PLuTo: 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

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 that takes 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 system show very high speedups for local and parallel execution on multi-cores over state-of-the-art compiler frameworks from the research community as well as the best native compilers. 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 microarchitecture are increasingly towards larger number of processing elements on a single chip. This has led to parallelism and multi-core architectures becoming mainstream. In addition, several specialized parallel architectures (accelerators) like the Cell processor and General-Purpose GPUs have emerged. 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 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 called 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 complex loop transformations in a completely mathematical setting relying on 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.

The task of program optimization (often for parallelism and locality) in the polyhedral model 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 code for the transformed program. Significant advances were made in the past decade on dependence analysis [15, 14, 35] and code generation [26, 21] in the polyhedral model, but the approaches suffered from scalability challenges. Recent advances in dependence analysis [46] and more importantly in code generation [37, 6, 45] 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. The key missing step is the absence of a scalable and practical approach for automatic transformation for parallelization and locality. Our work addresses this problem by developing a compiler framework that enables end-to-end fully automatic parallelization and locality optimization.

Tiling is a key transformation 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 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. One of the key aspects of our transformation framework is to find good ways of performing tiling.

Existing automatic transformation frameworks [31, 30, 29, 20] have one or more drawbacks or restrictions that limit their effectiveness. A significant problem is the lack of a realistic cost function to choose among the large space of legal transformations that

are suitable for coarse-grained parallel execution, as is used in practice with manually developed/optimized parallel applications. Most previously proposed approaches also do not consider locality and parallelism together. 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 end-to-end design and implementation of a practical parallelizer and locality optimizer in the polyhedral model. Finding good ways to tile for parallelism and locality directly directly through an affine transformation framework is the central idea. Our approach is thus a departure from scheduling-based approaches in this field [16, 17, 13, 20] as well as partitioning-based approaches [31, 30, 29] (due to incorporation of more concrete optimization criteria), however, is build on the same mathematical foundations and machinery. 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; this issue has not been addressed previously. We also 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 [47, 51], and ATLAS is well-known. 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. Also, since our transformation system operates entirely in the polyhedral abstraction, it is not just limited to C or Fortran code, but could accept any high-level language from which polyhedral domains can be extracted.

The rest of this report is organized as follows. Section 2 provides mathematical background on the polyhedral model and affine transformations. Section 3 provides an overview of our theoretical framework that drives automatic transformation. Sec. 4 and Section 5 discuss the design of our system, mainly focusing on techniques for generation of tiled and shared memory parallel code from transformations found. Finally, Sec 6 provides experimental results from the implemented system. Section 7 discusses related work and conclusions are presented in Section 8.

2. Background and Notation

This section provides background on the polyhedral model. 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 $\mathbf{h}.\vec{x} = k$, for $k \in \mathbf{Z}$, forms an affine hyperplane.*

In other words, a hyperplane is a higher dimensional analog of a (2-d) plane in three-dimensional space. 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. 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). *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.*

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 the 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 of $\mathcal{D}$:

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

The non-negative constants λ_k are referred to as Farkas multipliers. For a detailed proof, see Schrijver [42].

Polyhedral representation of programs. Given a program, each dynamic 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. Whenever the loop bounds are linear combinations of outer loop indices and program parameters (typically, symbolic constants representing problem sizes), the set of iteration vectors belonging to a statement define a polytope. Let $\mathcal{D}_S$ represent the polytope and its dimensionality be m_S. Let $\vec{p}$ be the vector of program parameters.

2.2 Polyhedral Dependences

Our dependence model is of exact affine dependences and same as the one used in [16, 30, 12, 46, 34]. Dependences are determined precisely, but we consider all dependences including anti (write-after-read), output (write-after-write) and input (read-after-read) dependences, i.e., input code does not require conversion to single-assignment form. Non-affine accesses or dynamic control can be handled conservatively; however, how such a conservative approximation is derived for general programs is not a part of this report. The Polyhedral Dependence Graph (PDG) 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 e. 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).

$$\mathcal{P}_e \equiv \left[\begin{array}{c} \mathcal{D}^{\mathcal{P}}{}_{\vec{s}} \quad\quad \mathcal{D}^{\mathcal{P}}{}_{\vec{t}} \\ \hline \mathsf{h_e} \end{array}\right] \left[\begin{array}{c} \vec{s} \\ \vec{t} \\ \vec{p} \\ 1 \end{array}\right] \left[\begin{array}{c} \geq 0 \\ \hline = 0 \end{array}\right] \tag{2}$$

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* [16]. The last access condition is not necessary though; in general, the equalities can be used to eliminate variables from $\mathcal{P}_e$. In the rest of this section, it is assumed for convenience that $\vec{s}$ can be completely eliminated using the h_e, being substituted by $h_e(\vec{t})$.

2.3 Statement-wise Affine Transforms

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

$$\begin{aligned} \phi_{s_k} &= \left[f_1 \; \ldots \; f_{m_{S_k}}\right] \left(\; \vec{i} \; \right) + f_0 \\ &= \mathbf{f}_{S_k}\vec{i} + f_0, \text{where } \mathbf{f}_{S_k} = [f_1, \ldots, f_{m_{S_k}}], \;\; f_i \in \mathbf{Z} \end{aligned} \tag{3}$$

A multi-dimensional affine transformation for a statement can now be represented by a matrix with each row being an affine hyperplane. If such a transformation matrix has full column rank, it completely specifies when and where an iteration executes. The total number of rows in the matrix may be much larger as some special rows, *splitters*, may represent unfused loops at a level. Consider the

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    S1: C[i,j] = 0;

for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    for (k=0; k<n; k++)
      S2: C[i,j] = C[i,j]
          + A[i,k]*B[k,j]

for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    for (k=0; k<n; k++)
      S3: D[i,j] = D[i,j]
          + E[i,k]*C[k,j]
```

	$S1$			$S2$			$S3$		
	i	j	$const$	ij	k	$const$	ij	k	$const$
c_1	1	0	0	10	0	0	00	1	0
c_2	0	1	0	01	0	0	01	0	0
c_3	0	0	0	00	0	0	00	0	1
c_4	0	0	0	00	1	0	10	0	0
c_5	0	0	0	00	1	0	10	0	0

