Affine Transformations for Communication Minimized 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

[2]Dept. of Electrical & Comp Engg.
Louisiana State University

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 [17] 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 [14, 36] and code generation [26, 20] 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 [43, 37, 4, 42] 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 [8, 18]. 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 [13, 14, 15, 16] have led to many research efforts on automatic parallelization in the polyhedral model. Existing automatic transformation frameworks [32, 31, 30, 3, 19] 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 [19], 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: (1) What is a good way to tile imperfectly nested loops for minimized communication in the processor space as well as improved locality at each processor? (2) 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 function 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 6 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 7 discusses related work and conclusions are presented in Section 8.

2 Background and Notation

In this section, we present a overview of the polyhedral model, and introduce notation used throughout the paper.

The set X of all vectors $x \in \mathbf{Z}^n$ such that $\vec{h}.\vec{x} = k$, for $k \in \mathbf{Q}$, forms an (affine) *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 $h.x_1 = h.x_2$. The set of all vectors $\mathbf{x} \in \mathbf{Z^n}$ such that $A\mathbf{x} + \mathbf{b} \geq 0$, where A is a constant matrix and $\mathbf{b}$ is a constant vector, defines a (convex) polyhedron. A polytope is a bounded polyhedron.

Each run-time instance of a statement S, in a program, is identified 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 program parameters (typically, symbolic constants representing the problem side). With conservative assumptions, input programs not satisfying these criteria can also be handled in the polyhedral model – techniques for such conservative approximations are not part of this work.

A well-known known result useful in the context of the polyhedral 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:

$$\psi(x) \equiv \lambda_0 + \sum_k \lambda_k(a_k x + b_k), \;\; \lambda_k \geq 0 \tag{1}$$

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 [41].

2.1 Polyhedral dependences

Our dependence model is of exact affine dependences and same as the one used in [8, 31, 35, 43]. The input need not be in single-assignment form. All dependences – flow, anti (write-after-read), output (write-after-write) and input (read-after-read) dependences are considered. The Data Dependence Graph (DDG) is a directed multi-graph with each vertex representing a statement, and an edge, $e \in E$, from node S_i to S_j representing a polyhedral dependence from a dynamic instance of S_i to one of S_j: it is characterized by a polyhedron, $\mathcal{P}_e$, called the *dependence polyhedron* that captures the exact dependence information corresponding to edge, e (see Fig. 1(b) for an example). The dependence polyhedron is in the sum of the dimensionalities of the source and target statement's polyhedra (with dimensions for program parameters as well). Though the equalities in $\mathcal{P}_e$ typically represent the affine function mapping the target iteration vector $\vec{t}$ to the particular source $\vec{s}$ that is the last access to the conflicting memory location, also known as the *h-transformation* [15]; the last access condition is not necessary; in general, the equalities can be used to eliminate variables from $\mathcal{P}_e$. In the rest of this section, we assume for convenience that $\vec{s}$ can be completely eliminated using h_e, being substituted by $h_e(\vec{t})$.

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    S1: A[i,j] = A[i,j]+u[i]*v[j];
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    S2: x[i] = x[i]+A[j,i]*y[j];
```

(a) original code

$$\mathcal{P}_{e_1}: \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & -1 \\ \hline 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ i' \\ j' \\ N \\ 1 \end{bmatrix} \begin{array}{l} \geq 0 \\ \geq 0 \\ \geq 0 \\ \geq 0 \\ \hline = 0 \\ = 0 \end{array}$$

(b) Dependence polyhedron for the inter-statement dependence on A

	S1			S2			
	i	j	$const$	i	j	$const$	
c_1	0	1	0	1	0	0	doall
c_2	1	0	0	0	1	0	doacross
c_3	0	0	0	0	0	1	scalar

(c) transformation

