# A Practical Automatic Polyhedral Parallelizer and Locality Optimizer

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

[1]Dept. of Computer Science and Engineering
The Ohio State University
{bondhugu,hartonoa,saday}@cse.ohio-state.edu

[2]Dept. of Electrical & Computer Engineering & CCT
Louisiana State University
jxr@ece.lsu.edu

## Abstract

We present the design and implementation of an 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.Unlike previous polyhedral frameworks, 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. The framework has been implemented into a tool to automatically generate OpenMP parallel code from C program sections. Experimental results from the tool show very high performance for local and parallel execution on multi-cores, when compared with state-of-the-art compiler frameworks from the research community as well as the best native production compilers. The system also enables the easy use of powerful empirical/iterative optimization for general arbitrarily nested loop sequences.

***Categories and Subject Descriptors*** D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization, Code generation

***General Terms*** Algorithms, Design, Experimentation, Performance

***Keywords*** Automatic parallelization, Locality optimization, Polyhedral model, Loop transformations, Affine transformations, Tiling

## 1. Introduction and Motivation

Current trends in microarchitecture are increasingly towards larger number of processing elements on a single chip. This has made parallelism and multi-core processors mainstream. 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.



*PLDI'08,* June 7–13, 2008, Tucson, Arizona, USA.



Many compute-intensive applications often spend most of their execution 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 only 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 [19, 18, 46] and code generation [31, 25] in the polyhedral model, but the approaches suffered from scalability challenges. Recent advances in dependence analysis and more importantly in code generation [47, 6, 55, 54] have solved many of these problems resulting in the polyhedral techniques being applied to code representative of real applications like the spec2000fp benchmarks [14, 22]. These advances have also made the polyhedral model practical in production compiler contexts [43] as a flexible and powerful representation to compose and apply transformations. The key missing step has been the demonstration of a scalable and practical approach for automatic transformation for parallelization and locality. Our work addresses this by developing a compiler, based on the theoretical framework we previously proposed [9], to enable end-to-end fully automatic parallelization and locality optimization.

Tiling [28, 58, 61] 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 partition-

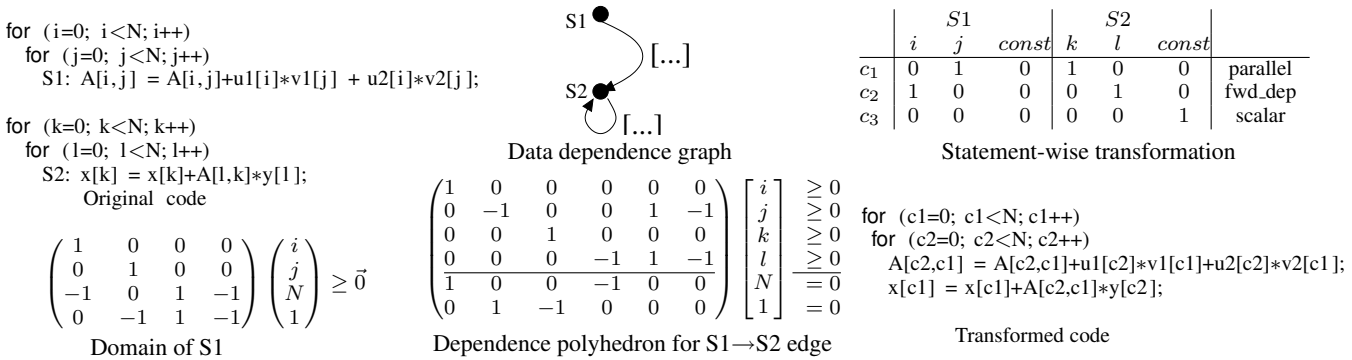



**Figure 1.** Polyhedral representation

ing 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 [37, 36, 35, 24] have one or more drawbacks or restrictions that limit their effectiveness. A common 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 PLuTo [1], a parallelization and locality optimization tool. Finding good ways to tile for parallelism and locality directly through an affine transformation framework is the central idea. Our approach is thus a departure from scheduling-based approaches in this field [20, 21, 17, 24, 15] as well as partitioning-based approaches [37, 36, 35] (due to incorporation of more concrete optimization criteria), however, is built 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. We evaluate the performance of the implemented system on a multicore processor using a number of application kernels that are non-trivial for any existing auto-parallelizer.

Model-driven empirical optimization and automatic tuning approaches (e.g., ATLAS) have been shown to be very effective in optimizing single-processor execution for some regular kernels like matrix-matrix multiplication [56, 63]. 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 and analyzed.

The rest of this paper is organized as follows. Section 3 provides an overview of our theoretical framework for automatic transformation that we proposed in [9]. Section 4 and Section 5 discuss some considerations in the design and techniques for generation of efficient tiled shared memory parallel code from transformations found. Section 6 describes the implemented system. Section 7 provides experimental results. Section 8 discusses related work and conclusions are presented in Section 9.

## 2. Background and Notation

This section provides background on the polyhedral model. All row vectors are typeset in bold.

### 2.1 The Polyhedral model

DEFINITION 1 (Affine Hyperplane). *The set $X$ of all vectors $x \in \mathbf{Z}^n$ such that $\mathbf{h}.\vec{x} = k$, for $k \in \mathbf{Z}$, is 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.*

***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.

***Polyhedral dependences.*** Our dependence model is of exact affine dependences and same as the one used in [20, 36, 14, 45]. Dependences are determined precisely through dataflow analysis [19], 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. The Data Dependence Graph (DDG) is a directed multi-graph with each vertex representing a statement, and an edge, $e \in E$, from node $S_i$ to $S_j$ representing a polyhedral dependence from a dynamic instance of $S_i$ to one of $S_j$: it is characterized by a polyhedron, $\mathcal{P}_e$, called the *dependence polyhedron* that captures the exact dependence information corresponding to $e$. 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). Let $\vec{s}$ represent the source iteration and $\vec{t}$ be the target iteration pertaining to a dependence edge $e$. It is possible to express the source iteration as an affine function of the target

iteration, i.e., to find the last conflicting access. This affine function is also known as the *h-transformation*, and will be represented by $h_e$ for a dependence edge $e$. Hence, $\vec{s} = h_e(\vec{t})$. The equalities corresponding to the h-transformation are a part of the dependence polyhedron and can be used to reduce its dimensionality. Figure 1 shows the polyhedral representation of a simple code.