```
for (c1=0; c1<n; c1++)
  for (c2=0; c2<n; c2++)
    C[c1,c2] = 0;
    for (c5=0; c5<n; c5++)
      C[c1,c2] = C[c1,c2] +
          A[c1,c5]*B[c5,c2]
    for (c5=0; c5<n; c5++)
      D[c5,c2] = D[c5,c2] +
          E[c5,c1] * C[c1,c2]
```

Figure 1. Statement-wise transformation and corresponding transformed code

code in Fig. 1 for example. Such transformations capture the fusion structure as well as compositions of permutation, reversal, relative shifting, and skewing transformations. This representation for transformations has been used by many researchers [17, 25, 12, 18], and directly fits with scattering functions that a code generator like CLooG [6] supports.

3. Overview of Automatic Transformation Approach

In this section, we give an overview of our theoretical framework for automatic transformation. Full details are available in another report [8].

3.1 Legality of tiling multiple domains with affine dependences

LEMMA 2. *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 $e \in E$:*

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

The above is a generalization of the classic condition proposed by Irigoin and Triolet [23] (as $h^T.R \geq \mathbf{0}$) for the legality of tiling a single domain. The tiling of a statement's iteration space by a set of 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. The above is a generalization to multiple iteration domains with affine dependences, with possibly different dimensionalities or corresponding to imperfectly nested input.

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). Then, $\{\phi^1_{s_1}, \phi^1_{s_2}, \ldots, \phi^1_{s_k}\}, \{\phi^2_{s_1}, \phi^2_{s_2}, \ldots, \phi^2_{s_k}\}$ represent rectangularly tilable loops in the transformed space. A tile can be formed by aggregating a group of hyperplane instances along $\phi^1_{s_i}$ and $\phi^2_{s_i}$. Due to (4), 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. Hence, any $\phi^j_{S_1}, \phi^j_{S_2}, \ldots, \phi^j_{S_n}$ that is a solution to (4) represents a common dimension (for all statements) in the transformed space with both inter and intra-statement affine dependences in the forward direction along it.

Partial tiling at any depth. 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., tiling at that level is legal. In the rest of this section, we use the term *affine transform* (with property (4)) and *tiling hyperplane* interchangeably.

3.2 Cost function, bounding approach and minimization

Consider the following affine form δ_e:

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

The affine form $\delta_e(\vec{t})$ is very significant. This function is the number of hyperplanes the dependence e traverses along the hyperplane normal ϕ. If ϕ is used as a space loop to generate tiles for parallelization, this function is a factor in the communication volume. On the other hand, if ϕ is used as a sequential loop, it gives us a measure of the reuse distance. 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 the bound, the same for cache misses at L1/L2 tile edges, or L1 cache loads for a register tile. Of particular interest is, if this function can be reduced to a constant amount or zero (free of a parametric component) by choosing a suitable direction for ϕ: if this is possible, then that particular dependence leads to constant boundary communication or no communication (respectively) for this hyperplane.

An attempt to minimize the above cost function ends up in an objective non-linear in 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 ends up with such a form when a dependence is not uniform or for an inter-statement dependence. The difficulty can be overcome by using a bounding function approach that allows the application of Farkas Lemma and casting the objective into an ILP formulation. Since the loop variables themselves are bounded by affine functions of the parameters, one can always find an affine form in the program parameters, $\vec{p}$, that bounds $\delta_e(\vec{t})$ for every dependence edge e, i.e., there exists $v(\vec{p}) = \mathbf{u}.\vec{p} + w$, such that

$$\begin{aligned} \phi_{s_i}(\vec{t}) - \phi_{s_j}(h_e(\vec{t})) \quad &\leq \quad v(\vec{p}), \ \vec{t} \in \mathcal{P}_e, \quad \forall e \in E \\ \text{i.e.,} \qquad v(\vec{p}) \ - \ \delta_e(\vec{t}) \quad &\geq \quad 0, \ \vec{t} \in \mathcal{P}_e, \quad \forall e \in E \end{aligned} \tag{6}$$

Such a bounding function approach was first used by Feautrier [16], but for a different purpose – to find minimum latency schedules. Now, Farkas Lemma can be applied to (6).

$$v(\vec{p}) - \delta_e(\vec{t}) \quad \equiv \quad \lambda_{e0} + \sum_{k=1}^{m_e} \lambda_{ek} \mathcal{P}_e^k, \quad {\lambda_{\mathbf{ek}}}^T \geq \vec{0}$$

where $\mathcal{P}_e^k$ is a face of $\mathcal{P}_e$. Coefficients of each of the iterators in $\vec{i}$ and parameters in $\vec{p}$ on the LHS and RHS can be gathered and equated, to obtain linear equalities and inequalities entirely in coefficients of the affine mappings for all statements, components of row vector $\mathbf{u}$, and w. The ILP system comprising the tiling legality constraints from (4) and the bounding constraints can be at once solved by finding a lexicographic minimal solution with $\vec{u}$ and w in the leading position. Let $\mathbf{u} = (u_1, u_2, \ldots u_k)$.

$$\text{minimize}_{\prec} \{u_1, u_2, \ldots, u_k, w, \ldots, c_i's, \ldots\} \tag{7}$$

Finding the lexicographic minimal solution is within the reach of the Simplex algorithm and can be handled by the Parametric Integer Programming (PIP) software [14]. Since the program parameters are quite large, their coefficients are minimized with the highest priority. The solution gives a hyperplane for each statement.

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 as many independent solutions (for a statement) as the dimensionality of its domain. Hence, once a solution is found, we augment the ILP formulation with new constraints that make sure of linear independence with solutions already found. This is done by constructing the orthogonal sub-space [33, 28, 9] of the transformation rows found so far (H_S) and forcing a non-zero component in it for the next solution.

$$H_S^{\perp} = I - H_S^T \left(H_S H_S^T \right)^{-1} H_S \tag{8}$$

Unified view of communication and locality optimization The best possible solution to (7) is with $(u = 0, w = 0)$, which is a hyperplane that has no dependence components along its normal – this is a fully parallel loop requiring no synchronization if 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 when they exist; these are followed by a set of hyperplanes requiring constant boundary communication ($u = 0; w > 0$). In the worst case, we have a hyperplane with $u > 0, w \geq 0$ resulting in long communication from non-constant dependences; such solutions are pushed to inner levels. Hence, all degrees of parallelism are found in the order of their preference. From the point of view of data locality, since the same hyperplanes used to scan the tile space scan points in a tile, cache misses at tile boundaries (that are equivalent to communication along processor tile boundaries) are minimized.

Algorithm 1 Affine transformation algorithm

INPUT Dependence polyhedra: $\mathcal{P}_e$, $e \in E$

1: **for** each dependence $e \in E$ **do**
2: Build legality constraints: apply Farkas Lemma on $\phi(\vec{t}) - \phi(h_e(\vec{t})) \geq 0$ under $\vec{t} \in \mathcal{P}_e$
3: Build communication volume/reuse distance bounding constraints: apply Farkas Lemma to $v(\vec{p}) - (\phi(\vec{t}) - \phi(h_e(\vec{t}))) \geq 0$ under $\vec{t} \in \mathcal{P}_e$
4: Aggregate constraints from both into $C_e(i)$
5: **end for**
6: **repeat**
7: $C \leftarrow \bigcup_{e \in E} C_e(i)$
8: Find lexicographic minimal solution for $(\mathbf{u}, w)$ and iteratively find as many independent solutions to C as possible
9: **if** no solutions were found **then**
10: Cut dependences between SCCs in the dependence graph
11: **end if**
12: Remove edges carried by solutions of Step 8/10 from E
13: **until** $H_{S_i}^{\perp} = \mathbf{0}$ for each S_i and $E = \emptyset$

OUTPUT A transformation matrix for each statement

Outer space and inner time. By minimizing $\phi(\vec{t}) - \phi(\vec{s})$ as we find hyperplanes from outermost to innermost, we push dependence carrying to inner loops, at the same time ensuring that the new loops have non-negative dependence components (to the extent possible) so that they can be tiled for locality and pipelined parallelism can be extracted in the presence of forward space dependences. If the outer loops are used as space (how many ever desired, say k), and the rest are used as time, communication in the processor space is minimal as the outer space loops are the k best ones. Whenever the loops are tiled, they result in coarse-grained parallelism as well as better reuse within a tile.

Fusion using hyperplanes Fusion across multiple iteration spaces that are weakly connected, as in sequences of producer-consumer loops is also enabled. Since the hyperplanes do not include coefficients for program parameters, a solution found corresponds to a fine-grained interleaving of different statement instances at that level (Fig. 1).

The algorithm is summarized as Algorithm. 1. Dependences from previously found hyperplanes are not removed as independent tiling hyperplanes are found (in Step 8) unless they have to be (Step 12) to allow finding the next band of tiling hyperplanes.

4. Design

We have implemented our algorithm to transform C/Fortran code completely automatically. Fig. 2 shows the entire tool-chain. We used the scanner, parser and dependence tester from the LooPo infrastructure [2]. LooPo is a polyhedral source-to-source transformation system that includes implementations of various polyhedral analyses and transformations from the literature. We used PipLib 1.3.3 [3, 14] as the ILP solver and CLooG 0.14.1 (with 64 bits) for code generation. Our tool takes as input dependence polyhedra from LooPo's dependence tester. Flow, anti and output dependences are considered for legality as well as the bounding function, while input dependences can optionally be considered for the bounding objective. We have also integrated the annotation-based transformation system of Norris et al. [32] to perform some syntactic transformations on the code generated from CLooG as a post-processing; these include register tiling and unrolling and scalar replacement.

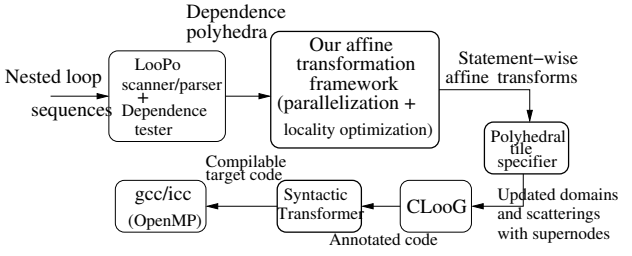


Figure 2. Our source-to-source transformation system

4.1 Handling input dependences

Input dependences need to be considered for optimization in many cases as reuse can be exploited by minimizing them. Clearly, legality (ordering between dependent RAR iterations) need not be preserved. We thus do not add legality constraints (4) for such dependences, but consider them for the bounding objective function. Since input dependences can be allowed to have negative components in the transformed space, they need to be bounded from both above and below. For every, $\mathcal{P}_e^R$ corresponding to a input dependence, we have the constraints:

$$\begin{aligned} \left| \phi_{s_j}(\vec{t}) - \phi_{s_i}(\vec{s}) \right| &\leq v(\vec{p}), \ \langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e^R \\ \text{i.e.,} \quad \phi_{s_j}(\vec{t}) - \phi_{s_i}(\vec{s}) &\leq v(\vec{p}), \ \langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e^R \\ \phi_{s_i}(\vec{s}) - \phi_{s_j}(\vec{t}) &\leq v(\vec{p}), \ \langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e^R \end{aligned}$$

