The Pluto+ Algorithm: A Practical Approach for Parallelization and Locality Optimization of Affine Loop Nests

UDAY BONDHUGULA and ARAVIND ACHARYA, Indian Institute of Science
ALBERT COHEN, INRIA and ENS

Affine transformations have proven to be powerful for loop restructuring due to their ability to model a very wide range of transformations. A single multidimensional affine function can represent a long and complex sequence of simpler transformations. Existing affine transformation frameworks such as the Pluto algorithm, which include a cost function for modern multicore architectures for which coarse-grained parallelism and locality are crucial, consider only a subspace of transformations to avoid a combinatorial explosion in finding transformations. The ensuing practical trade-offs lead to the exclusion of certain useful transformations: in particular, transformation compositions involving loop reversals and loop skewing by negative factors. In addition, there is currently no proof that the algorithm successfully finds a tree of permutable loop bands for all affine loop nests. In this article, we propose an approach to address these two issues (1) by modeling a much larger space of practically useful affine transformations in conjunction with the existing cost function of Pluto, and (2) by extending the Pluto algorithm in a way that allows a proof for its soundness and completeness for all affine loop nests. We perform an experimental evaluation of both, the effect on compilation time, and performance of generated codes. The evaluation shows that our new framework, Pluto+, provides no degradation in performance for any benchmark from Polybench. For the Lattice Boltzmann Method (LBM) simulations with periodic boundary conditions, it provides a mean speedup of 1.33× over Pluto. We also show that Pluto+ does not increase compilation time significantly. Experimental results on Polybench show that Pluto+ increases overall polyhedral source-to-source optimization time by only 15%. In cases in which it improves execution time significantly, it increased polyhedral optimization time by only 2.04×.

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

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Automatic parallelization, locality optimization, polyhedral model, loop transformations, affine transformations, tiling

ACM Reference Format:
Uday Bondhugula, Aravind Acharya, and Albert Cohen. 2016. The Pluto+ algorithm: A practical approach for parallelization and locality optimization of affine loop nests. ACM Trans. Program. Lang. Syst. 38, 3, Article 12 (April 2016), 32 pages.
DOI: http://dx.doi.org/10.1145/2896389

1. INTRODUCTION

Affine transformation frameworks for loop optimization have been known to be powerful due to their ability to model a wide variety of loop-reordering transformations [Aho et al. 2006]. Affine transformations preserve the collinearity of points as well as the

This work was supported in part by the INRIA Associate Team PolyFlow.
Authors' addresses: U. Bondhugula and A. Acharya, Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560012, India; emails: {uday, aravind.acharya}@csa.iisc.ernet.in; A. Cohen, INRIA and ENS 45 rue d'Ulm, 75005 Paris, France; email: Albert.Cohen@inria.fr.

DOI: http://dx.doi.org/10.1145/2896389

ratio of distances between collinear points. This makes the problem of generating code tractable after application of such transformations. Several polyhedral transformation frameworks [Feautrier 1992b; Kelly and Pugh 1995; Lim and Lam 1998; Ahmed et al. 2001; Pluto 2008; Meister et al. 2009; Grosser et al. 2011] employ affine transformations for execution reordering. The transformations are typically applied on integer points in a union of convex polyhedra. Several works have studied the problem of code generation under affine transformations of such integer sets [Ancourt and Irigoin 1991; Kelly et al. 1995; Quilleré et al. 2000; Bastoul 2004; Verdoolaege 2010; Chen 2012], and code generators such as Cloog [2004], Omega+ [Chen 2012], and ISL [Verdoolaege 2010] exist.

Affine loop nests typically refer to loop nests that have regular data accesses and static control. The simple relationship between iterators and data locations accessed allows precise compile-time analysis. Such loop nests encompass several interesting computational domains of interest to high-performance computing, including stencil computations, image-processing computations, Lattice-Boltzmann Method (LBM) simulations, and dense linear algebra routines. In several domains, the computation itself as written by programmers may not fit into the class of affine loop nests; however, the data dependences at a higher level of abstraction may be affine in nature. Polyhedral techniques for optimization thus could also be employed on a representation extracted from a higher-level specification. Such optimization has been used for domain-specific languages (DSLs) [Henretty et al. 2013; Yuki et al. 2013; Pananilath et al. 2015; Mullapudi et al. 2015].

Dependence analysis, automatic transformation, and code generation are the three key stages of an automatic polyhedral optimizer. A number of model-driven automatic transformation algorithms for parallelization exist in the literature. Among algorithms that work for the entire generality of the polyhedral framework, there are those of Feautrier [1992a, 1992b], Lim and Lam [1997], and Griebl [2004], as well as the Pluto algorithm based on Bondhugula et al. [2008a, 2008b]. The Pluto algorithm, the most recent among these, has been shown to be suitable for architectures in which extracting coarse-grained parallelism and locality are crucial—prominently modern general-purpose multicore processors.

The Pluto algorithm employs an objective function based on minimization of dependence distances [Bondhugula et al. 2008a]. The objective function makes certain practical trade-offs to avoid a combinatorial explosion in determining transformations. The trade-offs restrict the space of transformations modeled to a subset of all valid ones. This results in the exclusion of transformations crucial for certain data-dependence patterns and application domains. In particular, the excluded transformations are those that involve negative coefficients. Negative coefficients in affine functions capture loop reversal, loop skewing by negative factors, and, more important, those transformations that include the former (reversal and negative skewing) as one or more components in a potentially long composed sequence of simpler transformations.

Although loop reversal and skewing by negative factors are just some among several well-known unimodular or affine transformations, they can be composed in numerous ways with other transformations in a sequence to form a compound transformation that may also enable loop tiling and fusion, for example. All such compound transformations currently end up being excluded from Pluto's space. In this article, we show that application domains that are affected include, but are not limited to, those that have symmetric dependence patterns, those of stencil computations, LBM simulations, and other computational techniques defined over periodic data domains.

In addition to the completeness issue in modeling the space of transformations, another missing link concerning the Pluto algorithm is the absence of a proof that the algorithm or a variant of it always succeeds in finding a tree of permutable loop

bands for any affine loop nest. A permutable loop band is a contiguous set of loops that are permutable [Aho et al. 2006]. Permutable loop nests are desirable primarily for loop tiling for parallelism and locality. For the case of single statements, it was intuitively clear that finding linearly independent hyperplanes as long as none violated any unsatisfied dependences would always yield a valid transformation. However, no formal proof was presented. More important, for multiple statements, the reasoning is nontrivial. We provide a counterexample, although contrived and rare, in which the original Pluto algorithm fails to find a solution.

In this article, we address both of these limitations of Pluto. In summary, we make the following contributions:

—extending the space of valid transformation modeled by Pluto in a way that makes it practically complete; and
—extending the algorithm in a way that allows its hyperplane search to be provably sound and complete, and providing a proof.

We call Pluto+ the algorithm resulting from these extensions. Pluto+ captures a much larger space of practically useful transformations than Pluto, while using the same objective function. The second extension to the Pluto algorithm is completely incremental in the way that it changes the original algorithm—it is noninvasive with respect to the existing optimization objectives and approach, while providing a theoretical guarantee of its successful termination for any affine loop nest. Through experimental evaluation, we demonstrate that (1) the enlarged transformation space includes transformations that provide a significant improvement in parallel performance for a number of benchmarks, and (2) Pluto+ does not pose a compilation time issue.

The rest of this article is organized as follows. Section 2 provides background, notation, and detail on the problem domains and patterns that motivate Pluto+. Section 3 and Section 4 present our solutions in detail. Experimental evaluation is presented in Section 5. Related work and conclusions are presented in Section 6 and Section 7, respectively.

2. BACKGROUND AND MOTIVATION

In this section, we introduce terminology used in the rest of this article, and describe our motivation with examples.

2.1. Notation

We use the standard notation of vectors being lexicographically greater ($\succ$) or less than ($\prec$) other vectors from Aho et al. [2006]. Vectors are named using lowercase letters. The $\cdot$ operator is used for a vector dot product or a matrix vector product. We use $\langle \vec{i}\ \vec{j} \rangle$ to represent the vector obtained by concatenating $\vec{i}$ and $\vec{j}$. We also use $\langle\rangle$ as an operator to denote iterative concatenation in a way similar to Σ. For convenience, vectors may not always be expressed in column form, but also as a comma separated tuple of its individual dimension components.

Matrices are denoted using uppercase letters. The number of rows in a matrix M is denoted by $nr(M)$. Individual rows of a matrix are referred to by using a subscript: $M[i]$ refers to row i of matrix M, $1 \leq i \leq nr(M)$; $M[i]$ is thus a row vector. The zero matrix of an appropriate dimensionality, depending on the context, is denoted in boldface by $\mathbf{0}$. $M^{\perp}$ represents the null space of M, that is, $M^{\perp}M^{T} = \mathbf{0}$, or alternatively, each row of $M^{\perp}$ is orthogonal to each row of M. As examples: if $M = [[1, 0, 0]]$, $M^{\perp} = [[0, 1, 0], [0, 0, 1]]$; if $M = [[1, 1]]$, $M^{\perp} = [[1, -1]]$. Note that $M^{\perp}$ is not unique for an M. $\vec{u}_i$ represents a

unit vector along the canonical dimension i ($i \geq 1$). I denotes the identity matrix, that is, $I[k] = \vec{u}_k$.

2.2. Index Sets and Transformations

In the polyhedral compiler framework, loops are abstracted as dimensions of polyhedra. Let S be a statement in the program with m_S loops surrounding it. The set of iterations or statement instances to be executed for a statement S is the domain or *index set* of S, and is denoted by I_S. Geometrically viewed, it is the set of integer points in a polyhedron of dimensionality m_S. As an example, an index set corresponding to a two-dimensional loop nest with loops i and j is compactly represented as:

$$\mathtt{I_S} = \{(\mathtt{i}, \mathtt{j}) : \mathtt{0} \leq \mathtt{i}, \mathtt{j} \leq \mathtt{N} - \mathtt{1}\}.$$

Any symbols in the RHS not appearing within brackets should be treated as *program parameters*. N is a program parameter here. Program parameters are symbols that are not modified in the part of the program being analyzed. These symbols typically represent problem sizes, and appear in upper bounds of loops. Let $\vec{p}$ be the vector of those program parameters, and let m_p be the number of program parameters.

An iteration of statement S is denoted by $\vec{i}_S$, that is, $\vec{i}_S \in I_S$. $\vec{i}_S$ has m_S components corresponding to loops surrounding S from outermost to innermost.

An affine transformation, T, for a k-dimensional vector, $\vec{i}$, is one that can be expressed in the form:

$$T(\vec{i}) = M \cdot \vec{i} + \vec{m},$$

where M is an $n \times k$ matrix and $\vec{m}$ is an n-dimensional vector. T is an n-dimensional affine transformation here. For all purposes in this article, all elements of these entities are integers, that is, $M \in \mathbb{Z}^{n \times k}$, $m \in \mathbb{Z}^n$, and $\vec{i} \in \mathbb{Z}^k$.

Each statement has a multidimensional affine transformation T_S, of dimensionality r, given by:

$$T_S(\vec{i}_S) = H_S \cdot \vec{i}_S + G_S \vec{p} + \vec{f}_S, \tag{1}$$

where $H_S \in \mathbb{Z}^{r \times m_S}$, $G_S \in \mathbb{Z}^{r \times m_p}$, and $\vec{f}_S \in \mathbb{Z}^r$. H_S should have full column rank for the transformation to be a one-to-one mapping. Although T_S may be called a schedule for statement S, each dimension of T_S may have other information associated with it, sequential, parallel, tilable as part of a band of loops, for example. Thus, T_S does not imply a sequential execution order, but just an order for the code generator to scan. Once scanned, appropriate loops will be marked parallel. Thus, the terms *schedule* and *transformation* are sometimes used interchangeably.

Each dimension of T_S is a one-dimensional affine function. We denote the affine function represented by the k^{th} dimension of T_S by $T_S[k]$ or ϕ_S^k, that is,

$$\phi_S^k(\vec{i}_S) = T_S[k](\vec{i}_S). \tag{2}$$

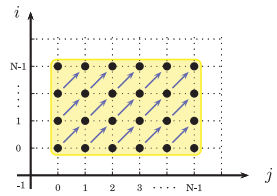
The ϕ_S^ks, $1 \leq k \leq n$, correspond to transformed loops from outermost to innermost. The ϕ functions can also be viewed as hyperplanes; throughout this article, we often refer to them as hyperplanes. The k^{th} one-dimensional affine transformation for S, ϕ_S^k, can be expressed with its complete structure as:

$$\begin{aligned} \phi_S^k(\vec{i}_S) \;=\; & \left(c_1^S \; c_2^S \; \ldots \; c_{m_S}^S\right) \cdot \vec{i}_S + \left(d_1^S \; d_2^S \; \ldots \; d_{m_p}^S\right) \cdot \vec{p} + c_0^S, \\ & c_0^S, c_1^S, \ldots, c_{m_S}^S, d_1^S, d_2^S, \ldots, d_{m_p}^S \in \mathbb{Z}. \end{aligned} \tag{3}$$

Each statement thus has its own set of c and d coefficients for each level k. For convenience, we do not specialize c_i^S, d_i^S coefficients with a k subscript, and we often drop

```
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
    A[i+1][j+1] = f(A[i][j]);
  }
}
```

(a) Code



(b) Visualization

Fig. 1. Example with dependence (1,1).

the superscript S if there is only one statement in context or if the discussion applies to any statement. The c_i^S coefficients, $1 \leq i \leq m_S$ correspond to dimensions of the index set of the statement; we will refer to them as the statement's *dimension coefficients*. The d_i^S, $1 \leq i \leq m_p$ correspond to parameters and represent parametric shifts. Finally, c_0^S represents a constant shift.