Let $S_1$, $S_2$, ..., $S_n$ be the statements of the program. A one-dimensional affine transform for statement $S_k$ is defined by:

$$\phi_{s_k}(\vec{i}) \quad = \quad \left[c_1 \ \ldots \ c_{m_{S_k}}\right] \left( \ \vec{i} \ \right) + c_0, \quad c_i \in \mathbf{Z} \qquad (1)$$

$\phi_{S_k}$ can also be called an affine hyperplane, or a *scattering function* when dealing with the code generator. A multi-dimensional affine transformation for a statement is represented by a matrix with each row being an affine hyperplane.

DEFINITION 3 (Dependence satisfaction). *An affine dependence with polyhedron $\mathcal{P}_e$ is satisfied at a level $l$ iff the following condition is satisfied:*

$$\begin{aligned} \forall k(1 \leq k \leq l-1) : \phi^k_{s_j}\left(\vec{t}\right) - \phi^k_{s_i}\left(\vec{s}\right) \quad &\geq \quad 0, \ \langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e \\ \text{and } \phi^l_{s_j}\left(\vec{t}\right) - \phi^l_{s_i}\left(\vec{s}\right) \quad &\geq \quad 1, \ \langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e \end{aligned}$$

# 3. Overview of Automatic Transformation Approach

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

## 3.1 Legality of tiling multiple domains with affine dependences

LEMMA 1. *Let $\phi_{s_i}$ 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 $e \in E$:*

$$\phi_{s_j}\left(\vec{t}\right) - \phi_{s_i}\left(\vec{s}\right) \quad \geq \quad 0, \ \langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e \qquad (2)$$

The above is a generalization of the classic condition proposed by Irigoin and Triolet [28] (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 and with possibly different dimensionalities coming from possibly 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 (2). 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 (2), 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 (2) 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 (2) is imposed on all dependences. However, if it is imposed only on dependences that have not been satisfied up to a certain depth, the independent $\phi$'s that satisfy the condition represent tiling hyperplanes at that depth, i.e., tiling at that level is legal.

## 3.2 Cost function, bounding approach and minimization

Consider the following affine form $\delta_e$:

$$\delta_e(\vec{s}, \vec{t}) \quad = \quad \phi_{s_j}(\vec{t}) - \phi_{s_i}(\vec{s}), \quad \langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e \qquad (3)$$

The affine form $\delta_e(\vec{s}, \vec{t})$ is very significant. This function is the number of hyperplanes the dependence $e$ traverses along the hyperplane normal $\phi$. If $\phi$ 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 $\phi$ 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 $\phi$: 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, $\phi(\vec{t}) - \phi(\vec{s})$ could be $c_1 i + (c_2 - c_3)j$, under $1 \leq i \leq N, 1 \leq j \leq N, i \leq j$. Such a form results 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 [20, 51] and casting the objective into an ILP formulation. Since the loop variables themselves can be bounded by affine functions of the parameters, one can always find an affine form in the program parameters, $\vec{p}$, that bounds $\delta_e(\vec{s}, \vec{t})$ for every dependence edge $e$, i.e., there exists $v(\vec{p}) = \mathbf{u}.\vec{p} + w$, such that

$$\begin{aligned} & \phi_{s_j}(\vec{t}) - \phi_{s_i}(\vec{s}) \quad \leq \quad v(\vec{p}), \ \langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e, \quad \forall e \in E \\ \text{i.e.,} \quad & v(\vec{p}) \ - \ \delta_e(\vec{s}, \vec{t}) \quad \geq \quad 0, \ \langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e, \quad \forall e \in E \qquad (4) \end{aligned}$$

Such a bounding function approach was first used by Feautrier [20], but for a different purpose – to find minimum latency schedules. Now, Farkas Lemma can be applied to (4).

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

where $\mathcal{P}^k_e$ 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 (2) 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 $\mathbf{u} = (u_1, u_2, \ldots u_k)$.

$$\text{minimize}_{\prec} \{u_1, u_2, \ldots, u_k, w, \ldots, c'_i s, \ldots\} \qquad (5)$$

Finding the lexicographic minimal solution is within the reach of the Simplex algorithm and can be handled by the Parametric Integer Programming (PIP) software [18]. Since the program parameters are quite large, their coefficients are minimized with the highest priority. The solution gives a hyperplane for each statement. Note the trivial zero solution is avoided by making a practical choice that is described in the next section.

***Iteratively finding independent solutions.*** Solving the ILP formulation in the previous section gives us a single solution to the coefficients of the best mappings for each statement. We need at least as many independent solutions (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 [40, 34] of the transformation rows found so far ($H_S$) and forcing a non-zero component in $H_S^{\perp}$ for the next solution.

$$H_S^{\perp} = I - H_S^T \left( H_S H_S^T \right)^{-1} H_S \tag{6}$$

Linearly independent (statement-wise) hyperplanes are found iteratively till all dependences are satisfied. Dependences from previously found hyperplanes are not removed as independent tiling hyperplanes are found unless they have to be to allow the next band of tiling hyperplanes to be found. Maximal sets of fully permutable loops are found like in the case of [59, 16, 36], however, with a optimization criterion (5) that goes beyond maximum degrees of parallelism.

***Outer space and inner time: communication and locality optimization unified*** The best possible solution to (5) 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*), or an inner parallel one 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 found last. 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. By minimizing $\phi(\vec{t}) - \phi(\vec{s})$ as we find hyperplanes from outermost to innermost, we push dependence satisfaction 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 if (forward) space dependences exist. 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.*** 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 (1), a solution found corresponds to a fine-grained interleaving of different statement instances at that level [8].

