Affine Transformations for Communication Minimal Parallelization and Locality Optimization of Arbitrarily Nested Loop Sequences

Uday Bondhugula[1], Muthu Baskaran[1], Sriram Krishnamoorthy[1],
J. Ramanujam[2], Atanas Rountev[1] and P. Sadayappan[1]

[1]Dept. of Computer Sci. and Engg.
The Ohio State University
2015 Neil Ave. Columbus, OH, USA

[2]Dept. of Electrical & Comp Engg.
Louisiana State University
Baton Rouge, LA

Email: bondhugu@cse.ohio-state.edu

OSU CSE Technical Report
OSU-CISRC-5/07-TR43

Abstract

A long running program often spends most of its time in nested loops. The polyhedral model provides powerful abstractions to optimize loop nests with regular accesses for parallel execution. Affine transformations in this model capture a complex sequence of execution-reordering loop transformations that improve performance by parallelization as well as better locality. Although a significant amount of research has addressed affine scheduling and partitioning, the problem of automatically finding good affine transforms for communication-optimized coarse-grained parallelization along with locality optimization for the general case of arbitrarily-nested loop sequences remains a challenging problem - most frameworks do not treat parallelization and locality optimization in an integrated manner, and/or do not optimize across a sequence of producer-consumer loops.

In this paper, we develop an approach to communication minimization and locality optimization in tiling of arbitrarily nested loop sequences with affine dependences. We address the minimization of inter-tile communication volume in the processor space, and minimization of reuse distances for local execution at each node. The approach can also fuse across a long sequence of loop nests that have a producer/consumer relationship. Programs requiring one-dimensional versus multi-dimensional time schedules are all handled with the same algorithm. Synchronization-free parallelism, permutable loops or pipelined parallelism, and inner parallel loops can be detected. Examples are provided that demonstrate the power of the framework. The algorithm has been incorporated into a tool chain to generate transformations from C/Fortran code in a fully automatic fashion.

1 Introduction and Motivation

Current trends in architecture are increasingly towards larger number of processing elements on chip. This has to led multi-core architectures becoming mainstream along with the emergence of several specialized parallel architectures or accelerators like the Cell processor, general-purpose GPUs (GPGPUs), FPGAs and MPSoCs. The difficulty of programming these architectures to effectively tap the potential of multiple on-chip processing units is a well-known challenge. Among several ways of addressing this issue, one of the very promising and simultaneously hard approach is automatic parallelization. This requires no effort on part of the programmer in the process of parallelization and optimization.

Long running computations often spend most of their running time in nested loops. This is particularly common in scientific applications. The polyhedral model [Fea96] provides a powerful abstraction to reason about transformations on such loop nests by viewing a dynamic instance (iteration) of each statement as an integer point in a well-defined space which is the statement's *polyhedron*. With such a representation for each statement and a particular precise view of inter or intra-statement dependences, it is possible to perform and reason about the correctness and goodness of a sequence of complex loop transformations using machinery from linear programming and linear algebra. The transformations finally reflect in the generated code as reordered execution with improved cache locality and/or loops that have been parallelized. The full power of the polyhedral model is applicable to loop nests in which the data access functions and loop bounds are affine combinations (linear combination with a constant) of the outer loop variables and parameters. Such code is also often called regular code. Irregular code or code with dynamic control can also be handled, but with conservative assumptions on some dependences.

Dependence analysis, transformations and code generation are the three major components of an automatic parallelization framework. In the nineties, dependence analysis [Fea91, Pug92] and code generation [KPR95, GLW98] in the polyhedral model suffered from scalability challenges while no work on automatically finding good transformations with a reasonable and practical cost model exists to date to the best of our knowledge. Hence, applicability was mainly limited to very small loop nests. Significant recent advances in dependence analysis and code generation [VBGC06, QRW00, Bas04, VBC06] have solved these problems resulting in the polyhedral techniques being applied to code representative of real applications like the spec2000fp benchmarks. However, current state-of-the-art polyhedral implementations still apply transformations manually and significant time is spent by an expert to determine the best set of transformations that lead to improved performance [CGP+05, GVB+06]. Our work fills this void and paves the way for a fully automatic parallelizing compiler.

Tiling and loop fusion are two key transformations in optimizing for parallelism and data locality. There has been a considerable amount of research into these two transformations, but very few studies have considered these two transformations in an integrated manner. Tiling has been studied from two perspectives - data locality optimization and parallelization. Tiling for data locality optimization requires grouping points in an iteration space into smaller blocks to maximize data reuse. Tiling for parallelism fundamentally involves partitioning the iteration space into tiles that may be concurrently executed on different processors with a reduced volume and frequency of inter-processor communication. Loop fusion involves merging a sequence of two or more loops into a fused loop structure with multiple statements in the loop body. Sequences of producer/consumer loops are commonly encountered in applications, where a nested loop statement produces an array that is consumed in a subsequent loop nest. In this context, fusion can greatly reduce the number of cache misses when the arrays are large - instead of first writing all elements of the array in the producer loop (forcing capacity misses in the cache) and then reading them in the consumer loop (incurring cache misses), fusion allows the production and consumption of elements of the array to be interleaved, thereby reducing the number of cache misses. Hence, one of the key aspects of an automatic transformation framework is to find good ways of performing tiling and fusion.

The seminal works of Feautrier [Fea88, Fea91, Fea92a, Fea92b] have led to many research efforts on automatic parallelization in the polyhedral model. Existing automatic transformation frameworks [LL98, LCL99, LLL01, AMP01, Gri04] have one or more drawbacks or restrictions that do not allow them to parallelize/optimize long sequences of loop nests. All of them lack a cost model. With the exception of Griebl [Gri04], all focus on one of the complementary aspects of parallelization or locality optimization. Hence, the problem of finding good transformations automatically with a cost model in the polyhedral model has not been addressed. In particular, we are unaware of any reported framework that addresses the following questions:

- What is the best set of tiling hyperplanes, to minimize the volume of communication between tiles (in processor space) as well as improve reuse at each processor?

- How can fusion be automatically enabled across a long sequence of nested loops with the goal of reducing the distance between a producer and a consumer?

The approach we develop in this report answers the above questions. One of our key contributions is the development of a powerful cost model within the polyhedral model that captures communication-minimized parallelism as well as improved reuse in the general case of multiple iteration spaces with affine dependences.

The rest of this report is organized as follows. Section 2 covers the notation and mathematical background for polyhedral model and affine transformations. In Section 3, we describe our algorithm in detail. Section 4 shows application of our approach through a detailed example. Section 5 outlines the entire end-to-end parallelizing compiler infrastructure we implemented our framework in, and provides experimental results on the running time of our tool; optimized code generated for some examples is also shown. Section 6 discusses related work and conclusions are presented in Section 7.

2 Background and Notation

This section provides background information on the polytope/polyhedral model, dependence abstraction, and affine machinery.

2.1 The polytope model

Definition 1 (Hyperplane) *The set X of all vectors $x \in \mathbf{Z}^n$ such that $\vec{h}.\vec{x} = k$, for $k \in \mathbf{Q}$, forms a hyperplane.*

The set of parallel *hyperplane instances* corresponding to different values of k is characterized by the vector $\vec{h}$ which is normal to the hyperplane. Each instance of a hyperplane is an $n-1$ dimensional subspace of the n-dimensional space. Two vectors x_1 and x_2 lie in the same hyperplane if $h.x_1 = h.x_2$.

Definition 2 (Polyhedron, polytope) *The set of all vectors $\mathbf{x} \in \mathbf{Q^n}$ such that $A\mathbf{x} + \mathbf{b} \geq 0$ defines a (convex) polyhedron. A polytope is a bounded polyhedron.*

Each run-time instance of a statement S is defined by its iteration vector $\vec{i}$ which contains values for the indices of the loops surrounding S, from outermost to innermost. A statement S is associated with a polytope D^S of dimensionality m_S. Each point in the polytope is an m_S-dimensional iteration vector, and the polytope is characterized by a set of bounding hyperplanes. This is true when the loop bounds are affine combinations of outer loop indices and structure parameters (typically, symbolic constants representing the problem side); our work is focused on programs with this property. Let the hyperplanes bounding the polytope of statement S_k be given by:

$$a_{S,k} \begin{pmatrix} \vec{i} \\ \vec{n} \end{pmatrix} + b_{S,k} \geq 0, \quad k = 1, m_S \tag{1}$$

where $\vec{n}$ is a vector of the structure parameters. A well-known known result useful in the context of the polytope model is the affine form of the Farkas lemma.

Lemma 1 (Affine form of Farkas Lemma) *Let $\mathcal{D}$ be a non-empty polyhedron defined by p affine inequalities or faces*

$$a_k.x + b_k \geq 0, \quad k = 1, p$$

Then, an affine form ψ is non-negative everywhere in $\mathcal{D}$ iff it is a positive affine combination of the faces:

$$\psi(x) \equiv \lambda_0 + \sum_k \lambda_k (a_k x + b_k),\ \lambda \geq 0 \tag{2}$$

The non-negative constants λ_k are referred to as Farkas multipliers. Proof of the *if* part is obvious. For the *only if* part, see Schrijver [Sch87].