An example of an affine transformation on a two-dimensional I_S, with $\vec{i}_S = (i, j)$, is:

$$\mathtt{T}\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i-j+N \\ i+j+1 \end{pmatrix} = \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \end{pmatrix} (N) + \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Both $i - j + N$ and $i + j + 1$ are one-dimensional affine functions of i, j, and N. More compactly, we write this as $T(i, j) = (i - j + N, i + j + 1)$.

2.3. Data Dependences

The data-dependence graph for polyhedral optimization has a precise characterization of data dependences between dynamic instances of statements. Each dependence edge is a relation between source and target iterations. Although, in some works, it is represented as a conjunction of constraints called the dependence polyhedron, it can be viewed as a relation between source and target iterations involving affine constraints as well as existential quantifiers. Let $G = (S, E)$ be the data-dependence graph. For each $e \in E$, let D_e be the data-dependence relation. Then, an iteration $\vec{s}$ of statement S_i depends on $\vec{t}$ of S_j through edge e when $\langle \vec{s}, \vec{t} \rangle \in D_e$. S_i is referred to as the source statement of the dependence, and S_j the target statement. As an example, the dependence polyhedron for the dependence in blue in Figure 1 is given by:

$$\begin{aligned} D_e &= \left\{ \langle \vec{s}, \vec{t} \rangle \mid \vec{s} = (i, j),\ \vec{t} = (i', j'),\ j' = j + 1,\ i' = i + 1,\ 0 \leq i \leq N - 2,\ 0 \leq j \leq N - 2 \right\} \\ &= \left\{ (i, j, i', j') \mid j' = j + 1,\ i' = i + 1,\ 0 \leq i \leq N - 2,\ 0 \leq j \leq N - 2 \right\}. \end{aligned}$$

The datadependence graph is a directed multigraph. Thus, there can be multiple edges between statements or multiple self-edges. Note that an edge is between two static statements while the dependence polyhedron associated with the edge captures information between dynamic instances (iterations) of the source and target statements. An edge that has the same statement as its source and target (sink) is referred to as an *intrastatement* edge while one between distinct statements is an *interstatement* edge. Strongly connected components (SCCs) in the data-dependence graph have special significance. They represent a group of statements that has at least one common surrounding loop [Aho et al. 2006], since an interleaved execution of iterations belonging to different statements can be achieved only if the statements have a common surrounding loop.

```
for (i = 0; i < N; i++) {
   b[i] = f(a[i]);
}
for (i = 0; i < N; i++) {
   c[i] = f(b[N-1-i]);
}
```

(a) Code

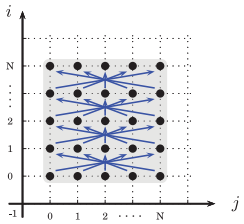
```
for (i = 0; i < N; i++) {
   b[i] = f(a[i]);
   c[N-1-i] = f(b[i]);
}
```

(b) Optimized

Fig. 2. Symmetric consumer.

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

(a) Code



(b) Visualization

Fig. 3. Symmetric dependences.

2.4. Motivation for Negative Coefficients in Transformations

This section presents the motivation for our first contribution, that of modeling a larger space of transformations than the one currently modeled by Pluto.

There are a number of situations when a transformation that involves a reversal or a negative skewing is desired. In this section, we discuss these computation patterns. The examples presented here are meant to be illustrative.

2.4.1. Reversals. Consider the two statements in Figure 2(a), S_1 and S_2, with index sets:

$$\mathtt{I}_{\mathtt{S}_1} = \{(\mathtt{i}) : 0 \leq \mathtt{i} \leq \mathtt{N} - 1\}, \quad \mathtt{I}_{\mathtt{S}_2} = \{(\mathtt{i}) : 0 \leq \mathtt{i} \leq \mathtt{N} - 1\}.$$

The schedule that corresponds to the original execution order is given by:

$$\mathtt{T}_{\mathtt{S}_1}(\mathtt{i}) = (0, \mathtt{i}), \quad \mathtt{T}_{\mathtt{S}_2}(\mathtt{i}) = (1, \mathtt{i}).$$

A transformation that exposes a parallel loop outside while fusing and exploiting locality inside involves a reversal for S_2, and is given by:

$$\mathtt{T}_{\mathtt{S}_1}(\mathtt{i}) = (\mathtt{i}, 0), \quad \mathtt{T}_{\mathtt{S}_2}(\mathtt{i}) = (\mathtt{N} - \mathtt{i} - 1, 1).$$

The transformed code is shown in Figure 2(b).

2.4.2. Skewing by Negative Factors for Communication-Free Parallelization. Consider the code in Figure 1. It has a single read-after-write (RAW) dependence represented by the distance vector (1,1).

The mapping $T(i, j) = (i - j, j)$ leads to a loop nest where the outer loop can be marked parallel while in the original code; the outer loop cannot be marked parallel. Note that this mapping involves a negative coefficient for j. $\theta(i, j) = i$ represents a valid scheduling, but this is not useful in any way for performance by itself unless a suitable placement is found for it.

2.4.3. Symmetric Dependences. Figures 2 and 3 show two other patterns in which the dependences are symmetric along a particular direction. Such examples were also studied in the synthesis of systolic arrays [Choffrut and Culik 1983; Yaacoby and

Cappello 1995], in which researchers used folding or reflections to make dependences uniform or long wires shorter. A more compiler- or code generation-centric approach is one in which an index set splitting is applied to cut the domain vertically into two halves [Bondhugula et al. 2014], and reverse and shift the halves in a way illustrated for periodic stencils in the next section. Such cuts or index set splitting are automatically generated by existing work [Bondhugula et al. 2014]; an implementation is available in Pluto.

2.4.4. Stencils on Periodic Domains. Stencils on periodic domains have long dependences in both directions along a dimension, as shown with red arrows in Figure 4(b). The corresponding code is shown in Figure 4(a). Such long dependences make tiling all loops invalid. Osheim et al. [2008] presented an approach to tile periodic stencils based on folding techniques [Choffrut and Culik 1983; Yaacoby and Cappello 1995]. Bondhugula et al. [2014] recently proposed an index set splitting technique that uses a hyperplane to cut close to the midpoints of all long dependences (Figure 4(c)). It opens the possibility of tiling through application of separate transformations on the split index sets to make dependences shorter. The sequence of transformations are shown in Figure 4(g). However, these separate transformations involve reversals as one in the sequence (Figure 4(d)). The sequence of transformations enable existing time tiling techniques [Wonnacott 2000; Bondhugula et al. 2008b; Strzodka et al. 2011; Bandishti et al. 2012], resulting in code that has been shown to provide dramatic improvement in performance, including on a SPEC benchmark (*swim* from SPECFP2000) [Bondhugula et al. 2014]. In addition, LBM codes share similarities with stencils, thus are a very significant class of codes that benefit from the approach that we propose.

2.5. Pluto Algorithm: Background and Challenges

Let $\mathbf{S}$ be the set of all statements in the portion of the program being represented in the polyhedral framework. Recall the general form of the transformation of a statement S:

$$\begin{aligned} T_S(\vec{i}) &= H_S \cdot \vec{i} + G_S \cdot \vec{p} + \vec{f}_S, \\ \phi_S^k(\vec{i}) &= T_S[k](\vec{i}) = \vec{h}_S \cdot \vec{i} + \vec{g}_S \cdot \vec{p} + c_0^S, \\ &\quad \text{where } \vec{h}_S = \left(c_1^S, c_2^S, \ldots, c_{m_S}^S\right),\ \vec{g}_S = \left(d_1^S, d_2^S, \ldots, d_{m_P}^S\right). \end{aligned} \tag{4}$$

H_S has to be full column-ranked to represent a one-to-one mapping. Certain dimensions of T_S where $\vec{h}_S = \mathbf{0}$ do not represent loops, but are what we call *scalar dimensions*. They often serve the purpose of fusion and distribution (see Section 2.4.1 example). Dimension k of T_S is a scalar dimension if $T_S[k]$ is a constant function. As an example, the first dimension of the statement transformations corresponding to Figure 2(a) are scalar dimensions; these are $T_{S_1}[0]$ and $T_{S_2}[0]$, and they distribute S_1 and S_2 at the outermost level. The number of rows in H_S (and the dimensionality of $\vec{f}_S$) is the same across all statements in spite of different statement index set dimensionalities. All zero rows can always be appended to H_S, that is, T_S can be padded with constant functions mapping all values to zero along those dimensions.

The following definitions on dependence satisfaction have been specified for a single statement for simplicity. These can be easily extended for dependences between distinct statements.

A hyperplane ϕ_S^k *weakly satisfies* a dependence e if

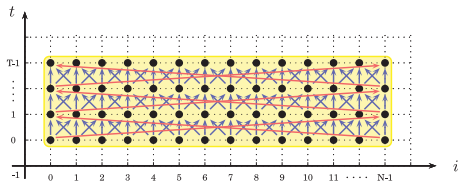
$$\phi_S^k(\vec{t}) - \phi_S^k(\vec{s}) \geq 0, \quad \forall \langle \vec{s}, \vec{t} \rangle \in D_e. \tag{5}$$

A hyperplane ϕ_S^k *satisfies* or *strongly satisfies* a dependence e if

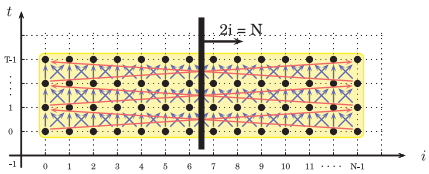
$$\phi_S^k(\vec{t}) - \phi_S^k(\vec{s}) \geq 1, \quad \forall \langle \vec{s}, \vec{t} \rangle \in D_e. \tag{6}$$

```
for (t = 0; t < T-1; t++) {
  for (i = 0; i < N; i++) {
    A[(t+1)%2][i] =
      ((i+1==N ? A[t%2][N-1] : A[t%2][i-1])
       + 2.0*A[t%2][i]
       + (i==0 ? A[t%2][0] : A[t%2][i+1])) / 4.0;
  }
}
```

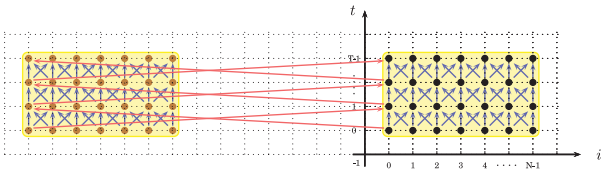
(a) A stencil on a periodic domain



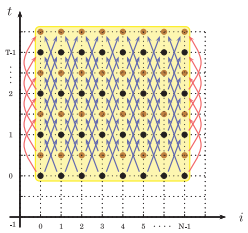
(b) Visualization for (a)



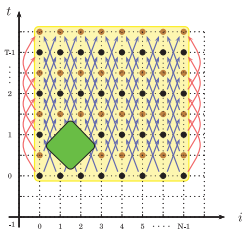
(c) Index Set Splitting



(d) Reversal of S^+: $T_{S^+}(t,i) = (t,-i)$



(e) Parametric shift (all dependences have become short)



(f) Diamond tiling

(1) ISS: $I_{S^-} = I_S \cap \{2i \leq N-1\}, \quad S^+ = I_S \cap \{2i \geq N\}$
(2) Reversal: $T_{I_{S^-}}(t,i) = (t,i), \quad T_{I_{S^+}}(t,i) = (t,-i)$
(3) Parametric shift: $T_{I_{S^-}}(t,i) = (t,i), \quad T_{I_{S^+}}(t,i) = (t,-i+N)$
(4) Diamond tiling: $T_{I_S^-}(t,i) = (t+i,t-i), \quad T_{I_S^+}(t,i) = (t-i+N,t+i-N)$

(g) Transformations

Fig. 4. A sequence of transformations that allows tiling for periodic stencils (with wraparound boundary dependences).

A dependence instance $\langle \vec{s}, \vec{t} \rangle \in D_e$ is satisfied or strongly satisfied at depth d if

$$\forall\, 0 \leq k \leq d-1 \;\; \phi_S^k(\vec{t}) - \phi_S^k(\vec{s}) = 0 \;\; \wedge \;\; \phi_S^d(\vec{t}) - \phi_S^d(\vec{s}) \geq 1. \tag{7}$$

Often, dependence edges are satisfied wholly at a particular level. A dependence $e \in E$ is considered satisfied at depth d if hyperplanes at depth d, that is, $\phi_S^d \forall S$, satisfy e as per Equation (6). A dependence is satisfied *by* depth d if all its instances are satisfied at depth $d' \leq d$ and at least one instance is satisfied at depth d. A transformation T

satisfies or strongly satisfies a dependence e if

$$T_{S_j}(\vec{t}) - T_{S_i}(\vec{s}) \succ \vec{0} \quad \forall \langle \vec{s}, \vec{t} \rangle \in D_e, \vec{s} \in I_{S_i}, \vec{t} \in I_{S_j}. \tag{8}$$

A dependence instance $\langle \vec{s}, \vec{t} \rangle \in D_e, \vec{s} \in I_{S_i}, \vec{t} \in I_{S_j}$ is *violated* by T if

$$T_{S_j}(\vec{t}) - T_{S_i}(\vec{s}) \prec \vec{0}. \tag{9}$$

A transformation T_S is *valid* if it does not violate any dependences. A loop is *parallel* if the hyperplanes in $T_S \forall S$ corresponding to that loop do not satisfy any dependence instances. Note that there can be multiple hyperplanes in $T_S \forall S$ that correspond to a loop since statements can be fused at a certain depth; however, each of those hyperplanes will be at the same depth of its respective T_S. Given a valid T_S, one can determine which loops satisfy dependences and thus the parallel loops. For a dimension k, starting from the outermost dimension ($k = 0$), let D_e^k be the instances of D_e that have not been satisfied in the previous $k-1$ levels; for $k = 0$, $D_e^0 = D_e$. Now Equation (6) can be used to check which dependence instances $\langle \vec{s}, \vec{t} \rangle$ have been satisfied at level k. This is done by intersecting the strong satisfaction constraint, $\phi^k(\vec{t}) - \phi^k(\vec{s}) \geq 1$, with D_e^k. The satisfied instances are then subtracted out from D_e^k to yield D_e^{k+1}, the yet unsatisfied instances, which will be checked for satisfaction in subsequent levels ($\geq k+1$). Alternatively, D_e^{k+1} can directly be obtained by intersecting $\phi^k(\vec{t}) - \phi^k(\vec{s}) = 0$ with D_e^k, since for any valid T_S, $\phi^k(\vec{t}) - \phi^k(\vec{s})$ cannot be negative for any $\langle \vec{s}, \vec{t} \rangle \in D_e^k$.