```
for (c1=0; c1<N; c1++)
  for (c2=0; c2<N; c2++)
    A[c2,c1] = A[c2,c1]+u[c2]*v[c1];
    x[c1] = x[c1]+A[c2,c1]*y[c1];
```

(d) transformed code

Figure 1. Polyhedral transformation and dependences

2.2 Transformations

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

$$\phi_{S_k} = \begin{bmatrix} c_1 & c_2 & \dots & c_{m_{S_k}} \end{bmatrix} \left(\vec{i} \right) + c_0 \tag{2}$$

$$= \begin{bmatrix} c_1 & c_2 & \dots & c_{m_{S_k}} & c_0 \end{bmatrix} \begin{pmatrix} \vec{i} \\ 1 \end{pmatrix} \tag{3}$$

A multi-dimensional affine transformation for a statement can now be represented by a matrix with each row being an affine hyperplane/transform. If such a transformation matrix has full column rank, it is a one-to-one mapping from the original iteration space to a target iteration space. Once the parallel loops are marked, the transformation completely specifies when and where an iteration executes. Hence, 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. Fig 1 shows a transformation. These transformations can capture a sequence of simpler transformations that include permutation, skewing, reversal, fusion, fission, and relative shifting.

The above representation for transformations is similar to that used by many researchers: earlier in [16, 27], and more recently in a systematic way [8, 18] and directly fits with scattering functions that a code generation tool like CLooG [4, 1] supports. On providing such a representation, the target code shown can be generated by scanning the statement polyhedra in the global lexicographic ordering with respect to 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

3.1 Legality of tiling imperfectly-nested loops

Theorem 1 *Let ϕ_{s_i} be a one-dimensional affine transform for statement S_i. For $\{\phi_{s_1}, \phi_{s_2}, \ldots, \phi_{s_k}\}$, to be a legal (statement-wise) tiling hyperplane, the following should hold for each edge e from S_i and S_j:*

$$\phi_{s_j}\left(\vec{t}\right) - \phi_{s_i}\left(\vec{s}\right) \quad \geq \quad 0, \quad \mathcal{P}_e \tag{4}$$

Proof. Tiling of a statement's iteration space 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. Let $\{\phi^1_{s_1}, \phi^1_{s_2}, \ldots, \phi^1_{s_k}\}$, $\{\phi^2_{s_1}, \phi^2_{s_2}, \ldots, \phi^2_{s_k}\}$ be two statement-wise 1-d affine transforms that satisfy (4). Consider a tile formed by aggregating a group of hyperplane instances along $\phi^1_{s_i}$ and $\phi^2_{s_i}$. Due to (4), for any dynamic dependence, the target iteration is mapped to the same hyperplane or a greater hyperplane than the source, i.e., 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 (ϕ^1 and ϕ^2 in this case). Similarly, all iterations outside of a tile influencing it are either in that tile or in the backward direction along one or more of the hyperplanes. The above argument holds true for both intra- and inter-statement dependences. For inter-statement dependences, this leads to an interleaved execution of tiles of iteration spaces of each statement when code is generated from these mappings. Hence, $\{\phi^1_{s_1}, \phi^1_{s_2}, \ldots, \phi^1_{s_k}\}$, $\{\phi^2_{s_1}, \phi^2_{s_2}, \ldots, \phi^2_{s_k}\}$ represent rectangularly tilable loops in the transformed space. If such a tile is executed on a processor, communication would be needed only before and after its execution. From locality point of view, if such a tile is executed with the associated data fitting in a faster memory, reuse is exploited in multiple directions.□