2.2 Dependence Abstraction

Our dependence model is the same as the one used by Feautrier [Fea92a] and Lim et al. [LL98, LCL99]. Dependences are determined precisely through dataflow analysis [Fea91]. The Generalized Dependence Graph (GDG) is a directed multi-graph represented by the tuple (V, E, D, R), where V is the set of vertices with each vertex representing a statement, E is the set of edges where an edge from node S_i to S_j represents a dependence from an instance of S_i to an instance of S_j. D is a function from V to the polytopes associated with the statements. R is a function from E to the corresponding dependence relations. There may be multiple edges between two statements, as well as a self-edge for a vertex. The notation we use is similar to that in [Fea92a].

For a dependence from S_i to S_j characterized by an edge $e \in E$, let R_e be the corresponding dependence relation. Then, exact dependence analysis makes sure that the dependence relation R_e can be expressed in a minimal form. A polyhedron P_e and an affine transformation f_e are obtained such that

$$\vec{p} \in D^{s_i},\ \vec{q} \in D^{s_j},\ \langle \vec{p}, \vec{q} \rangle \in R_e \quad \equiv \quad (\vec{p} = f_e(\vec{q}) \wedge \vec{q} \in P_e) \tag{3}$$

where $P_e = \{\vec{q} \mid \vec{q} \in D^{s_j},\ f_e(\vec{q}) \in D^{s_i},\ \langle f_e(\vec{q}), \vec{q} \rangle \in R_e\}$ and f_e represents the affine function mapping the target iteration vector $\vec{q}$ to the source $\vec{p}$. f_e is known as the *h-transformation* and P_e is the dependence polyhedron. We have such an h-transformation and a dependence polyhedron for every dependence, $e \in E$. Let P_e be defined by:

$$c_{e,k} \begin{pmatrix} \vec{i} \\ \vec{n} \end{pmatrix} + d_{e,k} \geq 0,\ \ k = 1, m_e \tag{4}$$

2.3 Affine transforms

A one-dimensional affine transform for statement S_k is defined by:

$$\begin{aligned} \phi_{S_k} &= \begin{bmatrix} c_1\ c_2\ \ldots\ c_{m_{S_k}} \end{bmatrix} \begin{pmatrix} \vec{i} \end{pmatrix} + c_0 \\ &= \begin{bmatrix} c_1\ c_2\ \ldots\ c_{m_{S_k}}\ c_0 \end{bmatrix} \begin{pmatrix} \vec{i} \\ 1 \end{pmatrix} \\ &= h_{S_k}\vec{i} + c_0 \end{aligned} \tag{5}$$

where $h_{S_k} = [c_1, c_2, \ldots, c_{m_{S_k}}]$ with $c_1, c_2, \ldots, c_{m_{S_k}} \in \mathbf{Z}$.

A multi-dimensional affine transform can be represented as a matrix and a vector. Each row of the matrix is a 1-d transform and has m_{S_k} columns, each column corresponding to a coefficient of of h_{S_k} in that order.

```
do i = 1, n
   do j = 1, n
      (S1) C[i,j] = 0;
   end do
end do

do i = 1, n
   do j = 1, n
      do k = 1, n
         (S2) C[i,j] = C[i,j] + A[i,k] * B[k,j]
      end do
   end do
end do

do i = 1, n
   do j = 1, n
      do k = 1, n
         (S3) D[i,j] = D[i,j] + E[i,k] * C[k,j]
      end do
   end do
end do
```

	$S1$			$S2$				$S3$			
	i	j	$const$	i	j	k	$const$	i	j	k	$const$
c_1	1	0	0	1	0	0	0	0	0	1	0
c_2	0	1	0	0	1	0	0	0	1	0	0
c_3	0	0	0	0	0	0	0	0	0	0	1
c_4	0	0	0	0	0	1	0	1	0	0	0

```
do c1=1, n
  do c2=1, n
    C[c1,c2] = 0;
    do c4=1, n
       C[c1,c2] = C[c1,c2] + A[c1,c4] * B[c4,c2]
    end do
    do c4=1, n
       D[c4,c2] = D[c4,c2] + E[c4,c1] * C[c1,c2]
    end do
  end do
end do
```

Figure 1. Statement-wise transformation and the corresponding transformed code

The vector carries the translation constants (c_0). Such a transformation matrix defines a one-to-one map from every point in the original iteration space to a point in the target iteration space if and only the matrix has full column rank, i.e., there are as many independent rows as the dimensionality of the iteration space of the corresponding statement. However, the total number of rows in the matrix may be much larger as some rows serve the purpose of representing partially fused or unfused loops at a level. Such a row has all zeros for the matrix row, and a particular constant for c_0: all statements with the same c_0 value are fused at that level and the unfused sets are placed in the increasing order of their c_0s.

We now give an example to show how statement-wise full-ranked affine transforms completely determine the transformed target loop structure and thus the new execution order. This includes fused, unfused, or partially fused loops as well as permutation, reversal, relative shifting, and skewing. Consider the code in Fig. 1. All three statements are transformed to a common space of dimensionality equal to the depth of the statement with the maximum depth (in this case three). Consider the transformation matrices shown. i loop of S1, i loop of S2, and k loop of S3 are fused as the new c_1 loop. c_2 target loop is obtained by fusion of the j loop of S1, j loop of S2, and j loop of S3. c_3 represents a split of S3 from S1, S2. The above transformation is indeed legal and allows immediate consumption of matrix C and it can be contracted to a scalar. As we will see later, our framework can enable such an transformation automatically.

The above representation for transformations is similar to that used by many researchers: earlier by [Fea92b, Kel96], and more recently in a systematic way by [CGP+05, GVB+06] and directly fits with scattering functions that a code generation tool like CLooG [Bas04, clo] supports. On providing such a representation, the target code shown can be generated by scanning the statement polyhedra in the global lexicographic ordering w.r.t the new loops $c_1, c_2, \ldots$. Our problem is thus to find the coefficients of the transformation matrices (along with the vectors) that are best for parallelism and locality.

3 Finding good affine transforms

Let there be a dependence from statement instance $\vec{p}$ of S_i to $\vec{q}$ of S_j corresponding to an edge e of the GDG of a program. After exact dependence analysis, we obtain,

$$\vec{p} = f_e(\vec{q}), \;\; \vec{q} \in P_e$$

where P_e and f_e are the dependence polyhedron and the h-transformation respectively as defined in (3).

Lemma 2 *Let ϕ_{s_i} be a one-dimensional affine transform for S_i. For ϕ_{s_1}, ϕ_{s_2}, ..., ϕ_{s_k}, to be a* coordinated valid tiling hyperplane *for all statements, the following should hold for each edge e from S_i and S_j:*

$$\begin{aligned} \phi_{s_j}(\vec{q}) - \phi_{s_i}(\vec{p}) &\geq 0, \;\; \langle p, q\rangle \in R_e \\ \phi_{s_j}(\vec{q}) - \phi_{s_i}(f_e(\vec{q})) &\geq 0, \;\; q \in P_e \end{aligned} \tag{6}$$

Proof. Tiling defined by a set of tiling hyperplanes is said to be legal if each tile can be executed atomically and a valid total ordering of the tiles can be constructed. This implies that there exists no two tiles such that they both influence each other. Since a dependent iteration is mapped to the same hyperplane or a greater hyperplane than the source, the set of all iterations that are outside of the tile and are influenced by it always lie in the forward direction along one of the independent tiling dimensions. Similarly, all outside iterations influencing a tile are either in that tile or in the backward direction along one or more of the tiling hyperplanes. Hence, an affine transform that satisfies the above constraint can be executed atomically with communication needed only before and after its execution. The above argument holds true for inter-statement dependences too. A dependence from $\vec{p}$ of S_i to $\vec{q}$ of S_j leads to $\vec{q}$ being mapped to a greater or a hyperplane with the same value, leading to an interleaved execution of tiles of iteration spaces of each statement (when code is generated from these mappings). Hence, in the presence of any number of dependences, tiles of S_1, S_2, ..., S_k defined by ϕ_{s_1}, ϕ_{s_2}, ..., ϕ_{s_k} can be executed in an interleaved fashion atomically.

The above condition was well known for the case of a single-statement perfectly nested loop with constant dependences from the work on Irigoin and Triolet [IT88] (as $H.D \geq 0$). We have generalized it above for multiple iteration spaces with affine dependences (constant and non-constant) with possibly different dimensionalities. Clearly, this exists a strong relationship between tiling hyperplanes of each statement and they are found in a coordinated fashion. The interleaved execution also leads to the notion of fusion that we discuss later in this section.

In the rest of this report, we use the term affine transform (with this property) and tiling hyperplane interchangeably, since after applying the transform, the target loop in the transformed iteration space can be blocked rectangularly as all dependences have positive components along that hyperplane.

Tiling hyperplanes are used for parallelization as well as for data locality optimization. Our goal is to find optimal hyperplanes for that purpose for all statements, i.e., the output of our algorithm should be, for each statement, as many hyperplanes as the dimensionality of its iteration space polytope, forming a spanning basis. *Since each hyperplane either carries a dependence or puts it in its null space (due to ≥ 0 in Eqn.6), and since the set of target hyperplanes span the entire polytope of the statement, every dependence is carried at some level or the other.* This approach of finding spanning bases is different from most works in this area that obtain good schedules as outer loops and then inner parallel loops. Parallel code generation in both cases is different; we also address this later.