The Pluto algorithm determines a T_S for each statement such that its H_S has a full column rank; that is, for any statement S with dimensionality m_S, m_S linearly independent hyperplanes are found. In addition, for any $\vec{i} \in I_{S_i}$, $\vec{j} \in I_{S_j}$, $\vec{i} \neq \vec{j} \Rightarrow T_{S_i}(\vec{i}) \neq T_{S_j}(\vec{j})$; that is, the functions T_{S_i}, $\forall S_i \in \mathbf{S}$ represent a one-to-one mapping for the union of all index sets. In particular, the Pluto algorithm iteratively finds one-dimensional affine transformations, or hyperplanes, starting from outermost to innermost while maximizing tilability. It finds ϕs satisfying the following constraint for all yet unsatisfied $\langle \vec{s}, \vec{t} \rangle \in D_e, \vec{s} \in I_{S_i}, \vec{t} \in I_{S_j}, e \in E$:

$$\phi_{S_j}(\vec{t}) - \phi_{S_i}(\vec{s}) \geq 0. \tag{10}$$

This can be viewed as an extension of the sufficient condition for the validity of tiling to the case of affine dependences, of having nonnegative dependence components along tiling directions [Irigoin and Triolet 1988].

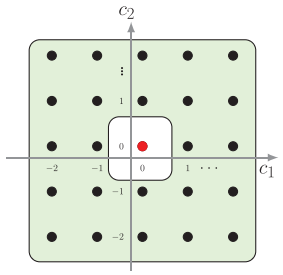
The objective function Pluto uses to find a hyperplane is that of reducing dependence distances using a bounding function. Intuitively, this reduces a factor in the reuse distances [Wolf and Lam 1991; Zhong et al. 2009], and in cache misses after synchronization or communication volume in the case of distributed memory. All dependence distances are bounded in the following way:

$$\phi_{s_j}(\vec{t}) - \phi_{s_i}(\vec{s}) \quad \leq \quad \vec{u} \cdot \vec{p} + w, \quad \langle \vec{s}, \vec{t} \rangle \in D_e. \tag{11}$$

The bounding function for the dependence distances is $\vec{u}.\vec{p} + w$, and the coefficients of $\vec{u}$ and the constant w are then minimized, in order of decreasing priority, by finding the lexicographic minimum:

$$\text{lexmin}(\vec{u}, w, \ldots, c_i^S, d_i^S, \ldots). \tag{12}$$

The lexicographic minimum objective, lexmin, for an ILP formulation was a special minimization objective first introduced by Feautrier [1991] for purposes of program analysis. It is a slight variation of a typical linear objective function. The objective finds



$$\begin{aligned} & c_1 \neq 0 \vee c_2 \neq 0, -2 \leq c_1, c_2 \leq 2, c_1, c_2 \in \mathbb{Z} \\ \Leftrightarrow \quad & c_1 + 3c_2 \geq 1 - 9\delta_S, \\ & -c_1 - 3c_2 \geq 1 - 9(1 - \delta_S),\ \delta_S \in \{0, 1\} \end{aligned}$$

Fig. 5. Avoiding the zero solution: $(c_1, c_2) \neq (0, 0)$, $-2 \leq c_1, c_2 \leq 2$.

the lexicographically smallest vector with respect to the order of variables specified in the ILP. For Equation (12), the best solution of $\vec{u} = \vec{0}$, $w = 0$ corresponds to a parallel loop, since it implies that the LHS of Equation (11) is 0 for all dependence instances.

The remaining background will be presented in Section 3 and Section 4 when our solution approach is described.

3. EXPANDING THE SPACE OF TRANSFORMATIONS MODELED

In this section, we present the first of our two contributions: the modeling of a larger, more practically complete space of valid affine transformations for a given input. Recall notation from Section 2.5. For convenience, at some places, we drop the superscript S from c_is and d_is if the discussion applies to all statements.

Note that Equation (10) and Equation (11) both have a trivial zero-vector solution ($\forall S$ $\forall i, c_i^S = 0, d_i^S = 0$), and getting rid of it is nontrivial if the dimension coefficients of ϕ^S, the c_is, $i \geq 1$, are allowed to be negative. Alternatively, if we trade off expressiveness for complexity and assume that all $c_i \geq 0$, the zero solution for a statement S is avoided easily and precisely with:

$$\sum_{i=1}^{m_S} c_i^S \geq 1. \tag{13}$$

Pluto currently makes this trade-off [Bondhugula et al. 2008b]. On the other hand, when negative coefficients are allowed, the removal of the zero solution from the full space leads to a union of a large number of convex spaces. For example, for a 3D statement, it leads to eight subspaces. For multiple statements, one would get a product of the number of these subspaces across all statements, leading to a combinatorial explosion: for example, for ten 3D statements, there would be more than 1 billion subspaces to consider at the outermost level. A similar issue exists when enforcing conditions for linear independence. We now propose a compact approach to address both problems, with compilation time on par with the original Pluto algorithm and implementation.

3.1. Excluding the Zero Solution

Intuitively, the challenge involved here is that removing a single point from the "middle" of a polyhedral space gives rise to a large disjunction of several polyhedral spaces, that is, several different cases to consider. As explained earlier, the possibilities grow exponentially with the sum of the dimensionalities of all statements. Figure 5 illustrates this for a two-dimensional space.

The approach that we propose relies on the following observation. The magnitudes of c_i, $1 \leq i \leq m_S$ represent loop scaling and skewing factors; in practice, we never need them to be very large. Let us assume that $\forall S$, $\forall i$, $-b \leq c_i^S \leq b$, for $b \geq 1$. Although the

description that follows will work for any constant $b \geq 1$, we will use $b = 4$ to make resulting expressions more readable. It also turns out to be the value that we chose for our implementation. Thus, $-4 \leq c_i^S \leq 4$, $1 \leq i \leq m_S$, $\forall S$. When c_i^Ss are bounded this way, the scenario under which all c_is $1 \leq i \leq m_S$ are zero is captured as follows:

$$(c_1^S, c_2^S, \ldots, c_{m_S}^S) = \vec{0} \Leftrightarrow \sum_{i=1}^{m_S} 5^{i-1} c_i^S = 0.$$

The reasoning is that, since $-4 \leq c_i \leq 4$, each c_i can be treated as an integer in base 5. Since $5^i > \sum 5^{i-1} c_i$ for every c_i in base 5, c_i, $1 \leq i \leq m_S$ will all be simultaneously zero only when $\sum_{i=1}^{m_S} 5^{i-1} c_i$ is zero. This, in turn, means:

$$(c_1^S, c_2^S, \ldots, c_{m_S}^S) \neq \vec{0} \Leftrightarrow \left| \sum_{i=1}^{m_S} 5^{i-1} c_i^S \right| \geq 1.$$

The absolute value operation can be eliminated by introducing a binary decision variable $\delta_S \in \{0, 1\}$, and enforcing the following constraints:

$$\sum_{i=1}^{m_S} 5^{i-1} c_i^S \geq 1 - \delta_S 5^{m_S}, \tag{14}$$

$$-\sum_{i=1}^{m_S} 5^{i-1} c_i^S \geq 1 - (1 - \delta_S) 5^{m_S}. \tag{15}$$

Explanation. Note that $5^{m_S} - 1$ and $1 - 5^{m_S}$ are the largest and the smallest values that the LHSs of Constraint (14) and Constraint (15) can take. Hence, when $\delta_S = 0$, Constraint (14) enforces the desired outcome and Constraint (15) does not restrict the space in any way. Similarly, when $\delta_S = 1$, Constraint (15) is enforcing and Constraint (14) does not restrict the space. Thus, $\delta_S = 0$ and $\delta_S = 1$, respectively, cover the half-spaces given by

$$\sum_{i=1}^{m_S} 5^{i-1} c_i^S \geq 1 \quad \text{and} \quad \sum_{i=1}^{m_S} 5^{i-1} c_i^S \leq -1.$$

Figure 5 illustrates this approach for a two-dimensional coefficient space with $-2 \leq c_i \leq 2$.

3.2. Modeling the Complete Space of Linearly Independent Solutions

When searching for hyperplanes at a particular depth, the Pluto algorithm incrementally looks for linearly independent solutions until all dependences are satisfied and transformations are full column-ranked, that is, one-to-one mappings for index sets. Determining constraints that enforce linear independence with respect to a set of previously found hyperplanes introduces a challenge similar in nature to, but harder than that of, zero solution avoidance.

Let H_S be the matrix with rows representing a statement's dimension coefficients $(c_i^S, 1 \leq i \leq m_S)$ already found from the outermost level to the current depth. Let $H_S^\perp$ be the subspace orthogonal to H_S, that is, each row of $H_S^\perp$ has a null component along the rows of H_S ($H_S^\perp \cdot H_S^T = 0$). For example, let H_S be [[1 0 0]]. Then,

$$H_S^\perp = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

For a hyperplane to be linearly independent of previous ones, it should have a nonzero component in the orthogonal subspace of previous ones, that is, a nonzero component along at least one of the rows of $H_S^{\perp}$. We thus have two choices for every row of $H_S^{\perp}$, for the component being ≥ 1 or ≤ -1. The Pluto algorithm currently chooses $c_2 \geq 0, c_3 \geq 0, c_2 + c_3 \geq 1$, a portion of the linearly independent space that can be called the *nonnegative orthant*. However, the complete linearly independent space is actually given by $\mid c_2 \mid + \mid c_3 \mid \geq 1$. The restriction to the nonnegative orthant among the four orthants may lead to the loss of interesting solutions, as the most profitable transformations may involve hyperplanes with negative coefficients. On the other hand, considering all orthants for each statement leads to a combinatorial explosion. For a 3D statement, the space of linearly independent solutions has four orthants once the first hyperplane has been found. If there are multiple statements, the number of cases to be considered, if all of the orthants have to be explored, is the product of the number of orthants across all statements.

To state it in general: for the existing Pluto algorithm, if H_S is the linear part of a transformation found, then the constraints used for enforcing linear independence for a future hyperplane to be found ($\vec{h}_S$) are

$$\forall S \left(\forall k \;\; H_S^{\perp} \cdot \vec{h}_S \geq 0, \quad \sum_{k=1}^{nr(H_S^{\perp})} H_S^{\perp}[k] \cdot \vec{h}_S \geq 1 \right). \tag{16}$$

However, the desired linear independence constraints for a complete modeling are captured by

$$\forall S \quad \sum_{k=1}^{nr\left(H_S^{\perp}\right)} \left| H_S^{\perp}[k] \cdot \left(c_1^S, c_2^S, \ldots, c_{ms}^S\right) \right| \geq 1. \tag{17}$$

We address this problem in a way similar to the way that we avoided the zero solution in the previous section. We consider each of the rows of $H_S^{\perp}$, and compute the minimum and maximum values it could take given that the coefficients are bounded below and above by $-b$ and b. To avoid all of the components in the orthogonal subspace being simultaneously zero, an expression is then constructed from Equation (17) similar to the decomposition of the zero-avoidance constraint into Constraints (14) and (15). As an example, consider $H_S = [[1, 1, 0]]$. Then, $H_S^{\perp} = [[1, -1, 0], [0, 0, 1]]$. If $b = 4$, the maximum value taken by any row will be 8. To ensure linear independence, we require at least one of the rows to be nonzero, that is,

$$(c_1 - c_2, c_3) \neq \vec{0} \;\Leftrightarrow\; |c_1 - c_2| + |c_3| \geq 1.$$

Given the bounds, this can now be captured through a single binary decision variable $\delta^l \in \{0, 1\}$, as follows:

$$\begin{aligned} 9(c_1 - c_2) + c_3 \;&\geq\; 1 - \delta_S^l 9^2, \\ -9(c_1 - c_2) - c_3 \;&\geq\; 1 - (1 - \delta_S^l)9^2. \end{aligned}$$

A strength of this technique from a scalability viewpoint is that only a single decision variable per statement is required irrespective of the dimensionality of the statement or the level at which the transformation is being found. One may observe that linear independence constraints, whenever they are added, automatically imply zero solution avoidance. However, we have proposed a solution to both of these problems in order to maintain generality and flexibility for future customization of modeling, and to make the presentation clearer as the latter follows from the former.

