A Practical and Fully Automatic Polyhedral Program Optimization System

Uday Bondhugula[1] J. Ramanujam[2] P. Sadayappan[1]

[1]Dept. of Computer Science and Engineering
The Ohio State University
2015 Neil Ave. Columbus, OH, USA
{bondhugu,saday}@cse.ohio-state.edu

[2]Dept. of Electrical & Computer Engg. and
Center for Computation & Technology
Louisiana State University
jxr@ece.lsu.edu

OSU-CISRC-10/07-TR70

Abstract

We present the design and implementation of a fully automatic polyhedral source-to-source transformation framework that can optimize regular programs (sequences of possibly imperfectly nested loops) for parallelism and locality simultaneously. Through this work, we show the practicality of analytical model-driven automatic transformation in the polyhedral model – far beyond what is possible by current production compilers. Unlike previous works, our approach is an end-to-end fully automatic one driven by an integer linear optimization framework taking an explicit view of finding good ways of tiling for parallelism and locality using affine transformations. We also address generation of tiled code for multiple statement domains of arbitrary dimensionalities under (statement-wise) affine transformations – an issue that has not been addressed previously. Experimental results from the implemented framework show very high speedups for local and parallel execution on multi-cores. The system also enables the easy use of powerful empirical/iterative optimization for general arbitrarily nested loop sequences.

1. Introduction and Motivation

Current trends in processor architecture are increasingly towards a larger number of processing elements on a chip. This has led to parallelism and multi-core architectures becoming mainstream, along with the emergence of several specialized parallel architectures or accelerators like the Cell processor and General-Purpose GPUs. The difficulty of programming these architectures to effectively tap the potential of multiple on-chip processing units is a significant challenge. Among several approaches to addressing this issue, one that is very promising but simultaneously very challenging is automatic parallelization. This requires no effort on part of the programmer in the process of parallelization and optimization and is therefore very attractive.

Many compute-intensive applications often spend most of their running time in nested loops. This is particularly common in scientific and engineering applications. The polyhedral model [16] 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 precise characterization of inter or intra-statement dependences, it is possible to reason about the correctness of a sequence of complex loop transformations using machinery from Linear Algebra and Linear Programming. The transformations finally reflect in the generated code as reordered execution with improved cache locality and/or loops that have been parallelized. 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 enclosing loop variables and parameters. While a precise characterization of data dependences is feasible for programs with static control structure and affine references/loop-bounds, codes with non-affine array access functions or code with dynamic control can also be handled, but with conservative assumptions on some dependences.

In the polyhedral framework, the task of automatic parallelization of an input sequential program comprised of a set of (possibly imperfectly) nested loops may be viewed in terms of three phases: (1) static dependence analysis of the input program, (2) transformations in the polyhedral abstraction, and (3) generation of efficient loop code for the transformed program. Significant advances were made in the nineties on dependence analysis [13, 33] and code generation [27, 20] in the polyhedral model, but the approaches suffered from scalability challenges. Hence, applicability was mainly limited to very small loop nests. Recent advances in dependence analysis [43] and more importantly in code generation [34, 6, 42, 41] have solved many of these problems resulting in the polyhedral techniques being applied to code representative of real applications like the spec2000fp benchmarks. CLooG [6, 1] is a powerful state-of-the-art code generator that captures most of these advances and is widely used. 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 [11, 17]. Thus, the current state-of-the-art is very advanced with respect to the first and third step of the three-phase process for program optimization in the polyhedral framework. Regarding the key middle

step, an important problem is to find an approach to drive transformations automatically. Our work addresses this problem, by completing this key link that enables fully automatic parallelization and locality optimization.

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. Tiling has been studied from two perspectives - data locality optimization and parallelization. Tiling for locality requires grouping points in an iteration space into smaller blocks (tiles) allowing reuse in multiple directions when the block fits in a faster memory (registers, L1, or L2 cache). Tiling for coarse-grained parallelism fundamentally involves partitioning the iteration space into tiles that may be concurrently executed on different processors with a reduced frequency and volume of inter-processor communication: a tile is atomically executed on a processor with communication required only before and after execution. 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. One of the key aspects of our transformation framework is to find good ways of performing tiling and fusion.

The works of Feautrier [12, 13, 14] have led to many research efforts on automatic parallelization in the polyhedral model. However, existing automatic transformation frameworks [31, 30, 29, 19] have one or more drawbacks or restrictions that limit their effectiveness for parallelizing/optimizing loop nests. A significant problem is the lack of a realistic cost model that is suitable for coarse-grained parallel execution, as is used in practice with manually developed parallel applications. Many previously proposed approaches also do not consider locality and parallelism in an integrated fashion. Hence, comprehensive performance evaluation on parallel targets using a range of test cases has not been done using a powerful and general model like the polyhedral model.

This paper presents the design and implementation of a practical parallelizer and locality optimizer in the polyhedral model. Finding good ways to tile for parallelism and locality directly in an affine transformation framework is the central idea. The contributions of this work are as follows:

- We show how tiled code generation for statement domains of arbitrary dimensionalities under statement-wise affine transformations is done for local and shared memory parallel execution
- We evaluate the performance of the implemented system by use of a number of non-trivial application kernels on a multi-core processor

Model-driven empirical optimization and automatic tuning approaches have been shown to be very effective in optimizing single-processor execution for some regular kernels like Matrix-matrix multiplication [45, 46, 51, 52], and ATLAS is widely used in practice. There is considerable interest in developing effective empirical tuning approaches for arbitrary input kernels. Our framework can enable such model-driven or guided empirical search to be applied to arbitrary affine programs, in the context of both sequential and parallel execution.

The rest of this report is organized as follows. Section 2 provides notation, mathematical background for the polyhedral model and affine transformations. Section 3 provides an overview of the theoretical framework that drives our automatic transformation. Sec. 4 discusses the design of our system, mainly focusing on tiled and shared memory parallel code generation. Sec 5 provides experimental results. Section 6 discusses the large body of relevant work in this field 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. All row vectors are typeset in bold.

2.1 The polytope model

DEFINITION 1 (Affine Hyperplane). *The set X of all vectors $x \in \mathbf{Z}^n$ such that $\mathbf{h}.\vec{x} + c = k$, for $k \in \mathbf{Z}$, 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 and the constant c. Each instance of a hyperplane is an $n-1$ dimensional affine sub-space of the n-dimensional space. Two vectors $\vec{x_1}$ and $\vec{x_2}$ lie in the same hyperplane if $\mathbf{h}.\vec{x_1} = \mathbf{h}.\vec{x_2}$.

DEFINITION 2 (Polyhedron, polytope). *The set of all vectors $\vec{x} \in \mathbf{Z^n}$ such that $A\vec{x} + \vec{b} \geq 0$, where A is an integer matrix, defines a (convex) integer polyhedron. A polytope is a bounded polyhedron.*

The notation we describe below is same as that used by Feautrier [14]. 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 size); our work is focused on programs with this property. Let $\vec{n}$ be the vector of the structure parameters. Let the hyperplanes bounding the polytope of statement S_k be given by:

$$\mathbf{a_{S,k}} \begin{pmatrix} \vec{i} \\ \vec{n} \end{pmatrix} + b_{S,k} \geq 0, \;\; 1 \leq k \leq m \tag{1}$$

A well-known known result useful for polyhedral analyses 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 s affine inequalities or faces*

$$\mathbf{a_k}.\vec{x} + b_k \geq 0, \;\; 1 \leq k \leq s$$

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

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

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 [39].

2.2 Dependence Abstraction

Our dependence model is of exact affine dependences and same as the one used by Feautrier [14] and Lim et al. [31, 30]. Dependences are determined precisely through dataflow analysis [13], but we consider all dependences including anti (write-after-read) and output (write-after-write) dependences, i.e., input code does not require conversion to single-assignment form. 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.