Consider the perfectly nested version of 1-d Jacobi shown in Fig. 2(a) as an example. This discussion also applies to the imperfectly nested version, but for convenience we first look at the single-statement perfectly nested version. We first describe solutions obtained by existing state of the art approaches - Lim and Lam's affine partitioning [LL98, LCL99] and Griebl's space and time tiling with FCO placement [Gri04]. Lim and Lam define legal time partitions which have the same property of tiling hyperplanes we described in the previous section. Their algorithm obtains affine partitions that minimize the *order* of communication while maximizing the *degree* of parallelism. Using the validity constraint in Eqn 6, we obtain the constraints: $(c_t \geq 0; c_i + c_j \geq 0; c_i - c_j \geq 0)$.

```
for t = 1,T do
  for I = 2,N-1 do
    a[t,i] = 0.33*(a[t-1,i] + a[t-1,i-1] + a[t-1,i+1])
  end for
end for
```

(a) 1-d Jacobi: perfectly nested

```
for t = 1 to T do
  for i = 2 to N-1 do
    S1: $b[i] = 0.33 * (a[i-1] + a[i] + a[i+1])$
  end for
  for i = 2 to N-1 do
    S2: $a[i] = b[i]$
  end for
end for
```

(b) 1-d Jacobi: imperfectly nested

Figure 2. 1-d Jacobi

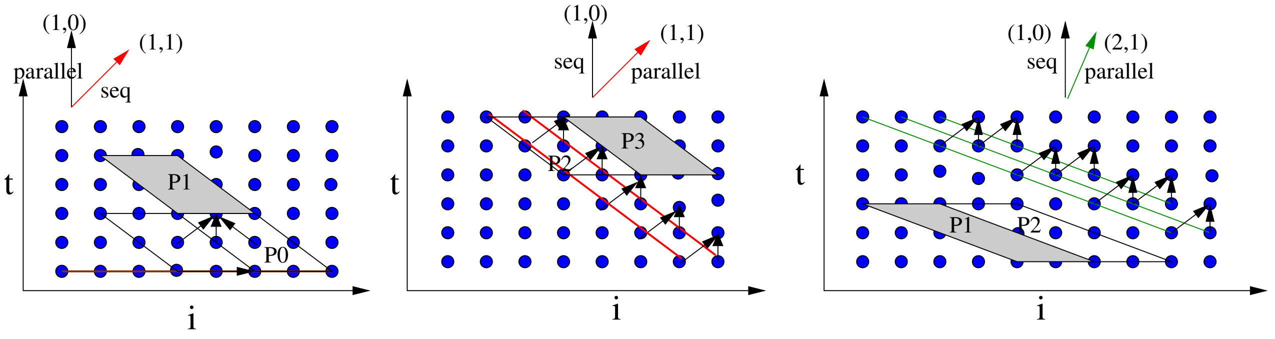

Figure 3. Communication volume with different valid hyperplanes for 1-d jacobi

There are infinitely many valid solutions with the same order complexity of synchronization, but with different communication volumes that may impact performance. Although it may seem that the volume may not effect performance considering the fact that communication startup time on modern interconnects dominates, for higher dimensional problems like n-d Jacobi, the ratio of communication to computation increases (proportional to tile size raised to $n-1$). Existing works on tiling [SD90, RS92, Xue97] can find near communication-optimal tiles for perfectly nested loops with constant dependences, but cannot handle arbitrarily nested loops. For 1-d Jacobi, all solutions within the cone formed by the vectors $(1,1)$ and $(1,-1)$ are valid tiling hyperplanes[1]. For imperfectly nested Jacobi, Lim's algorithm [LL98] finds two valid independent solutions without optimizing for any particular criterion. In particular, the solutions found by their algorithm (Algorithm A in [LL98]) are $(2,-1)$ and $(3,-1)$ which are clearly not the best tiling hyperplanes to minimize communication volume, though they do minimize the *order* of synchronization which is $O(N)$ (in this case any valid hyperplane has $O(N)$ synchronization). Figure 3 shows that the required communication increases as the hyperplane gets more and more oblique. For a hyperplane with normal $(k,1)$, one would need $(k+1)T$ values from the neighboring tile.

Using Griebl's approach, we first find that only space tiling is enabled with Feautrier's schedule being $\theta(t,i)=t$. With FCO placement along (1,1), time tiling is enabled that can aggregate iterations into time tiles thus decreasing the frequency of communication. However, note that communication in the processor space occurs along (1,1), i.e., two lines of the array are required. However, using (1,0) and (1,1) as tiling hyperplanes with (1,0) as space and (1,1) as inner time and a tile space schedule of (2,1) leads to only one line of communication along (1,0). Our algorithm finds such a solution.

We now develop a cost metric for an affine transform that captures reuse distance and communication volume.

3.1 Cost function

We define an affine form δ_e:

$$\delta_e(\vec{q}) \quad = \quad \phi_{s_i}(\vec{q}) - \phi_{s_j}(f_e(\vec{q})), \;\; \vec{q} \in P_e \tag{7}$$

The affine form $\delta_e(\vec{q})$ holds much significance. This function is also the number of hyperplanes the dependence e traverses along the hyperplane normal. It gives us a measure of the reuse distance if the hyperplane is used as time, i.e., if the hyperplanes are executed sequentially. Also, this function is a rough measure of communication volume if the hyperplane is used to generate tiles for parallelization and used as a processor space dimension. An upper bound on this function would mean that the number of hyperplanes that would be communicated as a result of the dependence at the tile boundaries would not exceed this bound. We are particularly interested if this function can be reduced to a constant amount or zero by choosing a suitable direction for ϕ: if this is possible, then that particular dependence leads to a constant or no communication for this hyperplane. Note that each δ_e is an affine function of the loop indices. The challenge is to use this function to obtain a suitable objective for optimization in the affine framework.

3.2 Challenges

The constraints obtained from Eqn 6 above only represent validity (permutability). We discuss below problems encountered when one tries to apply a performance factor to find a good tile shape out of the several possibilities.

The Farkas lemma has been used by many approaches in the polyhedral model [Fea92a, Fea92b, LL98, Gri04, CGP$^+$05, PBCV07] to eliminate loop variables from constraints by getting equivalent linear inequalities. The affine form in the loop variables is expressed equivalently as a positive linear combination of the

[1]For the imperfectly nested version of 1-d Jacobi, the valid cone is $(2,1)$ and $(2,-1)$

faces of the dependence polyhedron. When this is done, the coefficients of the loop variables on the left and right hand side are equated to eliminate the constraints of variables. This is done for each of the dependences, and the constraints obtained are aggregated. The resulting constraints are entirely in the coefficients of the tile mappings and Farkas multipliers. All Farkas multipliers can be eliminated, some by Gaussian elimination and the rest by Fourier-Motzkin [Sch87]. However, an attempt to minimize communication volume ends up in an objective function involving both loop variables and hyperplane coefficients. For example, $\phi(\vec{q}) - \phi(f_e(\vec{q}))$ could be $c_1 i + (c_2 - c_3) j$, where $1 \leq i \leq N \wedge 1 \leq j \leq N \wedge i \leq j$. One could possibly end up with such a form when one or more of the dependences are not uniform, making it infeasible to construct an objective function involving only the unknown hyperplane coefficients.

A possible approach touched upon by Feautrier is to visit all vertices of the polyhedron in the hyperplane coefficients space characterized by the constraints that express validity. It is likely that vertices will dominate all other points in the solution space. However, this procedure is not scalable beyond the smallest inputs. For example, for a sequence of two nested loops, each with a 3-d iteration space, the number of coefficients is at least 14. p unknowns could lead to exploration of up to 2^p vertices (hypercube) in the worst case.

It is also plausible that a positive spanning basis to the set of constraints obtained is better than other solutions. This is due to the fact that any valid tiling hyperplane can be expressed as a positive linear combination of the vectors in the positive spanning basis and that the basis represents the tight extreme vectors for the cone of solutions. This is indeed true for the perfectly nested 1-d Jacobi for which $(1,1)$ and $(1,-1)$ are good hyperplanes. However, we do not know whether this holds in the general case, but clearly they are sub-optimal when compared to (1,0) and (1,1) for perfectly nested 1-d Jacobi.

3.3 Details of Approach

We first discuss a result that would take us closer to the solution.

Lemma 3 *If all iteration spaces are bounded, there exists at least one affine form v in the structure parameters $\vec{n}$, that bounds $\delta_e(\vec{q})$ for every dependence edge e, i.e., there exists*

$$v(\vec{n}) = u.\vec{n} + w \tag{8}$$

such that

$$\begin{aligned} v(\vec{n}) \;-\; \left(\phi_{s_i}(\vec{q}) - \phi_{s_j}(f_e(\vec{q}))\right) \quad &\geq \quad 0, \;\; \vec{q} \in P_e, \forall e \in E \\ v(\vec{n}) \;-\; \delta_e(\vec{q}) \quad &\geq \quad 0, \;\; \vec{q} \in P_e, \forall e \in E \end{aligned} \tag{9}$$

The idea behind the above is that even if δ_e involves loop variables, one can find large enough constants in u that would be sufficient to bound $\delta_e(\vec{p})$. Note that the loop variables themselves are bounded by affine functions of the parameters, and hence the maximum value taken by $\delta_e(\vec{p})$ will be bounded by such an affine form. Also, since $v(\vec{n}) \geq \delta_e(\vec{p}) \geq 0$, v should increase with an increase in the structural parameters, i.e., the coordinates of u are positive. The reuse distance or communication volume for each dependence is bounded in this fashion by the same affine form.