# 4. More design considerations

In this section, we discuss a few enhancements to the framework as well as some practical choices for scalability.

## 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 (2) for such dependences, but consider them for the bounding objective function ( 4). 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:

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

## 4.2 Avoiding combinatorial explosion

There are two situations when there is a possibility of combinatorial explosion (with the number of statements) if reasonable choices are not made.

1. Avoiding the trivial zero vector solution to the hyperplanes
2. Construction of linearly independent sub-space for each statement's transformation

Removing the trivial zero solution to (5) on a per-statement basis leads to a non-convex space, and in this case a union of a large number of convex spaces each of which has to be tried. Similarly, while constructing a linearly independent sub-space for each statement, there are several choices for each statement and the number of choices to be exhaustively tried will be a product of all these [8]. The above difficulties can be solved at once by only looking for non-negative transformation coefficients. Then, the zero solution can be avoided with the constraint of $\sum c_i \geq 1$, $1 \leq i \leq m_{S_k}$, for each statement $S_k$. Doing so mainly excludes transformations that include loop reversal, and in practice, we do not find this to be a concern at all. The current implementation of Pluto [1] is with this choice, and scales very well without loss of good transformations. Exploring all possible choices if one wishes is still possible for around up to ten statements while keep the running time within a few tens of seconds.

# 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 the polyhedral code generator CLooG [13, 6]. 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 [57, 42] (which in turn uses the Chernikova algorithm [33]) for its core operations, and the code generated is far more efficient than that by older code generators based on Fourier-Motzkin variable elimination like Omega Codegen [46] or LooPo's internal code generator [25, 24]). Also, code generation time and memory utilization are much lower [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.

## 5.1 Tiling the transformed AST vs. Tiling the scattering functions

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), (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.

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 Figure 3(a) for example. If code is generated by just applying the transformation first, we get code shown in Figure 3(b). Even though the transformation framework obtained two tiling hyperplanes, the transformed code in Figure 3(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 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 describe an approach to accomplish the latter which is the subject of Section 5. For example, for the code in Figure 3, 2-d tiled code can be generated in one pass, both applying the transformation as well as accomplishing tiling.

### 5.2 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 $c_1, c_2$ constituting a permutable band; 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 [4]; but the scatterings are duplicated for the tile space too. 'T' subscript is used to denote the corresponding tile space iterator. The tile space and intra tile loop scattering functions are specified as follows.

| Domain | Scattering |
|---|---|
| $0 \le i \le N-1$ | $c_{1T} = i_T$ |
| $0 \le j \le N-1$ | $c_{2T} = i_T + j_T$ |
| $0 \le i - 32i_T \le 31$ | $c_1 = i$ |
| $0 \le (i+j) - 32(i_T + j_T) \le 31$ | $c_2 = i + j$ |
| $(c_{1T}, c_{2T}, c1, c2)$ | $\leftarrow scatter(i_T, j_T, i, j)$ |

$c_{1T}$ 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.

With this, we formally state the algorithm to modify the original domain and updating the statement-wise transformations (Algorithm 1). 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 matrix computed for each statement has the same number of rows.

**Algorithm 1** Tiling for multiple stmts under transformations

**INPUT** Hyperplanes (statement-wise) belonging to a tilable band of width $k$: $\phi_S^i, \phi_S^{i+1}, \ldots, \phi_S^{i+k-1}$, expressed as affine functions of corresponding original iterators, $\vec{i}_S$; 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 $\phi_S^j = \mathbf{f}^j(\vec{i}_S) + f_0$ **do**
4:  Increase the domain ($\mathcal{D}_\mathcal{S}$) dimensionality by creating supernodes for all original iterators that appear in $\phi_S^j$
5:  Let the supernode iterators be $i\vec{T}$
6:  Add the following two constraints to $\mathcal{D}_S$: $\tau_j * \mathbf{f}^j(i\vec{T}_S) \le \mathbf{f}^j(\vec{i}_S) + f_0^j \le \tau_j * \mathbf{f}^j(i\vec{T}_S) + \tau_j - 1$
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 to $\mathcal{D}_S$ in Step 4
13:  **for** each $\phi_S^j = \mathbf{f}^j(\vec{i}_S) + f_0^j, j = i, \ldots, i+k-1$ **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

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

Since, $\phi_S^j$, $i \le j \le i+k-1$ satisfy (2) 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.□

Figure 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 by a factor of two w.r.t time (as opposed to skewing by a factor of one that is required for the space memory-inefficient perfectly nested version).

***Example: 3-d tiles for LU*** The transformation obtained for the LU decomposition code 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 Figure 2. The code is shown in Figure 9.

***Tiling multiple times*** The same tiling hyperplanes can be used to tile multiple times (due to Theorem 1), for registers, L1, L2 caches, 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. Such a guarantee is

```
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];
    }
}
```

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

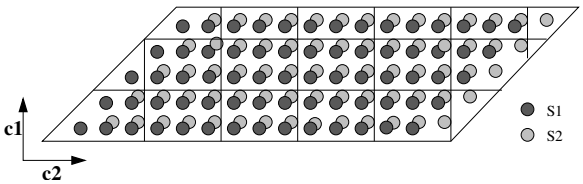
$$\begin{bmatrix}1&0\\2&1\\0&0\end{bmatrix}\begin{bmatrix}t\\j\end{bmatrix}+\begin{bmatrix}0\\1\\1\end{bmatrix}$$

(a) Original 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)




(c) Tiles under a transformation

```
#define S1(t,i) {b[i]=0.333∗(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) ;
    }
  }
 }
}
```