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 form that conveys when exactly the dependence exists between a dynamic instance of the source statement S_i, namely $\vec{s}$, and a dynamic instance of the target S_j, namely $\vec{t}$. It can be captured in a polyhedron in the sum of dimensionalities of the source and target statement's polyhedra (with dimensions for structure parameters as well).

$$\begin{aligned} & \vec{s} \in D^{s_i},\ \vec{t} \in D^{s_j},\ \langle \vec{s}, \vec{t} \rangle \in R_e \\ \equiv \quad & \begin{bmatrix} \cdots & \cdots & \\ \vdots & \vdots & \\ \cdots & \cdots & \end{bmatrix} \begin{bmatrix} \vec{s} \\ \vec{t} \\ \vec{n} \\ 1 \end{bmatrix} \begin{matrix} \cdots \\ \geq 0 \\ \cdots \\ = 0 \\ \cdots \end{matrix} \end{aligned} \tag{3}$$

We call the polyhedron on the RHS of (3) – the *exact dependence polyhedron*, denoted by P_e. The equalities in 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* [14], thus excluding any transitively covered dependences. However, the last access condition is not necessary; in general, the equalities can be used to eliminate variables from P_e. We have such a dependence polyhedron for every dependence, $e \in E$. In the rest of this section, we assume for convenience that $\vec{s}$ can be completely eliminated using the h-transformation, being substituted by $f_e(\vec{t})$, f_e being the h-transformation. Let the simplified P_e be defined by:

$$\mathbf{c}_{e,k} \begin{pmatrix} \vec{t} \\ \vec{n} \end{pmatrix} + d_{e,k} \geq 0,\ \ 1 \leq k \leq m_e. \tag{4}$$

2.3 Statement-wise Affine Transforms

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

$$\begin{aligned} \phi_{s_k} &= \begin{pmatrix} c_1 & \ldots & c_{m_{S_k}} \end{pmatrix} \begin{pmatrix} \vec{i} \end{pmatrix} + c_0 \qquad (5) \\ &= \begin{pmatrix} c_1 & \ldots & c_{m_{S_k}} & c_0 \end{pmatrix} \begin{pmatrix} \vec{i} \\ 1 \end{pmatrix} = h_{S_k} \vec{i} + c_0 \\ & \quad \text{where } h_{S_k} = (c_1, \ldots, c_{m_{S_k}}),\ \ c_0, c_1, \ldots, c_{m_{S_k}} \in \mathbf{Z} \end{aligned}$$

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 h_{S_k} in that order. The vector carries the translation constants (c_0). A single row along with the translation can be viewed as an affine hyperplane. 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 combinations of 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 a transformation automatically.

The above representation for transformations is similar to that used by many researchers: first by [15, 25], and more recently in a systematic way by [11, 17] and directly fits with scattering functions that a code generation tool like CLooG [6, 1] 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 $c_1, c_2, \ldots$. The next section provides an overview of an approach that finds the coefficients of the transformation matrices (along with the vectors) that are best for parallelism and locality.

3. Overview of Automatic Transformation Approach

In this section, we give an overview of our affine transformation framework that can optimize a sequence of imperfectly nested loops for parallelism and locality by embedding a flexible and powerful cost function in a linear optimization framework. Full details can be found in another report [8].

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:*

$$\phi_{s_j}(\vec{t}) - \phi_{s_i}(\vec{s}) \quad \geq \quad 0,\ \ \langle \vec{s}, \vec{t} \rangle \in R_e \tag{6}$$