Now, we apply the Farkas lemma to (9).

$$v(\vec{n}) - \delta_e(\vec{q}) \quad \equiv \quad \lambda_{e0} + \sum_{k=1}^{m_e} \lambda_{ek} \left(c_{ek} \begin{pmatrix} \vec{i} \\ \vec{n} \end{pmatrix} + d_{ek} \right) \tag{10}$$

The above is an identity and the coefficients of each of the loop indices in $\vec{i}$ and parameters in $\vec{n}$ on the left and right hand side can be gathered and equated. We now get linear inequalities entirely in coefficients of the affine mappings for all statements, components of row vector $\vec{u}$, and w. The above inequalities can be at once be solved by finding a lexicographic minimal solution with $\vec{u}$ and w in the leading position, and the other variables following in any order.

$$\text{minimize}_{\prec} \, \{u_1, u_2, \ldots, u_k, w, \ldots, c_i's, \ldots\} \tag{11}$$

Finding the lexicographic minimal solution is within the reach of the simplex algorithm and can be handled by the PIP software [Fea88]. Since the structural parameters are quite large, we first want to minimize their coefficients. We do not lose the optimal solution since an optimal solution would have the smallest possible values for u's. Note that the relative ordering of the structural parameters and their values at runtime may effect the solution, but considering this is beyond the scope of this approach.

The solution gives a hyperplane for each statement. Note that the application of the Farkas lemma to (9) is not required in all cases. When a dependence is uniform, the corresponding δ_e is independent of any loop variables, and application of the Farkas lemma is not required. In such cases, we just have $w \geq \delta_e$.

Finding independent solutions. Minimizing the objective function with the simplex algorithm gives us a single solution to the coefficients of the best mappings for each statement. We need at least as many independent solutions as the dimensionality of the polytope associated with each statement. Hence, once a solution is found, we run the LP solver again to find the next solution with the constraints being augmented by new ones. The new constraints make sure of independence with solutions found so far. Let the rows of H_S represent the solutions found so far for a statement S. Then, the sub-space orthogonal to H_S is given by:

$$J = I - H_S^T \left(H_S H_S^T\right)^{-1} H_S \tag{12}$$

Note that $J.{H_S}^T$ is $\mathbf{0}$, i.e., the rows of H_S are orthogonal to those of J. Any one of the inequalities given by, $J.\vec{h'_S} > \vec{0}$ or $J.\vec{h'_S} < \vec{0}$ gives the necessary constraint to be added for statement S to make sure that h'_S has a non-zero component in the sub-space orthogonal to H_S. Note that orthogonality is satisfied if either one of the constraints in $J.\vec{h'_S} \geq 1$ or $J.\vec{h'_S} \leq -1$ is satisfied for a given statement. Ideally, all cases have to be tried and the best among those chosen. When the number of statements is large, this leads to a large number of cases. In such cases, we restrict ourselves to the sub-space of the orthogonal space where all the constraints are positive. This leads to the addition of the constraints: $J.h'_S \geq 0 \wedge \sum J_i.h'_S \geq 1$, where J_i is a row of J, i.e., a single case in which a fixed number of constraints is added for each statement. As more independent solutions are found, the dimensionality of the orthogonal sub-space reduces and the fixed number of constraints added for each statement decreases. By just considering a particular portion of the orthogonal sub-space, we are mainly discarding solutions that involve loop reversals or combination of reversals with other transformations. However, in practice, we believe that this may not make a significant difference.

The mappings found are independent on a per-statement basis. When there are statements with different dimensionalities, the number of such independent mappings found for each statement is equal to the number of outer loops it has in the original program. Hence, no more orthogonality constraints need be found for statements for which enough independent solutions have been found (the rest of the rows get automatically filled with zeros). As mentioned in Sec. 2.3, the number of rows in the transformation matrix is the same for each statement and the depth of the deepest loop nest in the target loop code is the same as that of the source loop nest. Overall, a hierarchy of fully permutable loop nests are found, and a lower level in the hierarchy will not be obtained unless constraints corresponding to dependences that have been carried by the parent permutable set have been removed.

3.4 Communication and locality optimization unified

From the algorithm described above, both synchronization-free and pipelined parallelism is found. Note that the best possible solution to Eqn. (11) is with $(u = 0, w = 0)$ and this happens when we find a hyperplane that has no dependence components along its normal, which is a fully parallel loop requiring no synchronization if it is at the outer level (*outer parallel*); it could be an inner parallel loop if some dependences were removed previously and so a synchronization is required after the loop is executed in parallel. Thus, in each of the steps that we find a new independent hyperplane, we end up first finding all synchronization-free hyperplanes; these are followed by a set of fully permutable hyperplanes that are tilable and pipelined parallel requiring constant boundary communication $(u = 0; w > 0)$ w.r.t the tile sizes. In the worst case, we have a hyperplane with $u > 0, w \geq 0$ resulting in long communication from non-constant dependences. It is important to note that the latter are pushed to the innermost level. By bringing in the notion of communication volume and its minimization, all degrees of parallelism are found in the order of their preference.

From the point of view of data locality, note that the hyperplanes that are used to scan the tile space are same as the ones that scan points in a tile. Hence, data locality is optimized from two angles: (1) cache misses at tile boundaries are minimized for local execution (as cache misses at local tile boundaries are equivalent to communication along processor tile boundaries); (2) by reducing reuse distances, we are increasing the size of local tiles that would fit in cache. The former is due to selection of good tile shapes and the latter by the right permutation of hyperplanes (which is implicit in the order in which we find hyperplanes).

3.5 Space and time in transformed iteration space.

By minimizing $\phi(q) - \phi(p)$ as we find hyperplanes from outermost to innermost, we push dependence carrying to inner loops and also ensure that no loops have negative dependences components so that all target loops can be blocked. Once this is done, if the outer loops are used as space (how many ever desired, say k), and the rest are used as time (note that at least one time loop is required unless all loops are synchronization-free parallel), communication in the processor space is optimized as the outer space loops are the k best ones. All loops can be tiled resulting in coarse-grained parallelism as well as better reuse within a tile. Hence, the same set of hyperplanes are used to scan points in a tile, while a transformation is necessary in the outer tile space loops to get a tile schedule for parallel code generation. This is addressed in Sec. 3.10.

3.6 Fusion in the affine framework

The same affine hyperplane partitioning algorithm described in the previous section can enable fusion across multiple iteration spaces that are weakly connected, as in sequences of producer-consumer loops.

Consider the sequence of two matrix-vector multiplies in Figure 4(a). Applying our algorithm on it first gives us only one solution:

$$\phi_{S_1} = (1,0), \qquad \phi_{S_2} = (0,1)$$

This implies fusion of the i loop of S_1 and the j loop of S_2. Putting the orthogonality constraint now, we do not obtain any more solutions. Hence, now removing the dependence dismissed by it, and running affine partitioning again does not yield any solutions as the loops cannot be fused further. The remaining unfused loops are thus placed lexicographically inside as shown in Figure 4(b).

```
do i = 1, N
  do j = 1, N
    S1: x[i] = x[i]+a[i,j]*y[j]
  end do
end do
do i = 1, N
  do j = 1, N
    S2: y[i] = y[i]+a[i,j]*x[j]
  end do
end do
```

(a) Original

```
do c1 = 1, N
  do c3 = 1, N
    S1: x[c1] = x[c1]+a[c1,c3]*y[c3]
  end do
  do c3 = 1, N
    S2: y[c3] = y[c3]+a[c3,c1]*x[c1]
  end do
end do
```

(b) Fused

S1			*S2*		
i	*j*	*const*	*i*	*j*	*const*
1	0	0	0	1	0
0	0	0	0	0	1
0	1	0	1	0	0

(c) Transformation

Figure 4. Two matrix vector multiplies

Solving for hyperplanes for multiple statements leads to a schedule for each statement such that all statements in question are *finely* interleaved: this is indeed fusion with either a permutation, a skew, and/or a constant shift. In many cases, we find it to be a permutation (as in Figure 4) or a constant shift (shown for 1-D imperfectly nested Jacobi later). Hence, a common tiling hyperplane also represents a fused loop, and can be employed for components that are weakly connected to reduce reuse distances. Note that it is important to leave the structure parameter $\vec{n}$ out of our affine transform definition in 6 for the above to hold true. The set of valid independent hyperplanes that can be found from our algorithm when multiple statements are involved is the maximum number of loops that can be fused for those statements.

Consider the sequence of three matrix vector multiplies such as $y = Ax$; $z = By$; $w = Cz$. Together, these three loop nests do not have a common surrounding loop. However, it is possible to fuse the first two or the last two. When our algorithm is run on all three components, no valid tiling hyperplane is found. There exists a valid tiling hyperplane when the algorithm is ran for M_1 and M_2, or for M_2 and M_3, similar to that in Figure 4. With this motivation, we generalize the notion of fusion in the affine framework.