3.3. Bounds on Coefficients

Recall that we bounded all c_i, $1 \leq i \leq m_S$ by b and $-b$ when we chose $b = 4$ for clearer illustration in the previous section. Besides these coefficients, the constant shifts c_0s, and parametric shifts d_is are relative between statements; thus, there is no loss of generality in assuming that all of them are nonnegative. In addition, the coefficients of $\vec{u}$ and w are nonnegative. Hence, all variables in our Integer Linear Programming (ILP) formulation are bounded from below while the statement dimension coefficients and the binary decision variables are bounded from both above and below.

3.4. Keeping the Coefficients Small

In the scenario in which all or a number of the coefficient values are valid solutions with the same cost, for practical reasons, it is desirable to choose smaller ones, that is, solutions as close to zero as possible. A technique that is perfectly suited to achieve this goal is one of minimizing the sum of the absolute values of c_i^S for each statement S, that is, for each S, we define an additional variable, c_{sum}^S such that

$$c_{sum}^S = \sum_{i=1}^{m_S} |c_i^S|.$$

This, in turn, can be put in a linear form by expressing the RHS as a variable larger than the maximum of 2^{m_S} constraints, each generated using a particular sign choice for c_i^S, then minimizing c_{sum}. For example, for a 2D statement, we obtain

$$c_{sum} \geq \max(c_1 + c_2, -c_1 + c_2, c_1 - c_2, -c_1 - c_2).$$

In general, for each statement S, we have that

$$c_{sum}^S \geq \max\left(\pm c_1^S \pm c_2^S \cdots \pm c_{m_S}^S \right). \tag{18}$$

c_{sum}^S is then included in the minimization objective at a particular position, which we will see in Section 3.5, so that it is guaranteed to be minimized to be equal to the sum of the absolute values of c_i^S.

3.5. The Objective Function

Now, with Constraints (14) and (15), and similar ones for linear independence, the δ_S, δ_S^ls can be plugged into Pluto's lexicographic minimum objective at the right places with the following objective function:

$$\begin{aligned} &\text{lexmin}\,(\vec{u}, w, \\ &\vdots \\ &c_{sum}^{S_i}, c_1^{S_i}, c_2^{S_i}, \ldots, c_{m_S}^{S_i}, d_1^{S_i}, d_2^{S_i}, \ldots d_{m_p}^{S_i}, c_0^{S_i}, \delta_{S_i}, \delta_{S_i}^l, \\ &\vdots \\ &). \end{aligned} \tag{19}$$

Note that c_{sum} coefficients appear ahead of the statement's dimension coefficients, and since Equation (18) are the only constraints involving c_{sum}, minimizing c_{sum} minimizes the sum of the absolute values of each statement's dimension coefficients, with c_{sum} in the obtained solution becoming equal to that sum.

3.6. Complexity

In summary, to implement our approach on top of the Pluto algorithm, we used three more variables for each statement, c_{sum}, δ, and δ^l, the last two of which are binary decision variables, that is,

(1) δ_S to avoid the zero solution,
(2) δ^l_S for linear independence,
(3) and c^S_{sum} to minimize the sum of absolute values of a statement's dimension coefficients.

In addition, due to Equation (18), the number of constraints added for each statement is exponential in the dimensionality of the statement, m_S. Given that m_S is small, the increase in number of constraints is low relative to the total number of constraints that are already in the system, in particular, the constraints resulting from dependence-wise tiling validity (Equation (10)) and dependence distance bounding (Equation (11)). Since the number of variables in the ILP formulation increases by three times the number of statements, this extension to the Pluto algorithm does not change its time complexity with respect to the number of statements and number of dependences. Note that our approach does not change the top-level iterative algorithm used in Pluto, but only the space of transformations modeled.

3.7. Completeness

When compared to the original Pluto algorithm, although transformations that involve negative coefficients are included in the space, due to the bounds that we place on coefficients corresponding to statement's dimensions ($-b$ and b), we are also excluding transformations that originally existed in Pluto's space. To some extent, this bound can be adjusted and increased to reasonable limits if desired. In Section 5, we present results on the sensitivity of the Pluto+ algorithm runtime to b, and show that b can be increased to very large values (tested up to 1023) with no impact on the runtime of Pluto+. However, we did not find the need to use larger values of b because, (1) coefficients larger than the value used ($b = 4$) are not needed for any of the kernels that we have encountered, and (2) large values (of the order of loop bounds when the latter are constant) introduce spurious shifts and skews, and should be avoided anyway. Note that no upper bound is placed on the shift coefficients of transformations. In summary, although we cannot theoretically claim a complete modeling of the transformation space, it is near-complete and complete for all practical purposes.

4. EXTENDING THE PLUTO ALGORITHM

In this section, we present our second contribution: extending the Pluto algorithm to make it provably sound and complete for any affine loop nest.

Let O_S be the *original schedule* for S. The original schedule specifies the sequential execution that captures the semantics of the program portion from which the polyhedral representation was extracted. It can be viewed as the multidimensional one-to-one affine transformation that, when applied, would reproduce the original affine loop nest. A polyhedral extraction front end, such as Clan or Pet, primarily produces as output, for each statement, its index set, data accesses, and its original schedule. This information is then used to compute dependences. For a single statement S, $O_S(\vec{i}) = I_S \cdot \vec{i}_S$, is the identity transformation. For multiple statements, O_S may have additional rows that are scalar dimensions that capture loop distribution, as explained in Section 2.5. Like T_S, O_S always has the same number of rows for all S.

We now introduce a result that is used in the formal proofs presented later in this section.

```
for(i = 0; i < N; i++) {
    for(j = 0; j < N; j++) {
        a[i] = f(i); /* S1 */
        a[j] = g(j); /* S2 */
    }
}
```

(a) Original

```
#pragma omp parallel for
for (t1 = 0; t1 <= N-1; t1++)
  for (t2 = 0; t2 <= N-1; t2++)
    for (t3 = max(0, -t1+t2);
         t3 <= min(N-1, -t1+t2+N-1); t3++) {
      if (t1 == t2) {
        a[t1] = f(t1);
      }
      if (t2 == t3) {
        a[t1] = g(t1);
      }
    }
```

(b) Parallelized

Fig. 6. A contrived example in which communication-free parallelism exists.

LEMMA 4.1. *If a hyperplane $\vec{h}_{n+1}$ depends linearly on hyperplanes $\vec{h}_1, \vec{h}_2, \ldots, \vec{h}_n$, and $\vec{h}_{n+1}$ strongly satisfies a dependence e, and $\vec{h}_1, \vec{h}_2, \ldots, \vec{h}_n$ weakly satisfy e, then the transformation obtained from the sequence of hyperplanes $\vec{h}_1, \ldots, \vec{h}_n$ strongly satisfies e.*

PROOF. Since $\vec{h}_{n+1} = \sum_{i=1}^{n} \lambda_i \vec{h}_i$, from Equation (6), $\sum_{i=1}^{n} \lambda_i \vec{h}_i \cdot (\vec{t} - \vec{s}) \geq 1 \; \forall \langle \vec{s}, \vec{t} \rangle \in D_e$. In our context, $\vec{h}_i \cdot (\vec{t} - \vec{s}) \in \mathbb{Z}$. As a result, for any $\langle \vec{s}, \vec{t} \rangle \in D_e$, there exists a $\vec{h}_k$, $1 \leq k \leq n$, such that $\vec{h}_k \cdot (\vec{t} - \vec{s}) \geq 1$. Since $\vec{h}_i \cdot (\vec{t} - \vec{s}) \geq 0$ (by Equation (5)) for $1 \leq i \leq n$ and $\forall \langle \vec{s}, \vec{t} \rangle \in D_e$, $[\vec{h}_1, \vec{h}_2, \ldots, \vec{h}_n] \cdot (\vec{t} - \vec{s}) \succ \vec{0}$. Hence, e is strongly satisfied by the transformation $[\vec{h}_1, \vec{h}_2, \ldots, \vec{h}_n]$ (by Equation (8)). □

The linear independence constraint used in Pluto and given by Equation (16), as well as the one introduced to model a larger space of transformations (Equation (17)), both enforce linear independence in a greedy way for all statements at a particular level. We now show that the greedy nature of this approach can lead to a scenario in which no more permutable loops are found and the algorithm makes no further progress, although a valid completion of transformation exists.

4.1. An Example

Consider the code in Figure 6(a). Communication-free parallelism can be obtained by parallelizing the outermost loop after applying the following transformation:

$$T(S_1) = (i, i, j, 0), \qquad T(S_2) = (j, i, i, 1). \tag{20}$$

The existing Pluto algorithm will be unable to find such a transformation. In fact, it fails to find a complete transformation. Once the first hyperplane ($\phi^1_{S_1}(i, j) = i$, $\phi^1_{S_2}(i, j) = j$) is found, subsequent hyperplanes cannot be found unless the linear independence constraint is relaxed for one of the two statements. The first hyperplane found is a communication-free parallel loop. The transformation is not yet complete and there are dependences that remain to be satisfied. We believe that a pattern such as this one is less common; it is the only one that we have encountered so far.

4.2. Eager and Lazy Modes in Search for Hyperplanes

Our extension to the algorithm modifies the greedy nature of the search for linearly independent hyperplanes. In the original Pluto algorithm, linear independence is enforced for *all* statements for which sufficient solutions have not been found. The observation that we make here is that, in order to make progress, linear independence need not be enforced for all statements at once. We introduce a new mode for the

algorithm that we call the *lazy* mode, in which progress is made slowly. We call the original approach of the Pluto algorithm the *eager* mode.

The objective of introducing a lazy mode is to allow the algorithm to make progress in all cases. The lazy mode is intended to be used only as a fallback. It allows slower progress than the eager mode but, crucially, always keeps the scheduling hyperplanes associated with the original schedule in its space. This is proved later in this section. We now first describe the exact constraints that are used in the lazy mode.

4.2.1. Linear Independence in the Lazy Mode. First, we note that the linear independence constraint of Pluto in Equation (16) can be changed to the following, which enforces linear independence for at least one statement:

$$\forall S \;\; \forall k \;\; H_S^{\perp}[k] \cdot \vec{h}_S \geq 0, \;\; \sum_{\forall S} \sum_{k=1}^{nr(H_S^{\perp})} H_S^{\perp}[k] \cdot \vec{h}_S \geq 1. \tag{21}$$

This is sufficient to make progress, although the progress is slow and this is not desirable in general. However, the key is that the original schedule is still in the space, which will allow progress where the original algorithm did not. As an example, consider the example of Section 4.1, in which the original Pluto algorithm failed to find any more hyperplanes after the first hyperplane ($\phi^1_{S_1}(i,j) = i$, $\phi^1_{S_2}(i,j) = j$) was found. The use of Constraint (16) would have led to the linear independence constraints, $c_2^{S_1} \geq 1, c_1^{S_2} \geq 1$, while using Constraint (21) would lead to $c_2^{S_1} + c_1^{S_2} \geq 1$. The former constraint does not allow a valid completion of the transformation since there are unsatisfied dependence instances from S_1 to S_2 and some from S_2 to S_1 – these neither allow a distribution of S_1 and S_2 nor a valid fusion (for example, $T_{S_1} = (i, j)$, $T_{S_2} = (j, i)$ is invalid). However, the latter constraint ($c_2^{S_1} + c_1^{S_2} \geq 1$) obtained from Constraint (21) allows the valid T_S shown in Equation (20) to be found.

Constraint (21) can be improved further. Statements that are in unconnected components of the dependence graph do not affect each other, that is, their transformations can be chosen independently. Hence, linear independence can be enforced on at least one statement in a connected component, and for every connected component, as opposed to enforcing it on at least one statement among all statements. This allows faster progress in a lazy mode without any loss of generality or expressiveness. If C_1, $C_2, \ldots, C_r$ are the connected components of the data-dependence graph, a better linear independence constraint will thus be given by

$$\forall C_i \left(\forall S \in C_i \;\; \forall k \;\; H_S^{\perp}[k] \cdot \vec{h}_S \geq 0, \;\; \sum_{\forall S \in C_i} \sum_{k=1}^{nr(H_S^{\perp})} H_S^{\perp}[k] \cdot \vec{h}_S \geq 1 \right). \tag{22}$$

Both Constraints (21) and (22) are sufficient to obtain a solution for the code in Figure 6. However, we still cannot guarantee progress unless we model the linear independence constraint in a manner that is aware of the SCCs of the dependence graph. Recall that statements inside an SCC have at least one common surrounding loop. Let R^i_1, R^i_2, $\ldots$, R^i_m be the SCCs of C_i. Let $H_{R^i_j}$ represent the matrix obtained by concatenating H_S for all $S \in R^i_j$ row-wise, that is, $H_{R^i_j} = \langle \rangle_{S \in R^i_j} H_S$, and let $H^{\perp}_{R^i_j}$ be its orthogonal subspace. Similarly, let $\vec{h}_{R^i_j}$ be the hyperplane obtained by concatenating hyperplanes of all $S \in R^i_j$ (see Section 2.1 for notation). Then, the linear independence

should be specified as follows:

$$\forall C_i \Bigg(\forall R_j \in C_i \;\; \forall \big(1 \le k \le nr\big(H^{\perp}_{R_j}\big)\big) \;\; H^{\perp}_{R_j}[k] \cdot \vec{h}_{R_j} \ge 0, \sum_{\forall R_j \in C_i} \sum_{k=1}^{nr\left(H^{\perp}_{R_j}\right)} H^{\perp}_{R_j}[k] \cdot \vec{h}_{R_j} \ge 1 \Bigg). \tag{23}$$

Constraint (23) ensures that a transformation corresponding to the common surrounding loop, in the original program, of the statements belonging to the same SCC is retained in the space. We prove this in Theorem 4.4. As mentioned in Section 3.2, the zero solution avoidance constraints are not needed since linear independence ensures avoidance of the zero solution.