4.2 Single-pass polyhedral tiling vs. post-transformation tiling

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 are generally referred to as tiling. Our approach models tiling in the transformation framework by finding affine transformations that

make rectangular tiling in the transformed space legal. 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 rectangularly tiled. Hence, the transformation framework guarantees legality of rectangular tiling in the new space. The final generation of tiled loops can be done in two ways broadly, (1) directly through the polyhedral code generator itself in one pass itself, or (2) as a post-pass on the abstract syntax tree generated after applying the transformation. Each has its merits and both can be combined too. Tiled code generation with parametric tile sizes within the polyhedral model was recently addressed by Renganarayana et al. [39]. Their approach only handles single domains that are rectangularly blockable. While the techniques could be extended to tile loops in an abstract syntax tree (AST) at any level, it falls into the second category, and we show that tiling the transformed AST is less powerful than generating the transformed-cum-tiled code in one pass through a code generator.

For transformations that possibly lead to imperfectly nested code, polyhedral tiling is a natural way to get tiled code from the code generator in one pass guaranteeing legality. Consider the code in Fig. 4(a) for example. If code is generated by just applying the transformation first, we get code shown in Fig. 4(b). Even though the transformation framework obtained two tiling hyperplanes, the transformed code in Fig. 4(b) has no 2-d perfectly nested kernel. Doing a simple unroll-jam of the imperfect loop nest is illegal in this case; hence, straightforward 2-d syntactic tiling violates dependences. The legality of syntactic tiling or unroll/jam (for register tiling) of such loops cannot be reasoned about in the target AST easily since once we obtain the transformed code, we are outside of the polyhedral model, unless advanced techniques like re-entrance [44] are used. Even when re-entrance is used to reason about legality through dependence analysis on the target AST, such an approach would miss ways of tiling that are possible by reasoning about the obtained tiling hyperplanes on original domains itself – we propose an approach to accomplish the latter which is the subject of Section 5. For example, for the code in Fig. 4, 2-d tiled code can be generated in one pass, both applying the transformation as well as accomplishing tiling.

In other cases when there exists a perfectly nested band of loops in the AST that carry most of the computation, it is easy to do multi-level parametric tiling with existing techniques [39, 27]; that would require another pass of CLooG on the AST and lead to much simpler code with separation of full and partial tiles, allowing unrolling for ILP, register reuse and avoiding max's and min's in the inner loop bounds for the core of the computation. Alternatively, such bands in the target AST can be tiled efficiently syntactically too.

4.3 Syntactic transformations

We have also integrated an annotation-based transformation system to perform syntactic transformations on the code generated from CLooG as a post-processing; these include syntactic tiling of innermost perfect nests in the AST, which as mentioned in some cases is more efficient, register tiling followed by unrolling, scalar replacement, and scalar bound replacement and hoisting to facilitate auto-vectorization. Such syntactic transformations fit nicely as a post-pass on code generated from CLooG since the automatically generated code has a known and fixed structure making it amenable to syntactic treatment. Also, for transformations like syntactic tiling and unroll-jamming, the legality is guaranteed by the transformation framework. In this paper, we do not discuss any further on how exactly these transformations are performed and the corresponding performance improvement. They are non-trivial for non-rectangular iteration spaces for example, or when a sequence of those has to be composed. The complementary benefits of syntactic tiling will be reported in future, and is not the focus of this paper. However, a preview of the potential performance improvement is provided for a kernel in the experimental evaluation section.

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

In this section, we describe how tiled code is generated from transformations found in the previous section. This is a key step in generation of high performance code.

We first give a brief description of CLooG. CLooG can scan a union of polyhedra, and optionally, under a new global lexicographic ordering specified as through 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 system. 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 instances are interleaved). CLooG uses PolyLib (which in turn uses the efficient Chernikova algorithm) for its core polyhedral operations, and the code generated is far more efficient than older code generators based on Fourier-Motzkin variable elimination, e.g. Omega Codegen [35] or LooPo's internal code generator [21, 20]). Also, code generation time and memory utilization are much lower, allowing code generation to be feasible for hundreds of statements with a number of free parameters without 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 paper.

We now 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 by incorporating [39, 27] to make it convenient for iterative/empirical optimization.

5.1 Tiles under a transformation

Our approach to tiling is to specify a modified higher dimensional domain and specify transformations for what would be the tile space loops in the transformed space. Consider a very simple example: a two-dimensional loop nest with original iterators: i and j. Let the transformation found be $c_1 = i$, and $c_2 = i+j$, with c_1 and c_2 both found in Step 8 of Algorithm 1; hence, they can be blocked leading to 2-d tiles. We would like to obtain target 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 like that proposed by Ancourt and Irigoin [5]. The tile space and intra tile loop scattering functions are specified as follows:

Domain	Scattering
$0 \leq i \leq N-1$	$c_1 T = iT$
$0 \leq j \leq N-1$	$c_2 T = iT + jT$
$0 \leq i - 32iT \leq 31$	$c_1 = i$
$0 \leq (i+j) - 32(iT+jT) \leq 31$	$c_2 = i + j$
$(c_1 T, c_2 T, c1, c2)$	$\leftarrow scatter(iT, jT, i, j)$

$c_1 T$ and $c_2 T$ 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

Algorithm 2 Tiling for multiple stmts under transformations

INPUT Statement-wise hyperplanes: $\phi_S^i, \phi_S^{i+1}, \ldots, \phi_S^{i+k-1}$ expressed as affine functions of corresponding original iterators $\vec{i}_S$ to tile; Original domains: $\mathcal{D}_\mathcal{S}$; Tile sizes: $\tau_i, \tau_{i+1}, \ldots, \tau_{i+k-1}$

1: /* Update the domains */
2: **for** each statement S **do**
3: **for** each $\phi_S^j = \mathbf{f}^j(\vec{i}_S) + f_0^j$ **do**
4: Increase the domain ($\mathcal{D}_\mathcal{S}$) dimensionality by creating supernodes for all original iterators that appear in ϕ_S^j
5: Let the supernode iterators be $i\vec{T}$
6: Add the following two constraints to $\mathcal{D}_S$: $\tau_j * \mathbf{f}^j(i\vec{T}_S) \ \leq \ \mathbf{f}^j(\vec{i}_S) + f_0^j \ \leq \ \tau_j * \mathbf{f}^j(i\vec{T}_S) + \tau_j - 1$
7: **end for**
8: **end for**
9: /* Update the transformation matrices */
10: **for** each statement S **do**
11: Add k new rows to the transformation of S at level i
12: Add as many columns as the number of supernodes added $\mathcal{D}_S$ in Step 4
13: **for** each $\phi_S^j = \mathbf{f}^j(\vec{i}_S)$, $j = i, \ldots, i+k-1$ **do**
14: Add a supernode for this hyperplane: $\phi T_S^j = \mathbf{f}^j(i\vec{T}_S)$
15: **end for**
16: **end for**

OUTPUT Updated domains ($\mathcal{D}_S$) and transformations/scatterings

direction – this is guaranteed and detected by the transformation framework (Step 8 of Algorithm 1).

With this, we formally state the algorithm to modify the original domain and updating the statement-wise transformations (Algorithm 2). The (higher-dimensional) tile space loops are referred to as supernodes in the description. For example, in the example above, iT, jT were supernodes in the original domain, while c_1T, c_2T are supernodes in the transformed space. Note that the transformation for each statement by Algo. 1 has the same number of rows.

THEOREM 1. *The set of scattering supernodes, ϕT_S^i, ϕT_S^{i+1}, …, ϕT_S^{i+k-1} obtained from Algorithm 2 satisfy the tiling legality condition (4)*

The proof is straightforward. Since, ϕ_S^j, $i \leq j \leq i+k-1$ satisfy (4) and since the supernodes step through an aggregation of parallel hyperplane instances, dependences continue to be in the forward direction for the scattering supernode dimensions too. This holds true for both intra and inter-statement dependences. $\phi T_{S_1}^j, \phi T_{S_2}^j, \ldots, \phi T_{S_n}^j$ thus represent a common supernode dimension in the transformed space with all affine dependences in its forward direction or null-space.□

5.2 Example: 3-d tiles for LU

The transformation obtained for LU (Fig. 10(a)) is:

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

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

Fig. 5.2 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 memory-inefficient perfectly nested version).

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

Figure 3. Tiled specification for LU

5.3 Tiling multiple times

The same tiling hyperplanes can be used to tile multiple times (Theorem 1) if one wishes to tile for registers, L1, L2, and for parallelism, and the legality of the same is guaranteed by the transformation framework. The scattering functions are duplicated for each such level as it was done for one level. An example is shown in Fig. 14(e) for a matrix-vector transpose kernel tiled for L1 and L2 caches. However, for parallel code generation, some additional treatment is needed. This is discussed shortly.

5.4 Parallel code generation

Once the algorithm in Sec. 5.1 is applied, outer parallel or inner parallel loops can be readily marked parallel (for example with openmp pragmas). However, unlike scheduling-based approaches, since we find tiling hyperplanes and the outer ones are used as space, there may not be a single loop in the transformed space that carries all dependences (even if the code admits a one dimensional schedule). Hence, when one or more of the space loops carries a (forward) dependence (also called doacross loops), care has to be taken while generating parallel code. Recall, the framework makes sure that the dependence is in the forward direction. Pipelined parallelism exists in such cases, and our approach to coarse-grained (tiled) shared memory parallel code generation is as described in Fig. 3.

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 obtains a valid schedule of tiles and preserves tile shapes. Also, not that all degrees of pipelined parallelism need not be exploited. In practice, we observe that a few are sufficient. Moreover, performing such a unimodular transformation to the tile space introduces very less additional code complexity (modulo's do not appear in the generated code due to unimodularity).

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