Fusion for a sequence of loop nests. Given the GDG, a set of nodes on a path can be tested for fusion using our algorithm. Let there be a path of strongly-connected components (a chain of strongly-connected components that are weakly connected) of length n with a maximum nesting depth of m (for some statement in it). If there exist m independent solutions to our constraints, the statements are fully fusable, i.e., they can be embedded into a perfectly nested loop. If no valid solution exists (as in the case of three matrix vector multiply sequences described earlier), there is no common loop for all of the n statements. The number of solutions found gives the number of fusible loops. If the number of solutions found is less than m, dependences between two adjacent strongly connected components need to be *cut*, and these dependences are ignored from both validity and objective constraints for solutions to be found yet. Note that there are multiple places where the cut can be placed. Currently, we use a simple heuristic based on the number of dependences crossing two adjacent strongly connected components. After transformation, the transformed structures need to be placed one after the other – this is expressed in the transformation matrix by adding a row (to the matrix of each statement) which maps all statements preceding the point where the cut was made to zero, and those after to one – i.e., $\phi(S_i) = 0$ if $i \leq u$ and $\phi(S_i) = 1$ if $i \geq v$, where S_u and S_v are the first and last statements respectively in the two strongly connected components between which the cut was made. The process is repeated recursively till as many independent solutions are found for the deepest statement as its nesting depth. Thus, the transformation matrix found for every statement has full column rank. Rows of the transformation matrix are in a suitable form to be expressed as scattering functions to a code generation tool like CLooG [clo, Bas04]. For example, for the sequence of matrix-vector multiplies, the transformation matrices corresponding to the transformation in Fig. 4 are below.

Multi-dimensional schedules. Code for which other techniques in the literature come up with multidimensional schedules (i.e., code that does not accept one-dimensional schedules), are handled with our algorithm without any special treatment.

```
do i = 1, N
  do j = 1, N
    s = s + a[i,j];
  end do
end do
```

Consider the above example from the literature [Fea92b]. Using our algorithm, (1,0) is the first and the only hyperplane found that satisfies the permutability constraint for all dependences. Unless dependences carried by it are removed, we cannot find another solution. When this is done, the dependence $(i, j) \rightarrow (i', j')$ where $i' = i+1 \wedge j = N \wedge j' = 1$ is removed; subsequently, (0,1) is found as the next hyperplane.

A loop nest is not amenable to any parallelization if a single solution to the mapping coefficients is obtained repeatedly at every level, and the last loop also carries a dependence. This corresponds to a completely sequential loop nest or multiple loop nests that have only one common surrounding loop (carrying a dependence) at any level.

3.7 Summary

We can broadly conclude the following from this section. For affine partitioning on a path in the GDG, the set of independent solutions obtained from our algorithm have the following properties.

If k independent solutions are found at any level by our algorithm, the following are equivalent:

1. There exist k fully permutable loops at that level
2. There exist at least $k-1$ degrees of parallelism
3. There exist k tilable loops at that level
4. There exist k common (fused) surrounding loops at that level

Our entire algorithm is summarized below.

Algorithm 1 Affine transformation algorithm

```
Input  Generalized dependence graph (includes h-transformations and dependence polyhedra)
 1: for each dependence $d$ do
 2:    Build legality constraints: if the dependence is not an intra-statement uniform dependence, apply
       Farkas lemma on $\phi(\vec{q}) - \phi(f_d(\vec{q})) \geq 0$ under $\vec{q} \in P_d$, and eliminate all Farkas multipliers
 3:    Build communication volume/reuse distance bounding constraints: apply Farkas lemma to $v(\vec{n}) -$
       $(\phi(\vec{q}) - \phi(f_d(\vec{q})) \geq 0$ under $\vec{q} \in P_d$, and eliminate all Farkas multipliers
 4:    Aggregate constraints from the above two into $C_d(i)$
 5: end for
 6: repeat
 7:    $C \leftarrow \emptyset$
 8:    for each dependence $d$ that is not carried do
 9:       $C \leftarrow C \cup C_d(i)$
10:    end for
11:    Find as many independent solutions as possible to $C$ (lexmin with $u$'s coefficients in the leading posi-
       tion); orthogonality constraints are added after each soln is found
12:    if no solutions were found then
13:       Cut dependences between two strongly-connected components in the GDG and insert the appropriate
          splitter in the transformation matrices of the statements
14:    end if
15:    Remove dependences carried by solutions found above
16: until Enough independent solutions for the statement with maximum iteration space dimensionality are
    found
Output  A transformation matrix for each statement
```

3.8 Accuracy of metric and refinement.

The metric we presented here can be refined while keeping the problem within ILP. The motivation behind taking a *max* is to avoid multiple counting of the same set of points that need to be communicated for different dependences. This happens when all dependences originate from the same data space and the same order volume of communication is required for each of them. Using the sum of max'es on a per-array basis is a more accurate metric. Also, even for a single array, sets of points with very less overlap or no overlap may have to be communicated for different dependences. Also, different dependences may have source dependence polytopes of different dimensionalities. Note that the image of the source dependence polytope under the data access function associated with the dependence gives the actual set of points to be communicated. Hence, just using the communication rate (number of hyperplanes on the tile boundary) as the metric may not be accurate enough. This can be taken care of by having different communication volume bounds for dependences with different orders of communication, and using the bound coefficients for dependences with higher orders of communication as the leading coefficients while finding the lexicographic minimal solution. Hence, the metric can be easily refined within the linear cost model we have.

3.9 Limitations

Trade-off between fusion and parallelization. Consider the sequence of matrix vector multiplies shown in Fig. 4. Fusing it allows better reuse, however it leads to loss of parallelism. Both loop nests can be parallelized in a synchronization-free fashion when each of them is treated separately, and a synchronization is needed between them. However, after fusion we only get an inner level of parallelism from the inner loop of S2. Our approach cannot select the better of these two. It would always fuse if it is legal.

Concurrent start. The choice of space loops in the transformed space need not be just made based on communication volume. When there is a pipelined start-up processor space, the pipelined start-up can be eliminated in some cases [KBB+07]. For 1-D Jacobi, it happens that concurrent start can be enabled by split or overlapped tiling along (1,1), but not along (1,0); using (1,1) as space with split tiling is shown to provide better performance [KBB+07].

3.10 Parallel code generation

Code generation under multiple affine mappings was first addressed by Kelly et al. [KPR95]. Significant advances were made by Quilleré et al [QRW00] and more recently by Bastoul [Bas04] and Vasilache et al [VBC06], and have been implemented into a freely available tool, CLooG [clo]. The transformations we produce can be readily given to CLooG as scattering functions. Unlike most other works, since we find sets of fully permutable loops, there may not be a single loop in the transformed space that carries all dependences (even if the code admits a one dimensional schedule). The outer loops we find are space loops and care has to be taken in the parallel code generation when these loops are pipelined parallel. Our approach to coarse-grained (tiled) parallel code generation is as follows.

1. At any level we have a set of fully permutable loops in the transformed space. Tile all loops in each such set.

2. Perform the following unimodular transformation on only the outer tile loops that step through the tile space (for each set):
   $(l_1, l_2, \ldots, l_k) \rightarrow (l_1 + l_2 + \cdots + l_k, l_2, \ldots, l_k)$. This gives us an outer loop that is a valid tile schedule

3. Place a barrier at the end of the tile schedule loop (inside it) - one barrier for each set of permutable loops found.

Fig. 5 shows this for a simple example with tiling hyperplanes (1,0) and (0,1). Since each target loop has a non-negative component for every dependence and since all dependence are carried at some level or the other, the sum of all ϕ's satisfies is a valid tile schedule. Note that communication still happens along boundaries of l_1, l_2, …, l_s, and the same old hyperplanes $l_1, l_2, \ldots, l_k$ are used to scan a tile. Hence, reuse distances are minimized while scanning a tile and communication in the processor space is minimal. Note that obtaining an affine schedule and then enabling time tiling would still lead to communication along a non-optimal hyperplane.

```
do i = 1, N do
  do j = 1, N do
    a[i,j] = a[i−1,j] + a[i,j−1]
  end do
end do
```

(a) Original

```
do t=2, 2n, B
  doall P=max(1,t−N), min(t−1,N), B do

    do tT=t−p, min(t−p+B,N), 1
      do pT = p, min(p+B,N), 1
        a[tT,pT] = a[tT−1,pT]+a[tT,pT−1];
      end do
    end do

  end doall
  barrier
end do
```

(b) Coarse-grained parallel barrier

Figure 5. Shared memory parallel code generation example

4 Examples

In this section, we apply our algorithm on different examples.

4.1 Example 1: Non-constant dependences

Figure 6 shows an example from the literature [DV97] with affine non-constant dependences. We exclude the constant c_0 from the inequalities as we have a single statement. Dependence analysis produces the following h-transformations and dependence polyhedra:

```
do i = 1, N
 do j = 2, N
  a[i,j] = a[j,i]+a[i,j−1]
 end do
end do
```

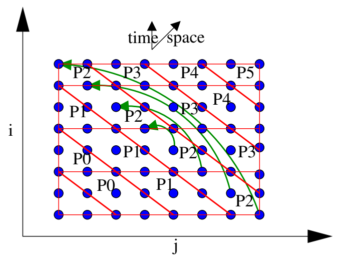

Figure 6. Example 1: Non-constant dependences

$$
\begin{aligned}
\text{flow} : a[i',j'] \rightarrow a[i,j-1] \\
h : i' = i, j' = j-1; \quad P_1 : 2 \le j \le N, 1 \le i \le N \\
\text{flow} : a[i',j'] \rightarrow a[j,i] \\
h : i' = j, j' = i; \quad P_2 : 2 \le j \le N,\ 1 \le i \le N, i-j \ge 1 \\
\text{anti} : a[j',i'] \rightarrow a[i,j] \\
h : j' = i, i' = j \quad P_3 : 2 \le j \le N,\ 1 \le i \le N,\ i-j \ge 1
\end{aligned}
$$