(d) Optimized with tiling ( tile size 256), cloog −f 3 −l 5

$$\begin{bmatrix}c_1\\c_2\\c_3\\c_4\\c_5\end{bmatrix}=\begin{bmatrix}1&0&0&0&0\\2&1&0&0&0\\0&0&1&0&0\\0&0&2&1&0\\0&0&0&0&0\end{bmatrix}\begin{bmatrix}t_T\\i_T\\t\\i\\1\end{bmatrix}\quad\text{S1}\qquad\begin{bmatrix}1&0&0&0&0\\2&1&0&0&0\\0&0&1&0&0\\0&0&2&1&1\\0&0&0&0&1\end{bmatrix}\begin{bmatrix}t_T\\j_T\\t\\j\\1\end{bmatrix}\quad\text{S2}$$

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

$$\begin{bmatrix}c_1\\\mathbf{c_2}\\c_3\\c_4\\c_5\end{bmatrix}=\begin{bmatrix}3&1&0&0&0\\2&1&0&0&0\\0&0&1&0&0\\0&0&2&1&0\\0&0&0&0&0\end{bmatrix}\begin{bmatrix}t_T\\i_T\\t\\i\\1\end{bmatrix}\quad\text{S1}\qquad\begin{bmatrix}3&1&0&0&0\\2&1&0&0&0\\0&0&1&0&0\\0&0&2&1&1\\0&0&0&0&1\end{bmatrix}\begin{bmatrix}t_T\\j_T\\t\\j\\1\end{bmatrix}\quad\text{S2}$$

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

**Figure 3.** Tiling imperfectly nested Jacobi

available even when syntactic tiling is to be done as a post-pass on a perfectly nest band in the target AST.

### 5.3 Parallel code generation

Once the algorithm in Sec. 5.2 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 satisfies all dependences (even if the code admits a one dimensional schedule). Hence, when one or more of the space loops satisfies a (forward) dependence (also called doacross loops), care has to be taken while generating parallel code. Hence, for pipelined parallel codes, our approach to coarse-grained (tiled) shared memory parallel code generation is as described in Figure 2.

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 $\phi T^1 + \phi T^2 + \cdots + \phi T^{p+1}$ satisfies all affine dependences satisfied by $\phi T^1, \phi T^2, \ldots, \phi T^{p+1}$,

**Algorithm 2** Tiled pipelined parallel code generation

**INPUT** Given that Algorithm 1 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: **end for**

**OUTPUT** Updated transformation matrices/scatterings

and gives a legal wavefront (schedule) of tiles. Since the transformation is only on the tile space, it preserves the shape of the tiles. Communication still happens along boundaries of $\phi^1$, $\phi^2$, ..., $\phi^s$, and the same shaped tiles are used to scan a tile, thus preserving the benefits of the optimization performed by the bounding approach. Moreover, performing such a unimodular transformation to the tile

Domains

| $S1$ | $S2$ |
|---|---|
| $0 \leq k \leq N-1$ | $0 \leq k \leq N-1$ |
| $k+1 \leq j \leq N-1$ | $k+1 \leq i \leq N-1$ |
| | $k+1 \leq j \leq N-1$ |
| $0 \leq k - 32k_T \leq 31$ | $0 \leq k - 32k_T \leq 31$ |
| $0 \leq j - 32j_T \leq 31$ | $0 \leq i - 32i_T \leq 31$ |
| | $0 \leq j - 32j_T \leq 31$ |

Scatterings

| S1 | S2 |
|---|---|
| $c_{1T} = k_T$ | $c_{1T} = k_T$ |
| $c_{2T} = j_T$ | $c_{2T} = j_T$ |
| $c_{3T} = k_T$ | $c_{3T} = i_T$ |
| $c_1 = k$ | $c_1 = k$ |
| $c_2 = j$ | $c_2 = j$ |
| $c_3 = k$ | $c_3 = i$ |
| $(c_{1T}, c_{2T}, c_{3T}, c1, c2, c3)$ | $(c_{1T}, c_{2T}, c_{3T}, c1, c2, c3)$ |
| $\leftarrow scatter(k_T, j_T, k, j)$ | $\leftarrow scatter(k_T, j_T, i_T, k, j, i)$ |

**Figure 2.** Tiled specification for LU

space introduces very less additional code complexity (modulo's do not appear in the generated code due to unimodularity).

