min_l d_{p_1,p_2}^l - l\vec{e}_N \right) \preceq \vec{a}_{p_1,p_2} \preceq \min_l d_{p_2,p_1}^l - l\vec{e}_N \tag{9}\]

**Proof** We prove the proposition by induction. In the base case, the final graph contains a single node, which means that all dependences are self-dependences and then (4) follows trivially from (7).

For the induction case, first note that we only need to consider simple cycles, i.e., those that visit a node at most once, since if (7) holds for all simple cycles, then it also holds for the other cycles. Any new simple cycle passes through \( p' \) and the corresponding minimal distance vector is therefore of the form either \( d_{p_1,p_2}^l + \vec{a}_{p_1,p_2} \) or \( d_{p_2,p_1}^l - \vec{a}_{p_1,p_2} \) (8). By substituting these two new distance vectors in (7) we obtain (9). It only remains to show that (9) always has a solution.

Suppose that (9) does not have a solution. This requires there exist \( l_1 \) and \( l_2 \) such that

\[-(d_{p_1,p_2}^l - l_1\vec{e}_N) > d_{p_2,p_1}^l - l_2\vec{e}_N.\]

But then, since \( d_{p_1,p_1}^l \preceq d_{p_2,p_1}^l + d_{p_1,p_2}^l \), we have

---

**Algorithm 1** Incremental loop fusion

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 combine\((G, p_1, p_2, \vec{a}_{p_1,p_2})\).
\[ \vec{d}_{p_1,p_2} + l_2 \vec{e}_{N} \preceq \vec{d}_{p_2,p_1} - l_1 \vec{e}_{N} \preceq \vec{d}_{p_1,p_2} - l_1 \vec{e}_{N} \prec \vec{0}, \] which contradicts the assumption (7), since \( \vec{d}_{p_1,p_2} \) is a minimal distance vector in the graph prior to combination. This completes the proof. □

Although (6) holds for any (imperative) program, (7) may be violated if a statement indirectly depends on itself through one or more loop-independent dependences (with distance vector \( \vec{0} \)) and the remaining distance vectors in the cycle do not add up to \( l \vec{e}_{N} \). The problem can easily be solved by adding an extra dimension to the common iteration space, representing the offset of the statements inside the inner loop. The \((N + 1)\)-dimensional offsets can then be obtained by applying Algorithm 1 to obtain the offsets in the first \( N \) dimensions followed by a topological sort to determine the offsets in the \( N + 1 \)st dimension.

More specifically, we relax (7) to
\[ \forall (p, p) \in E^*_G : \vec{d}^*_p \preceq \vec{e}_N \] for the initial graph and to \( \forall (p, p) \in E^*_G : \vec{d}^*_p \preceq \vec{0} \) for all the intermediate graphs, including the final graph, where \( E^*_G \) contains indirect links of any length and \( \vec{d}^*_p \) is the minimal distance vector over the indirect link \((p_1, p_2)\). Consequently, (9) is relaxed to
\[ -\vec{d}^*_p \preceq \vec{\alpha}_{p_1,p_2} \preceq \vec{0} \] (11)

The resulting set of \( N \)-dimensional translation offsets does not necessarily satisfy (4) as some of the translated distance vectors may be zero. Let \( G_Z \) be the graph resulting from the initial graph \( G \) by removing all the links with a non-zero minimal distance vector. Because of (10), \( G_Z \) is acyclic and its nodes can be topologically sorted. The sequence numbers of this sort can be used as the \( N + 1 \)st dimension of the translation offsets such that (4) does hold for the \((N + 1)\)-dimensional translation offsets.

Note that by using (10) instead of (7), we not only extend the application domain to all fusible programs, but the calculations are also greatly simplified since the circuit lengths need no longer be taken into account.

4: Locality heuristic

Algorithm 1 does not describe a single, but rather a set of algorithms, since it specifies neither which two nodes to combine in step 3 nor which relative offset to choose from the valid range (11) in step 4. As a heuristic for the first choice, we first consider the links with a minimal distance vector determined by a flow dependence, rather than an output or anti-dependence, since at least one dependence can be fully optimized over such a link. From those links (or from the others after exhaustion) we choose the one with the highest number of data elements involved in the corresponding dependence.

As to the second choice, observe that the lower bound of (11) corresponds to the optimal value for at least one (chain of) dependence(s), since it will reduce the distance vector to \( \vec{0} \) for that dependence. As a simple but effective heuristic, we therefore choose this lower bound as the offset. In case input dependences are also to be optimized, an appropriate cost function has to be evaluated for all the optimal offsets of input dependences that fall within the legal range.
for (i = 0; i <= n; i++)
    a[i]=f(i); // A
for (i = 1; i <= n; i++)
    b[i]=g(b[i-1], a[i-1]); // B
for (i = 0; i <= n; i++)
    c[i]=h(b[n], a[i]); // C

Figure 1. Example program with dependences.

for (i = 0; i <= n-1; i++) {
    a[i]=f(i);
    b[i+1]=g(b[i], a[i]);
}
c[0]=h(b[n],a[0]); a[n] = f(n);
for (i = n; i <= 2*n-1; i++)
    c[i-n+1]=h(b[n], a[i-n+1]);

Figure 2. Complete fusion

5: Example

For ease of exposition, we will demonstrate the algorithm on a simple one-dimensional example, shown in Fig. 1 on the left with the individual dependences on the right. The dependence graph of this example program and the associated dependence polytopes are shown in Fig. 3. Since the graph was generated directly from an imperative program, (10) automatically holds as can be verified from the single (direct) self-dependence from B to itself.