Dependence 1: Tiling legality constraint:

$$c_i i + c_j j - c_i i - c_j(j-1) \ge 0 \quad \Rightarrow \quad c_j \ge 0$$

Since this is a constant dependence, the volume bounding constraint gives:

$$w - c_j \ge 0$$

Dependence 2: Tiling legality constraint:

$$(c_i i + c_j j) - (c_i j + c_j i) \ \ge\ 0, \quad (i,j) \in P_2$$

Applying Farkas lemma, we have:

$$
\begin{aligned}
(c_i - c_j)i \quad + \quad & (c_j - c_i)j \\
\equiv \quad & \lambda_0 + \lambda_1(N-i) + \lambda_2(N-j) \\
& +\lambda_3(j-i-1) + \lambda_4(i-1) + \lambda_5(j-1) \\
& \lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5 \ge 0
\end{aligned} \tag{13}
$$

LHS and RHS coefficients for i, j, N and the constants are equated in (13) and the Farkas multipliers are eliminated through Fourier-Motzkin. The reader may verify that doing this yields:

$$c_i - c_j \ge 0$$

Volume bounding constraint:

$$u_1 N + w - (c_i j + c_j i - c_i i - c_j j) \ge 0, \quad (i,j) \in P_2$$

Application of Farkas lemma in a similar way as above and elimination of the multipliers yields:

$$\begin{aligned} u_1 &\geq 0 \\ u_1 - c_i + c_j &\geq 0 \\ 3u_1 + w - c_i + c_j &\geq 0 \end{aligned} \tag{14}$$

Dependence 3: Due to symmetry with respect to i and j, the third dependence does not give anything more than the second one.

Finding the transformation. Aggregating legality and volume bounding constraints for all dependences, we obtain:

$$\begin{aligned} c_j &\geq 0 \\ w - c_j &\geq 0 \\ c_i - c_j &\geq 0 \\ u_1 &\geq 0 \\ u_1 - c_i + c_j &\geq 0 \\ 3u_1 + w - c_i + c_j &\geq 0 \\ \text{minimize}_{\prec}\ &(u_1, w, c_i, c_j) \end{aligned} \tag{15}$$

The lexicographic minimal solution for the vector $(u_1, w, c_i, c_j) = (0, 1, 1, 1)$[2]. Hence, we get $c_i = c_j = 1$. Note that $c_i = 1$ and $c_j = 0$ is not obtained even though it is a valid tiling hyperplane as it involves more communication: it requires u_1 to be positive.

The next solution is forced to have a positive component in the subspace orthogonal to $(1,1)$ given by (12) as (1,-1). This leads to the addition of the constraint $c_i - c_j \geq 1$ or $c_i - c_j \leq -1$ to the existing formulation. Adding $c_i - c_j \geq 1$ to (15), the lexicographic minimal solution is (1, 0, 1, 0), i.e., $u_1 = 1, w = 0, c_i = 1, c_j = 0$ ($u_1 = 0$ is no longer valid). Hence, $(1,1)$ and $(1,0)$ are the best tiling hyperplanes. (1,1) is used as space with one line of communication between processors, and the hyperplane (1,0) is used as time in a tile. The outer tile schedule is (2,1) (= (1,0) + (1,1)).

This transformation is in contrast to other approaches based on schedules which obtain a schedule and then the rest of the transformation matrix. Feautrier's greedy heuristic gives the schedule $\theta(i, j) = 2i + j - 3$ which carries all dependences. However, using this as either space or time does not lead to communication or locality optimization. The (2,1) hyperplane has non-constant communication along it. In fact, the only hyperplane that has constant communication along it is (1,1). This is the best hyperplane to be used as a space loop if the nest is to be parallelized, and is the first solution that our algorithm finds. The (1,0) hyperplane is used as time leading to a solution with one degree of pipelined parallelism with one line per tile of near-neighbor communication (along (1,1)) as shown in Fig. 4.1. Hence, a good schedule that tries to carry all dependences (or as many as possible) is not necessarily a good loop for the transformed iteration space.

[2] The zero vector is a trivial solution and is avoided

4.2 Example 2: Imperfectly Nested 1-d Jacobi

Consider the code in Figure 2(b). The affine dependences and the dependence polyhedra are as follows:

$$\begin{array}{rl}
(S_1, b[i]) \rightarrow (S_2, b[j]) & t = t' \wedge j = i \\
(S_2, b[j]) \rightarrow (S_1, b[i]) & t = t'+1 \wedge j = i \\
(S_2, a[j]) \rightarrow (S_1, a[i]) & t = t'+1 \wedge j = i \\
(S_1, a[i]) \rightarrow (S_2, a[j]) & t = t' \wedge j = i \\
(S_1, a[i+1]) \rightarrow (S_2, b[j]) & t = t' \wedge j = i+1 \\
(S_1, a[i-1]) \rightarrow (S_2, b[j]) & t = t' \wedge j = i-1 \\
(S_2, a[j]) \rightarrow (S_1, a[i+1]) & t = t'+1 \wedge j = i+1 \\
(S_2, a[j]) \rightarrow (S_1, a[i-1]) & t = t'+1 \wedge j = i-1
\end{array}$$

Our algorithm obtains $(c_t, c_i) = (1,0)$ with $c_0 = 0$, followed by $(c_t, c_i) = (2,1)$ with $c_0 = 1$, and $c'_t = c_t$ and $c'_i = c_i$. The solution is thus given by:

$$\phi_{s_1} = \begin{pmatrix} 1 & 0 \\ 2 & 1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} t \\ i \end{pmatrix} \qquad \phi_{s_2} = \begin{pmatrix} 1 & 0 \\ 2 & 1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} t' \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

Both iteration spaces have the same hyperplanes, with $(2,1)$ hyperplane of S_2 having a constant shift; the resulting transformation is equivalent to a constant shift of S2 relative to S1, fusion and skewing the i loop with respect to the t loop by a factor of 2. The (1,0) hyperplane has the least communication: no dependence crosses more than one hyperplane instance along it.

5 Implementation

We have implemented our transformation framework using PipLib 1.3.3 [Fea88] and Polylib 5.22.3 [pol]. Our tool takes as input dependence information (dependence polyhedra and h-transformations) from LooPo's [loo] dependence tester and generates statement-wise affine transformations. Flow, anti and output dependences are considered for legality as well as the minimization objective. The transforms generated by our tool are provided to CLooG [Bas04] as scattering functions. Fig. 7 shows the flow. The goal is to get tiled shared memory parallel code, for example, OpenMP code for multi-core architectures.

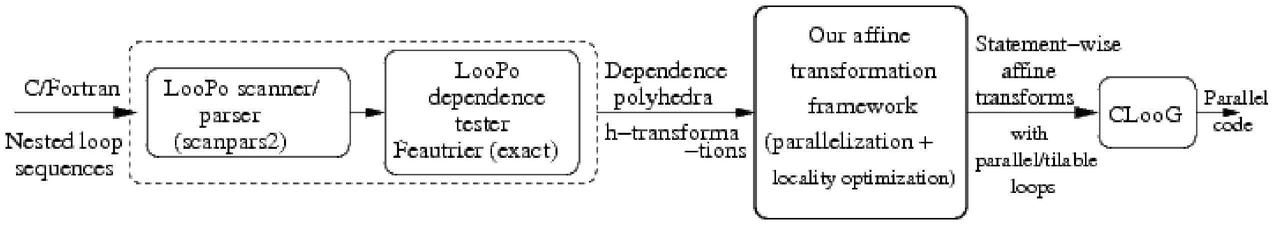
C/Fortran
Nested loop sequences
LooPo scanner/ parser (scanpars2)
LooPo dependence tester Feautrier (exact)
Dependence polyhedra
h−transforma −tions
Our affine transformation framework (parallelization + locality optimization)
Statement−wise affine transforms
with parallel/tilable loops
CLooG
Parallel code

Figure 7. Implementation status

The following are transformations obtained by our tool automatically from C/Fortran source code. The transformed code was generated using CLooG 0.14.0 [clo]. Note that for examples that follow, CLooG was run with options to not optimize control so that all statements are embedded into the innermost nest wherever possible; this is to show that the loops can be blocked in a straightforward fashion (since our framework finds permutable loops). This adds additional conditional guards which would affect performance; we do not intend to use these options for final tiled code generation. The original code is shown on the left while the transformed code with the transformation matrices is on the right. Note that the tiled code is not shown (but all loop nests are transformed to a tree of fully permutable nests). doall/forall indicates a fully parallel loop while doallpp/forallpp represents a pipelined parallel loop (a set of pipelined parallel space loops along with one time loop can enable pipelined parallelism).

5.1 2-d imperfectly nested jacobi

```
do t=1, n
  do i=2,n−1
    do j=2,n−1
      b[i,j] = a[i−1,j]+a[i,j]+a[i+1,j]
               + a[i,j−1]+a[i,j+1];
    end do
  end do
  do k=2,n−1
    do l=2,n−1
      a[k,l] = b[k,l];
    end do
  end do
end do
```

```
forallpp (c1=1;c1<=n;c1++){
  forallpp (c2=2∗c1+2;c2<=2∗c1+n;c2++){
    for (c3=max(c2−n+3,2∗c1+2);
                    c3<=min(c2+n−3,2∗c1+n);c3++){
      if  (c1 >= max(ceild(c3−n+1,2),ceild(c2−n+1,2))){
        S1(t=c1,  i=−2∗c1+c3, j=−2∗c1+c2);
      }
      if  (c1 <= min(floord(c2−3,2), floord(c3−3,2))){
        S2(t=c1,  i=−2∗c1+c3−1, j=−2∗c1+c2−1);
      }
    }
  }
}
```

		S1				S2	
t	*i*	*j*	*const*	*t*	*k*	*l*	*const*
1	0	0	0	1	0	0	0
2	1	0	0	2	1	0	1
2	0	1	0	2	0	1	1

Figure 8. Imperfectly nested 2-d Jacobi

Fig. 8 shows the code and the transformation. The transformation implies shifting the *i* and *j* loop of statement S2 by one iteration each, fusion with S1, skewing of the fused *i* and *j* loops with respect to the time loop by two. This allows tiling of all three loops and extraction of two degrees of pipelined parallelism.