4.3. The Pluto+ Algorithm

The updated algorithm is presented as Algorithm 1, which we call the Pluto+ algorithm. The proposed extension takes effect only when the eager mode fails. Hence, there is no additional overhead for cases it does not apply to or benefit. The changes are in Step 12 and Step 14 for the linear independence constraints addition, and Steps 18 to 21. Step 14 finds a band of permutable loops. The linear independence constraints added in Step 12 depend on the mode. Linear independence constraints in the eager mode are given by Constraint (17), and by Constraint (23) in the lazy mode. Step 15 computes which dependences were satisfied by solutions found at Step 14. Step 27, the termination condition, checks whether all transformations are one-to-one (full-ranked) and whether all dependences have been satisfied.

The Pluto or Pluto+ algorithm can be viewed as transforming to a tree of permutable hyperplane bands, that is, each node of the tree is a set of permutable hyperplanes. Step 14 of Algorithm 1 finds such a band of permutable hyperplanes. If all hyperplanes are tilable, there is just one node containing all permutable hyperplanes. On the other extreme, if no loops are tilable, each node of the tree has just one loop, and no tiling is possible. At least two hyperplanes should be found at any level (without dependence removal/cutting) to enable tiling. Dependences from previously found solutions are thus not removed unless they have to be (Step 16) to allow the next permutable band to be found, and so on. Hence, partially tilable or untilable input is handled. Loops in each node of the target tree can be strip-mined/interchanged when there are at least two of them; however, it is invalid to move a strip-mined loop across different levels in the tree.

4.4. Proof of Soundness and Completeness

We now prove that the Pluto algorithm terminates with a valid transformation for a single statement, and that the Pluto+ extension always terminates for any affine loop nest. We first give a brief sketch of the proof. The main argument relies on the fact that the algorithm always has one of the original hyperplanes in its space until it has found enough hyperplanes. Hence, if dependences come from an affine loop nest for which the original schedule is a valid one, we show that there is always progress and the terminating condition is met. The extension proposed is not required to prove that the algorithm is sound and complete for the single statement case, that is, for the single statement, the following proof shows that the Pluto algorithm has these properties. However, for the case of multiple statements, the extension is necessary.

THEOREM 4.2. *Transformations found by the Pluto + algorithm are always valid.*

ALGORITHM 1: The Pluto+ Algorithm

Require: Data-dependence graph $G = (\mathbf{S}, E)$, $D_e \forall e \in E$

1: $mode \leftarrow$ EAGER
2: **for** each dependence $e \in E$ **do**
3: Compute validity constraints V_e: apply Farkas Lemma [Feautrier 1992a; Schrijver 1986] to Equation (10) under D_e, and eliminate all Farkas multipliers
4: Compute bounding function constraints B_e: apply Farkas Lemma to Equation (11) under D_e, and eliminate all Farkas multipliers
5: $C_e \leftarrow V_e \cup B_e$
6: **end for**

7: **repeat**
8: $C \leftarrow \varnothing$
9: **for** each dependence edge $e \in E$ **do**
10: $C \leftarrow C \cup C_e$
11: **end for**
12: L $\leftarrow$ Get linear independence constraints using Equation (17) if *mode* is EAGER else using Equation (23)
13: $C \leftarrow C \cup L$
14: Find as many linear independent solutions to C as possible with the lexicographic minimal objective in Equation (19), while adding linear independence constraints (depending on *mode*) after each solution is found. Each obtained solution is of the form $\phi_S(\vec{i}_S) = \vec{h}_S \cdot \vec{i}_S + \vec{g}_S \cdot \vec{p} + c_0^S$, and appends a new dimension to $T_S, \forall S$
15: $E_c \leftarrow$ dependences satisfied by solutions found in Step 14
16: $E \leftarrow E - E_c$
17: **if** no solutions were found in Step 14 **then**
18: **if** no edges exist between SCCs of G **then**
19: /* Assertion should never fail by design */
20: **assert** ($mode$ = EAGER);
21: $mode \leftarrow$ LAZY
22: **else**
23: Select two or more SCCs of G, and order statements belonging to these SCCs by inserting *scalar dimensions* in T_S; let E_r be the dependences between these SCCs
24: $E \leftarrow E - E_r$
25: **end if**
26: **end if**
27: **until** $H_{S_i}^{\perp} = \mathbf{0}\ \forall S_i \in \mathbf{S}$ and $E = \varnothing$

Ensure: $T_S\ \forall S \in \mathbf{S}$

PROOF. Due to the validity constraint of Step 27, a dependence will never be violated at any level. The termination condition ensures that all dependences are satisfied, that is, two different time points are assigned to dependent iterations respecting order. Hence, if the repeat-until loop of Algorithm 1 terminates, the transformations found are always valid. □

We now present proofs for the completeness of the Pluto+ algorithm. We first present the proof of termination for the single statement case. The proof for the multiple-statements case builds on the former. The proof for the single-statement case also shows that the original Pluto algorithm always terminates for a single statement.

THEOREM 4.3. *The Pluto and Pluto+ algorithms always terminate for a single-statement affine loop nest.*

PROOF. Let us assume, to the contrary, that the termination condition of Algorithm 1 (Step 27) is never reached, that is, either a sufficient number of solutions were not

found ($H_S^\perp \neq \mathbf{0}$) or there were unsatisfied edges ($E \neq \varnothing$), or both. In this case, there is trivially one strongly connected component comprising the single statement, and $O_S(\vec{i}) = I_S \cdot \vec{i}_S$.

Case $H_S^\perp \neq \mathbf{0}$: In this case, enough linearly independent solutions were not found for S, and there would be no progress only if no solutions were found at Step 14. Since there is only one SCC, in the case of Pluto+, the algorithm would have switched to lazy mode without any further change to the constraints. We now argue that Step 12 would at least find one solution, and the dimensionality of the image space of $H_S^\perp$ would reduce every iteration of the repeat-until loop. Since the original computation comes from a valid affine loop nest, the original schedule, O_S, comprising hyperplanes represented by unit vectors $\vec{u}_1 = (1, 0, \ldots, 0)$, $\vec{u}_2 = (0, 1, \ldots, 0), \ldots, \vec{u}_{m_S} = (0, 0, \ldots, 1)$ is always a valid transformation. Now, at least one of these unit vectors is linearly independent to all rows in H_S since $H_S^\perp \neq \mathbf{0}$. Of all these that are linearly independent, let $\vec{u}_m$ be the lexicographically largest. Now, the hyperplanes in I_S that are lexicographically larger than $\vec{u}_m$, if any, are linearly dependent on the hyperplanes of H_S, that is, all dependences that are satisfied by those hyperplanes are satisfied by H_S (Lemma 4.1). Thus, $\vec{u}_m$ cannot violate any dependences that are not satisfied by H_S. If it did, O_S would not have been a valid transformation. Thus, $\vec{u}_m$ is a valid hyperplane linearly independent of H_S, thus is a candidate that will be found in Step 12. This reduces the dimensionality of $H_S^\perp$ by one until $H_S^\perp = \mathbf{0}$.

Case $H_S^\perp = \mathbf{0}$, $E \neq \varnothing$: Now, consider the case in which sufficient linearly independent solutions have been found ($H_S^\perp = 0$), but E still has an (unsatisfied) edge, say, e. This implies that there exists an $\langle \vec{s}, \vec{t} \rangle \in D_e$, $\vec{s} \neq \vec{t}$, such that $H_S \cdot \vec{t} - H_S \cdot \vec{s} = \vec{0}$, that is, $\vec{t} - \vec{s}$, which is not $\vec{0}$, is in H_S's null space, which is a contradiction.

Note that this proof holds even if the lazy mode did not exist. It thus holds even for the original Pluto algorithm. □

THEOREM 4.4. *The Pluto+ algorithm terminates for any affine loop nest with multiple statements.*

Let us assume, to the contrary, that the algorithm made no progress. This means that the mode had been switched to lazy, no solutions were subsequently found, and no unsatisfied dependences between SCCs existed (Step 18). Since the termination condition was not met, there either exists at least one statement S with $H_S^\perp \neq \mathbf{0}$, or $E \neq \varnothing$.

Case $H_{S_k}^\perp \neq \mathbf{0}$ for some S_k: similar to the single-statement case, let the hyperplane from the original schedule, which is linearly independent of the already found hyperplanes in H_{S_k} and the lexicographically largest among all such hyperplanes, be $\vec{u}_m^{S_k}$. We now show that there exists a transformation that corresponds to a level of the original schedule that is still within our space, and that Step 14 should have found it.

Now, as argued in Theorem 4.3, $\vec{u}_m^{S_k}$ does not violate any intrastatement dependences not satisfied by hyperplanes in H_{S_k}. A dependence violation resulting from appending $\vec{h}_m$ to H_{S_k} would imply that the same dependence was violated by the original schedule, O_{S_k} (recall O_S defined at start of this section), which is a contradiction. For interstatement dependences, clearly, Step 24 can cut and satisfy all interstatement dependences where the statements lie in two different SCCs, until we reach a scenario in which we are left with only unconnected strongly connected components, that is, G contains SCCs that are themselves not connected. Consider such a scenario, and let $R_1, R_2, \ldots, R_n$ be the SCCs. Consider a single SCC R_i since each R_i can now be treated in isolation. In this case, $\langle \rangle_{S_j \in R_i} O_{S_j}[m]$ is linearly independent of H_{R_i}. We now show that $O_{S_j}[m]$ is a valid hyperplane for $S_j \in R_i$, and is modeled in the space restricted by Constraint (22).

Assume, to the contrary, that an edge e in R_i is violated by this transformation. Since all dimensions $\langle\rangle_{S_j \in R_i} O_{S_j}[l]$, $1 \leq l \leq m-1$ are linearly dependent on the ones $H_{R_i}[l]$, $1 \leq l \leq m-1$, these dimensions will not satisfy any more dependences than the ones satisfied by the latter (Lemma 4.1). Hence, the edge e will also be violated by the original schedule O_S—a contradiction. This argument holds for all SCCs, R_i. Hence, more solutions can be found, and there is always progress at Step 14 if $H_{S_k}^{\perp} \neq 0$ for some S_k until $H_{S_k}^{\perp}$ becomes $\mathbf{0}$ $\forall S_k$.

Case $\forall S_i$ $H_{S_i}^{\perp} = 0$, $E \neq \varnothing$: Now, $e \in E$ has to be an interstatement edge; it cannot be an intrastatement edge since Theorem 4.3 would apply in that case to lead to a contradiction. e also cannot be an edge between two SCCs since Step 24 should have removed such an edge from E, and inserted a scalar dimension in the transformations that satisfies the edge. e also cannot be an edge between two statements in the same SCC since the presence of two statements in an SCC implies that more hyperplanes corresponding to common surrounding loops remain to be found, contradicting $H_S^{\perp} = \mathbf{0}$. Thus, $e \notin E$, which is a contradiction. Hence, there is always progress until $E = \varnothing$. □

4.5. Impact of Lazy Mode

For the code in Figure 6, relaxing the linear independence constraint in the way proposed in this section allows Pluto+ to find the complete transformation, and the outer loop after transformation can be marked parallel. As stated earlier, such scenarios arise only occasionally; the lazy mode is provided for completeness as a fallback. Note that the switch to lazy mode happens only when the eager mode fails, and the hyperplanes found up until then (in the eager mode) are still part of the final solution. Thus, one has all the intended behavior with regard to the objective (communication-free parallelism, permutability, and so on) up until then. In the lazy mode, since linear independence may not be ensured for all statements, the objective may not have the desired effect to the same extent. However, a valid completion is still useful—for example, we find outer parallelism with fusion for the example in Figure 6. For all benchmarks in Polybench [2010] and the ones that we evaluate in the next section, the eager mode is sufficient, and the algorithm never switches to the lazy mode.

Negative coefficients and lazy mode. Our objective in using the lazy mode was only that of ensuring that the original schedule was included in the modeled space. Hence, instead of using the new modeling proposed in Section 3, the simpler approach of enforcing linear independence used in the original Pluto Algorithm (Equation (16)) was used as the basis for Constraints (21), (22), and (23); we use this in our implementation. However, the new modeling presented in Section 3 is compatible with the lazy mode. Constraints (21), (22), or (23) can be encoded in exactly the same way as done in Section 3.2. In addition, in Section 3, we had introduced a bound b on the statement's dimension coefficients, that is, coefficients of $\vec{h}_S$. In the lazy mode, since the original schedule is provably in our space and has coefficients that are either one or zero, a valid solution can always be obtained even with the smallest possible value of b, that is, $b = 1$, for any affine loop nest. Thus, the lazy mode and the modeling introduced to handle negative coefficients are compatible with each other.

5. EXPERIMENTS

In this section, we study (1) the scalability of our new approach and (2) the impact of newly included transformations on performance.