Application of the algorithm proceeds as follows. The dependence involving the largest amount of array elements (n) is the one between A and C, compared to n − 1 for (A, B) and 1 for (B, C). Since the corresponding link is not part of a cycle (11) only yields a lower bound (1−n) ≪ ̄α_{A,C}, with ̄d^*_{A,C} = 1+(−n). Following our heuristic, we choose ̄α_{A,C} = n−1. The minimal distance vectors after combining A and C are ̄d_{AC,AC} = ̄d_{AC} + ̄α_{A,C} = 0+(n−1), ̄d_{AC,B} = ̄d_{A,B} = 1 and ̄d_{B,AC} = ̄d_{B,C} − ̄α_{A,C} = −n+(n−1) resulting in a constraint on the relative offset between AC and B with identical lower and upper bounds: −1 ≪ ̄α_{AC,B} ≪ −1 and we have no choice but to use ̄α_{AC,B} = −1. We arbitrarily choose ̄α_A = ̄α_{ABC} = 0 and obtain ̄α_B = 1 and ̄α_C = n−1.

Two links in the original graph, (A, B) and (B, C), now have zero-valued distance vectors and so G_Z consists of those two links. A topologically sort of the nodes in this graph assigns 0 to A, 1 to B and 2 to C. The resulting fused program is shown in Fig. 2, 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 statements inside the inner loop.
Table 1. Effect of translation on memory compaction

<table>
<thead>
<tr>
<th>Problem</th>
<th>LL18</th>
<th>CD</th>
<th>USVD</th>
<th>APP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declared size</td>
<td>1440000</td>
<td>3984964</td>
<td>3050498</td>
<td>1020000</td>
</tr>
<tr>
<td>Size after MC</td>
<td>1437191</td>
<td>1227845</td>
<td>3030203</td>
<td>1000200*</td>
</tr>
<tr>
<td>Size after fusion and MC</td>
<td>322407</td>
<td>616013</td>
<td>2060205</td>
<td>1009899</td>
</tr>
</tbody>
</table>

6: Experimental results

Preliminary experiments show that even the simple heuristic of Sect. 4 improves locality and memory requirements. Table 1 shows the result of memory compaction (MC) [10] both before and after fusion (without linear transformation) on a number of applications. LL18 is loop 18 of the Livermore benchmark set, APP is the algebraic path problem, Cavity Detection (CD) is an algorithm used in medical applications and USVD is an updating singular value decomposition algorithm frequently used in wireless signal processing applications. The most dramatic improvements are obtained for programs that only contain uniform dependences (the first two in the table). Both APP and USVD contain non-uniform dependences and relatively complex iteration domains. Although for APP almost no gains are obtained, the USVD does show a significant reduction in memory requirements. Note that the size after MC (without fusion) marked by * in the figure for APP is based on a known optimal scheduling of a program written in an applicative language in a form which is not procedurally executable. Since the program does satisfy (10), our fusion algorithm finds a valid scheduling automatically, which is the main benefit of our tool here since this is difficult to achieve manually.

To evaluate the effect of our transformations on data locality, we calculate the (backward) reuse distance (BRD) [1] of each memory access, which equals the number of different memory elements that are accessed between the given access to a certain memory element and the previous access to the same memory element. Figure 4 shows the result of this calculation for three of the applications. For each application, the relative amount of accesses with the $2^\log$ of their BRD up to a certain value are shown for all of the original version, the fused version, the original version with MC applied and the fused version with MC applied. For example, for the original formulation of the USVD, almost half of the accesses that reuse an element have a BRD up to $2^{15}$. The other half have a BRD between $2^{15}$ and $2^{16}$. The accesses with infinite BRD (those that access a memory element for the first time) are not shown in the figure and the bars are scaled according to the maximal amount of accesses with finite BRD for each application.