5.2 LU decomposition

Fig. 9 shows the original and transformed code.

5.3 Sequence of Matrix-Matrix multiplies

For the sequence of matrix-matrix multiplies in Fig. 10, each of the original loop nests can be parallelized, but a synchronization is needed after the first loop nest is executed. The transformed loop nest has one outer

```
do k=1, n
  do j=k+1,n
    S1: a[k,j] = a[k,j]/a[k,k];
  end do
  do i=k+1,n
    do j=k+1,n
      S2: a[i,j] = a[i,j]
            − a[i,k]∗a[k,j];
    end do
  end do
end do
```

```
doallpp c1=1, n−1
  doallpp c2=c1+1, n
    S1(k = c1,j = c2)
    do c3=c1+1, n
      S2(k = c1,i = c3,j = c2)
    end do
  end do
end do
```

		$S1$		$S2$		
k	j	$const$	k	i	j	$const$
1	0	0	1	0	0	0
0	1	0	0	0	1	0
1	0	0	0	1	0	0

Figure 9. LU decomposition

parallel loop ($c1$), but reuse is improved as each element of matrix C is consumed immediately after it is produced (C can be contracted to a single scalar). This transformation is non-trivial and cannot be obtained from existing frameworks.

5.4 Multiple statement stencils

This code (Fig. 11) is representative of multimedia applications. The transformed code enables immediate reuse of data produced by each statement at the next statement.

5.5 TCE four-index transform

This is a sequence of four nested loops, each of depth five (Fig. 12), occurring in Tensor Contraction Expressions that appear in computational quantum chemistry problems [CS90]. Our tool transforms the code as shown where the producing/consuming distances between the loops have been reduced.

5.6 Experimental results on running time

Table 5.6 shows the running times of a preliminary implementation of our transformation framework for five different compute kernels - imperfectly-nested 2-d Jacobi, Haar's 1-d discrete wavelet transform, LU decomposition, TCE 4-index transform and swim kernel (from spec2000fp). Running times were measured on an Intel Core 2 Duo 2.4 GHz processor (2 MB L2 cache) running Linux kernel version 2.6.20. Results show that the tool with preliminary optimizations already runs very fast. The number of loops shown in the table is the sum of the number of outer loops of all statements in the original code. We plan an extensive study on performance achieved with codes generated by the developed framework - this will require additional work, including some adaptations to CLooG to generate OpenMP code for the loops we detect as parallel (with barrier's placed where necessary) and tile size determination and fusion structure selection through use of an empirical search engine. Given the current speed of the tool, we are confident that the model can be further refined to any degree if needed to enable transformations that work best.

```
do i = 1, n
  do j = 1, n
    do k = 1, n
      S1: C[i,j] = C[i,j] + A[i,k] * B[k,j]
    end do
  end do
end do

do i = 1, n
  do j = 1, n
    do k = 1, n
      S2: D[i,j] = D[i,j] + E[i,k] * C[k,j]
    end do
  end do
end do
```

```
doall c1 = 1, n
  do c2 = 1, n
    do c4 = 1, n
      S1(i=c1, j=c2, k=c4)
    end do
    do c4 = 1, n
      S2(i=c4, j=c2, k=c1)
    end do
  end do
end do
```

S1				*S2*			
i	*j*	*k*	*const*	*i*	*j*	*k*	*const*
1	0	0	0	0	0	1	0
0	1	0	0	0	1	0	0
0	0	0	0	0	0	0	1
0	0	1	0	1	0	0	0

Figure 10. Sequence of MMs

Code	Num of statements	Num of loops	Num of deps	Running time
2-d Jacobi	2	6	20	0.05s
Haar 1-d	3	5	12	0.018s
LU	2	5	10	0.022s
TCE 4-index	4	20	15	0.20s
Swim	58	110	639	20.9s

Table 1. Transformation tool running time (preliminary)

6 Related work

Iteration space tiling [IT88, Wol89, WL91, RS92] is a standard approach for aggregating a set of loop iterations into tiles, with each tile being executed atomically. In addition, researchers have considered the problem of selecting tile shape and size to minimize communication, improve locality or minimize finish time [SD90, RS92, BDRR94, Xue97, HCF97, HCF99, HS02, RR04]. These works are restricted to single perfectly nested loops with uniform dependences. Some of these works (especially those in the nineties) [SD90, RS92, BDRR94, Xue97, HCF97] gave key insights into the notion of partitioning and its implication on parallel performance. However, such input are *toy examples* by current standards given the need to optimize real world code.

Loop parallelization has been studied extensively. The reader is referred to [BDSV98] for a detailed survey of older parallelization algorithms which accepted restricted input and/or are based on weaker dependence abstractions outside of the polyhedral model. Kelly and Pugh's algorithm finds one dimension of parallelism for programs with arbitrary nesting and sequences of loops [KP93, KP96]. Their program transforms include loop permutations and reversals, but not loop skewing. The exclusion of loop skewing enables them to enumerate all the possible transformation choices and select the best one based on communication cost.

Scheduling with affine functions using faces of the polytope by application of the Farkas algorithm was

```
do i = 2, n-1
  a1[i] = a0[i-1] + a0[i] + a0[i+1];
end do
do i = 2, n-1
  a2[i] = a1[i-1] + a1[i] + a1[i+1];
end do
do i = 2, n-1
  a3[i] = a2[i-1] + a2[i] + a2[i+1];
end do
do i = 2, n-1
  a4[i] = a3[i-1] + a3[i] + a3[i+1];
end do
do i = 2, n-1
  a5[i] = a4[i-1] + a4[i] + a4[i+1];
end do
```

```
do c1=2, n+3
  if (c1 <= n-1) then
    S1(i = c1)
  end if
  if ((c1 >= 3) .and. (c1 <= n)) then
    S2(i = c1-1)
  end if
  if ((c1 >= 4) .and. (c1 <= n+1)) then
    S3(i = c1-2)
  end if
  if ((c1 >= 5) .and. (c1 <= n+2)) then
    S4(i = c1-3)
  end if
  if (c1 >= 6) then
    S5(i = c1-4)
  end if
end do
```

$S1$		$S2$		$S3$		$S4$		$S5$	
i	$const$	i	$const$	i	$const$	i	$const$	i	$const$
1	0	1	1	1	2	1	3	1	4

Figure 11. Multi statement stencil

first proposed by Feautrier [Fea92a]. Feautrier explored various possible approaches to obtain good affine schedules that minimize latency. The one-dimensional schedules (wherever they can be found) carry all dependences and so all the inner loops are parallel. Using reasonable heuristics usually yields good solutions. However, transforming to permutable loops that are amenable to tiling or detecting outer parallel loops is not addressed. As discussed and shown in Sec. 3, using this schedule as one of the loops for parallel code generation does not necessarily optimize communication or locality. Hence, schedules need not be good hyperplanes for tiling. Several works [Gri04, CGP$^+$05, PBCV07] make use of such schedules. Though this approach yields maximal inner parallelism, tiling the time loop is not possible unless communication in the space loops is in the forward direction (dependences have positive components along all dimensions). Enforcing (6) for each dependence while finding hyperplanes enables detection of fully permutable loop nest sets as well as allows us to put the minimization objective for synchronization and reuse distances. Our algorithm ends up finding inner parallel loops only if permutable loops are not available. Overall, Feautrier's works [Fea92a, Fea92b] are geared towards finding minimum latency schedules and maximum fine-grained parallelism as opposed to tilability for coarse-grained parallelization with minimized communication and better locality.

Lim and Lam [LL98, LCL99] use the same exact dependence model as us and propose an affine framework that identifies outer parallel loops (communication-free space partitions) and permutable loops (pipelined parallel or tilable loops) with the goal of minimizing the order of synchronization. They employ the same machinery for blocking [LLL01]. Several (infinitely many) solutions equivalent in terms of the criterion they optimize for result from their algorithm, and these significantly differ in communication cost; no metric is provided to differentiate between these solutions. Also, tiling for locality is not handled in an integrated way with parallelization. Fusion across a sequence of weakly connected components to optimize a sequence of producer/consumer loops is not addressed. The structure of the transformation matrices in the presence of statements with different dimensionalities, partially fused loops and the associated code generation is not discussed. Our solution addresses all of these aspects.

Ahmed et al. [AMP01] propose a framework for data locality optimization of imperfectly nested loops for sequential execution. The approach determines the embedding for each statement into a product space, which is then considered for locality optimization through another transformation matrix. Their framework was among the first to address tiling of imperfectly nested loops. However, the technique used for minimizing reuse distances is based on a rather ad-hoc heuristic that does not guarantee good solutions or even a solution. The reuse distances in the target space for some dependences are set to zero with the goal of obtaining solutions to the embedding function/transformation matrix coefficients. However, the choice of the dependences and the number (which is crucial) is determined heuristically. Moreover, setting some reuse classes to zero (or a constant) need not completely determine the embedding function or transformation matrix coefficients. Exploring all possibilities is infeasible. Overall, the automatability of the approach is unclear from the description.