```
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];
    }
}
```

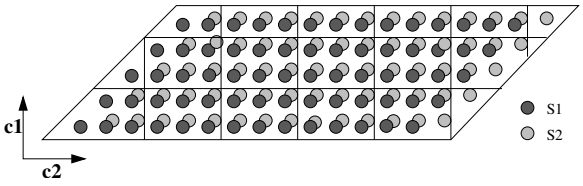
$$\begin{bmatrix}1 & 0\\ 2 & 1\\ 0 & 0\end{bmatrix}\begin{bmatrix}t\\ i\end{bmatrix}+\begin{bmatrix}0\\ 0\\ 0\end{bmatrix} \qquad \begin{bmatrix}1 & 0\\ 2 & 1\\ 0 & 0\end{bmatrix}\begin{bmatrix}t\\ j\end{bmatrix}+\begin{bmatrix}0\\ 1\\ 1\end{bmatrix}$$

(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) Transformed (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 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

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

(e) Transformation for generation of locally tiled code in (c)

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

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

Figure 4. Tiling imperfectly nested 1-d Jacobi

Algorithm 3 Tiled pipelined parallel code generation

INPUT Given that Algorithm 2 has been applied, a set of k (statement-wise) supernodes in the transformed space belonging to a tilable band: $\phi T_S^1, \phi T_S^2, \ldots, \phi T_S^k$

1: To extract m ($< k$) degrees of pipelined parallelism:
2: /* Update transformation matrices */
3: **for** each statement S **do**
4: Perform the following unimodular transformation on only the scattering supernodes: $\phi T^1 \rightarrow \phi T^1 + \phi T^2 + \cdots + \phi T^{m+1}$
5: Mark $\phi T^2, \phi T^3, \ldots, \phi T^{m+1}$ as parallel
6: Leave $\phi T^1, \phi T^{m+2}, \ldots, \phi T^k$ as sequential
7: Place a barrier at the end of the tile schedule loop, $\phi T^1 + \phi T^2 + \cdots + \phi T^{m+1}$
8: Intra-tile loops (and thus the tile shapes) are untouched
9: **end for**

OUTPUT Updated transformation matrices/scatterings

to scan a tile preserving the benefits of the optimization performed by the bounding approach. Fig. 5 shows a simple example with tiling hyperplanes (1,0) and (0,1). In contrast, obtaining an affine (fine-grained) schedule and then enabling time tiling would lead to shapes different from above our approach. Our technique resembles that of [31] where (permutable) time 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. Fig. 10 shows the parallel code generated for LU.

The above scheme allows clean generation of parallel code without any syntactic treatment. Alternate ways of generating pipelined parallel code exist that insert special post/notify or wait-/signal directives to handle dependences in the space loops; however, these require syntactic treatment. In practice, a few degrees of pipelined parallelism may be sufficient. Using several degrees could introduce code complexity with diminishing return.