The figure shows that MC always has a positive effect on (temporal) locality for each of the three applications, although for the non-fused Livermore 18 the improvement is invisibly small. For Livermore 18, fusion is always a win: to ensure that all accesses with finite BRD are cache hits (assuming a fully associative cache) a cache of size $2^{21}$ is needed for the original program (with MC) and one of only $2^{13}$ is needed for the fused program (with MC) and the number of accesses with finite BRD is also increased: a cache of only $2^{5}$ can handle more accesses in the fused program than one of $2^{20}$ in the original program. For USVD, fusion is a win for all cache sizes except $2^{15}$. For cavity detection, the results are more ambiguous. For example, for a $2^{4}$ cache the non-fused program is clearly preferable, i.e., the result of our heuristic is not optimal for all cache sizes.
Figure 4. Backward reuse distances

7: Related work

Many researchers have investigated automatic loop transformations. The initial individual loop transformations were generalized to linear (mostly unimodular) transformations for perfectly nested loop nests and later to affine transformations for each statement in one or multiple nests, mainly focusing on parallelism [7, 11, 18], although recently locality has also received some attention [19]. Most of these approaches use exact dependence information, which leads to relatively expensive algorithms and limits the scope of applicability.

Rather than splitting affine transformations into a linear and a translation step, most researchers build up the affine transformation matrices row by row. One such example is the research by Darte et al. [9]. Although the heuristic they propose is oriented towards parallelism, their approach is similar to ours since they also use the distance vector abstraction and since their algorithm allows more freedom in the choice of cost function.

The approach proposed by van Swaaij et al. [24] consists of an affine transformation of each statement to a common iteration space, followed by an additional linear transformation of the common iteration space, called the ordering phase. Danckaert [3] further proposes to split the first step into a linear and a translation substep, but focuses on the linear phase and provides no evidence that a translation substep can still be performed after his linear substep. In [25] we show that incremental loop fusion is indeed possible in this setting with an extra ordering phase, but considerably more complex than the approach presented here.

Early research mostly focused on loop fusion without loop shifting, e.g., [17]. Manjikian and Abdelrahman [20] 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.

The most closely related research is that of Fraboulet et al. [12] and its extensions by Song et al. [23]. The former only consider uniform dependences and although the latter do not explicitly impose this restriction, their assumption on the maximal dependence distances only allows for some exceptional cases of non-uniform dependences. Their algorithms only apply to one dimension, which, in the case of Song et al., can be the result of coalescing multiple dimensions, and a heuristic is used for multiple dimensions. Darte and Huard [6] do handle non-uniform dependences and provide an ILP formulation and complexity results for optimizing array contraction for simple loops.

The key features of our proposed incremental approach for loop fusion are the applicability to a general class of programs, the low complexity which makes it scalable, and the independence of cost function of the algorithm itself. The heuristic to steer the optimiza-
tion obviously does depend on that (set of) cost function(s). Although the authors of [23] 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, we take 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.

The algorithm proposed by [23] is only valid for the particular cost function they use, which in turn depends on their limited program model. The cost function minimizes the maximal dependence distance for a given array. If a given array element is read in multiple statements, then only the last access is optimized, ignoring the opportunity to also optimize locality over the other dependences. Furthermore, although the authors claim their algorithm 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. The simple heuristic we propose in Sect. 4 suffers from the same short-sightedness, but it can more easily be extended to include such effects since the algorithm does not depend on the cost function.

As to complexity, if $v$ and $e$ represent the number of nodes and links respectively, then $e$ ranges between $v$ for a dependency chain and $v^2$ for a fully connected dependence graph, i.e., $e = v^2$, with $1 \leq x \leq 2$. The algorithm requires exactly $v - 1$ iterations and in each iteration at most $v$ minimal distance vectors need to be updated (in the generalized case) to ensure legality. For our heuristic, at most $e$ links have to be updated and/or examined. This leads to a worst-case time complexity of $O(v(v + e)) = O(v^{x+1})$, which compares favorably to the time complexity $O((e \log v)(e + \log v)) = O(v^{2x} \log v)$ cited by Song et al. If optimality is of lesser concern, then the complexity can be further reduced by splitting the dependence graph into strongly connected components and applying the algorithm on each component and subsequently combining them. By also ignoring the heuristic, i.e., selecting two arbitrary nodes in each step, the time complexity reduces to $O(vv' + e)$, where $v'$ is the size of the largest component.

8: Conclusions and future work

We have shown that the translation part of affine transformations can be used as a general multi-dimensional loop fusion. The main result is a simple greedy algorithm for performing this loop fusion incrementally on a large class of programs independently of the cost function used and a proof of its validity. Finally, we have proposed a simple heuristic to optimize data locality using the incremental loop fusion algorithm.

It should be clear that the quality of the fusion depends on the preceding linear transformation step. In particular, this step should identify the dimensions of different loop nests, especially if they have different nesting levels. That is, the dummy dimensions referred to in Sect. 2 should preferably be added at the appropriate places by the linear transformation step. In the future we will therefore focus on this linear transformation step as well as on improving the translation step by investigating and comparing other cost functions. More specifically, experiments similar to those in Sect. 6 have shown that our current heuristic, targeted at temporal locality, is not as effective for spatial locality, resulting in smaller improvements and in some cases even in deterioration. The heuristic will therefore have to be extended when spatial locality also needs to be exploited.
References