We implemented the techniques proposed in Sections 3 and 4 by modifying the automatic transformation component in Pluto version 0.11.3 (from the *pet* branch of its public git repository); we refer to our new system as Pluto+. In the rest of

Table I. Architecture Details

Intel Xeon E5-2680	
Clock	2.70GHz
Cores/socket	8
Total cores	16
L1 cache/core	32KB
L2 cache/core	512KB
L3 cache/socket	20MB
Peak GFLOPs	172.8
Compiler	Intel C compiler (icc) 14.0.1
Compiler flags	-O3 -xHost -ipo -restrict -fno-alias -ansi-alias -fp-model precise -fast-transcendentals
Linux kernel	3.8.0-44

this section, *Pluto* refers to the existing Pluto version 0.11.3 (git branch *pet*). Other than the component that automatically determines transformations, Pluto+ shares all remaining components with Pluto. With both Pluto and Pluto+, ISL [Verdoolaege 2010] was used for dependence testing, Cloog-isl [Bastoul 2004] 0.18.2 for code generation, and PIP [Feautrier 1988] 1.4.0 as the lexmin ILP solver. For large ILPs with Pluto+ (100+ variables), GLPK [GNU 2012] 4.45 was used. It was significantly faster than PIP or ISL's solver for such problems. With Pluto, GLPK did not make any significant difference, as the ILP formulations were smaller. Typical options used for generating parallel code with Pluto were used for both Pluto and Pluto+; these were –isldep –lastwriter –parallel –tile. Index set splitting (ISS) implementation made available by Bondhugula et al. [2014] was enabled (–iss). For periodic stencils, diamond tiling [Bandishti et al. 2012] was enabled (–partlbtile).

All performance experiments were conducted on a dual socket NUMA Intel Xeon E5-2680 system (based on the Intel Sandybridge microarchitecture) running at 2.70GHz with 64GB of DDR3-1600 RAM, with Intel's C compiler (icc) 14.0.1. Details of the setup, along with compiler flags, are provided in Table I. The default thread to core affinity of the Intel KMP runtime was used (KMP_AFFINITY granularity set to *fine, scatter*), that is, threads are distributed as evenly as possible across the entire system. Hyperthreading was disabled, and the experiments were conducted with up to 16 threads running on 16 cores. Both Pluto and Pluto+ themselves were compiled using gcc 4.8.1 and run single-threaded. We used the regular wallclock timer (gettimeofday) and were able to reproduce all execution times up to within $\pm$ 0.2ms.

Benchmarks. Evaluation was done on 27 out of the 30 benchmarks/kernels from Polybench/C version 3.2 [Polybench 2010]; three of the benchmarks—trmm, adi, and reg-detect—were excluded, as they were reported to be clearly not representative of the intended computations [Yuki 2014]. Polybench's standard dataset sizes were used. In addition, we also evaluate Pluto+ on 1D, 2D, and 3D heat equation applications with periodic boundary conditions. These can be found in the Pochoir suite [Tang et al. 2011]. Then, we also evaluate on LBM simulations that simulate lid-driven cavity flow (lbm-ldc) [Chen and Doolen 1998], flow past cylinder (lbm-fpc-d2q9), and Poiseuille flow [Zou and He 1997] (lbm-poi-d2q9). An *mrt* version of LBM [d'Humières 2002] involves multiple relaxation parameters for a single timestep, and therefore a higher operational intensity. Periodic boundary conditions were employed for these. We consider two different configuration classes for lbm: d2q9 and d3q27. d3q27 signifies a 3D data grid with a 27-neighbor interaction, while d2q9, a 2D data grid with 9 neighbor interactions. These are full-fledged real applications as opposed to kernels. Finally, we also evaluate on *171.swim* from the SPEC CPU2000 FP suite. The *swim*

Table II. Problem Sizes for Heat, Swim, and LBM Benchmarks

Benchmark	Problem size
heat-1dp	$1.6 \cdot 10^6 \times 1000$
heat-2dp	$16000^2 \times 500$
heat-3dp	$300^3 \times 200$
swim	$1335^2 \times 800$
lbm-ldc-d2q9	$1024^2 \times 50000$
lbm-ldc-d2q9-mrt	$1024^2 \times 20000$
lbm-fpc-d2q9	$1024 \times 256 \times 40000$
lbm-poiseuille-d2q9	$1024 \times 256 \times 40000$
lbm-ldc-d3q27	$256^3 \times 300$

benchmark [Sadourny 1975; Swarztrauber 2000] solves shallow-water equations on a 2D grid with periodic boundary conditions. Problem sizes used are shown in Table II.

5.1. Impact on Transformation and Compilation Times

Table III provides the time spent in automatic transformation as well as the total time spent in source-to-source polyhedral transformation. The automatic transformation time reported therein is the time taken to compute or determine the transformations, that is, Pluto or Pluto+ algorithm times, as opposed to the total time taken to transform the source. The total time taken during polyhedral optimization to transform the source is reported as total polyhedral compilation time. Dependence analysis and code-generation times are two other key components besides automatic transformation. The *Misc/other* component primarily includes time spent in post-transformation analyses, such as computing hyperplane properties (parallel, sequential, tilable) and precise tilable band detection. This is potentially done multiple times for various late transformations such as intratile optimization for better vectorization and spatial locality, and creating tile wavefronts. The lower part of Table III lists those benchmarks in which Pluto+ actually finds a significantly different transformation involving a negative coefficient—to enable tiling and other optimization. All of these are partial differential equation solvers of some form in which periodic boundary conditions are used. When there is no periodicity, both Pluto and Pluto+ find the same transformations.

Figure 7 shows the breakdown across various components of the polyhedral source-to-source parallelization. Note that code-generation time as well as other post-transformation analysis times are affected by transformations obtained. We observe from Table III and Figure 7 that, in a majority of cases, the auto-transformation time is at most few tens of milliseconds, and that the code-generation time dominates for both Pluto and Pluto+ in many cases. Surprisingly, auto-transformation times are, on average, lower with Pluto+—11% lower on Polybench and 38% lower on the periodic stencil benchmarks. This is due to the fact that additional checks had been used in Pluto to ensure that the single orthogonal subspace chosen to derive linear independence constraints was a feasible one. Such checks are obviously not needed with Pluto+. Swim presented the most challenging case with 219 variables in the Pluto+ ILP, and was the only benchmark for which Pluto+ used GLPK.

We find Pluto+ to be surprisingly scalable and practical on such a large set of kernels, benchmarks, and real applications from certain domains. None of these posed a serious scalability issue on the total polyhedral source-to-source transformation time. In a majority of the cases, the overall compilation time itself was low, from under a second to a few seconds. In nearly all cases in which the total compilation time was significant and showed a significant increase with Pluto+, it was because code generation had taken much longer since a nontrivial transformation had been performed with Pluto+ while Pluto did not change the original schedule (Table III). We thus see this as an opportunity

Table III. Impact on Polyhedral Compilation Time (All Times Reported are in Seconds)

	Auto-transform (s)		Total (s)		Increase factor	
Benchmark	Pluto	Pluto+	Pluto	Pluto+	in transformation	overall
correlation	0.470	0.214	0.726	0.719	0.45	0.99
covariance	0.043	0.027	0.188	0.227	0.62	1.21
2mm	0.023	0.033	0.246	0.282	1.47	1.15
3mm	0.070	0.126	0.423	0.536	1.81	1.27
atax	0.004	0.004	0.041	0.061	0.97	1.49
bicg	0.002	0.003	0.112	0.151	1.21	1.36
cholesky	0.048	0.025	0.328	0.366	0.52	1.12
doitgen	0.018	0.017	0.178	0.198	0.94	1.11
gemm	0.005	0.006	0.066	0.095	1.23	1.44
gemver	0.010	0.007	0.070	0.097	0.73	1.37
gemvsumv	0.004	0.004	0.039	0.064	1.12	1.63
mvt	0.002	0.003	0.026	0.036	1.28	1.38
symm	0.052	0.039	0.564	0.281	0.75	0.50
syr2k	0.009	0.010	0.100	0.147	1.20	1.48
syrk	0.004	0.005	0.066	0.095	1.24	1.44
trisolv	0.004	0.004	0.055	0.081	1.02	1.48
durbin	0.045	0.029	0.101	0.105	0.63	1.05
dynprog	0.171	0.103	0.545	1.007	0.60	1.85
gramschmidt	0.098	0.069	0.250	0.260	0.71	1.04
lu	0.006	0.005	0.117	0.164	0.88	1.40
ludcmp	0.973	0.263	1.209	0.569	0.27	0.47
floyd-warshall	0.009	0.012	0.298	0.400	1.29	1.34
fdtd-2d	0.043	0.057	2.424	2.527	1.31	1.04
fdtd-apml	1.581	0.563	2.158	1.475	0.36	0.68
jacobi-1d-imper	0.006	0.007	0.130	0.13	1.28	1.00
jacobi-2d-imper	0.015	0.026	0.705	0.751	1.71	1.07
seidel-2d	0.011	0.009	0.249	0.260	0.75	1.04
Mean (geometric)					**0.89**	**1.15**
heat-1dp	0.033	0.016	0.337	0.648	0.48	1.92
heat-2dp	0.190	0.078	2.430	9.282	0.41	3.82
heat-3dp	2.585	0.810	10.77	16.86	0.31	1.57
lbm-ldc-d2q9	0.701	0.330	3.535	8.152	0.47	2.31
lbm-ldc-d3q27	8.726	2.704	35.52	56.36	0.31	1.59
lbm-mrt-d2q9	0.321	0.177	5.180	9.068	0.55	1.75
lbm-fpc-d2q9	0.324	0.175	5.282	9.017	0.54	1.71
lbm-poi-d2q9	0.323	0.176	5.226	9.001	0.55	1.72
swim	1.489	14.46	16.74	47.35	9.71	2.83
Mean (geometric)					**0.62**	**2.04**

to further improve code-generation techniques, including integrating hybrid polyhedral and syntactic approaches such as PrimeTile [Hartono et al. 2009] when suitable. In conclusion, the increase in compilation time due to Pluto+ itself is quite acceptable.

5.2. Performance Impact on Periodic Heat Equation, LBM Simulations, and Swim

On all benchmarks from Polybench, Pluto+ obtained the same transformation (or equivalent ones) as Pluto. Polybench does not include any kernels where negative coefficients help, at least as per our cost function. We thus obtain the same performance on all of Polybench.

Figure 8 shows results from our comparison of Pluto, Pluto+, and ICC on the heat equation, swim, and LBM simulations. *icc-omp-vec* represents the original code optimized with little effort—by manually inserting an OpenMP pragma on the outermost space loop and `#pragma ivdep` for the innermost space loop to enable vectorization.

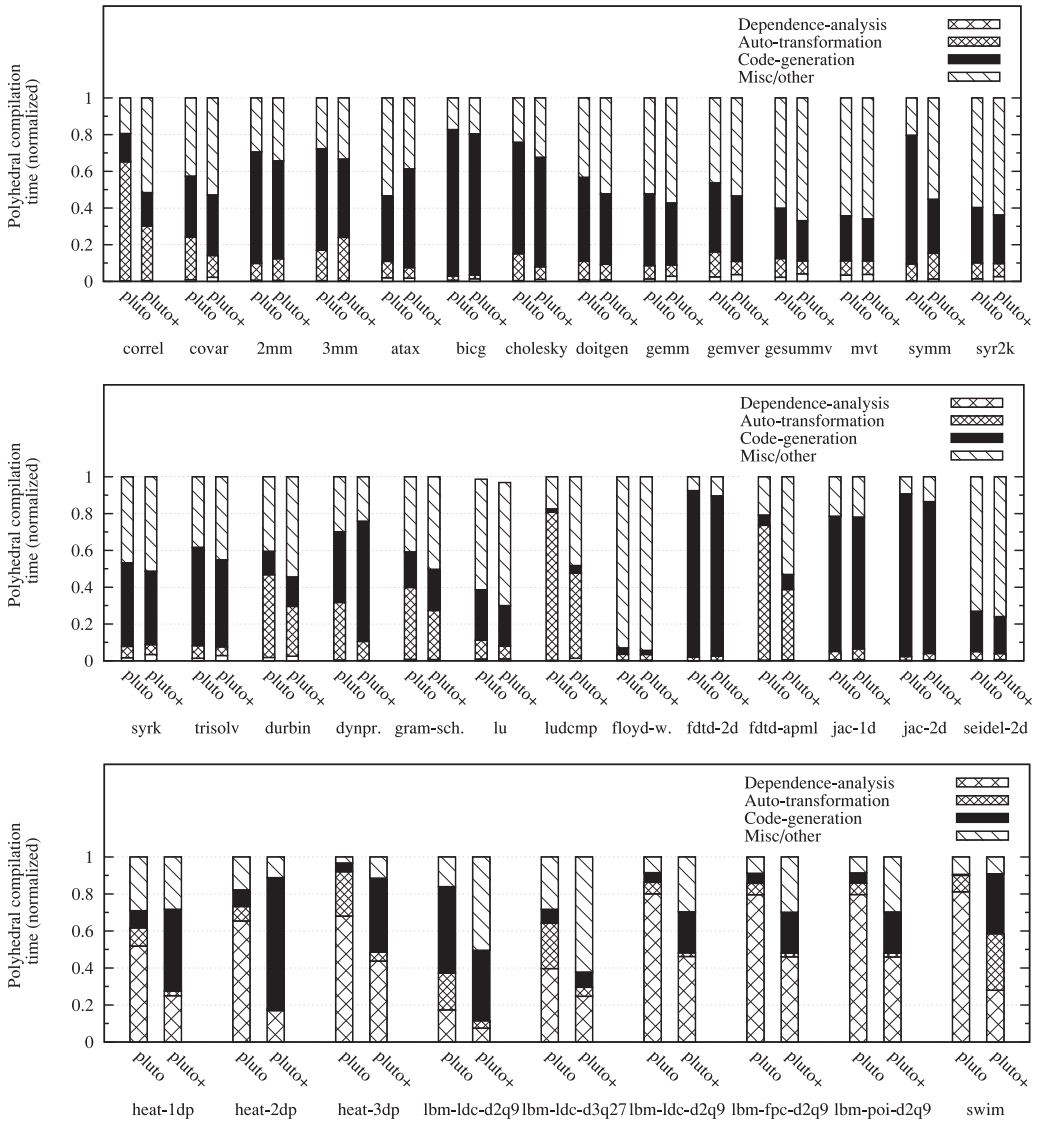


Fig. 7. Polyhedral compilation time breakdown into dependence analysis, transformation, code generation, and miscellaneous (some benchmark names have been abbreviated; full names can be found in Table III). Both Pluto and Pluto+ times are individually normalized to their respective total times for readability. Table III can be used in conjunction with this figure to determine the absolute time for any component.