```
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 5. Shared memory parallel code generation example

5.5 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 5.5 shows the sensitivity in performance for a 1-d stencil code shown in Fig. 4. 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 is very effective. Also, it also avoids large numbers in the intermediate operations performed by code generators, that could possibly lead to PolyLib exceptions for large tile sizes or deep loop nest tiling.

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

Table 1. Performance sensitivity of L1-tiled imperfectly nested stencil code with codegen options: $N = 10^6$, $T = 1000$, tSize=2048x2048

The described tile code generation schemes have been fully implemented into the system to generate compilable parallel-cum-locally tiled code.

6. 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. However, significant jumps were made in the process of going from the compiler framework to evaluation. A direct comparison is difficult since the implementations of those approaches (with the exception of Griebl's) 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, or by picking solutions from a large space of solutions characterized. In addition, a common 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 Forward-Communication-Only allocations to enable time tiling [20], 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) with a 32 KB L1 D cache, 8MB of L2 cache (4MB shared per core pair), and 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 -funroll-loops" (-openmp for parallelized code); the '-fast' option turns on -O3, -ipo, -static, -no-prec-div on x86-64 processors – these options also enable auto-vectorization in icc. Whenever gcc is used, it is 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. For easier presentation and analysis, local tiling for most codes is done for the L1 cache, with equal tile sizes used along all dimensions; they were set empirically (without any comprehensive search) and agreed with the cache size quite well. In all cases, the optimized code for our framework was obtained automatically in a turn-key fashion from the input source code. When comparing with approaches of Lim/Lam and Griebl, the best tile sizes for each approach were almost always the same.

Our transformation framework itself runs quite fast – within a fraction of a second for all benchmarks considered. Along with code generation time, the entire source-to-source transformation does not take more than a few seconds in any of the cases. . The OpenMP "parallel for" directive(s) achieves the distribution of the blocks of the tile space loop(s) among processor cores. Hence, execution on each core is a sequence of L1 or L2 tiles. No explicit tile sizes are chosen for the time loops at the processor level since the L2 time tile sizes are already quite large – hence, L1 or L2 is

the last level at which time tiling is done. Analysis is more detailed for the first example which is simple.

6.1 Imperfectly nested stencil code

The original code, code optimized by our system without tiling, and optimized tiled code are shown in Fig. 4. The performance of the optimized codes are shown in Fig. 6.1. 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]) which finds (2,-1), (3,-1) as the maximally independent time partitions. These do minimize the order of synchronization and maximize the degree of parallelism (O(N)), but any legal independent time partitions 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 does not imply 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. This is the case with icc's auto parallelizer; hence, we just show the sequential run time for icc in this case. Fig. 7 shows L1 cache misses with each approach for a problem size that completely fits in the L2 cache. Though Griebl's technique incurs considerably lesser cache misses than Lim-Lam's, the schedule introduces non-unimodularity leading to modulo comparison in inner loops; it is possible to remove the modulo through an advanced technique using non-unit strides [44] that CLooG does not implement yet. Our code incurs two times lesser number of cache misses than Griebl's and nearly 50 times lesser cache misses than Lim/Lam's scheme. Note that both Lim-Lam's and our transformation in this case are unimodular and hence have the same number of points in a tile for a given tile size. Comparison with gcc is provided in Fig. 6(d) (gcc used to compile all codes) to demonstrate that the relative benefits of our source-to-source system will be available when used in conjunction with any sequential compiler.