The above condition was well-known for the case of a single-statement perfectly nested loops from the work of Irigoin and Triolet [24] (as $h^T.R \geq \mathbf{0}$). We have generalized it above for multiple iteration spaces with exact affine dependences with possibly different dimensionalities and imperfect nestings for statements.

Tiling at an arbitrary depth. Note that the legality condition as written in (4) is imposed on all dependences. However, if it is imposed only on dependences that have not been carried up to a certain depth, the independent ϕ's that satisfy the condition represent tiling hyperplanes at that depth, i.e., rectangular blocking (stripmine/interchange) at that level in the transformed program is legal.

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 one. We first describe solutions obtained by existing state of the art approaches - Lim and Lam's affine partitioning [32, 31] and Griebl's space and time tiling with FCO placement [19]. 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 4, 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

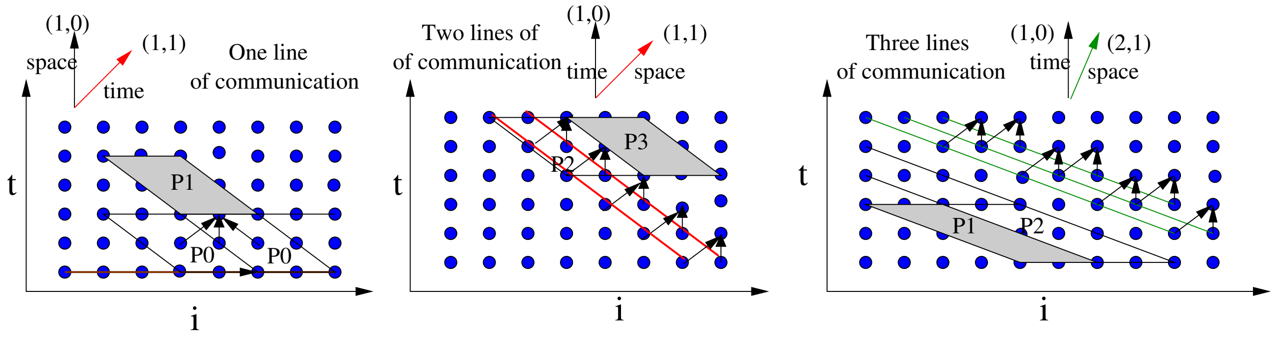

Figure 3. Communication volume with different valid hyperplanes for 1-d jacobi

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 [40, 38, 48] 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 [32] finds two valid independent solutions without optimizing for any particular criterion. In particular, the solutions found by their algorithm (Algorithm A in [32]) 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.

3.2 Cost function

Consider the affine form defined as:

$$\delta_e(\vec{t}) \quad = \quad \phi_{s_i}(\vec{t}) - \phi_{s_j}(h_e(\vec{t})), \quad \vec{t} \in \mathcal{P}_e \tag{5}$$

The affine form, $\delta_e(\vec{t})$, 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

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

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.

3.3 Challenges

The constraints obtained from Eqn 4 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 [15, 16, 32, 19, 8, 35] 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 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 [41]. However, an attempt to minimize communication volume ends up in an objective function involving both loop variables and hyperplane coefficients. For example, $\phi(\vec{t})-\phi(h_e(\vec{t}))$ 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.

3.4 Cost Function Bounding and Minimization

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

Lemma 2 *If all iteration spaces are bounded, there exists at least one affine form v in the structure parameters $\vec{p}$, that bounds $\delta_e(\vec{t})$ for every dependence edge e, i.e., there exists*

$$v(\vec{p}) = u.\vec{p} + w \tag{6}$$

such that

$$\begin{aligned} v(\vec{p}) \ - \ \left(\phi_{s_i}(\vec{t}) - \phi_{s_j}(h_e(\vec{t}))\right) \ &\geq \ 0, \quad \vec{t} \in \mathcal{P}_e, \forall e \in E \\ v(\vec{p}) \ - \ \delta_e(\vec{t}) \ &\geq \ 0, \quad \vec{t} \in \mathcal{P}_e, \forall e \in E \end{aligned} \tag{7}$$

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 $v(\vec{p}) \geq \delta_e(\vec{p}) \geq 0$, v should either increase or stay constant with an increase in the structural parameters, i.e., the coordinates of u are non-negative. 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 (7).

$$v(\vec{p}) - \delta_e(\vec{t}) \ \equiv \ \lambda_{e0} + \sum_{k=1}^{m_e} \lambda_{ek} \left(c_{ek} \begin{pmatrix} \vec{i} \\ \vec{p} \end{pmatrix} + d_{ek} \right) \tag{8}$$