Pluto is unable to perform time tiling or any useful optimization on stencils with periodicity: this includes the heat equation benchmarks (Figures 8(a), 8(b), 8(c)), LBM benchmarks (Figures 8(d), 8(e), 8(f), 8(g), 8(h)), and swim (Figure 8(i)). Its performance is thus the same as that achieved via *icc-omp-vec* through parallelization of the space loop immediately inside the time loop and vectorization of the innermost loop. On the other hand, Pluto+ performs time tiling for all periodic stencils using the diamond tiling technique [Bandishti et al. 2012]. We obtain speedups of 2.72, 6.73, and 1.4 over *icc-omp-vec* or Pluto for heat-1dp, heat-2dp, and heat-3dp, respectively. For LBM,

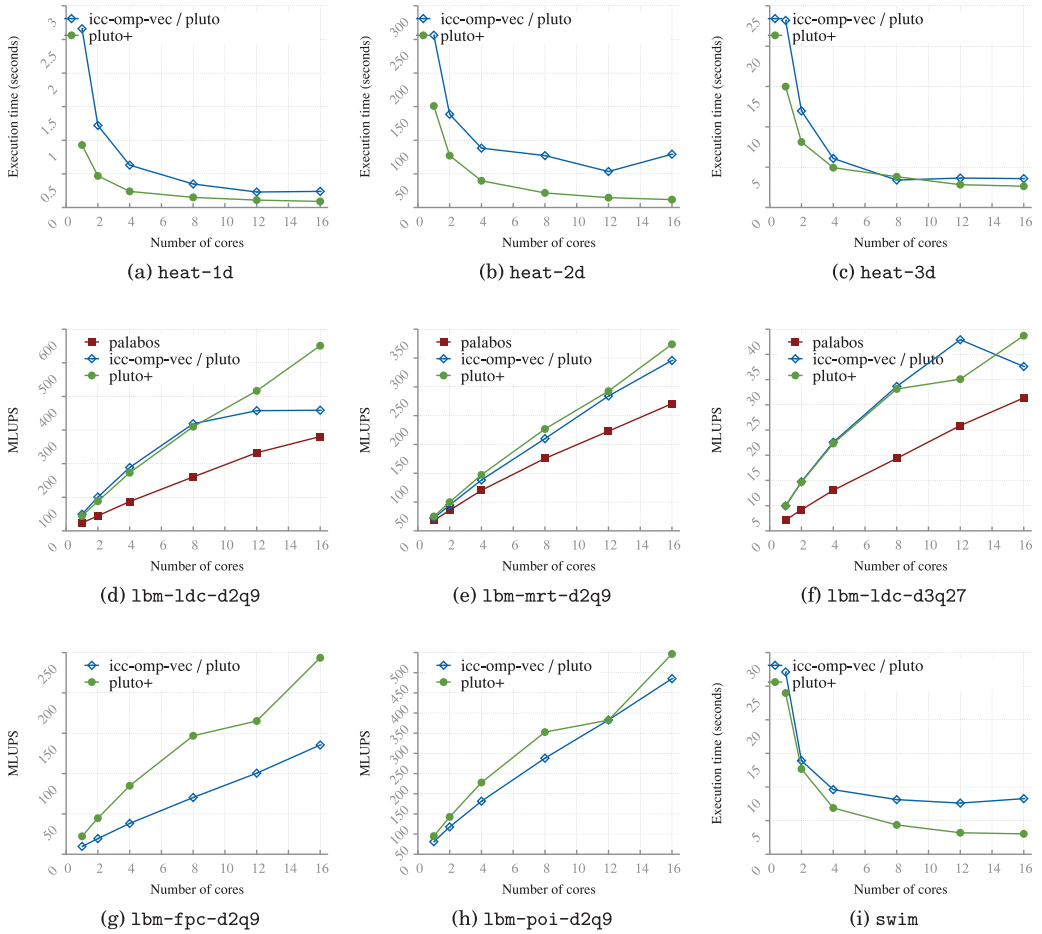
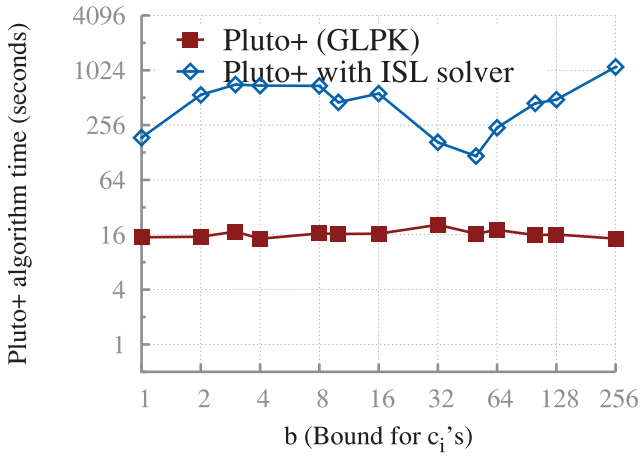


Fig. 8. Performance impact.

we also compare with Palabos [2012], a well-known stable library package for LBM computations for which a user manually provides the specification in C++. Palabos results are provided as a reference point as opposed to for a direct comparison since the approach falls into a different class. The performance of LBM benchmarks is usually reported in million lattice site updates per second (MLUPS). MLUPS is computed by dividing the total of grid updates (timesteps times number of grid points) by the execution time. For flow-past cylinder and Poiseuille flow, we could not find a way to express them in Palabos; hence, those comparisons are not provided. For LBM, we obtain a mean speedup of 1.33× over *icc-omp-vec* and Pluto, and 1.62× over Palabos. All of these performance improvements are due to the improved locality as a result of time tiling enabled by Pluto+: due to reduced memory bandwidth utilization, besides improvement in single-thread performance, it led to better scaling with the number of cores.

For the benchmarks on three-dimensional data grids (heat-3d and lbm-ldc-d3q27), we see little or no improvement in some cases (Figures 8(c) and 8(f)). The 3D benchmarks were particularly hard to optimize and we found that more work was necessary to automate for certain complementary aspects. For lbm-ldc-d3q27, the drop in



b	Pluto+ time (s)	
	GLPK	ISL
1	15.00	186
2	15.24	550
4	14.46	694
8	16.56	690
16	16.42	569
32	20.60	166
64	18.11	239
128	16.12	488
256	14.52	1112

Fig. 9. Variation of Pluto+ algorithm time on *swim* with b.

icc-omp-vec or Pluto's performance was due to NUMA locality effects, an aspect that we do not model or optimize for. This was deduced from the observation that changing KMP's affinity setting from "scatter" to "compact" made the results follow a predictable pattern when running on 12 threads or more. However, we found the default "scatter" setting to be delivering the best performance. Effective vectorization is also more difficult for 3D stencils than for lower-dimensional ones. Previous works that studied this problem in detail [Henretty et al. 2011, 2013; Kong et al. 2013] demonstrated little or no improvement for 3D cases.

For the swim benchmark, our tool takes as input a C translation of it (since the front end only accepts C) with all three *calc* routines inlined. We verified that the Intel compiler provided exactly the same performance on this C version as on the original Fortran version. The inlining thus does not affect icc's optimization. Index set splitting application is fully automatic (from Bondhugula et al. [2014]) just like for the other periodic stencils. On swim, we obtain a speedup of 2.73× over icc's auto-parallelization when running on 16 cores (Figure 8(i)).

5.3. Variation with *b*

We now show how the time taken by the Pluto+ algorithm (auto-transformation time) varies with b, the bound on the statements' dimension coefficients introduced in Section 3. As discussed in Section 3.7, b is related to completeness, and a reasonable value makes the modeled space complete for all practical purposes. All evaluation presented so far was with $b = 4$. We now analyze the sensitivity with respect to the magnitude of b for the most complex of all cases, swim. For swim, any $b \geq 1$ yields the same transformation. In most other cases, auto-transformation time is not significant, both absolutely and in relation to other components. For all benchmarks considered, transformations obtained never had a c_i ($1 \leq i \leq m_S$) value higher than two or lower than -2, even when experimenting with a value of b as large as 1023.

Figure 9 shows the variation in runtime of Pluto+ for swim (compile time for the input program) with b. The results show that Pluto+'s runtime is not affected by large values of b, and that GLPK is able to deal with such formulations with no pattern of increase in solver time. The variation is within 25% across all values of b. We observed similar behavior with even larger values b (tested up to 1023). On the other hand, when ISL is employed as the solver for Pluto+, the runtime is highly sensitive to the value of b. Although there is no monotonic increase or decrease with b, Pluto+ time with ISL as

solver is generally higher for larger b values, and dramatically higher when compared with GLPK as the solver. For $b = 4$, Pluto+ is 48× faster with GLPK than with ISL; 76× faster for $b = 256$, and 25× faster, on average. Note that GLPK implements a large number of techniques and heuristics for solving LPs and MIPs; ISL's lexmin function is much simpler and minimal in comparison. Almost all past efforts in polyhedral optimization employed PIP [Feautrier 1988] or ISL, with the exception of Bondhugula et al. [2010] and Vasilache et al. [2012], where COIN-OR [2001], Lougee-Heimer [2003], and Gurobi were used, respectively, but no comparative experiments were conducted.

6. RELATED WORK

Affine schedules computed by Feautrier's scheduling algorithms [Feautrier 1992a, 1992b] allow for coefficients to be negative. However, the objectives used in finding such schedules are those of reducing the dimensionality of schedules and typically minimizing latency. Thus, they do not capture conditions for the validity of tiling, pipelined parallelism, communication-free parallelism, and are very different from the objective used in Pluto or Pluto+. The latter objective requires additional constraints for zero-solution avoidance and linear independence, which, in turn, necessitates techniques developed in this article for modeling the space of affine transformations.

The approach of Griebl [2004] finds Forward Communication Only (FCO) placements for schedules found by Feautrier [1992a, 1992b]: placements determine where (processor) iterations execute while schedules determine when they execute. The FCO constraint is the same as the tiling constraint in Constraint (10). FCO placements, when found, will allow tiling of both schedule dimensions and placement dimensions. Once the FCO constraint is enforced, the approach proposed to pick from the space of valid solutions relies on finding the vertices, rays, and lines (extremal solutions) of the space of valid transformations. From among these extremal solutions, the one that leads to a zero communication (zero-dependence distance) for the maximum number of dependences is picked (Griebl [2004], Section 7.4). Linear independence is not an issue since all extremal solutions are found, and those linearly dependent on scheduling dimensions can be excluded. Finding the vertices, rays, and lines of the space of valid transformations is very expensive given the number of dimensions (transformation coefficients) involved. The practicality and scalability of such an approach has not been demonstrated.

The approach in Lim and Lam [1997], Lim and Lam [1998], and Lim et al. [1999] was the first affine transformation framework to take a partitioning view, as opposed to a scheduling-cum-placement view, with the objective to reduce the frequency of synchronization and improve locality. The algorithm proposed to finally determine transformation coefficients, Algorithm A in Lim and Lam [1998], does not use a specific objective or cost function to choose among valid ones. It instead proposes an iterative approach to finding all linearly independent solutions to the set of constraints representing the space of valid solutions. No bound is provided on the number of steps that it takes, and no proof is provided on its completeness in general. In addition, the objective was to maximize the number of degrees of parallelism – this was a significant advance over prior art and provided better parallelization with tiling for a much more general class of codes. It has already been shown that maximizing only the number of degrees of parallelism is not sufficient, and solutions that exhibit the same number of degrees of parallelism can perform very differently [Bondhugula et al. 2008b].

The approach of Pouchet et al. [2008] for iterative optimization is driven by search and empirical feedback as opposed to by a model, and is on the same space of schedules as those modeled in Feautrier [1992b]. In spite of supporting negative coefficients in schedules, there is no notion of tiling or communication-free parallelization, whether

or not they required negative coefficients. Its limitations are thus independent of our contributions here.

Semiautomatic transformation frameworks based on the polyhedral framework exist [Rudy et al. 2011; Yuki et al. 2013]. Such frameworks allow a user to explore and experiment with transformations manually, typically through a script. Pluto or Pluto+ can be employed in such a setting to model and enumerate the space of valid transformations to reduce the manual component in exploration.

Vasilache [2007] proposes a formulation that models the entire space of multidimensional schedules, that is, all levels at a time. However, the modeling does not include an objective function. An objective function such as that of Pluto and Pluto+ requires linear independence and trivial-solution avoidance; there is no known way to achieve this with the formulation in Vasilache [2007]. Constraints to ensure linear independence often necessitate a level-by-level approach [Li and Pingali 1994; Bondhugula et al. 2008a]; doing so without losing interesting valid solutions requires modeling such as the one we propose in this article. Pouchet et al. [2011] developed a hybrid iterative-cum-analytical model-driven approach to explore the space of all valid loop fusion/distributions, while ensuring tilability and parallelization of the fused loop nests. The transformation coefficients were limited to nonnegative ones to enable use of Pluto.