In contrast, obtaining an affine (fine-grained) schedule and then enabling time tiling would lead to shapes different from above our approach. The above technique of adding up 1-d transforms resembles that of [37] 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 dismisses all dependences.

```
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 tile schedule

**Figure 4.** Shared memory parallel code generation example

Figure 4 shows a simple example with tiling hyperplanes (1,0) and (0,1). Our 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 [36, 24], but, these require syntactic treatment. Note that not all degrees of pipelined parallelism need be exploited. In practice, a few degrees are sufficient; using several could introduce code complexity with diminishing return.

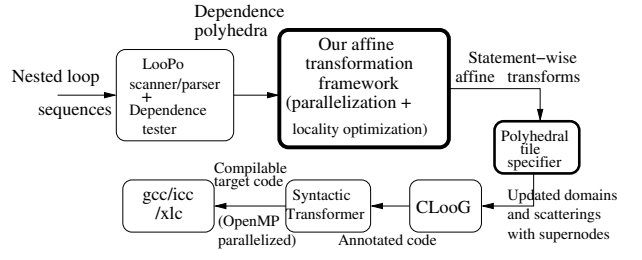



**Figure 5.** The PLuTo source-to-source transformation system

### 5.4 Intra-tile reordering

Due to the nature of our algorithm, even within a local tile (L1) that is executed sequentially, the intra-tile loops that are actually parallel do not end up being outer in the tile (Sec. 3.2): this goes against vectorization of the transformed source for which we rely on the native compiler. Also, the polyhedral tiled code is often complex for a compiler to further analyze and say, permute and vectorize. Hence, as part of a post-process in the transformation framework, we move the parallel loop within a tile innermost and make use of ignore dependence pragmas to explicitly force vectorization. Similar reordering is possible to improve spatial locality that is not considered by our cost function due to the latter being fully dependence-driven. Note that the tile shapes or the schedule in the tile space is not altered by such post-processing.

## 6. Implementation

The proposed framework has been implemented into a tool, PLuTo [1]. Figure 5 shows the entire tool-chain. We used the scanner, parser and dependence tester from the LooPo infrastructure [38], which is a polyhedral source-to-source transformer including implementations of various polyhedral analyses and transformations from the literature. We used PipLib 1.3.3 [41, 18] as the ILP solver and CLooG 0.14.1 [13] for code generation. The transformation framework takes as input, polyhedral domains and dependence polyhedra from LooPo's dependence tester, computes transformations and provides it to Cloog. Compilable OpenMP parallel code is finally output after some post-processing on the Cloog code.

***Syntactic post-processing.*** We have also integrated an annotation-driven system of Norris et al. [39] to perform syntactic transformations on the code generated from Cloog as a post-processing; these include register tiling followed by unrolling or unroll/jamming. The choice of loops to perform these transformations on is specified by the transformation framework, and hence legality is guaranteed. 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 to perform for non-rectangular iteration spaces, for example. The complementary benefits of syntactic post-processing will be reported in future. However, a preview of the potential performance improvement is provided for one kernel in the experimental evaluation section.

## 7. Experimental evaluation

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

#### 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 (with the exception of Griebl's [24, 38]) is not available; further most previously presented

studies did not use an end-to-end automatic implementation, but performed some manual code generation based on solutions generated by a transformation framework, or by selecting a solution from a large space of solutions characterized.

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. [2]). For others [37, 36, 35], 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 or generate compilable optimized code, we have not attempted an exhaustive comparison for all the cases. For the above reasons, the first kernel chosen (imperfectly-nested 1-d Jacobi) is a relatively simple one.

The current state-of-the-art with respect to optimizing code has been semi-automatic approaches that require an expert to manually guide transformations [22]. As for scheduling-based approaches, the LooPo system [38] 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 [24] that will be referred to as 'Scheduling-based (time tiling)' and (2) Lim/Lam's affine partitioning [37, 36, 35] referred to as 'Affine partitioning (max degree parallelism, no cost function)' in the graphs. 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. By doing so, these compared alternate approaches also get all the benefits of CLooG and our tiled code generation scheme.

***Experimental setup.*** The results were obtained 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 was 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 ensure auto-vectorization in icc. The OpenMP implementation of icc supports nested parallelism needed to exploit multiple degrees of pipelined parallelism when they exist.

Our transformation framework itself runs quite fast – within a fraction of a second for all benchmarks considered here. Along with code generation time, the entire source-to-source transformation does not take more than a few seconds for 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 L2 tiles (or L1 tiles if L2 tiling is not done). Tile sizes were set automatically using a very rough model. Equal tile sizes were used along all dimensions, except when loops were marked for vectorization (Sec.5.4), in which case the tile size of the loop to be vectorized was increased. In all cases, the optimized code for our framework was obtained automatically in a turn-key fashion from the input source code.

***Imperfectly nested stencil code.*** The original code, code optimized by our system without tiling, and optimized tiled code are shown in Figure 3. The performance of the optimized codes are shown in Figure 6. Speedups ranging from 4x to 7x are obtained for single core execution due to locality enhancement. The parallel speedups are compared with Lim/Lam's technique (Algorithm A in [37]) which obtains (2,-1), (3,-1) as the maximally independent time partitions, and Griebl's time tiling technique which uses an FCO allocation of $2t + i$ along with the schedule $2t$ for S1, $2t + 1$ for S2. Just space tiling in this case does not expose sufficient parallelism granularity and an inner space parallelized code has very poor performance. This is also the case with icc's auto parallelizer; hence, we just show the sequential run time for icc in this case. Analysis of cache misses with each of the schemes is presented in a more detailed report [10].