The above is an identity and the coefficients of each of the loop indices in $\vec{i}$ and parameters in $\vec{p}$ 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.

$$\text{minimize}_{\prec} \ \{u_1, u_2, \ldots, u_k, w, \ldots, c_i's, \ldots\} \tag{9}$$

Finding the lexicographic minimal solution is within the reach of the simplex algorithm and can be handled by the PIP software [13]. 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 (7) 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 $w \geq \delta_e$.

3.5 Iteratively Finding Independent Solutions

Solving the ILP formulation in the previous section 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 augment the ILP formulation with new constraints and obtain the next solution; the new constraints ensure linear independence with solutions already found. Let the rows of H_S represent the solutions found so far for a statement S. Then, the sub-space orthogonal to H_S [29, 34] is given by:

$$H_S^{\perp} = I - H_S^T \left(H_S H_S^T\right)^{-1} H_S \tag{10}$$

Note that $H_S^{\perp}.{H_S}^T = \mathbf{0}$, i.e., the rows of H_S are orthogonal to those of $H_S^{\perp}$. Let h_S^* be the next row (linear portion of the hyperplane) to be found for statement S. Let ${H^i}_S^{\perp}$ be a row of $H_S^{\perp}$. Then, any *one* of the inequalities given by $\forall i,\ {H^i}_S^{\perp}.\vec{h_S^*} > 0, {H^i}_S^{\perp}.\vec{h_S^*} < 0$ gives the necessary constraint to be added for statement S to ensure that h_S^* has a non-zero component in the sub-space orthogonal to H_S. This leads to a non-convex space, and ideally, all cases have to be tried and the best among those kept. When the number of statements is large, this leads to a combinatorial explosion. In such cases, we restrict ourselves to the sub-space of the orthogonal space where all the constraints are positive, i.e., the following constraints are added to the ILP formulation for linear independence:

$$\forall i, {H^i}_S^{\perp}.{h^*}_S \geq 0 \quad \wedge \quad \sum_i {H^i}_S^{\perp} h_S^* \geq 1 \tag{11}$$

By just considering a particular convex portion of the orthogonal sub-space, we discard solutions that usually involve loop reversals or combination of reversals with other transformations; however, we believe this does not make a difference in practice. 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. Hence, no more orthogonality constraints need be added for statements for which enough independent solutions have been found (the rest of the rows get automatically filled with zeros or linearly dependent rows). 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 code is the same as that of the source loop nest. Overall, a hierarchy of fully permutable loop nest sets is 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.

3.6 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. (9) 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).

3.7 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 schedule to tiles for parallel code generation [5].

3.8 Fusion in the affine transformation 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:

$$(c_i, c_j, c'_i, c'_j) = (1, 0, 0, 1)$$

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 one after the other as shown in Figure 4(b). This generalization of fusion is same as the one proposed in [8, 18]. We show that this naturally integrates into our automatic transformation approach.

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    S1: x[i] = x[i]+a[i,j]*y[j]


for (i'=0; i'<N; i'++)
  for (j'=0; j'<N; j'++)
    S2: y[i'] = y[i'] + a[i',j']*x[j']
```