The tiling of a statement's iteration space 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 exist no two tiles such that they both depend on 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 (6). Consider a tile formed by aggregating a group of hyperplane instances along $\phi^1_{s_i}$ and $\phi^2_{s_i}$. Due to (6), 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 the point of view of data locality, 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 single-statement perfectly nested loops from the work of Irigoin and

```
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

Triolet [23] (as $h^T.R \geq \mathbf{0}$). We have generalized it above for multiple iteration spaces with affine dependences, with possibly different dimensionalities and imperfect nestings for statements. The interleaved execution also leads naturally to the notion of fusion.

Tiling at an arbitrary depth. Note that the legality condition as written in (6) 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.

In the rest of this section, we use the term affine transform (with this property) and tiling hyperplane interchangeably, since after applying the transformation, the target loops in the transformed iteration space can be orthogonally blocked.

3.2 Cost function

Consider the following affine form δ_e:

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

The affine form $\delta_e(\vec{t})$ holds much significance. This function is 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 factor in the 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.

The constraints obtained from (6) only guarantee legality of tiling (permutability). However, an attempt to minimize communication volume ends up in an objective function involving both loop variables and hyperplane coefficients. For example, $\phi(\vec{t}) - \phi(f_e(\vec{t}))$ 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 ends up with such a form when a dependence is not uniform or for an inter-statement dependence, making it hard to construct an objective function involving only the unknown hyperplane coefficients. We use a bounding function approach that allows the application of Farkas Lemma and casting into an ILP formulation.

3.3 Cost function bounding and minimization

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

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

such that

$$v(\vec{n}) \; - \; \left(\phi_{s_i}(\vec{t}) - \phi_{s_j}(f_e(\vec{t}))\right) \quad \geq \quad 0, \;\; \vec{t} \in P_e, \quad \forall e \in E$$

$$\textit{i.e.,} \qquad v(\vec{n}) \; - \; \delta_e(\vec{t}) \quad \geq \quad 0, \;\; \vec{t} \in P_e, \quad \forall e \in E$$

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 $\delta_e(\vec{s})$. Note that the loop variables themselves are bounded by affine functions of the parameters, and hence the maximum value taken by $\delta_e(\vec{s})$ will be bounded by such an affine form. Also, since $v(\vec{n}) \geq \delta_e(\vec{s}) \geq 0$, 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. Such a bounding function was used by Feautrier [14] to find minimum latency schedules.

Now, we apply Farkas Lemma to (9).

$$v(\vec{n}) - \delta_e(\vec{t}) \quad \equiv \quad \lambda_{e0} + \sum_{k=1}^{m_e} \lambda_{ek} \left(\mathbf{c}_{ek} \begin{pmatrix} \vec{t} \\ \vec{n} \end{pmatrix} + d_{ek} \right), \;\; \lambda_{ek} \geq 0 \tag{9}$$

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 $\mathbf{u}$, and w. The above inequalities can 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. Let $\mathbf{u} = (u_1, u_2, \ldots u_k)$.

$$\text{minimize}_{\prec} \{u_1, u_2, \ldots, u_k, w, \ldots, c_i's, \ldots\} \tag{10}$$

Finding the lexicographic minimal solution for a system of linear inequalities is within the reach of the simplex algorithm and can be handled by the Parametric Integer Programming (PIP) software [12]. Since the structural parameters are quite large, we first want to minimize their coefficients. The solution gives a hyperplane

for each statement. Note that the application of the Farkas Lemma to (9) is not required in all cases.

3.4 Iteratively finding independent solutions through orthogonal basis.

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 make sure of linear independence with solutions already found, by constructing the orthogonal sub-space of the transformation rows found so far (H_S) as follows:

$$H_S^{\perp} = I - H_S^T \left(H_S H_S^T \right)^{-1} H_S \tag{11}$$

3.5 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 (10) 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 the 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 local cache tile sizes. 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).

3.6 Space and time in transformed iteration space.

By minimizing $\phi(\vec{t}) - \phi(\vec{s})$ as we find hyperplanes from outermost to innermost, we push dependence carrying to inner loops and also ensure that loops do not have negative dependence components (to the extent possible) so that target loops can be tiled. 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. Whenever the loops can be tiled, they result in coarse-grained parallelism as well as better reuse within a tile. Note that all loops detected as parallel need not be marked parallel. If a degree of communication-free parallelism exists, that particular loop (assuming it has a large extent) is sufficient to expose enough coarse-grained parallelism.

3.7 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.

Algorithm 1 Affine transformation algorithm

Input Generalized dependence graph $G = (V, E)$ (includes dependence polyhedra 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 $\phi(\vec{t}) - \phi(f_e(\vec{t})) \geq 0$ under $\vec{t} \in P_e$, and eliminate all Farkas multipliers
4: Build communication volume/reuse distance bounding constraints: apply Farkas Lemma to $v(\vec{n}) - (\phi(\vec{t}) - \phi(f(\vec{t}))) \geq 0$ under $\vec{t} \in P_e$, 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; update necessary dependence polyhedra (when a portion of it is satisfied)
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)

The algorithm is summaried in Fig. 1. Note that the transformations found are compound ones that realize a sequence of simpler transformations. Correctness and completeness results can be found in [8].

4. Tiled code generation for arbitrarily-nested loops under statement-wise transformations

In this section, we describe how tiled code is generated from the transformations found in the previous section.

Code generation under multiple affine mappings was first addressed by Kelly et al. [27]. Significant advances were made by Quilleré et al [34] and more recently by Bastoul [6] and Vasilache et al [42], and have been implemented into a open-source tool, CLooG [1] CLooG uses PolyLib (which uses the efficient Chernikova algorithm) for its core polyhedral operations, and code generated is much more efficient than previous code generators based on Fourier-Motzkin variable elimination (e.g. Omega [33] or LooPo's internal code generator [2]). Also, code generation time and memory utilization are much lower, allowing code to be generated for hundreds of statements with a number of free parameters without any memory explosion [6]. Such a powerful and efficient code generator is essential in conjunction with the transformation

framework we develop, since the statement-wise transformations found when coupled with tiling lead to complex execution reordering. This is especially so for imperfectly nested loops and generation of parallel code, as will be seen in the rest of this section.

Fig. 2 shows the entire tool-chain that has been implemented. We used PipLib 1.3.3 [12, 3] as the ILP solver and CLooG 0.14.1 (with 64 bits) for code generation. Our tool takes as input dependence information (dependence polyhedra and h-transformations) from LooPo's [2] dependence tester and generates statement-wise affine transformations. Flow, anti and output dependences are considered for legality as well as the bounding function, while read dependences can optionally be considered for the bounding objective. The transforms generated by our tool are provided to CLooG as scattering functions. Thanks to the various options provided by CLooG, readily compilable/runnable code can be generated by using it as a library. We first give a brief description of the state-of-the-art code generator CLooG. CLooG can scan a union of polyhedra efficiently, under a new global lexicographic ordering specified as the scattering functions. Scattering functions are specified statement-wise, and the legality of scanning the polyhedron with these dimensions in the specified order should be guaranteed by the specifier – in our case, an automatic transformation approach. The code generator does not have any information on the dependences and hence, in the absence of any scattering functions would scan the union of the statement polyhedra in the global lexicographic order of the original iterators (statement insteances are interleaved).

Before proceeding further, we differentiate between using the term 'tiling' for, (1) modeling and enabling tiling through a transformation framework (as was described in the previous section), and (2) final generation of tiled code from the hyperplanes found. Both could be referred to as tiling and hence the need to clarify. Our approach models tiling in the transformation framework and finds good bands of permutable loops. The hyperplanes found are the new *basis* for the loops in the transformed space and have special properties that have been detected when the transformation is found – e.g. being parallel, sequential or belonging to a band of loops that can now be orthogonally tiled (possibly multiple times). Hence, the transformation framework guarantees legality of such rectangular tiling. The final generation of tiled loops can be done directly through the polyhedral code generator itself or syntactically (in some simple cases) as a post-pass on the generated target abstract syntax tree. We discuss tiled code generation from such transforms now.

Tiled code generation with parametric tile sizes within the polyhedral model has recently been addressed by Renganarayana et al. [36]. But their approach only handles single domains that are rectangularly blockable, While the techniques could be extended to tile loops in an AST at any level, tiling the transformed AST is less powerful than generating the transformed-cum-tiled code in one pass through the code generator. We explain our tiled code generation scheme for any number of domains (with possibly different dimensionalities) under (domain-wise) scattering functions. We initially use fixed tile sizes in our framework; we plan to extend it to use parametric tiling – which would be convenient for iterative/empirical optimization.

4.1 Tiles under a transformation

Our approach to tiling in a polyhedral framework is to specify the domain and scattering constraints for the tiled higher-dimensional iteration space directly to CLooG, i.e. explicitly generate and provide the scattering functions for the outer tile space loops.

Consider a two-dimensional loop nest with i and j iterators. Let the transformation found be $c_1 = i$, and $c_2 = i + j$, with c_1 and c_2 both found in Step 12 of Algo 1; hence, they can be blocked. We would like to obtain code that is tiled rectangularly along c_1 and c_2. The domain supplied to the code generator is a higher dimensional domain with the tile shape constraints exactly like proposed by Ancourt and Irigoin [5]. The tile space and intra tile loop scattering functions are specified as follows:

$$\begin{array}{rc} \text{Domain} & \text{Scattering} \\ 0 \leq i \leq N-1 & c_1T = iT \\ 0 \leq j \leq N-1 & c_2T = iT + jT \\ 0 \leq i - 32iT \leq 31 & c_1 = i \\ 0 \leq (i+j) - 32(iT + jT) \leq 31 & c_2 = i + j \\ (c_1T, c_2T, c1, c2) & \leftarrow scatter(iT, jT, i, j) \end{array}$$

c_1T and c_2T are the tile space loops in the transformed space. This approach can seamlessly tile across statements of arbitrary dimensionalities, irrespective of original nesting structure, as long as the $c_i's$ have dependences (inter-stmt and intra-stmt) in the forward direction – this is guaranteed and detected by the transformation framework (Step 12 of Algorithm 1).

Fig. 4.1 shows tiles for imperfectly nested 1-d Jacobi. Note that tiling it requires a relative shift of S2 by one and skewing the space loops skewing by a factor of two (as opposed to skewing by a factor of one that is required for the space inefficient perfectly nested version).

4.2 3-d tiles for LU

The transformation obtained for LU (Fig. 8(a)) by our tool is:

$$S1 : \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} k \\ j \end{bmatrix}$$

$$S2 : \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} k \\ i \\ j \end{bmatrix}$$

Hyperplanes c_1, c_2 and c_3 are identified as belonging to one tilable band. 3-d tiles for LU decomposition from the above transformation are specified as follows:

4.3 Preventing code expansion in tile space

The overhead of floor and ceil operations as well as conditionals in the tile-space loops (at the outer levels) is insignificant. Hence, we would like to have compact code at the outer level while allowing code expansion in the intra-tile loops to decrease control complexity. This improves performance while keeping the code size under control.

Table 4.3 shows the sensitivity in performance for a 1-d stencil code shown in Fig. 3. Using the default options with tiling specified as described leads to significant code expansion since the transformed space we are tiling is a shifted and skewed space. Preventing any code expansion at all leads to an if condition in the innermost loop, resulting in very low performance. However, optimizing only the intra-tile loops for control (the -f and -l CLooG flags representing first and last loop depths to optimize for control) is very effective. Besides, it also avoids large numbers in the intermediate operations performed by code generators, that could possibly lead to PolyLib overflows for large tile sizes or deep loop nest tiling.

Whenever there is a perfectly nested band of loops in the AST - it is easy to do multi-level syntactic tiling, followed by unroll of the innermost intra-tile loop (register tile). Alternatively, such loops can be tiled efficiently with parametric tile sizes with existing techniques [36, 28] that would require another pass of CLooG on the AST. However, for transformations that possibly lead to imperfectly nested code, polyhedral tiling is a natural way to get tiled

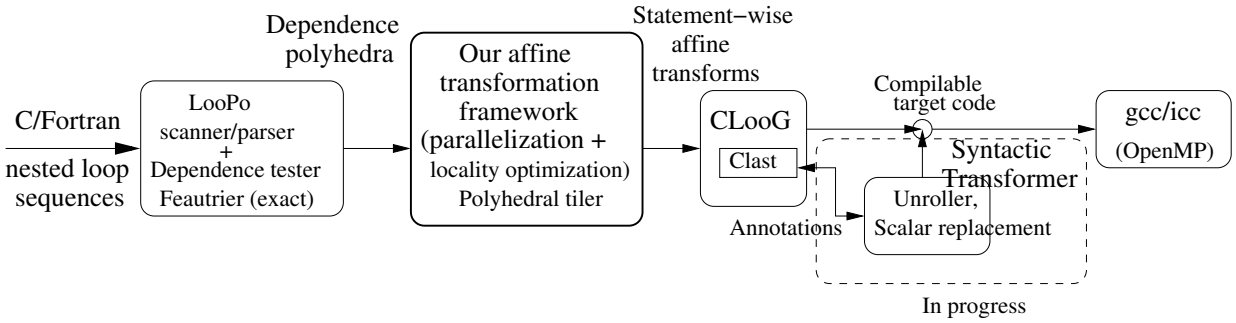


Figure 2. The Automatic Transformation System

CLooG	Code size (lines)	Codegen time	Performance of code	Speedup (base: icc -fast -O3)
Full code expansion	2226	1.84s	2.57s	2.7x
Only tile space expansion (-f3 -l5)	40	0.04s	1.6s	4.3x
No code expansion (-short)	15	0.01s	17.6s	0.39x

Table 1. Code size and performance sensitivity with CLooG codegen options – Imperfectly nested stencil tiled (L1) ($N = 10^6$, $T = 1000$, tile size=2048)

Domains

$S1$	$S2$
$0 \leq k \leq N-1$	$0 \leq k \leq N-1$
$k+1 \leq j \leq N-1$	$k+1 \leq i \leq N-1$
	$k+1 \leq j \leq N-1$
$0 \leq k - 32kT \leq 31$	$0 \leq k - 32kT \leq 31$
$0 \leq j - 32jT \leq 31$	$0 \leq i - 32iT \leq 31$
	$0 \leq j - 32jT \leq 31$

Scatterings

S1	S2
$c_1T = kT$	$c_1T = kT$
$c_2T = jT$	$c_2T = jT$
$c_3T = kT$	$c_3T = iT$
$c_1 = k$	$c_1 = k$
$c_2 = j$	$c_2 = j$
$c_3 = k$	$c_3 = i$
$(c_1T, c_2T, c_3T, c1, c2, c3)$	$(c_1T, c_2T, c_3T, c1, c2, c3)$
$\leftarrow scatter(kT, jT, k, j)$	$\leftarrow scatter(kT, jT, iT, k, j, i)$

code from the code generator in one pass guaranteeing legality. The legality of syntactic tiling or unroll/jam of such loops cannot be guaranteed since once we obtain the syntax tree of the transformed code, we are outside of the polyhedral model, unless advanced techniques like re-entrance are used [41]. Even if legality of such tiling is guaranteed, such an approach would miss ways of tiling possible by just viewing the original domains and the transformations. Consider the code in Fig. 3(b) for example. Even though the transformation obtained has the forward dependence property – once transformed code is generated, tiling the imperfectly nested loop cannot be reasoned about in the target AST. Doing a simple unroll-jam of the imperfect loop nest leads to an illegal tiling.

4.4 Parallel Tiled Code generation

Unlike other transformation approaches, since we find tiling hyperplanes, 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 generating parallel code when one or more of these loops carry a forward dependence: the framework makes sure that the dependence is in the forward direction. Our approach to coarse-grained (tiled) shared memory parallel code generation is as follows.

Algorithm 2 Parallel code generation under a tiling transformation

1: At any level, we have a set of k fully permutable loops in the transformed space. All loops in each such set are tilable.
2: To extract p ($< k$) degrees of pipelined parallelism, perform the following unimodular transformation on only the outer tile loops that step through the tile space (for a set): $(\phi^1, \phi^2, \ldots, \phi^k) \rightarrow (\phi^1 + \phi^2 + \cdots + \phi^{p+1}, \phi^2, \ldots, \phi^k)$. This gives us an outer loop that is a valid tile schedule.
3: Mark $\phi^2, \phi^3, \ldots, \phi^{p+1}$ as parallel
4: Leave $\phi^1, \phi^{p+2}, \ldots, \phi^k$ as sequential
5: Place a barrier at the end of the tile schedule loop $\phi^1 + \phi^2 + \cdots + \phi^{p+1}$

Once the technique described in the previous section is applied to generate the tile space scatterings and intra-tiled loops – dependence components are all forward and non-negative for any band of tile space loops. Hence, the sum $\phi^1 + \phi^2 + \cdots + \phi^{p+1}$ carries all affine dependences carried by ϕ^1, ϕ^2, ..., ϕ^{p+1}, and gives a legal wavefront of tiles. This tile space transformation thus merely obtains a valid schedule of tiles and preserves tile shapes. Note that communication still happens along boundaries of ϕ^1, ϕ^2, ..., ϕ^s, and the same old hyperplanes $\phi^1, \phi^2, \ldots, \phi^k$ are used to scan a tile. Note that obtaining an affine (fine-grained) schedule and then enabling time tiling would lead to shapes different from above our approach. This technique is similar the approach used in [31] where permutable partitions are summed up for maximal dependence dismissal; however, we do this in the tile space as opposed to for finding a schedule that provides the maximum degree of parallelism.

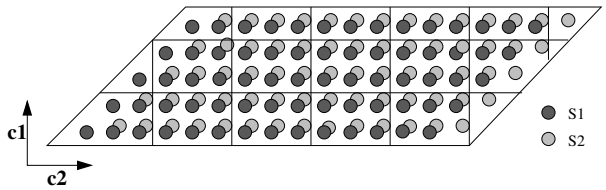
```
for (t=0; t<T; t++) {
    for (i=2; i<N-1; i++) {
      b[i] = 0.333*(a[i-1] + a[i] + a[i+1]);
    }
    for (j=2; j<N-1; j++) {
      a[j] = b[j];
    }
}
```

(a) Original sequential code