```
for (t=0; t<tmax; t++) {
  for (j=0; j<ny; j++)
    ey[0][j] = exp(-coeff0*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]);
}
```

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

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

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

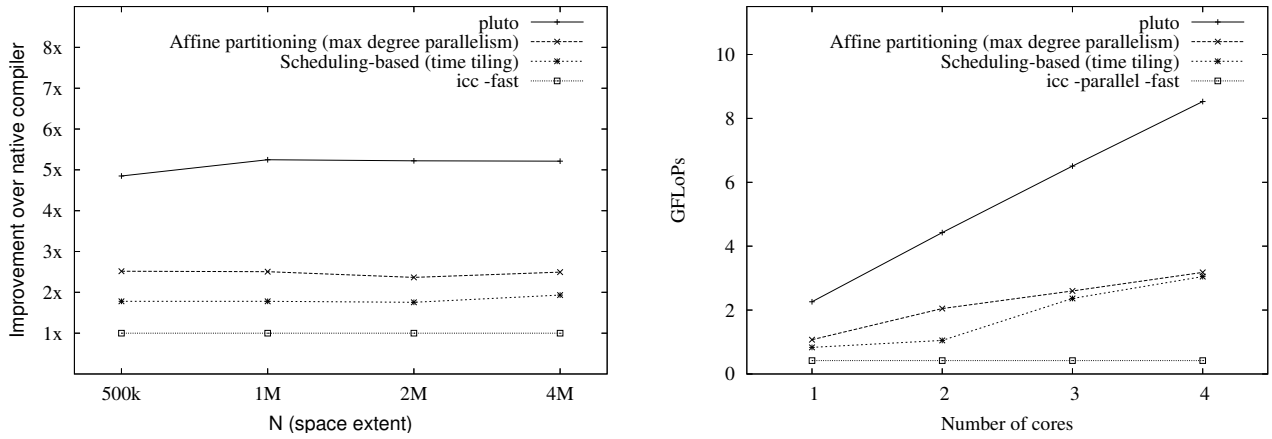
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}$$

**Figure 7.** 2-d FDTD (original code) and transformation

***Finite Difference Time Domain electromagnetic kernel.*** The FDTD (Finite Difference Time Domain) code is as shown in Figure 7. The arrays $ex$ and $ey$ 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 represents a combination of shifting, fusion and time skewing. Parallel performance results shown are for $nx = ny = 2000$ and $tmax = 500$. Results are shown in Figure 8.

***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 auto-parallelize this. Performance results on the quad core machine are shown in Figure 10(b). Scheduling-based parallelization performs poorly mainly due to code complexity arising out of a non-unimodular transformation, that also inhibits vectorization.

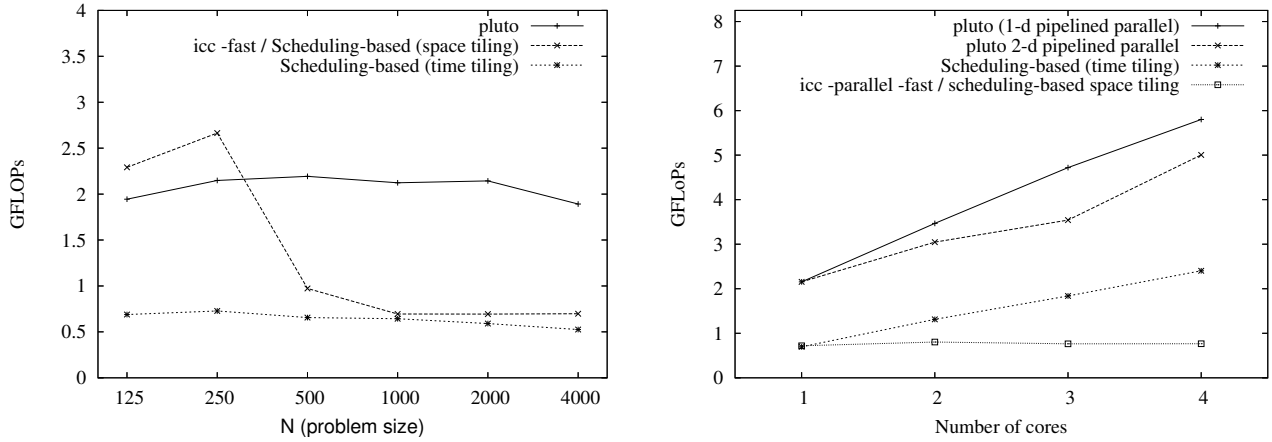
***Matrix vector transpose.*** The MVT kernel is a sequence of two matrix vector transposes as shown in Figure 11. It is found within an outer convergence loop with the Biconjugate gradient algorithm. The only inter-statement dependence is a non-uniform read/input on matrix A. The cost function bounding (4) leads to 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 satisfies 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 do




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

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

**Figure 6.** Imperfectly nested Jacobi stencil




(a) Single core: T=500

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

**Figure 8.** 2-d FDTD

not consider input dependences. Hence, other approaches only extract synchronization-free parallelism from each of the MVs separately with a barrier between the two, giving up reuse on array A. Figure 12 shows the results for a problem size $N = 8000$. Fusion of ij with ij does not exploit reuse on matrix A, whereas the code generated by our tool performs the best – it fuses ij with ji, tiles it and extracts a degree of pipelined parallelism. For this case, results are also shown with further syntactic transformations performed on the Pluto code.

***3-D Gauss-Seidel successive over relaxation.*** The Gauss-Seidel computation allows tiling of all three dimensions after skewing. The transformation obtained by our tool skews 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. Figure 13 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 does not parallelize this. The absolute GFLoPs performance here is on the lower side when compared to other codes due to a unique dependence structure that prevents auto-vectorization.

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    x1[i] = x1[i] + a[i,j]*y1[j];

for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    x2[i] = x2[i] + a[j,i]*y2[j];
```

(a) Original

```
for (c1=0;c1<=N-1;c1++)
  for (c2=0;c2<=N-1;c2++)
    x1[c1] = x1[c1]+a[c1,c2]*y1[c2];
    x2[c2] = x2[c2]+a[c1,c2]*y2[c1];
```

(b) Transformed

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

**Figure 11.** Matrix vector transpose

### 7.1 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

```
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

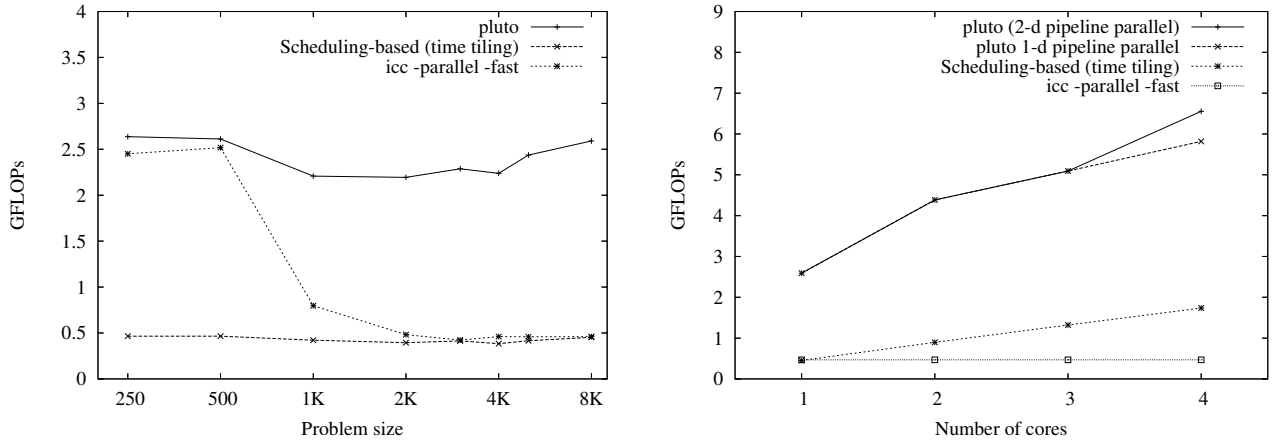
```
#define S1(zT0,zT1,k,j) {a[k][j]=a[k][j]/a[k][k];}
#define S2(zT0,zT1,zT2,k,i,j)    {a[i][j]=a[i][j]-a[i][k]*a[k][j];}

/* 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 9.** LU decomposition




(a) Single core (L1 and L2 tiled)

(b) On a quad core: N=8000

**Figure 10.** LU performance

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 comparable to hand optimized versions reported in [29]. 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 (Figure 8(b), Figure 6(b)) – scheduling-based approaches that do not perform time tiling, or production compilers' auto-parallelizers perform such transformations.