(a) Original

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
     S1: x[i] = x[i]+a[i,j]*y[j]
  }
  for (i'=0; i'<N; i'++) {
     S2: y[i'] = y[i']+a[i',i]*x[i]
  }
}
```

(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

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. In most cases, we find it to be automatically enabled in combination with a permutation (as in Figure 4) or a constant shift (shown for imperfectly nested Jacobi later). Note that it is important to leave the structure parameter $\vec{p}$ out of our affine transform definition in 3 for the above to hold true. Hence, a common tiling hyperplane also represents a fused loop, and reuse distances between components that are weakly connected can be reduced with our cost function. The set of valid independent hyperplanes that can be iteratively found from our algorithm for multiple statements (at a given depth) is the maximum number of loops that can be fused at that depth.

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. If no valid solution exists, there is no common loop for all the statements. The number of solutions found gives the number of fusable 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 legality 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 ($\phi_{s_i} = (0, 0, \ldots, 0)$), and those after to one ($\phi_{s_i} = (0, 0, \ldots, 1)$. We call such a special row of the transformation matrix a *splitter*. The process is repeated recursively till as many independent solutions are found for the deepest statement as its nesting depth and all dependences are carried. Rows of the transformation matrix are in a suitable form to be expressed as scattering functions to a code generation tool like CLooG [1, 4]. For example, for the sequence of matrix-vector multiplies, the corresponding transformation matrices are shown in Fig. 4.

3.9 Summary

The algorithm is summarized below. Our approach can be viewed as transforming to a tree of permutable loop nests sets/bands - each node of the tree is a good permutable loop nest set. Step 12 of the repeat-until block in Algorithm 1 finds such a band of permutable loops. If all loops are tilable, there is just one node containing all the loops that are permutable. On the other extreme, if no loops are tilable, each node of the tree has just one loop and so no tiling is possible. At least two hyperplanes should be found at any level (without dependence removal/cutting) to enable tiling. Dependences from previously found solutions are thus not removed unless they have to be (Step 17): to allow the next permutable band to be found, and so on. Hence, partially tilable or untilable input is all handled. Loops in each node of the target tree can be stripmined/interchanged when there are at least two of them in it; however, it is illegal to move a stripmined loop across different levels in the tree.

Choice of dependences to cut. The algorithm as described is geared towards maximal fusion, i.e., dependences are cut at the deepest level (as a last resort). However, it does not restrict how the following decisions is made – which dependences between SCCs to cut when

Algorithm 1 Overview of algorithm

Input Generalized dependence graph $G = (V, E)$ (includes dependence polyhedra $\mathcal{P}_e$, $e \in E$)