```
#define S1(t,i) {b[i]=(0.333*(a[1+i]+a[i]+a[i-1]);}
#define S2(t,j) {a[j]=b[j];}

for (c1=0;c1<=T-1;c1++) {
  S1(c1,2) ;
  for (c2=2*c1+3;c2<=2*c1+N-2;c2++) {
    S1(c1,-2*c1+c2) ;
    S2(c1,-2*c1+c2-1) ;
  }
  S2(c1,N-2) ;
}
```

(b) Optimized without tiling



(d) Tiles under a transformation

```
#define S1(t,i) {b[i]=((double)(333))/1000*(a[1+i]+a[i]+a[i-1]);}
#define S2(t,j) {a[j]=b[j];}
/* Generated from PLuTo-produced CLooG file by CLooG v0.14.1 64 bits in 0.02s. */
for (c1=0;c1<=floord(T-1,256);c1++) {
  for (c2=max(0,ceild(512*c1-253,256));
       c2<=min(floord(N+2*T-3,256),floord(512*c1+N+509,256));c2++){
    if ((c1 <= floord(256*c2-N+1,512)) && (c2 >= ceild(N-1,256))) {
      if ((-N+1)%2 == 0) {
        S2(c1,-2*c1+c2,(256*c2-N+1)/2,N-2) ;
      }
    }
  }
  for (c3=max(max(ceild(256*c2-N+2,2),256*c1),0);
       c3<=min(min(T-1,256*c1+255),floord(256*c2-N+256,2));c3++){
    for (c4=256*c2;c4<=2*c3+N-2;c4++) {
      S1(c1,-2*c1+c2,c3,-2*c3+c4) ;
      S2(c1,-2*c1+c2,c3,-2*c3+c4-1) ;
    }
    S2(c1,-2*c1+c2,c3,N-2) ;
  }
  for (c3=max(max(0,256*c1),ceild(256*c2-N+257,2));
    c3<=min(min(256*c1+255,T-1),128*c2-2);c3++){
    for (c4=256*c2;c4<=256*c2+255;c4++) {
      S1(c1,-2*c1+c2,c3,-2*c3+c4) ;
      S2(c1,-2*c1+c2,c3,-2*c3+c4-1) ;
    }
  }
  for (c3=max(max(128*c2-1,0),256*c1);
    c3<=min(min(128*c2+126,256*c1+255),T-1);c3++){
    S1(c1,-2*c1+c2,c3) ;
    for (c4=2*c3+3;c4<=256*c2+255;c4++) {
      S1(c1,-2*c1+c2,c3,-2*c3+c4) ;
      S2(c1,-2*c1+c2,c3,-2*c3+c4-1) ;
    }
  }
 }
}
```

(c) Optimized with tiling (tile size 256), cloog -f 3 -l 5

$$S1: \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 2 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} t \\ i \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad S2: \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 2 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} t \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$$

(d) Transformation obtained

$$S1: \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} tT \\ iT \\ t \\ i \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad S2: \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} tT \\ jT \\ t \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

(e) Transformation for generation of locally tiled code

$$S1: \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{bmatrix} = \begin{bmatrix} 3 & 1 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} tT \\ iT \\ t \\ i \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad S2: \begin{bmatrix} c_1 \\ \mathbf{c_2} \\ c_3 \\ c_4 \\ c_5 \end{bmatrix} = \begin{bmatrix} 3 & 1 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} tT \\ jT \\ t \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

(f) Transformation for generation of parallelized + locally tiled code

Figure 3. Tiling imperfectly nested 1-d Jacobi

Fig. 4 shows a simple example with tiling hyperplanes (1,0) and (0,1).

Note that, in practice, a few degrees of pipelined parallelism may be sufficient. Using several degrees could introduce code complexity. Note that all loops need not be tiled; this choice can be made either empirically or through an additional model. If we have reuse along each, in theory, it should only lead to improved performance. However, having too many tiled loops could be detrimental.

```
for(i=1; i<N; i++){
    for(j=1; j<N; j++){
        a[i,j] = a[i-1,j] + a[i,j-1];
    }
}
```

(a) Original sequential code