As mentioned before, tile sizes were not optimized through any search or a concrete model. In addition, studying the interplay of the transformed codes with prefetching is important. Using cost

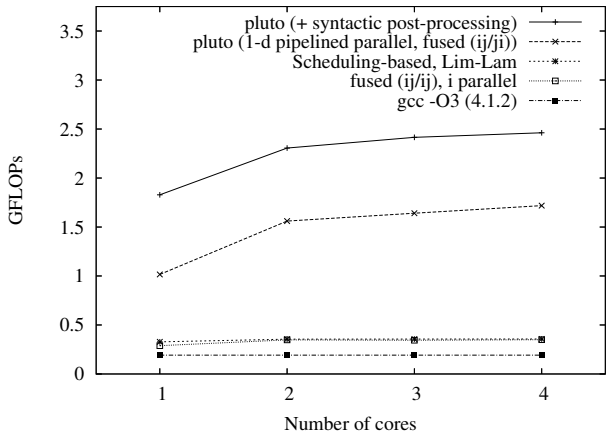



**Figure 12.** MVT performance on a quad core: N=8000

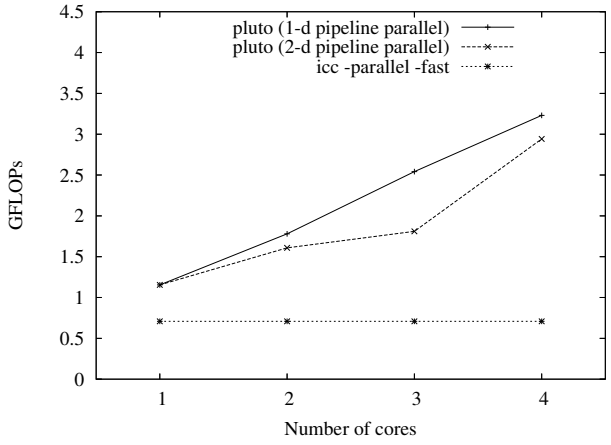



**Figure 13.** 3-D Gauss Seidel on a quad core: $N_x = N_y = 2000$; T=1000

models for effective tile size determination with some amount of empirical search, in a manner decoupled with the pure model-driven scheme presented here, we expect to move 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. We are also integrating complementary syntactic transformations on the generated code: note that this latter phase fully relies on the transformation framework for correctness and systematic application. Additional experimental results and comments about the optimized codes and transformations are available in an extended report [10].

## 8. Related work

Iteration space tiling [28, 58, 48, 61] 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, improve locality and minimize communication. Researchers have considered the problem of selecting tile shape and size to minimize communication, improve locality or minimize finish time [5, 11, 26, 27, 48, 50, 60]. However, these studies were restricted to very simple codes – like single perfectly nested loop nests with uniform dependences and/or sometimes loop nests of a particular depth. To the best of our knowledge, these works have not been extended to more general cases and the cost functions proposed therein not been implemented to allow a direct comparison for those restricted cases. Our work is in the direction of a practical cost function that works for the general case (any polyhedral program or one that can be approximated into it) as opposed to a more sophisticated function for restrictive input. With such a function, we are able to keep the problem linear, and since sparse ILP formulations that result here are solved very quickly, we are at a sweet-spot between cost-function sophistication and scalability to real-world programs. Refinements to the function that still keep it linear are discussed in Section 3.10 of [8]. Note that our function does not capture tile size optimization, but our results show that decoupling optimization of tile shapes and sizes is a practical and very effective approach; all the performance improvement shown were with tile sizes that were selected with rough thumb rules automatically.

Ahmed et al. [2] proposed a framework for data locality optimization of imperfectly nested loops for sequential execution. It was among the first attempts to tile imperfectly nested loops. Based on the description of the heuristic approach to minimizing reuse distances [2], it would appear that it is not scalable [53]. Some specialized works [52, 62] also exist on tiling a restricted class of imperfectly nested loops for locality.

Loop parallelization has been studied extensively. The reader is referred to the survey of Boulet et al. [12] for a detailed summary of earlier parallelization algorithms – these restricted the input loop forms and/or were based on weaker dependence abstractions than exact polyhedral dependences [3, 17, 16, 59]. Automatic parallelization efforts in the polyhedral model broadly fall into two classes: (1) scheduling/allocation-based, and (2) partitioning-based. The works of Feautrier [20, 21], Darte and Vivien [17] and Griebl [24] (to some extent) fall into the former class, while Lim/Lam's approach [37, 36, 35] falls into the second class. We now compare our approach with previous approaches from both classes.

Pure scheduling-based approaches are geared towards finding minimum latency schedules or maximum fine-grained parallelism, as opposed to tileability 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, 24, 14, 45, 44].

Griebl [24] 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's FCO approach finds an allocation that satisfies the forward communication-only constraint: this enables time tiling. As shown elsewhere [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.

Lim and Lam [37, 36] proposed a 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 [35]. 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 are sought, without using any cost function. As shown through this work, without a cost func-

tion, solutions obtained even for simple input may be unsatisfactory with respect to communication cost, locality, and target code complexity.

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 a polyhedral transformation framework, thereby enabling the effective extraction of coarse-grained parallelism along with data locality optimization. At the same time, codes which cannot be tiled or only partially tiled are all handled, and traditional transformations are captured.