Kong et al. [2013] present a framework that finds schedules that enable vectorization and better spatial locality in addition to the dependence- minimization objective of Pluto or Pluto+. Their work is thus complementary, and the limitations addressed by Pluto+ exist in their modeling of the space of transformations as well.

The R-Stream compiler [Meister et al. 2011] includes techniques that model negative coefficients [Leung et al. 2010]. The approach is different from ours and uses a larger number of decision variables. For each statement, one decision variable is used to model the direction choice (positive or negative) associated with each row of the orthogonal subspace basis ($H_S^{\perp}$ in Section 3.2)—in effect, multiple decision variables for each statement capture the choice of the orthant. In contrast, our approach encodes linear independence for a statement using a single decision variable. We achieved this by exploiting bounds on the components associated with each row of $H_S^{\perp}$. The R-Stream approach is not available for a direct experimental comparison.

Bondhugula et al. [2014] present an automatic index set-splitting technique to enable tiling for periodic stencils. The approach requires techniques developed in this work to be able to ultimately find the right transformations once such index set splitting has been performed. The same index set-splitting technique is used in conjunction with Pluto+ to enable a richer variety of transformations; we are able to reproduce performance improvements similar to those reported in Bondhugula et al. [2014] for 1D, 2D, and 3D heat equations with periodicity, and the swim benchmark. Pananilath et al. [2015] present a domain-specific code generator for LBM computations while using Pluto for time tiling; results are presented only in the absence of any periodic boundary conditions, however.

This work presents two significant improvements to the original Pluto algorithm presented in Bondhugula et al. [2008a]. Section 3 presents the first improvement, a prior version of which was presented in Acharya and Bondhugula [2015]. Our contributions in Section 4 and the analysis of the runtime on the coefficient bounds are new contributions over Acharya and Bondhugula [2015]. Bondhugula [2008] presents proofs (different from the ones presented here) for the soundness of the original Pluto algorithm, but they do not cover all cases and thus do not prove completeness. Finally, a backtracking method to extend the Pluto algorithm has been proposed for the PPCG research compiler [Verdoolaege et al. 2013], theoretically lifting the restriction to nonnegative coefficients. Yet, as implemented in PPCG, the method becomes unscalable for problems of the size and complexity of the larger benchmarks studied in this article.

7. CONCLUSIONS

In this article, we propose significant extensions to the Pluto algorithm for parallelization and locality optimization. First, we propose an approach to model a significantly larger space of affine transformations than previous approaches in conjunction with a cost function, and second, an extension that makes the algorithm provably sound and complete. We also provide proofs for its soundness and completeness for affine loop nests in general. Experimental evaluation showed that Pluto+ provided no degradation in performance in any case. For LBM simulations with periodic boundary conditions, it provided a mean improvement of 1.33× over Pluto while running on 16 cores of a modern multicore SMP. On the swim benchmark from the SPEC CPU2000 FP suite, Pluto+ obtains a speedup of 2.73× over Intel C compiler's auto-parallelization. We also show that Pluto+ does not induce specific scalability issues: it is generally as scalable as Pluto, with the overall polyhedral compilation time on Polybench increasing by only 15%, on average. In cases in which it improved execution time significantly due to a different transformation, the total polyhedral compilation time was increased by 2.04×. As for the time component involved in computing transformations, Pluto+ is faster than Pluto, on average. We believe that these contributions are a significant advance over Pluto, both from a theoretical and a practical standpoint.

ACKNOWLEDGMENTS

We thank Louis-Noel Pouchet for the example that showed that the original Pluto algorithm did not terminate due to the greedy nature of its linear independence constraints.

REFERENCES

Aravind Acharya and Uday Bondhugula. 2015. PLUTO+: Near-complete modeling of affine transformations for parallelism and locality. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)*. ACM, New York, NY.

N. Ahmed, Nikolay Mateev, and Keshav Pingali. 2001. Synthesizing transformations for locality enhancement of imperfectly-nested loops. *International Journal of Parallel Programming* 29, 5.

Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, and Monica S. Lam. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Prentice Hall, Upper Saddle River, NJ.

Corinne Ancourt and Francois Irigoin. 1991. Scanning polyhedra with DO loops. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, New York, NY, 39–50.

Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. 2012. Tiling stencil computations to maximize parallelism. In *Supercomputing*. Article 40, 11 pages.

Cédric Bastoul. 2004. Code generation in the polyhedral model is easier than you think. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 7–16. DOI:http://dx.doi.org/10.1109/PACT.2004.11

Uday Bondhugula. 2008. *Effective Automatic Parallelization and Locality Optimization Using the Polyhedral Model*. Ph.D. Dissertation. The Ohio State University, Columbus, OH.

Uday Bondhugula, Vinayaka Bandishti, Albert Cohen, Guillain Potron, and Nicolas Vasilache. 2014. Tiling and optimizing time-iterated computations on periodic domains. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'14)*. 39–50.

Uday Bondhugula, M. Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. 2008a. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction (ETAPS CC'08)*.

Uday Bondhugula, Oktay Gunluk, Sanjeeb Dash, and Lakshminarayanan Renganarayanan. 2010. A model for fusion and code motion in an automatic parallelizing compiler. In *International Conference on Parallel Architectures and Compilation Techniques*. 343–352.

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008b. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Symposium on Programming Languages Design and Implementation (PLDI'08)*. ACM, New York, NY, 101–113.

Chun Chen. 2012. Polyhedra scanning revisited. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*. ACM, New York, NY, 499–508.

Shiyi Chen and Gary D. Doolen. 1998. Lattice Boltzmann method for fluid flows. *Annual Review of Fluid Mechanics* 30, 1, 329–364.

C. Choffrut and K. Culik. 1983. Folding of the plane and the design of systolic arrays. *Information Processing Letters* 17, 3, 149–153.

Cloog 2004. The CLooG Code Generator in the Polyhedral Model. http://www.cloog.org.

COIN-OR. 2001. Computational Infrastructure for Operations Research. Retrieved March 2, 2016 from http://www.coin-or.org.

Dominique d'Humières. 2002. Multiple–relaxation–time Lattice Boltzmann models in three dimensions. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 360, 1792, 437–451.

P. Feautrier. 1988. Parametric integer programming. *RAIRO Recherche Opérationnelle* 22, 3, 243–268.

P. Feautrier. 1991. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming* 20, 1, 23–53.

P. Feautrier. 1992a. Some efficient solutions to the affine scheduling problem: Part I, one-dimensional time. *International Journal of Parallel Programming* 21, 5, 313–348.

P. Feautrier. 1992b. Some efficient solutions to the affine scheduling problem: Part II, multidimensional time. *International Journal of Parallel Programming* 21, 6, 389–420.

GNU. 2012. GLPK (GNU Linear Programming Kit). Retrieved March 2, 2016 from https://www.gnu.org/software/glpk/.

Martin Griebl. 2004. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. Habilitation thesis. University of Passau, Passau, Germany.

Tobias Grosser, Hongbin Zheng, Ragesh Aloor, Andreas Simbrger, Armin Grolinger, and Louis-Noël Pouchet. 2011. Polly polyhedral optimization in LLVM. In *Proceedings of the 1st International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*.

Albert Hartono, Muthu Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, and J. Ramanujam. 2009. A parametric multi-level tiler for imperfect loop nests. In *International Conference on Supercomputing (ICS'09)*.

Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam, and P. Sadayappan. 2011. Data layout transformation for stencil computations on short SIMD architectures. In *ETAPS International Conference on Compiler Construction (CC'11)*. 225–245.

Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. 2013. A stencil compiler for short-vector SIMD architectures. In *ACM International Conference on Supercomputing*. ACM, New York, NY.

F. Irigoin and R. Triolet. 1988. Supernode partitioning. In *ACM SIGPLAN Principles of Programming Languages*. ACM, New York, NY, 319–329.

W. Kelly and W. Pugh. 1995. *A Unifying Framework for Iteration Reordering Transformations*. Technical Report CS-TR-3430. Department of Computer Science, University of Maryland, College Park, MD.

W. Kelly, W. Pugh, and E. Rosser. 1995. Code generation for multiple mappings. In *International Symposium on the Frontiers of Massively Parallel Computation*. 332–341.

Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. 2013. When polyhedral transformations meet SIMD code generation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. ACM, New York, NY.

A. Leung, N. T. Vasilache, B. Meister, and R. A. Lethin. 2010. Methods and apparatus for joint parallelism and locality optimization in source code compilation. (June 2010). US Patent 2010/0070956 A1. Filed September 16, 2009, Issued March 18, 2010.

Wei Li and Keshav Pingali. 1994. A singular loop transformation framework based on non-singular matrices. *International Journal of Parallel Programming* 22, 2, 183–205.

A. Lim, Gerald I. Cheong, and Monica S. Lam. 1999. An affine partitioning algorithm to maximize parallelism and minimize communication. In *ACM International Conference on Supercomputing (ICS'99)*. 228–237.

A. Lim and Monica S. Lam. 1997. Maximizing parallelism and minimizing synchronization with affine transforms. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 201–214.

A. Lim and Monica S. Lam. 1998. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing* 24, 3–4, 445–475.

Robin Lougee-Heimer. 2003. The common optimization interface for operations research. *IBM Journal of Research and Development* 47, 1, 57–66.

Benoit Meister, Allen Leung, Nicolas Vasilache, David Wohlford, Cédric Bastoul, and Richard Lethin. 2009. Productivity via automatic code generation for PGAS platforms with the R-stream compiler. In *Workshop on Asynchrony in the PGAS Programming Model*.

Benoît Meister, Nicolas Vasilache, David Wohlford, Muthu Baskaran, Allen Leung, and Richard Lethin. 2011. R-stream compiler. In *Encyclopedia of Parallel Computing*. 1756–1765.

Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic optimization for image processing pipelines. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*.

Nissa Osheim, Michelle Mills Strout, Dave Rostron, and Sanjay Rajopadhye. 2008. Smashing: Folding space to tile through time. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'08)*. Springer, Berlin, 80–93.

Palabos. 2012. Palabos. Retrieved March 2, 2016 from http://www.palabos.org/.

Irshad Pananilath, Aravind Acharya, Vinay Vasista, and Uday Bondhugula. 2015. An optimizing code generator for a class of Lattice-Boltzmann computations. *ACM Transactions on Architecture and Code Optimization.* 12, 2, 14:1–14:23.

Pluto 2008. PLUTO: An automatic parallelizer and locality optimizer for affine loop nests. http://pluto-compiler.sourceforge.net.

Polybench 2010. Polybench suite. Retrieved March 2, 2016 from http://polybench.sourceforge.net.

L.-N. Pouchet, Cédric Bastoul, John Cavazos, and Albert Cohen. 2008. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN Symposium on Programming Languages Design and Implementation (PLDI)*.

Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. 2011. Loop transformations: Convexity, pruning and optimization. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'11)*.

Fabien Quilleré, Sanjay V. Rajopadhye, and Doran Wilde. 2000. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming* 28, 5, 469–498.

Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen, and Jacqueline Chame. 2011. A programming language interface to describe transformations and code generation. In *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing (LCPC'10)*. Springer, Berlin, 136–150.

Robert Sadourny. 1975. The dynamics of finite-difference models of the shallow-water equations. *Journal of the Atmospheric Sciences* 32, 4.

Alexander Schrijver. 1986. *Theory of Linear and Integer Programming*. John Wiley & Sons, Hoboken, NJ.

Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel. 2011. Cache accurate time skewing in iterative stencil computations. In *International Conference on Parallel Processing (ICPP'11)*. 571–581.

Paul N. Swarztrauber. 2000. 171.swim SPEC CPU2000 Benchmark Description File. Standard Performance Evaluation Corporation. Retrieved March 2, 2016 from http://www.spec.org/cpu2000/CFP2000/171.swim/docs/171.swim.html.

Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. 2011. The Pochoir stencil compiler. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11)*. New York, NY, 117–128.

Nicolas Vasilache. 2007. *Scalable Program Optimization Techniques in the Polyhedral Model*. Ph.D. Dissertation. Université de Paris-Sud, INRIA Futurs, Orsay, France.

Nicolas Vasilache, Benoit Meister, Muthu Baskaran, and Richard Lethin. 2012. Joint scheduling and layout optimization to enable multi-level vectorization. In *International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*.

Sven Verdoolaege. 2010. ISL: An integer set library for the polyhedral model. In *Mathematical Software—ICMS 2010*, Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama (Eds.). Lecture Series in Computer Science, Vol. 6327. Springer, Berlin, 299–302.

Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4, 54:1–54:23.

M. Wolf and Monica S. Lam. 1991. A data locality optimizing algorithm. In *ACM SIGPLAN Symposium on Programming Languages Design and Implementation*. ACM, New York, NY, 30–44.

D. Wonnacott. 2000. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *IPDPS*. 171–180.

Yoav Yaacoby and Peter R. Cappello. 1995. Converting affine recurrence equations to quasi-uniform recurrence equations. *VLSI Signal Processing* 11, 1–2, 113–131.

Tomofumi Yuki. 2014. Understanding Polybench/C 3.2 kernels. In *International Workshop on Polyhedral Compilation Techniques (IMPACT'14)*.

Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. 2013. AlphaZ: A system for design space exploration in the polyhedral model. In *Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, Vol. 7760. Springer, Berlin, 17–31.

Yutao Zhong, Xipeng Shen, and Chen Ding. 2009. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems* 31, 6, Article 20, 39 pages.

Qisu Zou and Xiaoyi He. 1997. On pressure and velocity boundary conditions for the Lattice Boltzmann BGK model. *Physics of Fluids (1994-present)* 9, 6, 1591–1598.

Received March 2015; revised December 2015; accepted December 2015