```
for (c1=-1;c1<=floord(N-1,16);c1++) {
  #pragma omp parallel for shared(c1,a) private (c2,c3,c4)
    for (c2=max(ceild(32*c1-N+1,32),0);
      c2<=min(floord(16*c1+15,16),floord(N-1,32));c2++) {
        for (c3=max(1,32*c2);c3<=min(32*c2+31,N-1); c3++) {
            for (c4=max(1,32*c1-32*c2);
              c4<=min(N-1,32*c1-32*c2+31); c4++) {
                S1(c2,c1-c2,c3,c4) ;
            }
        }
    }
    /* barrier happens only here (in tile space) */
}
```

(b) Coarse-grained parallel barrier after a tile-space transformation

Figure 4. Shared memory parallel code generation example

5. Experimental evaluation

In this section, we evaluate the performance of the transformed codes generated by our implementation.

Comparison with previous approaches

Several previous papers on automatic parallelization have presented experimental results. A direct comparison is difficult since the implementations of those approaches is not available; further most previously presented studies did not use an end-to-end automatic implementation, but performed manual code generation based on solutions generated by a transformation framework. Often, a missing link in the chain was the lack of a powerful and efficient code generator like CLooG , which has only recently become available.

In assessing the effectiveness of our system, we compare performance of the generated code with that generated by production compilers, as well as undertaking a best-effort fair comparison with previously presented approaches from the research community. The comparison with other approaches from the literature is in some cases infeasible because there is insufficient information for us to reconstruct a complete transformation (e.g. [4]). For others [31, 30, 29], a complete description of the algorithm allows us to manually construct the transformation; but since we do not have access to an implementation that can be run to determine the transformation matrices, we have not attempted an exhaustive comparison for all the cases.

The current state-of-the-art with respect to optimizing code has been semi-automatic approaches that require an expert to manually guide transformations. As for scheduling-based approaches, the LooPo system [2] includes implementations of various polyhedral scheduling techniques including Feautrier's multi-dimensional time scheduler which can be coupled with Griebl's space and FCO time tiling techniques. We thus provide comparison for some number of cases with the state of the art – (1) Griebl's approach that uses Feautrier's schedules along with FCO to enable time tiling [19], and (2) Lim/Lam's affine partitioning [31, 30, 29]. For both of these previous approaches, the input code was run through our system and the transformations were forced to be what those approaches would have generated. Hence, these techniques get all benefits of CLooG and our fixed tile size code generation scheme.

Experimental setup. The results were taken on a quad-core Intel Core 2 Quad Q6600 CPU clocked at 2.4 GHz (1066 MHz FSB) featuring a 32 KB L1 D cache and 8MB of L2 cache (4MB shared per core pair) with 2 GB of DDR2-667 RAM, running Linux kernel version 2.6.22 (x86-64). ICC 10.0 is the primary compiler used to compile the base codes as well as the source-to-source transformed codes; it was run with "-fast -O3 -funroll-loops" (-openmp for parallelized code) – these options also enable auto vectorization in ICC. Whenever gcc was used for comparison, it was GCC 4.1.1 with options "-O3 -funroll-loops" (-fopenmp for parallelized code). The OpenMP implementation of ICC supports nested parallelism – this is needed for exploiting multiple degrees of pipelined parallelism when they exist. Due to the large size of the L2 cache, and for easier presentation, local tiling for almost all codes is for L1 cache, with equal tile sizes used along all dimensions. Tiling for L2 and full tile/partial tile separation with register tiling [28] and unrolling with a systematic tile size selection approach would give further improvement. In all cases, the optimized code for our framework was obtained fully automatically in a turn-key fashion from the input source code. Tile sizes were set empirically and agreed with the cache size quite well.

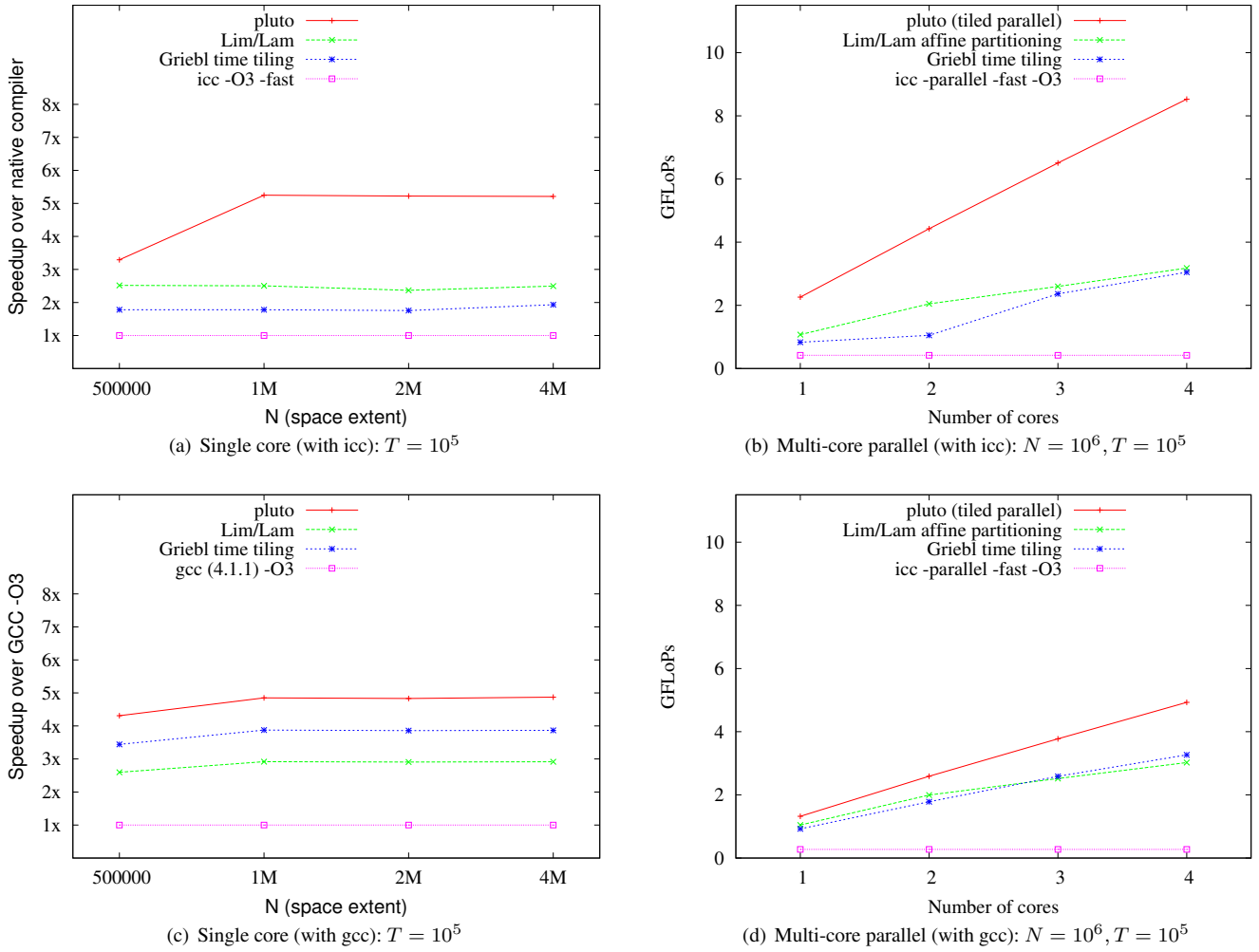
The transformation framework itself runs quite fast – within a fraction of a second for all benchmarks considered. Since the same tiling hyperplanes are used for parallelization and locality optimization, parallel code looks same as the local tiled code except that there is a transformation on the tile space to get a tile schedule in the parallel code (as explained in Sec. 4.4) and the right loop(s) are marked parallel. The OpenMP "parallel for" directive(s) achieves the distribution of the blocks of the tile space loop(s) among the processors. Hence, the execution on each core is a sequence of L1 tiles.

5.1 Imperfectly nested stencil code

The original code, code optimized by our system without tiling, and optimized tiled code are shown in Fig. 3. The performance of the optimized codes are shown in Fig. 5.1. Speedup's ranging from 4x to 7x are obtained for single core execution due to locality enhancement. The best tile size proved to be 2048 (2*2048*8 = 32KB same as L1 D cache size). The parallel speedups are compared with Lim/Lam's technique (Algorithm A in [31]) that generates (2,-1), (3,-1) as the tiling hyperplanes. These do minimize the order of synchronization and maximize the degree of parallelism (O(N)), but any legal tiling hyperplanes would have one degree of pipeline parallelism. With scheduling-based techniques, the schedules found by LooPo's Feautrier scheduler are $2t$ and $2t + 1$ for S1 and S2, respectively (note that this doesn't lead to fusion). An FCO allocation here is given by $2t + i$, and this enables time tiling. Just space tiling in this case does not expose sufficient parallelism granularity and an inner space parallelized code has very poor performance. The same is the case with ICC's auto parallelizer; hence, we just show the sequential run time for ICC in this case. The comparison is also provided with gcc – where gcc was used to compile all the optimized codes. This is to show that ICC's loop transformations were not in any way interfering with our transformed code and that the relative benefits of our source-to-source transformation will be available when used with any sequential compiler.

5.2 Finite Difference Time Domain

The code is as shown in Fig. 6. ex, ey represent the electric fields in x and y directions, while hz represents the magnetic field. The code has four statements - three of them are 3-dimensional and one two dimensional and are nested imperfectly. The transformation framework finds three tiling hyperplanes (all in one band -



(a) Single core (with icc): $T = 10^5$

(b) Multi-core parallel (with icc): $N = 10^6, T = 10^5$

(c) Single core (with gcc): $T = 10^5$

(d) Multi-core parallel (with gcc): $N = 10^6, T = 10^5$

Figure 5. Imperfectly nested 1-d Jacobi stencil

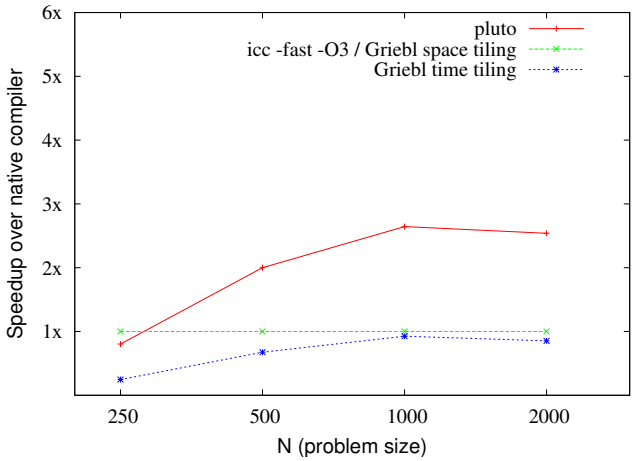
fully permutable). The results shown are for $nx = ny = 2000$ and $tmax = 512$. A tile size of 32 was used for each of three dimensions. Results are shown in Fig. 6. Note that the outer loop can be written as sequential and the inner loops are all parallel – this is the solution obtained by polyhedral scheduling-based techniques as well as ICC's auto parallelizer. Note that this does not fuse the inner loops; also, synchronization has to be done every time step. With our optimizer, all three dimensions are tiled and the loops are fused, and each processor executes a 3-d tile (which itself is a sequence of 3-d L1 tiles) and a synchronization is done only before and after it. The parallel results exhibit highly super-linear speedups. Note that here we have two degrees of pipelined parallelism – to exploit both, we need a tile wavefront of (1,1,1); however, to exploit one, we just need a wavefront of (1,1,0) (Sec. 4.4. The quality of code generated when using a wavefront of (1,1,1) to exploit only one degree of pipelined parallelism has an impact on performance.

5.3 LU decomposition

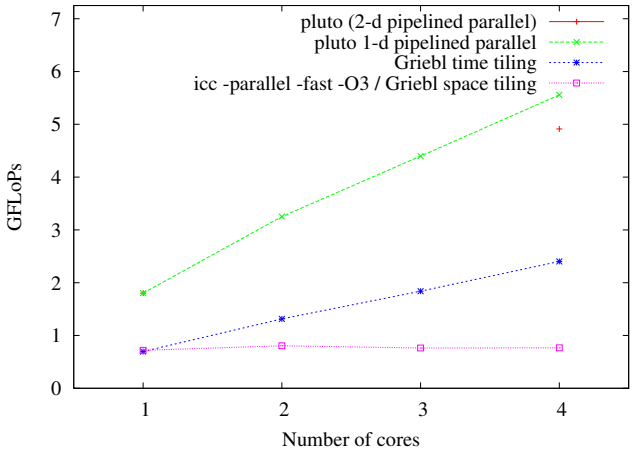
Three tiling hyperplanes are found – all belonging to a single band of permutable loops. Thus, there are two degrees of pipelined parallelism. Exploiting both degrees of pipelined parallelism requires a tile wavefront of (1,1,1) while exploiting only one requires (1,1,0). The code for the latter is likely to be less complex, however, has a lesser computation to communication ratio. Performance results on the quad core machine are shown in Fig. 10. A tile size of 32 was used for all three dimensions. The first statement though lower-dimensional is naturally sunk into a a 3-dimensional fully permutable space. ICC is unable to parallelize such code. Note that two degrees of parallelism are only meaningful for four cores or more, and only when the number of cores is not a prime number.

5.4 Matrix vector transpose

The MVT kernel is a sequence of two matrix vector transposes as shown in Fig. 9 (a). It is encountered in a time loop in Biconjugate gradient. The only inter-statement dependence is a non-uniform read/input on matrix A. The cost function bounding 8 leads to reduction of this dependence distance by fusion of the first MV with the permuted version of the second MV (note that $\phi(i) - \phi(i')$ for this dependence becomes 0 for both $c1$ and $c2$). This however leads to loss of synchronization-free parallelism, since, in the fused form, each loop carries a dependence. However, since these dependences are in the forward direction, the parallel code is generated corresponding to one degree of pipelined parallelism. Existing techniques, even if they consider input dependences, cannot automatically fuse the first MV with the permuted version of the second MV. Note that each of the matrix vector multiplies is



(a) Single core (L1 tiled) T=500



(b) Parallel: $Nx = Ny = 2000, T = 500$

Figure 7. 2-D FDTD