```
1: $S_{max}$: statement with maximum domain dimensionality
2: **for** each dependence $e \in E$ **do**
3:     Build legality constraints: apply Farkas Lemma on $\phi(\vec{t}) - \phi(h_e(\vec{t})) \geq 0$ under $\vec{t} \in \mathcal{P}_e$,
       and eliminate all Farkas multipliers
4:     Build communication volume/reuse distance bounding constraints: apply Farkas
       Lemma to $v(\vec{p}) - (\phi(\vec{t}) - \phi(f(\vec{t}))) \geq 0$ under $\vec{t} \in \mathcal{P}_e$, and eliminate all Farkas multipliers
5:     Aggregate constraints from both into $C_e(i)$
6: **end for**
7: **repeat**
8:     $C = \emptyset$
9:     **for** each dependence edge $e \in E$ **do**
10:        $C \leftarrow C \cup C_e(i)$
11:    **end for**
12:    Compute lexicographic minimal solution with $u's$ coefficients in the leading position
       followed by $w$ to iteratively find independent solutions to $C$ (orthogonality constraints
       are added as each soln is found)
13:    **if** no solutions were found **then**
14:        Cut dependences between two strongly-connected components in the GDG and insert
           the appropriate *splitter* in the transformation matrices of the statements
15:    **end if**
16:    Compute $E_c$: dependences carried by solutions of Step 12/14
17:    $E \leftarrow E - E_c$; reform the GDG $(V, E)$
18: **until** $H^{\perp}_{S_{max}} = \mathbf{0}$ and $E = \emptyset$
```

Output A transformation matrix for each statement (with the same number of rows) fused loops are not found (Step 14)?. Aggressive fusion may kill parallelism and increases the running time of the transformation framework (in the presence of large strongly-connected components), while cutting too early may give up reuse. We plan to tune our algorithm to optimize for this trade-off between fusion and parallelization in future. Our notion of a cut is equivalent to introducing a parametric shift to separate loops. Loop shifting was used for parallelization and fusion with a simplified representation of dependences and transformations by Darte et al. [10, 11] and more recently for correcting illegal loop transformations by Vasilache et al. [44]. Example 5.5 explains through an example how more sophisticated transformations can be enabled than with techniques based purely on loop shifting.

3.10 Accuracy of cost function 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 bounding functions 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 tuned while keeping the problem linear.

3.11 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.

3.12 Correctness and Completeness

Theorem 2 *A transformation is always found by Algorithm 1.*

Proof. We show that the termination condition (Step 18) of Algorithm 1 is always reached. Firstly, every strongly-connected component of the dependence graph has at least one common surrounding loop, if the input comes from a valid computation. Hence, Step 12 is guaranteed to find at least one solution when all dependences between strongly-connected components have eventually been cut (iteratively in Step 14 whenever solutions are not found). Hence, enough linearly independent solutions are found for each statement such that $H_S^{\perp}$ eventually becomes $\mathbf{0}$ for every $S \in V$, i.e., H_S becomes full-ranked for each statement. Now, we show that the condition $E = \emptyset$ is also eventually satisfied. Let us consider the two groups of dependences: (1) self-edges (or intra-statement dependences), and (2) inter-statement dependences. Since H_S becomes full-ranked and does not have a null space, all dependent iterations comprising a self-edge are satisfied at one level or the other (since $\phi(\vec{t}) - \phi(\vec{s}) \geq 0$ stays in the formulation till satisfaction). Now, consider an inter-statement dependence from S_i to S_j. If at Step 14, the dependences between S_i and S_j were cut, all uncarried dependences between S_i and S_j will immediately be carried by the *splitter* row introduced in the transformation matrices (since ϕ_{S_j} is set to the scalar one and ϕ_{S_i} is set to zero). However, if S_i and S_j belong to the same strongly-connected component, then a solution will be found at Step 12, and eventually they will belong to separate strongly-connected components and dependences between them will be cut (if not satisfied). Hence, both intra and inter-statement dependences are eventually carried, and the condition $E = \emptyset$ is met.□

Theorem 3 *The transformation found by Algorithm 1 is always legal.*

Proof. Given the proof for Theorem 2, the proof for legality is straightforward. Since we keep $\phi(\vec{t}) - \phi(\vec{s}) \geq 0$ in the formulation till satisfaction, no dependence is violated. The termination condition for the repeat-until block thus ensures that all dependences are carried. Hence, the transformations found are always legal.□

3.13 Bound on iterative search in Algorithm 1

Each iteration of the repeat-until block in Algorithm 1 incurs a call to PIP. The number of times this block executes depends on how dependences across strongly-connected components are handled in Step 14. Consider one extreme case when all dependences between any two strongly-connected components are cut whenever no solutions are found: then, the number of PIP calls required is $2d + 1$ at worst, where d is the depth of the statement with maximum dimensionality. This is because, in the worst case, exactly one solution is found at Step 12, and the rest of the d times dependences between all SCCs are cut (both happen in an alternating fashion); the last iteration of the block adds a splitter that specifies the ordering of the statements in the innermost loop(s). Now, consider the other extreme case, when dependences are cut very conservatively between SCCs; in the worst case, this would increase the number of iterations of the block by the number of dependences in the program. The running times shown in Fig. 6.1 correspond to a very conservative cutting scheme. Even with such a scheme, for the largest input (swim kernel) with 58 statements and 600 odd dependences, we find that 50 iterations of the repeat-until block were executed and the running time is a few tens of seconds.

4 Examples

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

4.1 Example 1: Non-constant dependences

Figure 5 shows an example from the literature [12] 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
```

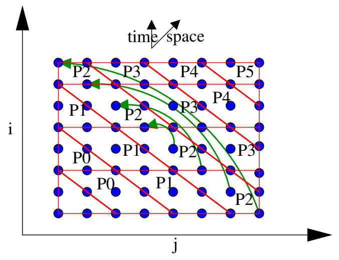

Figure 5. Example 1: Non-constant dependences

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

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:

$$w - c_j \geq 0$$

Dependence 2: Tiling legality constraint:

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

Applying Farkas lemma, we have:

$$\begin{aligned}
(c_i - c_j)i \;\; &+ \;\; (c_j - c_i)j \\
&\equiv \;\; \lambda_0 + \lambda_1(N - i) + \lambda_2(N - j) \\
& \;\; +\lambda_3(i - j - 1) + \lambda_4(i - 1) + \lambda_5(j - 1) \\
& \;\; \lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5 \geq 0
\end{aligned} \tag{12}$$