Griebl [Gri04] presents an integrated framework for optimizing data locality and parallelism with space and time tiling. Though Griebl's approach enables time tiling by using a forward communication-only placement with an existing schedule, it does not necessarily lead to communication/locality-optimized solutions. This is mainly due to tiling being modeled and enabled as a post-processing as opposed to being integrated into a transformation framework. Also, loop fusion is not addressed. Overall, as described earlier (Sec. 3), using schedules as time loops (even with some post-processing) is not best for coarse-grained parallelization.

Cohen et al., Girbal et al. [CGP$^+$05, GVB$^+$06] proposed and developed a framework (URUK/WRAP-IT) to compose sequences of transformations in a semi-automatic fashion. Transformations are manually specified by an expert and are applied automatically. Pouchet et al. [PBCV07] searches the space of transformations to find good ones through iterative optimization by employing performance counters. This is because the space of valid transformations for loop nests is very large. Our approach automatically and directly obtains good transformations without search. However, in many cases empirical and iterative optimization is required to choose transforms that work best in practice. This is true, for example, when we need to choose among several different fusion structures and our algorithm cannot differentiate between them, or when there is a trade-off between fusion and parallelization (Sec. 3.6). Also, effective determination of tile sizes and unroll factors for transformed whole-programs may only be possible through empirical search through performance measurement. A combination of our affine transformation algorithm and empirical search in a smaller space is an interesting approach to pursue. Alternatively, more powerful cost models like computing Ehrhart polynomials [VSB$^+$04] can be employed once solutions in a smaller space can be enumerated.

7 Conclusions

We have presented a single affine transformation framework that can optimize imperfectly nested loop sequences for parallelism and locality simultaneously. Our framework is also more advanced than previous frameworks on each of the two complementary aspects of coarse-grained parallelization and locality. The approach also enables fusion in the presence of producing-consuming loops. The framework has been implemented into a tool to perform transformations in a fully automatic way from C/Fortran code.

Acknowledgments

We would like to thank Martin Griebl and team (FMI, Universität Passau, Germany) for the LooPo infrastructure. We would also like to thank Cédric Bastoul (Paris-Sud XI University) and all other contributors for CLooG. This work is supported in part by the U.S. National Science Foundation through grants 0121676, 0121706, 0403342, 0509442, and 0509467.

References

[AMP01] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. *International Journal of Parallel Programming*, 29(5), October 2001.

[Bas04] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *IEEE Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, September 2004.

[BDRR94] Pierre Boulet, Alain Darte, Tanguy Risset, and Yves Robert. (Pen)-ultimate tiling? *Integration, the VLSI Journal*, 17(1):33–51, 1994.

[BDSV98] Pierre Boulet, Alain Darte, Georges-André Silber, and Frédéric Vivien. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Computing*, 24(3–4):421–444, 1998.

[CGP$^+$05] Albert Cohen, Sylvain Girbal, David Parello, M. Sigler, Olivier Temam, and Nicolas Vasilache. Facilitating the search for compositions of program transformations. In *ACM ICS*, pages 151–160, June 2005.

[clo] CLooG: The Chunky Loop Generator. http://www.cloog.org.

[CS90] Lawrence A. Covick and Kenneth M. Sando. Four-index transformation on distributed-memory parallel computers. *Journal of Computational Chemistry*, pages 1151 – 1159, November 1990.

[DV97] Alain Darte and Frederic Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *Int. J. Parallel Programming*, 25(6):447–496, December 1997.

[Fea88] P. Feautrier. Parametric integer programming. *Operationnelle/Operations Research*, 22(3):243–268, 1988.

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, 1992.

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem. part II. multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[Fea96] Paul Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, pages 79–103, 1996.

[GLW98] Martin Griebl, Christian Lengauer, and S. Wetzel. Code generation in the polytope model. In *IEEE PACT*, pages 106–111, 1998.

[Gri04] Martin Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. FMI, University of Passau, 2004. Habilitation Thesis.

[GVB$^+$06] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Intl J. of Parallel Programming*, 34(3):261–317, June 2006.

[HCF97] K. Högstedt, Larry Carter, and Jeanne Ferrante. Determining the idle time of a tiling. In *Proceedings of PoPL '97*, pages 160–173, 1997.

[HCF99] Karin Hogstedt, Larry Carter, and Jeanne Ferrante. Selecting tile shape for minimal execution time. In *SPAA*, pages 201–211, 1999.

[HS02] Edin Hodzic and Weijia Shang. On time optimal supernode shape. *IEEE Trans. Par. & Dist. Sys.*, 13(12):1220–1233, 2002.

[IT88] F. Irigoin and R. Triolet. Supernode partitioning. In *PoPL*, pages 319–329, 1988.

[KBB$^+$07] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective Automatic Parallelization of Stencil Computations. In *ACM SIGPLAN PLDI 2007*, July 2007.

[Kel96] Wayne Kelly. Optimization within a unified transformation framework. Technical Report CS-TR-3725, 1996.

[KP93] W. Kelly and W. Pugh. Determining schedules based on performance estimation. Technical Report UMIACS-TR-9367, University of Maryland, College Park, December 1993.

[KP96] Wayne Kelly and William Pugh. Minimizing communication while preserving parallelism. In *ICS*, pages 52–60, 1996.

[KPR95] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *FRONTIERS '95: Proceedings of Symposium on the Frontiers of Massively Parallel Computation*, page 332, 1995.

[LCL99] Amy W. Lim, Gerald I. Cheong, and Monica S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *ACM ICS*, pages 228–237, 1999.

[LL98] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3-4):445–475, 1998. Extended version of PoPL'97 paper.

[LLL01] A. Lim, S. Liao, and M. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *ACM SIGPLAN PPoPP*, pages 103–112, 2001.

[loo] LooPo - Loop parallelization in the polytope model. http://www.fmi.uni-passau.de/loopo.

[PBCV07] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *ACM CGO*, March 2007.

[pol] PolyLib - A library of polyhedral functions.
http://icps.u-strasbg.fr/polylib/.

[Pug92] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, August 1992.

[QRW00] Fabien Quilleré, Sanjay V. Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *Intl J. of Parallel Programming*, 28(5):469–498, 2000.

[RR04] L. Renganarayana and Sanjay Rajopadhye. A geometric programming framework for optimal multi-level tiling. In *SC*, 2004.

[RS92] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–230, 1992.

[Sch87] Alexander Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1987. SchRI a 87:1 1.Ex.

[SD90] R. Schreiber and J. Dongarra. Automatic blocking of nested loops. Technical report, University of Tennessee, Knoxville, TN, August 1990.

[VBC06] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. Polyhedral code generation in the real world. In *International Conference on Compiler Construction (ETAPS CC'06)*, pages 185–201, March 2006.

[VBGC06] Nicolas Vasilache, Cédric Bastoul, Sylvain Girbal, and Albert Cohen. Violated dependence analysis. In *ACM ICS*, June 2006.

[VSB$^+$04] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Analytical computation of ehrhart polynomials: enabling more compiler analyses and optimizations. In *CASES*, pages 248–258, September 2004.

[WL91] Michael Wolf and Monica S. Lam. A data locality optimizing algorithm. In *PLDI '91*, pages 30–44, 1991.

[Wol89] M. Wolf. More iteration space tiling. In *Proceedings of Supercomputing '89*, pages 655–664, 1989.

[Xue97] Jingling Xue. Communication-minimal tiling of uniform dependence loops. *Journal of Parallel and Distributed Computing*, 42(1):42–59, 1997.

```
do a = 1, N
  do q = 1, N
    do r = 1, N
      do s = 1, N
        do p = 1, N
          T1[a,q,r,s] = T1[a,q,r,s]
                  + A[p,q,r,s]*C4[p,a]
        end do
      end do
    end do
  end do
end do

do a = 1, N
  do b = 1, N
    do r = 1, N
      do s = 1, N
        do q = 1, N
          T2[a,b,r,s] = T2[a,b,r,s]
                + T1[a,q,r,s]*C3[q,b]
        end do
      end do
    end do
  end do
end do

do a = 1, N
  do b = 1, N
    do c = 1, N
      do s = 1, N
        do r = 1, N
          T3[a,b,c,s] = T3[a,b,c,s]
                + T2[a,b,r,s]*C2[r,c]
        end do
      end do
    end do
  end do
end do

do a = 1, N
  do b = 1, N
    do c = 1, N
      do d = 1, N
        do s = 1, N
          B[a,b,c,d] = B[a,b,c,d] + T3[a,b,c,s]*C1[s,d]
        end do
      end do
    end do
  end do
end do
```

(a) Original

```
doall c1=1, N
  do c2=1, N
    doall c4=1, N
      doall c5=1, N
        do c7=1, N
          S1(i = c1,j = c5,k = c4,l = c2,m = c7)
        end do
        do c7=1, N
          S2(i = c1,j = c7,k = c4,l = c2,m = c5)
        end do
      end do
    end do
    doall c4=1, N
      doall c5=1, N
        do c7=1, N
          S3(i = c1,j = c5,k = c4,l = c2,m = c7)
        end do
        do c7=1, N
          S4(i = c1,j = c5,k = c4,l = c7,m = c2)
        end do
      end do
    end do
  end do
end do
```

(b) Transformed code

Figure 12. TCE 4-index transform