In addition to model-based approaches, semi-automatic and search-based transformation frameworks in the polyhedral model also exist [30, 14, 22, 45, 44]. Cohen et al. [14] and Girbal et al. [22] 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. A limitation of the recent iterative polyhedral compilation approaches [45, 44] is that the constructed search space does not include tiling and its integration poses a non-trivial challenge. Though our system now is fully model-driven, empirical iterative optimization would be beneficial on complementary aspects, such as determination of optimal tile sizes and unroll factors, and in other cases when interactions with the underlying hardware and native compiler cannot be well-captured.

Code generation under multiple affine mappings was first addressed by Kelly et al. [31]. Significant advances relying on new algorithms and mathematical machinery were made by Quilleré et al. [47] and recently by Bastoul et al. [6], resulting in a powerful open-source code generator, CLooG [13]. Our tiled code generation scheme uses Ancourt and Irigoin's [4] 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 the computed transformations. Goumas et al. [23] reported an alternate tiled code generation scheme to Ancourt and Irigoin's [4]) to address the inefficiency involved in using Fourier-Motzkin elimination – however, this is no longer an issue as the state-of-the-art uses efficient algorithms [47, 6] based on PolyLib [57, 42] and its implementation of the Chernikova algorithm [33]. Techniques for parametric tiled code generation [49, 32] were recently proposed for single statement domains for which rectangular tiling is valid. These techniques complement our system very well and we intend to explore the possibility of integrating them.

## 9. 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. We have implemented our framework in a tool to generate OpenMP parallel code from C program sections automatically. Experimental results show significantly higher performance for single core and parallel execution on multi-cores, when compared with production compilers as well as state-of-the-art research approaches. 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 exceed 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 and analyzed. Since our entire transformation framework works in the polyhedral abstraction, only the polyhedra extractor and dependence tester need to be adapted to accept a different language. It could be applied for example to very high-level languages or domain-specific languages to generate high-performance parallel code.

## 10. Availability

A beta release of the Pluto system including all codes used for experimental evaluation in this paper are available at [1].

## Acknowledgments

We would like to thank Cédric Bastoul (Paris-Sud XI University, Orsay, France) very much for CLooG. We wish to acknowledge Martin Griebl and team (FMI, Universität Passau, Germany) for the LooPo infrastructure. We thank the reviewers of the submission for detailed comments. In addition, we thank Alain Darte for useful feedback that has helped us improve the presentation. This work was supported in part by the U.S. National Science Foundation through grants 0121676, 0121706, 0403342, 0508245, 0509442, 0509467, and 0541409.

## References

[1] PLuTo: A polyhedral automatic parallelizer and locality optimizer for multicores. http://pluto-compiler.sourceforge.net.

[2] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loops. *Intl. J. of Parallel Programming*, 29(5), Oct. 2001.

[3] R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Trans. on Programming Languages and Systems*, 9(4):491–542, 1987.

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

[5] R. Andonov, S. Balev, S. Rajopadhye, and N. Yanev. Optimal semi-oblique tiling. *IEEE Trans. Par. & Dist. Sys.*, 14(9):944–960, 2003.

[6] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'04)*, pages 7–16, Sept. 2004.

[7] C. Bastoul and P. Feautrier. Improving data locality by chunking. In *Intl. Conf. on Compiler Construction (ETAPS CC)*, pages 320–335, Warsaw, Apr. 2003.

[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, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Intl. Conf. on Compiler Construction (ETAPS CC)*, Apr. 2008.

[10] U. Bondhugula, J. Ramanujam, and P. Sadayappan. Pluto: A practical and fully automatic polyhedral parallelizer and locality optimizer. Technical Report OSU-CISRC-10/07-TR70, The Ohio State University, Oct. 2007.

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

[12] 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.

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

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

[15] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhauser Boston, 2000.

[16] A. Darte, G.-A. Silber, and F. Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. *Parallel Processing Letters*, 7(4):379–392, 1997.

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

[18] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[19] P. Feautrier. Dataflow analysis of scalar and array references. *Intl. J. of Parallel Programming*, 20(1):23–53, Feb. 1991.

[20] P. Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *Intl. J. of Parallel Programming*, 21(5):313–348, 1992.

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

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

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

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

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

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

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

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

[29] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yellick. Implicit and explicit optimization for stencil computations. In *ACM SIGPLAN workshop on Memory Systems Perofmance and Correctness*, 2006.

[30] 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.

[31] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Intl. Symp. on the frontiers of massively parallel computation*, pages 332–341, Feb. 1995.

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

[33] H. LeVerge. A note on chernikova's algorithm. Technical Report Research report 635, IRISA, Feb. 1992.

[34] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *Intl. J. of Parallel Programming*, 22(2):183–205, 1994.

[35] 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.

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

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

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

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

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

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

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

[43] S. Pop, A. Cohen, C. Bastoul, S. Girbal, P. Jouvelot, G.-A. Silber, and N. Vasilache. GRAPHITE: Loop optimizations based on the polyhedral model for GCC. In *Proc. of the 4th GCC Developper's summit*, Ottawa, Canada, June 2006.

[44] L.-N. Pouchet, C. Bastoul, J. Cavazos, and A. Cohen. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *PLDI'08*, Tucson, Arizona, June 2008.

[45] 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.

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

[47] 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.

[48] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *JPDC*, 16(2):108–230, 1992.

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

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

[51] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

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

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

[54] N. Vasilache, C. Bastoul, and A. Cohen. Polyhedral code generation in the real world. In *Intl. Conf. on Compiler Construction (ETAPS CC)*, pages 185–201, Mar. 2006.

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

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

[57] D. K. Wilde. A library for doing polyhedral operations. Technical Report RR-2157, IRISA, 1993.

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

[59] M. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):452–471, 1991.

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

[61] J. Xue. *Loop tiling for parallelism*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

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

[63] 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.