LHS and RHS coefficients for i, j, N and the constants are equated in (12) and the Farkas multipliers are eliminated through Fourier-Motzkin. The reader may verify that doing this yields:

$$c_i - c_j \geq 0$$

Volume bounding constraint:

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

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

$$\begin{aligned}
u_1 &\geq 0 \\
u_1 - c_i + c_j &\geq 0 \\
3u_1 + w - c_i + c_j &\geq 0
\end{aligned} \tag{13}$$

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:

$$\begin{aligned} c_j &\geq 0 \\ w - c_j &\geq 0 \\ c_i - c_j &\geq 0 \\ u_1 &\geq 0 \\ u_1 - c_i + c_j &\geq 0 \\ 3u_1 + w - c_i + c_j &\geq 0 \\ \text{minimize}_{\prec}\ &(u_1, w, c_i, c_j) \end{aligned} \tag{14}$$

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 (10) 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 (14), 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:

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

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 More Examples

The following are transformations obtained by our tool automatically from C/Fortran source code. The transformed code was generated using CLooG 0.14.0 [1]. 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 doacross 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
```

```
doacross(c1=1;c1<=n;c1++){
  doacross(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 6. Imperfectly nested 2-d Jacobi

Fig. 6 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. 7 shows the original and transformed code. All three loops can be blocked and two degrees of pipelined parallelism can be exploited.

5.3 Sequence of Matrix-Matrix multiplies

For the sequence of matrix-matrix multiplies in Fig. 8, 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 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.

```
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 7. LU decomposition

5.4 Multiple statement stencils

This code (Fig. 9) is representative of multimedia applications. The code is a sequence of producing consuming loops. Constant relative shifts are needed to enable fusion of all loops – our algorithm is able to do this. 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. 10), occurring in Tensor Contraction Expressions that appear in computational quantum chemistry problems [9]. Our tool transforms the code as shown where the producing/consuming distances between the loops have been reduced. One of the dimensions of arrays T1, T3 can now be contracted. There are other maximal fusion structures that can be enumerated, but we do not show them due to space constraints. It is extremely tedious to reason about the legality of such a transformation manually. With a semi-automatic framework accompanied with loop shifting to automatically correct transformations [44], such a transformation cannot be found unless the expert has applied the right permutation on each loop nest before fusing them. In this case, correction purely by shifting after straightforward fusion will introduce shifts at the outer levels itself, giving up reuse opportunity.

6 Implementation

We have implemented our transformation framework using PipLib 1.3.3 [13] and Polylib 5.22.3 [2]. Our tool takes as input dependence information (dependence polyhedra and h-

```
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 8. Sequence of MMs

transformations) from LooPo's [33] 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 [4] as scattering functions. The goal is to get tiled shared memory parallel code, for example, OpenMP code for multi-core architectures. Experimental results along with code generation can be found elsewhere [5].

6.1 Experimental results on running time

Table 6.1 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.