```
for (t=0; t<tmax; t++){
  for (j=0; j<ny; j++)
      ey[0][j] = exp(-t1);

  for (i=1; i<nx; i++) {
    for (j=0; j<ny; j++) {
      ey[i][j] = ey[i][j] - coeff1*(hz[i][j]-hz[i-1][j]);
    }
  }

  for (i=0; i<nx; i++) {
    for (j=1; j<ny; j++) {
      ex[i][j] = ex[i][j] - coeff1*(hz[i][j]-hz[i][j-1]);
    }
  }

  for (i=0; i<nx; i++) {
    for (j=0; j<ny; j++) {
      hz[i][j] = hz[i][j] -
        coeff2*(ex[i][j+1]-ex[i][j]+ey[i+1][j]-ey[i][j]);
    }
  }
}
```

Figure 6. 2-d FDTD

one strongly connected component. Hence, previous approaches are only able to extract synchronization-free parallelism from each of the MVs separately with a barrier between the two, giving up reuse on array A. Though Lim/Lam's approach does consider optimization between two adjacent SCCs through a near-neighbor constraint [30], it is a set-and-test approach and its automatability is not clear from the description. Fig. 10 shows the results for a problem size $N = 12000$. Note that both the optimized versions were tiled for the L1 cache. Fusion of ij with ij does not exploit reuse on matrix A, whereas the code that our tool comes up with performs best – it fuses ij with ji, tiles it and exploits a degree of pipelined parallelism. Note that this computation is memory-bandwidth bound and not likely to scale as well as the other compute-intensive kernels evaluated.

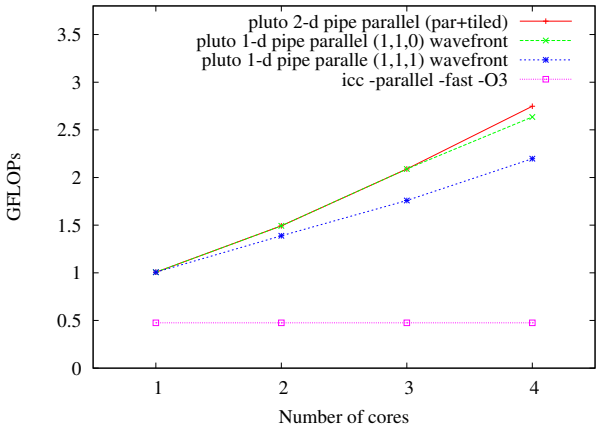


Figure 10. LU performance on a quad core: N=2000

5.5 3-D Gauss-Seidel successive over relaxation

The Gauss-Seidel computation allows tiling of all three dimensions after skewing the space loops with respect to time. Hence, two degrees of pipelined parallelism can be extracted after that. Our tool automatically finds the transformation (can be seen as the minimum amount of skewing in this case) to allow tiling of all three dimensions, followed by a tile space transformation to obtain a legal tile schedule. Fig. 12 shows the performance improvement achieved with the 2-d pipelined parallel space as well as 1-d: the latter is better in practice mainly due to simpler code. Due to the transformation applied, the inner loops have a number of min's and max's that are possibly affecting the absolute GFLoPs; again, separation of full and partial tiles will address this. In contrast, generating parallel code by hand is extremely tedious – time skewing adds complexity to the bounds, this coupled by the fact that the code has to be tiled at least for one level of local cache, and a 2-d pipelined parallel schedule of 3-d tiles is to be obtained.

Summary. All experiments show very high speedups with our approach, even with just single thread execution. Super-linear speedups are seen for almost all compute-intensive kernels considered here. Matrix vector transpose is memory-bandwidth bound;

```
for (k=0; k<N; k++) {
  for (j=k+1; j<N; j++)   {
    a[k][j] = a[k][j]/a[k][k];
  }
  for(i=k+1; i<N; i++)    {
    for (j=k+1; j<N; j++)   {
      a[i][j] = a[i][j] - a[i][k]*a[k][j];
    }
  }
}
```

(a) Original code

S1

$$\begin{bmatrix} c_1 \\ \mathbf{c_2} \\ c_3 \\ c_4 \\ c_5 \\ c_6 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} kT \\ jT \\ k \\ j \end{bmatrix}$$

S2

$$\begin{bmatrix} c_1 \\ \mathbf{c_2} \\ c_3 \\ c_4 \\ c_5 \\ c_6 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} kT \\ iT \\ jT \\ k \\ i \\ j \end{bmatrix}$$

c_2 is marked omp parallel

(b) 1-d pipelined parallel

```
#define S1(zT0,zT1,k,j) {a[k][j]=a[k][j]/a[k][k];}
#define S2(zT0,zT1,zT2,k,i,j)    {a[i][j]=a[i][j]-a[i][k]*a[k][j];}