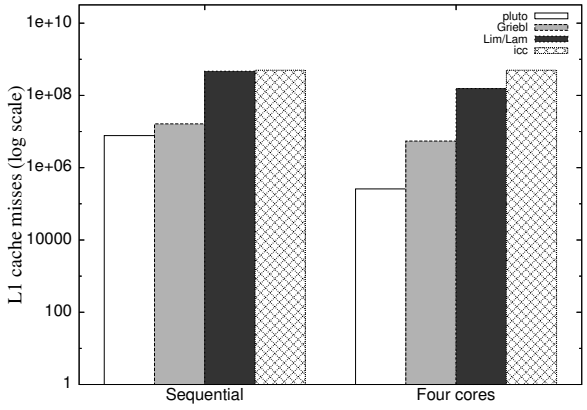


Figure 7. 1-d Jacobi: L1 tiling: $N = 10^5, T = 10000$

6.2 Finite Difference Time Domain electromagnetic kernel

FDTD code is as shown in Fig. 8. ex, ey represent electric fields in x and y directions, while hz is the magnetic field. The code has four statements - three of them 3-d and one 2-d and are nested imperfectly. Our transformation framework finds three tiling hyperplanes (all in one band - fully permutable). The transformation represent a combination of shifting, fusion and time skewing. Parallel performance results shown are for $nx = ny = 2000$ and

```
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]);
}
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

Figure 8. 2-d Finite Difference Time Domain code

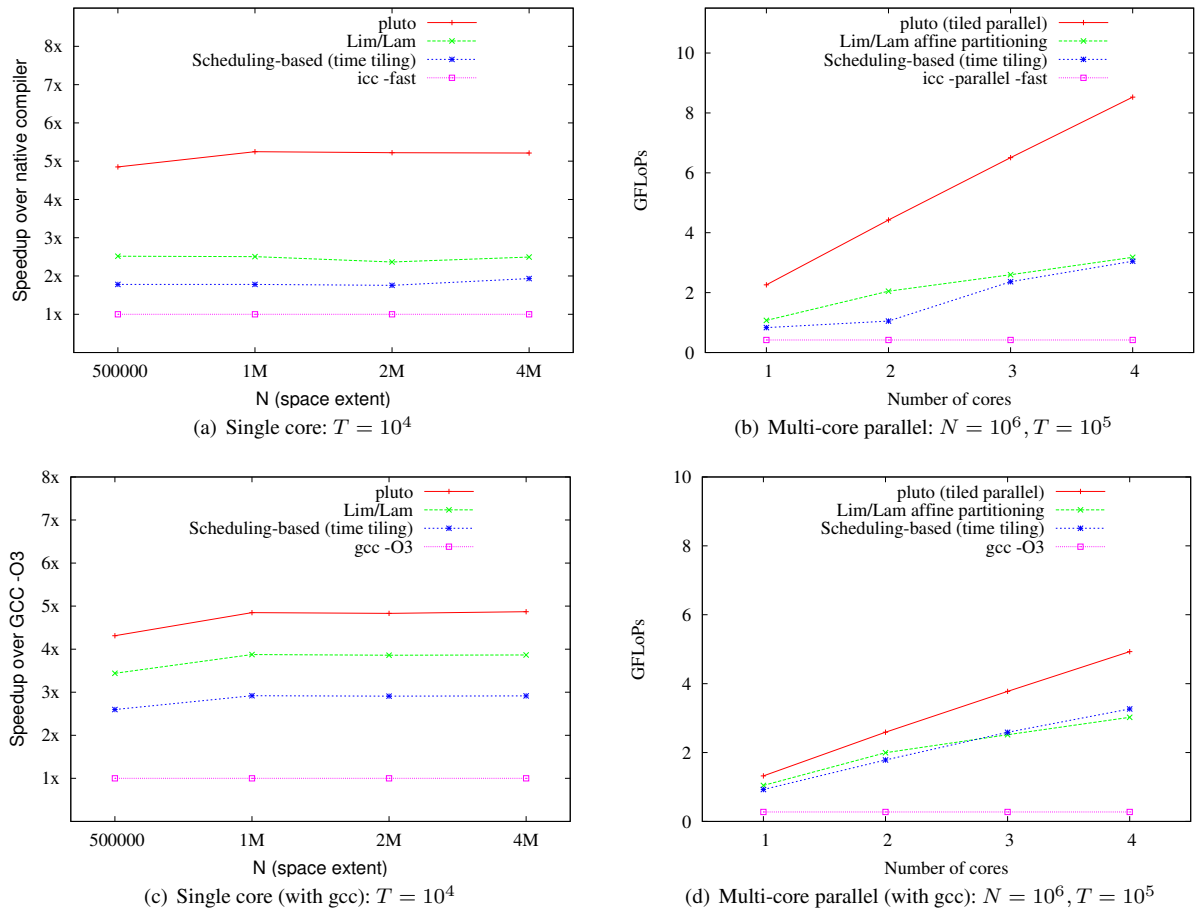
$tmax = 500$. L1 and L2 tile sizes of 32 and 256 were used for each of the three dimensions. Results are shown in Fig. 9. With polyhedral scheduling-based techniques, the outer loop is identified as the sequential schedule loop and the inner loops are all parallel – this is also the transformation applied by icc's auto parallelizer. This does not fuse the inner loops, and synchronization has to be done every time step. With our approach, all three dimensions are tiled (due to a relative shift followed by a skew), the loops are fused, and each processor executes a 3-d tile (which itself is a sequence of 3-d L2 tiles) before synchronization. Multi-core results exhibit highly super-linear speedups. We have two degrees of pipelined parallelism here – to exploit both, a tile space wavefront of (1,1,1) is needed; however, to exploit one, we just need a wavefront of (1,1,0) (Sec. 5.4 leading to much simpler code. Note that two degrees of parallelism are only meaningful when the number of cores is not prime. The slight drop in performance for $N = 4000$ for the sequential case is due to sub-optimal L2 cache tile sizes.

6.3 LU decomposition

Three tiling hyperplanes are found – all belonging to a single band of permutable loops. The first statement though lower-dimensional is naturally sunk into a a 3-dimensional fully permutable space. Thus, there are two degrees of pipelined parallelism. Icc is unable to parallelize such code. 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. 11. The GFLOPs is computed using an operation count of $\frac{2N^3}{3}$. Tiling was done for both L1 and L2 caches. An L1 tile size of 16x16x300 (kij) and an L2 tile size of 256x256x1200 was used. With scheduling-based approaches, $2k$ and $2k+1$ are the schedules found for S1 and S2 respectively. In this case, time tiling is readily enabled by choosing a simple orthogonal allocation (since they have positive non-uniform dependence components).

6.4 Matrix vector transpose

The MVT kernel is a sequence of two matrix vector transposes as shown in Fig. 14 (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 (6) leads to



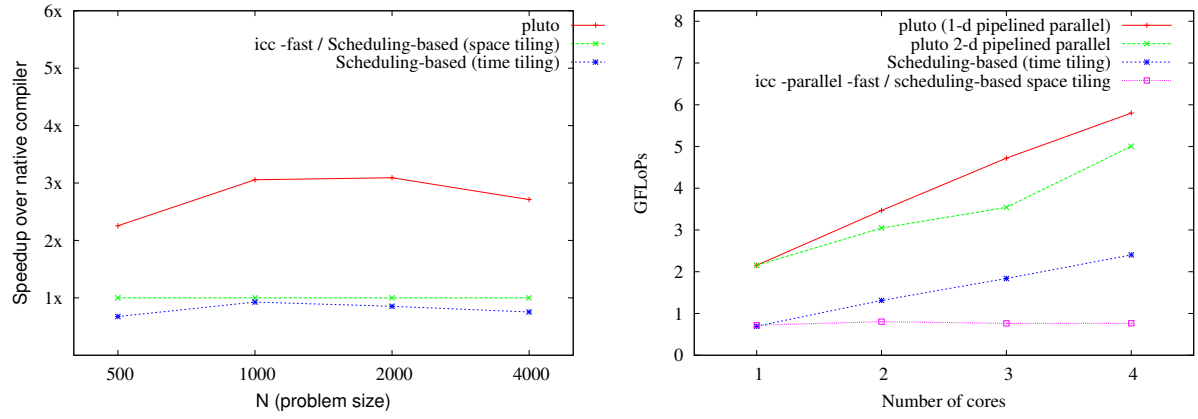
(a) Single core: $T = 10^4$

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

(c) Single core (with gcc): $T = 10^4$

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

Figure 6. Imperfectly nested 1-d Jacobi stencil



(a) Single core: T=500

(b) Parallel: $nx = ny = 2000, tmax = 500$

Figure 9. 2-d FDTD

minimization of this dependence distance by fusion of the first MV with the permuted version of the second MV (note that $\phi(\vec{t}) - \phi(\vec{s})$ 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 one strongly