7 Related work

Iteration space tiling [24, 46, 47, 38] 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

```
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 9. Multi statement stencil

locality or minimize finish time [40, 38, 6, 48, 22, 23, 21, 39]. These works are restricted to single perfectly nested loops with uniform dependences or similar restrictions.

Loop parallelization has been studied extensively. The reader is referred to [7] 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 [25, 28]. 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. Their transformation framework had a systematic way to represent transformations (including fusion structures), however, was based on search. Automatically finding good transformations which could itself represent a compound sequence of simpler transformations (like fusion enabled by appropriate amounts of shifts or tiling of imperfectly-nested stencil code enabled by shifting, fusion and skewing) was not addressed.

Scheduling with affine functions using faces of the polytope by application of the Farkas algorithm was first proposed by Feautrier [15]. Feautrier explored various possible approaches to obtain good affine schedules that minimize latency. The one-dimensional schedules (wherever

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)

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 [19, 8, 35] 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). Overall, Feautrier's works [15, 16] 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 [32, 31] 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 [30]. 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. Also, it is not mentioned how linear independence is maintained across multiple levels of permutable loop nest sets – this situation always arises whenever there is no one-dimensional schedule or even when a 1-d schedule exists. Fusion across a sequence of weakly connected components to optimize a sequence of producer/consumer loops is not addressed. Our solution addresses all of these aspects.

Ahmed et al. [3] proposed 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 heuristic used for minimizing reuse distances is not concrete. The reuse distances in the target space for some dependences are set to zero (or a constant) with the goal of obtaining solutions to the embedding function/transformation matrix coefficients.

However, there is no concrete procedure to determine the choice of the dependences and the number (which is crucial), and how a new choice is made when no feasible solution is found. Moreover, setting some reuse classes to zero (or a constant) need not completely determine the embedding function or transformation matrix coefficients. Exploring all possibilities here leads to a combinatorial explosion and is infeasible.

Griebl [19] 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. [8], Girbal et al. [18] proposed and developed a framework (URUK/WRAP-IT) to compose sequences of transformations in a semi-automatic manner. Polyhedral transformations are manually specified by an expert and are applied automatically. These works demonstrated the practicality and feasibility of polyhedral optimization for real-world code in light of advances made in code generation [37, 4]. Pouchet et al. [35] searches the space of transformations to find good ones through empirical iterative optimization. However, their search space does not include tiling transformations. Our approach is completely model-driven and automatically finds good transformations without search. However, in several cases empirical and iterative optimization may be required in a complementary fashion to choose from 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. 3.8). Also, effective determination of tile sizes and unroll factors for transformed whole-programs may only be possible through empirical search as complex interactions with hardware may not be fully captured even in a sophisticated model. A combination of our transformation framework and empirical search in a smaller space is an interesting approach to pursue. Alternatively, more powerful cost models like those based on computing Ehrhart polynomials [45] can be employed once solutions in a smaller space can be enumerated.

8 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

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

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

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

[4] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *IEEE PACT*, pages 7–16, September 2004.

[5] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. PLuTo: A practical and fully automatic polyhedral parallelizer and locality optimizer. Technical Report OSU-CISRC-5/07-TR70, The Ohio State University, October 2007.

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

[7] 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.

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

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

[10] Alain Darte and Guillaume Huard. Loop shifting for loop compaction. *IJPP*, 28(5):499–534, 2000.

[11] Alain Darte and Guillaume Huard. Loop shifting for loop parallelization. Technical Report RR2000-22, ENS Lyon, May 2000.

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

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

[14] Paul Feautrier. Dataflow analysis of array and scalar references. *IJPP*, 20(1):23–53, 1991.

[15] Paul Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *IJPP*, 21(5):313–348, 1992.

[16] Paul Feautrier. Some efficient solutions to the affine scheduling problem. part II. multidimensional time. *IJPP*, 21(6):389–420, 1992.

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

[18] 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. *IJPP*, 34(3):261–317, June 2006.

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

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

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

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

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

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

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

[26] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers 95*, page 332, 1995.

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

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

[29] Wei Li and Keshav Pingali. A singular loop transformation framework based on non-singular matrices. *IJPP*, 22(2):183–205, 1994.

[30] 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.

[31] 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.

[32] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3-4):445–475, 1998.

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

[34] Roger Penrose. A generalized inverse for matrices. *Proceedings of the Cambridge Philosophical Society*, 51:406–413, 1955.

[35] 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.

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

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

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

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

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

[41] Alexander Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1987.

[42] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. Polyhedral code generation in the real world. In *CC*, pages 185–201, March 2006.

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

[44] Nicolas Vasilache, Albert Cohen, and Louis-Noel Pouchet. Automatic correction of loop transformations. In *PACT'07*, 2007.

[45] 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.

[46] M. Wolf. More iteration space tiling. In *SC*, pages 655–664, 1989.

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

[48] Jingling Xue. Communication-minimal tiling of uniform dependence loops. *JPDC*, 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