/* Generated from lu.par110.cloog by CLooG v0.14.1 64 bits in 0.02s. */
for (c1=-1;c1<=floord(2*N-3,32);c1++) {
    lb = max(max(ceild(16*c1-15,32),ceild(32*c1-N+2,32)),0);
    ub = min(floord(32*c1+31,32), floord(N-1,32));
#pragma omp parallel for shared(c1,lb,ub,a) private (c2,c3,c4,c5,c6,i,j,k,l,m,n)
  for (c2=lb;c2<=ub;c2++) {
    for (c3=max(ceild(16*c1-16*c2-465,496),ceild(16*c1-16*c2-15,16));c3<=floord(N-1,32);c3++) {
      if (c1 == c2+c3) {
        for (c4=max(0,32*c3);c4<=min(min(32*c3+30,N-2),32*c2+30);c4++) {
          for (c5=max(32*c2,c4+1);c5<=min(N-1,32*c2+31);c5++) {
            S1(c1-c2,c2,c4,c5) ;
            for (c6=c4+1;c6<=min(32*c3+31,N-1);c6++) {
              S2(c1-c2,c1-c2,c2,c4,c6,c5) ;
            }
          }
        }
      }
      for (c4=max(0,32*c1-32*c2);c4<=min(min(32*c1-32*c2+31,32*c3-1),32*c2+30);c4++) {
        for (c5=max(32*c2,c4+1);c5<=min(N-1,32*c2+31);c5++) {
          for (c6=32*c3;c6<=min(32*c3+31,N-1);c6++) {
            S2(c1-c2,c3,c2,c4,c6,c5) ;
          }
        }
      }
      if ((-c1 == -c2-c3) && (c1 <= min(floord(32*c2+N-33,32),floord(64*c2-1,32)))) {
        for (c5=max(32*c1-32*c2+32,32*c2);c5<=min(32*c2+31,N-1);c5++) {
          S1(c1-c2,c2,32*c1-32*c2+31,c5) ;
        }
      }
    }
  }
}
```

(c) LU (1-d pipelined parallel + locality tiled) (tile size 32), cloog -f4 -l7

$$S1: \begin{bmatrix} c_1 \\ \mathbf{c_2} \\ \mathbf{c_3} \\ c_4 \\ c_5 \\ c_6 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} kT \\ jT \\ k \\ j \end{bmatrix} \qquad S2: \begin{bmatrix} c_1 \\ \mathbf{c_2} \\ \mathbf{c_3} \\ c_4 \\ c_5 \\ c_6 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} kT \\ iT \\ jT \\ k \\ i \\ j \end{bmatrix}$$

c2, c3 are marked omp parallel

(d) 2-d pipelined parallel

Figure 8. LU decomposition (3-d tiling)

hence the lower speedup. A significant portion of the large difference in icc/gcc performance is due to vectorization performed by ICC. Note that the tile sizes were not optimized through any extensive search or a model. Tiling at multiple levels (L2) and register tiling followed by full unrolling of the register tile has been shown to give further significant benefit [28] and is complementary to our system. Integration of these techniques is needed to achieve performance closer to the machine peak and is in progress. The performance of the optimized stencil codes through our framework has already surpassed that of hand optimized versions [24], where space inefficient versions of the stencil computations were used to keep the code perfectly nested (adding a redundant time dimension to the array), and make tiling by hand feasible.

As mentioned earlier, there is a natural path to address issues in order to move achieved performance closer to the machine peak. This includes integration of a model-driven empirical optimization framework to select good tile sizes, separation of full and partial tiles, and unrolling of the last-level tile (register tile). We are also addressing the fusion/parallelization trade-off that is key to effectively optimizing longer sequences of loop nests.

6. Related work

Iteration space tiling [23, 47, 48, 35] 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 [38, 35, 9, 49, 22, 21, 37]. But these studies were restricted to perfectly nested loops with uniform dependences or had other restrictions that limited their applicability to simple codes. Some specialized works [40, 50] also exist on tiling a restricted class of imperfectly nested loops.

Loop parallelization has been studied extensively. The reader is referred to the survey of Boulet et al.[10] for a detailed summary of older parallelization algorithms which accepted restricted input and/or were based on weaker dependence abstractions than exact polyhedral dependences. Kelly and Pugh's algorithm finds

```
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        x1[i] = x1[i] + a[i][j] * y_1[j];
    }
}

for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        x2[i] = x2[i] + a[j][i] * y_2[j];
    }
}
```

(a) Original code

```
#define S1(i,j) {x1[i]=a[i][j]*y_1[j]+x1[i];}
#define S2(i,j) {x2[i]=a[j][i]*y_2[j]+x2[i];}

/* Generated from PLuTo-produced CLooG
 file by CLooG v0.14.1 64 bits in 0.00s. */
for (c1=0;c1<=N-1;c1++) {
  for (c2=0;c2<=N-1;c2++) {
    S1(c1,c2) ;
    S2(c2,c1) ;
  }
}
```

(b) Optimized without tiling