```
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

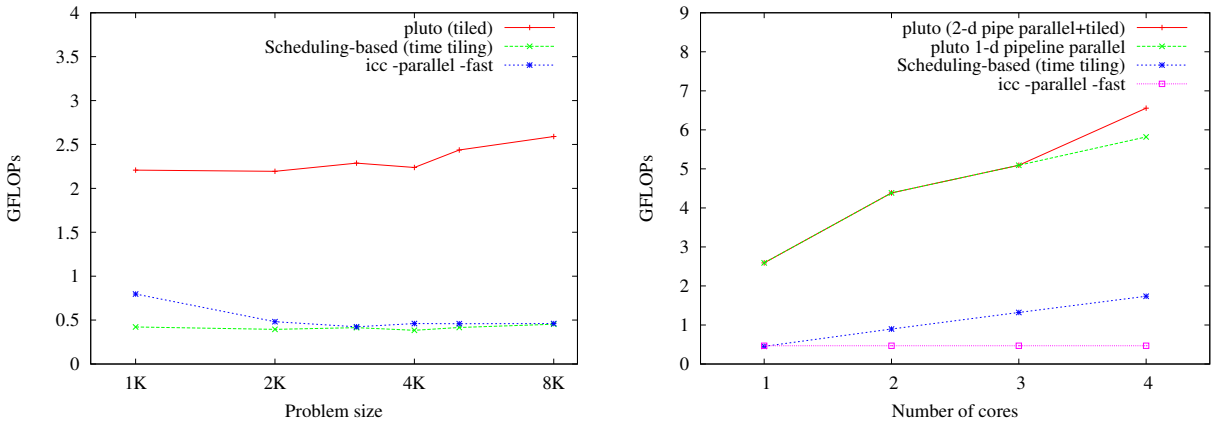
```
#define S1(zT0,zT1,k,j) {a[k][j]=a[k][j]/a[k][k];}
#define S2(zT0,zT1,zT2,k,i,j)    {a[i][j]=a[i][j]-a[i][k]*a[k][j];}

/* Generated 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 + L1 tiled) (tile size 32) cloog -f 4 -l 7

Figure 10. LU decomposition (3-d tiling)



(a) Single core (L1 and L2 tiled)

(b) On a quad core: N=8000

Figure 11. LU performance

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. 12 shows the results for a problem size $N = 8000$. 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. Results are also shown for this case with further syntactic transformations performed on our code to serve as a preview.

6.5 3-D Gauss-Seidel successive over relaxation

The Gauss-Seidel computation allows tiling of all three dimensions after skewing. The transformation our tool obtains skews each of the two space dimensions by a factor of one and two, respectively, w.r.t time. Two degrees of pipelined parallelism can be extracted subsequently, and all three dimensions can be tiled. Fig. 15 shows the performance improvement achieved with 2-d pipelined parallel space as well as 1-d: the latter is better in practice mainly due to simpler code. Again, icc is unable to parallelize such code due to a simple dependence-level based parallel loop detection. The GFLoPs performance is on the lower side when compared to other

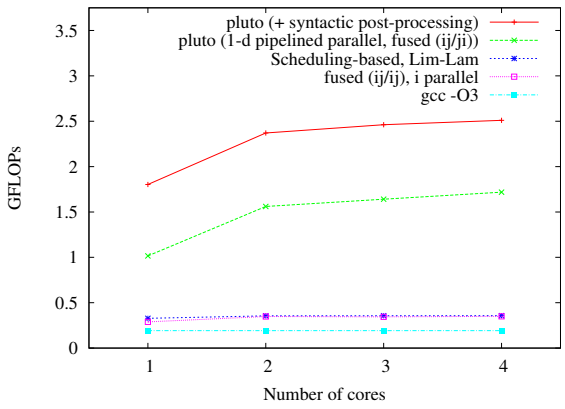


Figure 12. MVT performance on a quad core: N=8000

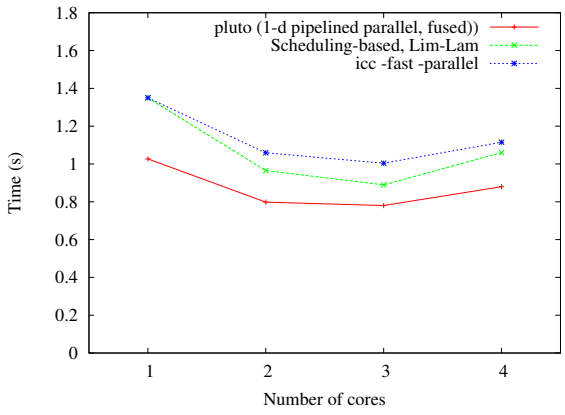


Figure 13. Advect3d performance on a quad core: nx=ny=nz=300

```
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

```
for (c1=0;c1<=N-1;c1++)
  for (c2=0;c2<=N-1;c2++)
    x1[c1] = x1[c1] + a[c1][c2]*y_1[c2];
    x2[c2] = x2[c2] + a[c1][c2]*y_2[c1];
```

(b) Transformed

$$\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \underset{\text{S1}}{\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}} \begin{bmatrix} i \\ j \end{bmatrix} \quad \underset{\text{S2}}{\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}} \begin{bmatrix} i \\ j \end{bmatrix}$$

Figure 14. Matrix vector transpose

schemes is due to the lack of auto-vectorization due to a unique dependence structure for this code.

6.6 Advect3d

Advect3d is a weather-modeling kernel (considered by Qasem et al. [36] for evaulation of their model-driven empirical search approach) that is a sequence of eight 3-d nested loops with a producer-consumer relationship. All arrays are 3-dimensional making the problem memory-bandwidth bound. Though there is a time loop around the eight statements and our tool was able to time tile it, generating 4-d time tiled code became infeasible in this case. Hence, no tiling was done with respect to the outer loop. Fusion is not legal unless constants shifts are applied along various dimensions. Our framework is able to fuse all loop nests (applying 6 shifts) and extract pipelined parallelism (since fusion introduces dependences). Existing approaches cannot enable fusion across such a long sequence SCCs. Results are shown in Fig. 13.

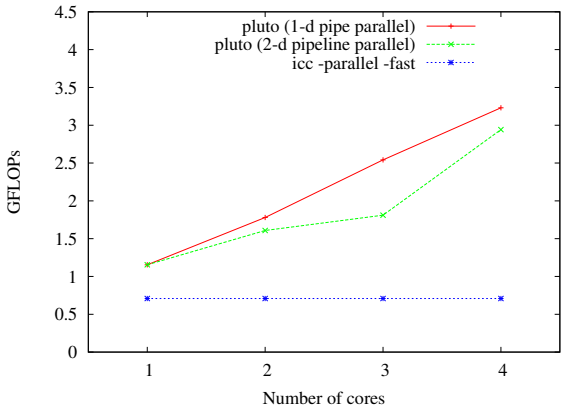


Figure 15. 3-D Gauss Seidel on a quad core: $N_x = N_y = 2000$; T=1000

6.7 Analysis.

All experiments show very high speedups with our approach, both for single thread and multicore parallel execution. The performance improvement is very significant over production compilers as well as state-of-the-art from the research community. Speedup ranging from 2x to 5x are obtained over previous automatic transformation approaches in most cases, while an order of 10x improvement is obtained over the best native production compilers. Linear to super-linear speedups are seen for almost all compute-intensive kernels considered here due to optimization for locality as well as parallelism. To the best of our knowledge, such speedup's have not been reported by any automatic compiler framework as general as ours.