```
#define S1(zT0,zT1,zT2,zT3,i,j) {x1[i]=a[i][j]*y_1[j]+x1[i];}
#define S2(zT0,zT1,zT2,zT3,i,j) {x2[i]=a[j][i]*y_2[j]+x2[i];}
/* Generated from PLuTo-produced CLooG file by CLooG v0.14.1 64 bits in 0.04s. */
for (c1=-1;c1<=floord(N-1,512);c1++) {
        lb=max(ceild(1024*c1-N+1,1024),0);
        ub=min(floord(1024*c1+1023,1024),floord(N-1,1024));
#pragma omp parallel for shared(c1,lb,ub) private(c2, c3, c4, c5, c6, c7)
        for (c2=lb; c2<=ub; c2++) {
      for (c3=max(0,32*c1-32*c2);c3<=min(32*c1-32*c2+31,floord(N-1,32));c3++) {
        for (c4=32*c2;c4<=min(32*c2+31,floord(N-1,32));c4++) {
          for (c5=32*c3;c5<=32*c3+31;c5++) {
            for (c6=32*c4;c6<=32*c4+31;c6++) {
              S1(c1-c2,c2,c3,c4,c5,c6) ;
              S2(c2,c1-c2,c4,c3,c6,c5) ;
            }
          }
        }
      }
    }
}
/* End of CLooG code */
```

(c) 1-d pipelined parallel tiled

$$S1: \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \qquad S2: \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$

(d) without tiling

$$S1: \begin{bmatrix} c_1 \\ \mathbf{c_2} \\ c_3 \\ c_4 \\ c_5 \\ c_6 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} iT' \\ jT' \\ iT \\ jT \\ i \\ j \end{bmatrix} \qquad S2: \begin{bmatrix} c_1 \\ \mathbf{c_2} \\ c_3 \\ c_4 \\ c_5 \\ c_6 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} iT' \\ jT' \\ iT \\ jT \\ i \\ j \end{bmatrix}$$

(e) with tiling and pipelined parallelism

Figure 9. Matrix vector transpose

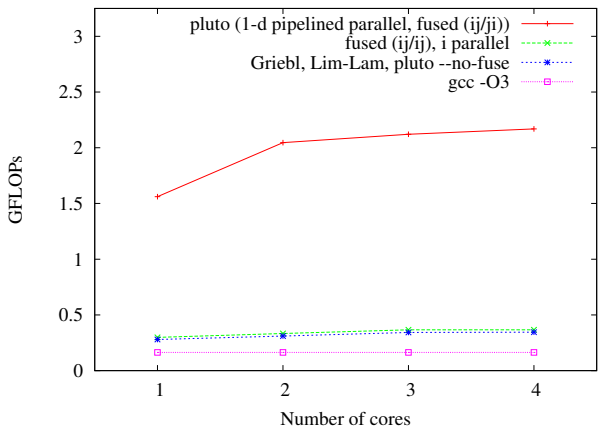


Figure 11. MVT performance on a quad core: N=48000

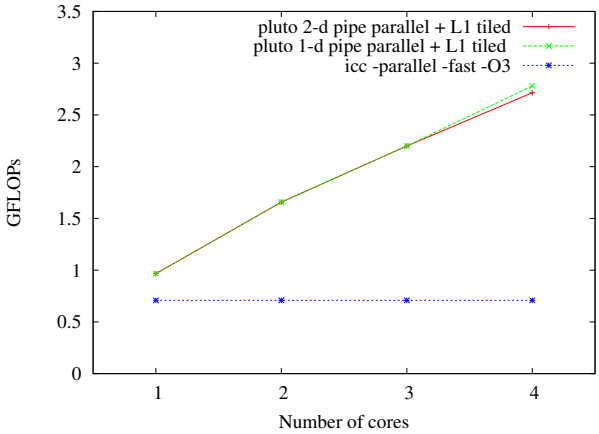


Figure 12. 3-D Gauss Seidel on a quad core: $N_x = N_y = 2000$; T=1000

one dimension of parallelism for loops with arbitrary nestings [26]. Their program transformation framework is search-based and includes only loop permutations and reversals, but not loop skewing.

Scheduling with affine functions using faces of the polytope by application of the Farkas algorithm was first proposed by Feautrier [14]. Feautrier explored various possible approaches to

obtain good affine schedules that minimize latency. Using reasonable heuristics usually yields very good solutions. Overall, Feautrier's classic works [14, 15] were geared towards finding minimum latency schedules and maximum fine-grained parallelism, as opposed to tilability for coarse-grained parallelization with minimized communication and improved locality. As argued in [8], using such schedules as one of the loops for parallel code generation does not necessarily optimize communication and locality. Hence, a scheduling-based approach may not lead to good tiling (as demonstrated through experimental results also). Several works [19, 7, 11, 32] make use of such schedules.

Ahmed et al. [4] proposed a framework for data locality optimization of imperfectly nested loops for sequential execution on a uniprocessor. Their approach first determines the embedding for each statement into a product space, which is then optimized for locality as a perfectly nested loop by using another transformation matrix to achieve permutability. The embedding function and the transformation is sought to minimizing reuse distances, based on a heuristic. The reuse distances in the target space for some dependences are set to zero (or constant), with the goal of obtaining solutions to the embedding function and transformation matrix coefficients. The decision of the number and specific choice for these dependences is made using heuristics. The robustness and automatability of the approach is unclear from the description; to the best of our knowledge no automated tool using the approach has been reported.

Lim and Lam [31, 30] proposed an affine partitioning framework that identifies outer parallel loops (communication-free space partitions) and permutable loops (pipelined parallel or tilable loops) with the goal of maximizing the degree of parallelism and minimizing the order of synchronization. They employ the same machinery for blocking [29]. Several (infinitely many) solutions equivalent in terms of the criterion they optimize for result from their algorithm, and these significantly differ in communication cost and locality; no metric is provided to differentiate between these solutions. As argued in [8] and demonstrated through experimental results, without a cost function, the solutions obtained even for simple inputs may be unsatisfactory – both with respect to performance and code generation. Also, they do not discuss parallel and tiled code generation for the general case. Our approach addresses all of these aspects.

Griebl [19] presents an integrated framework for optimizing locality and parallelism with space and time tiling, and is one of the few works that addresses both locality and coarse-grained parallelism through tiling. However, tiling is treated as a post-processing step using a scheduling-allocation approach, as opposed to being integrated into the transformation framework. Though Griebl's approach enables time tiling by using a forward-communication-only (FCO) allocation with an existing schedule, fixing schedules as loops can lead to sub-optimal solutions with respect to communication, locality, and target code complexity [8]. Also, loop fusion (both within an SCC or across SCCs) is not addressed, since a schedule specifying maximum parallelism need not interleave operations of different statements.

Cohen et al., Girbal et al. [11, 17] proposed and developed a powerful framework (URUK/WRAP-IT) to compose and apply sequences of transformations in a semi-automatic fashion. Transformations are applied automatically, but specified manually by an expert. Pouchet et al. [32] searches the space of transformations (one-dimensional schedules) to find good ones through iterative optimization by employing performance counters. On the other hand, our approach is model-driven and fully-automatic and directly obtains good transformations without search; codes that need multi-dimensional schedules are naturally handled. Empirical and iterative optimization is still required to choose transforms that work best in practice. Effective determination of register-tile sizes and unroll factors for transformed whole-programs are likely best optimized through empirical search. A combination of our affine transformation framework and empirical search in a smaller space is an attractive approach that we intend to explore in the near future. Alternatively, more powerful cost models like those based on computing Ehrhart polynomials [44] may be employed once transformations in a smaller space are enumerated.

Our tiled code generation scheme uses Ancourt and Irigoin's [5] well-known approach to just specify domains with fixed tile sizes, but combines it with CLooG's support for scattering functions. Goumas et al. [18] reported an alternate (to [5]) tiled code generation scheme to avoid the inefficiency involved in using Fourier-Motzkin (FM) – however, this is no longer an issue as CLooG uses polylib completely avoiding FM. Note that the work of Goumas et al. generates code still with the original basis used to scan tiles and the tile space.

7. Conclusions

We have presented the design and implementation of a fully automatic polyhedral source-to-source program optimizer that can simultaneously optimize sequences of arbitrarily nested loops for parallelism and locality. Through this work, we have shown the practicality and promise of automatic transformation beyond what is possible by current production compilers. Experimental results show very significant speedup for single core and parallel execution on multi-cores. Our system also leaves a lot of flexibility for future optimization, mainly iterative and empirical and/or through more sophisticated cost models, and promise to achieve performance close to or beat manually developed codes.

The transformation system presented here is not just applicable to C or Fortran code, but to any input language from which polyhedra can be extracted. Since our entire framework works on the polyhedral abstraction, only the frontend (scanner/parser and dependence tester) needs to be adapted to accept a future high-productivity language. It could be applied for example to very high-level languages like MATLAB or domain-specific languages to generate efficient parallel code.

Acknowledgments

We would like to acknowledge Cédric Bastoul (Paris-Sud XI University, Orsay, France) and all other contributors to CLooG for this code generation masterpiece. We would also like to thank Martin Griebl and team (FMI, Universität Passau, Germany) for the LooPo infrastructure. 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] LooPo - Loop parallelization in the polytope model. http://www.fmi.uni-passau.de/loopo.

[3] PIP: The Parametric Integer Programming Library. http://www.piplib.org.

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

[5] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *PPoPP'91*, pages 39–50, 1991.

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

[7] C. Bastoul and P. Feautrier. More legal transformations for locality, Aug. 2004.

[8] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Affine transformations for communication minimal parallelization and locality optimization for arbitrarily-nested loop sequences. Technical Report OSU-CISRC-5/07-TR43, The Ohio State University, May 2007.

[9] P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling? *Integration, the VLSI Journal*, 17(1):33–51, 1994.

[10] P. Boulet, A. Darte, G.-A. Silber, and F. Vivien. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Computing*, 24(3–4):421–444, 1998.

[11] A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ACM ICS*, pages 151–160, June 2005.

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

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

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

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

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

[17] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Intl J. of Parallel Programming*, 34(3):261–317, June 2006.

[18] G. Goumas, M. Athanasaki, and N. Koziris. Code Generation Methods for Tiling Transformations. *Journal of Information Science and Engineering*, 18(5):667–691, Sep. 2002.

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

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

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

[22] K. Hogstedt, L. Carter, and J. Ferrante. Selecting tile shape for minimal execution time. In *SPAA*, pages 201–211, 1999.

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

[24] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yellick. Implicit and explicit optimization for stencil computations. In *ACM MSPC*, 2006.

[25] W. Kelly and W. Pugh. A unifying framework for iteration reordering transformations. Technical Report CS-TR-3430, Dept. of Computer Science, University of Maryland, College Park, 1995.

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

[27] 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.

[28] D. Kim, L. Renganarayanan, M. Strout, and S. Rajopadhye. Multi-level tiling: 'm' for the price of one. In *SC'07*, 2007.

[29] 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.

[30] A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *ACM ICS*, pages 228–237, 1999.

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

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

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

[34] F. Quilleré, S. V. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *Intl J. of Parallel Programming*, 28(5):469–498, 2000.

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

[36] L. Renganarayana, D. Kim, S. Rajopadhye, and M. M. Strout. Parameterized tiled loops for free. In *PLDI'07*, pages 405–414, 2007.

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

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

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

[40] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 215–228, 1999.

[41] N. Vasilache. *Scalable Program Optimization Techniques in the Polyhedral Model*. PhD thesis, Université de Paris-Sud, INRIA, Futurs, Sept. 2007.

[42] N. Vasilache, C. Bastoul, and A. Cohen. Polyhedral code generation in the real world. In *ETAPS CC'06*, pages 185–201, Mar. 2006.

[43] N. Vasilache, C. Bastoul, S. Girbal, and A. Cohen. Violated dependence analysis. In *ACM ICS*, June 2006.

[44] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Analytical computation of ehrhart polynomials: enabling more compiler analyses and optimizations. In *CASES*, pages 248–258, September 2004.

[45] R. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). In *Proc. Supercomputing '98*, 1998.

[46] R. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing Journal*, 2000.

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

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

[49] J. Xue. Communication-minimal tiling of uniform dependence loops. *JPDC*, 42(1):42–59, 1997.

[50] Q. Yi, K. Kennedy, and V. Adve. Transforming complex loop nests for locality. *J. Supercomput.*, 27(3):219–264, 2004.

[51] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. A. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *PLDI'03*, pages 63–76, 2003.

[52] K. Yotov, K. Pingali, and P. Stodghill. Think globally, search locally. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 141–150, 2005.