Hand-parallelization of many of the examples we considered here is extremely tedious and not feasible in some cases, especially when time skewed code has to be pipelined parallelized or imperfectly nested loops are involved; this coupled by the fact that the code has to be tiled for at least for one level of local cache, and a 2-d pipelined parallel schedule of 3-d tiles is to be obtained makes manual optimization very complex. The performance of the optimized stencil codes through our system is already in the range of that of hand optimized versions reported in [24]. Also, for many of the codes, a simple parallelization strategy of exploiting inner parallelism and leaving the outer loop sequential (i.e., no time tiling) hardly yields any parallel speedup (Fig. 9(b), Fig. 6(b)). Scheduling-based approaches that do not perform time tiling, or production compilers' auto-parallelizers come up with such transformations.

As mentioned before, tile sizes were not optimized through any extensive search or a model. In addition, studying the interplay of the transformed codes with vectorization and prefetching is crucial. Using cost models for effective tile size determination, with some amount of empirical search, in a manner decoupled with the pure model-driven scheme presented is a reasonably guaranteed approach to take this performance closer to the machine peak. Integration of these techniques is in progress. For simpler codes like matrix-matrix multiplication, this latter phase of optimization, though very simple and straightforward when compared to the rest of our system, brings most of the benefits.

7. Related work

Iteration space tiling [23, 48, 38] is a standard approach for aggregating a set of loop iterations into tiles, with each tile being executed atomically. It is well known that it can improve register reuse, locality and optimize communication. Researchers have considered the problem of selecting tile shape and size to minimize communication, improve locality or minimize finish time [41, 38, 10, 49, 22, 40]. But these studies were restricted to perfectly nested loops with uniform dependences or had other restrictions that limited their applicability to very simple codes. Some specialized works [43, 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.[11] for a detailed summary of older parallelization algorithms which accepted restricted input and/or were based on weaker dependence abstractions than exact polyhedral dependences. Overall, automatic parallelization efforts in the polyhedral model broadly fall in two classes: (1) scheduling/allocation-based, and (2) partitioning-based. The works of Feautrier [16, 17], Darte/Vivien [13] and Griebl [20] (to some extent) fall into the former class, while Lim/Lam's approach falls into the second class. We now compare with approaches from both classes.

Pure scheduling-based approaches are geared towards finding minimum latency schedules or maximum fine-grained parallelism, as opposed to tilability for coarse-grained parallelization with minimized communication and improved locality. Clearly, on most modern parallel architectures, at least one level of coarse-grained parallelism is desired as communication/synchronization costs matter, and so is improving locality. Several works are based on such schedules [7, 20, 12, 34].

Griebl [20] presents an integrated framework for optimizing locality and parallelism with space and time tiling, by treating tiling as a post-processing step after a schedule is found. When schedules are used, the inner parallel (space) loops can be readily tiled. In addition, if coarser granularity of parallelism is desired, Griebl finds an allocation that satisfies the forward communication-only constraint: this enables time tiling. As argued in [8] from a theoretical standpoint and as demonstrated here through experiments, using schedules as one of the loops is not best suited for communication and locality optimization as well as target code complexity. 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.

Lim et al. [31, 30] proposed an affine partitioning framework that identifies outer parallel loops (communication-free space partitions) and permutable loops (time partitions) to maximize the degree of parallelism and minimize 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 performance. No metric is provided to differentiate between these solutions as maximally independent solutions without a cost function are sought. As shown through this work, without a cost function, solutions obtained even for simple input may be unsatisfactory with respect to communication cost, locality, and target code complexity. Also, tiled and parallel code generation for the general case are not discussed. Our approach addresses all of these aspects.

Our approach is closer to the latter class of partitioning-based approaches. However, to the best of our knowledge, it is the first to explicitly model tiling in the transformation framework thereby enabling it to find good tiling hyperplanes for parallelism and locality. The view of tiling hyperplanes on the original domain is preserved till code generation. At the same time, input which cannot be tiled or only partially tiled is all handled, and standard transformations are captured.

In addition to model-based approaches, semi-automatic and search-based transformation frameworks in the polyhedral model also exist [25, 12, 18, 34]. Cohen et al., Girbal et al. [12, 18] 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. Though our system now is fully model-driven, some amount of empirical and iterative optimization may be required on complementary aspects, like tile size and unroll factor determination. Also, decision problems involved with fusion are good candidates for empirical search. Alternatively, more powerful cost models may be employed once transformations in a smaller space are enumerated.

Code generation under multiple affine mappings was first addressed by Kelly et al. [26]. Significant advances were made by Quilleré et al [37] and more recently by Bastoul [6] and Vasilache et al [45, 44], resulting in a powerful open-source code generator, CLooG [1]. Our tiled code generation scheme uses Ancourt and Irigoin's [5] classic approach to specify domains with fixed tile sizes and shape information, but combines it with CLooG's support for scattering functions to allow generation of tiled code for multiple domains under transformations obtained from our theoretical framework. Goumas et al. [19] reported an alternate tiled code generation scheme (to [5]) to address the inefficiency involved in using Fourier-Motzkin (FM) – however, this is no longer an issue as the state-of-the-art uses PolyLib. Techniques for parametric tiled code generation [39, 27] were recently proposed for single statement domains for which rectangular tiling is valid. Such techniques complement our parallelization framework very well and we plan to integrate them into our system.

8. 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 in the polyhedral model, 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/Fortran code, but to any input language from which polyhedra can be extracted. Since our entire transformation framework works in the polyhedral abstraction, only the polyhedral frontend (polyhedra and dependence information extractor) 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 high-performance 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. *IJPP*, 29(5), Oct. 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. In *Euro-Par'10*, pages 272–283, 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 of arbitrarily-nested loop sequences. Technical Report OSU-CISRC-5/07-TR43, The Ohio State University, May 2007.

[9] U. Bondhugula, J. Ramanujam, and P. Sadayappan. Automatic mapping of nested loops to FPGAs. In *ACM SIGPLAN PPoPP'07*, Mar. 2007.

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

[11] 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.

[12] 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.

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

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

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

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

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

[18] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations. *IJPP*, 34(3):261–317, June 2006.

[19] 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.

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

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

[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 *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, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *FRONTIERS*, page 332, 1995.

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

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

[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] B. Norris, A. Hartono, and W. Gropp. *Annotations for performance and productivity*. 2007. Preprint ANL/MCS-P1392-0107.

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

[34] 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.

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

[36] A. Qasem and K. Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *ICS*, pages 249–258, 2006.

[37] F. Quilleré, S. V. Rajopadhye, and D. 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. *JPDC*, 16(2):108–230, 1992.

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

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

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

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

[43] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *PLDI*, pages 215–228, 1999.

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

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

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

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

[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.