Tiling and Optimizing Time-Iterated Computations over Periodic Domains

Uday Bondhugula
Department of Computer Science and Automation,
Indian Institute of Science
Bangalore 560012 India
uday@csa.iisc.ernet.in

Vinayaka Bandishti
Department of Computer Science and Automation,
Indian Institute of Science
Bangalore 560012 India
vbandishti@csa.iisc.ernet.in

Albert Cohen
INRIA and École Normale Supérieure
45 rue d'Ulm
Paris 75005, France
Albert.Cohen@inria.fr

Guillain Potron
École Normale Supérieure
and Indian Institute of Science
45 rue d'Ulm
Paris 75005, France
guillain.potron@ens.fr

Nicolas Vasilache
Reservoir Labs
632 Broadway
New York, NY 10012, USA
vasilache@reservoir.com

ABSTRACT

This paper deals with optimizing time-iterated computations on periodic data domains. These computations are prevalent in computational sciences, particularly in partial differential equation solvers. We propose a fully automatic technique suitable for implementation in a compiler or in a domain-specific code generator for such computations. Dependence patterns on periodic data domains prevent existing algorithms from finding tiling opportunities. Our approach augments a state-of-the-art parallelization and locality-enhancing algorithm from the polyhedral framework to allow time-tiling of stencil computations on periodic domains. Experimental results on the swim SPEC CPU2000fp benchmark show a speedup of 5× and 4.2× over the highest SPEC performance achieved by native compilers on Intel Xeon and AMD Opteron multicore SMP systems, respectively. On other representative stencil computations, our scheme provides performance similar to that achieved with no periodicity, and a very high speedup is obtained over the native compiler. We also report a mean speedup of about 1.5× over a domain-specific stencil compiler supporting limited cases of periodic boundary conditions. To the best of our knowledge, it has been infeasible to manually reproduce such optimizations on swim or any other periodic stencil, especially on a data grid of two-dimensions or higher.

Categories and Subject Descriptors

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

PACT '14 August 24 - 27 2014, Edmonton, AB, Canada

http://dx.doi.org/10.1145/2628071.2628106.

Keywords

Tiling; Stencils; Periodic; Automatic parallelization; Polyhedral model

1. INTRODUCTION

Stencil-style computations are widely used in solving partial differential equations over discretized domains. They have been extensively studied by the parallel and high-performance computing community. Stencil computations involve updating points in a data grid of certain dimensionality repeatedly. The computation performed at each point in the grid uses values from its immediate or short distance neighbors. These updates to the grid are repeated a certain number of times or until convergence. Hence, as originally viewed and specified, the computation accesses the entire grid each iteration before accessing it again the next iteration. Such an execution order is memory bandwidth-bound when the data grid does not fit in the last level cache.

Stencil computations can be performed on discretized domains that are either non-periodic or periodic. Non-periodic domains have boundaries that do not change. However, periodic domains are often used to model a portion of a larger space. Periodicity also arises when one models a hollow object. The object is cut and unrolled flat in a lower dimensional space. For example, domains like hollow spheres, cylinders, or tori can be cut and flattened out in 2-d space. Similarly, a ring can be cut and flattened into a 1-d array. When this is done, the points on either boundary have to be treated as neighbors with respect to one another, resulting in wrap-around dependences. These wrap-around dependences create cyclic dependences between tiles in an otherwise valid tiling. Although a lot of effort has been put in optimizing stencil computations, there is very little work on optimizing those with periodicity. This is an important domain of numerical simulation, since periodic boundary conditions are prevalent in partial differential equation solvers. The fact that the SPEC CPU2000fp included the swim (171.swim) as one of its benchmarks is strong evidence of this. The swim benchmark is a weather prediction program that performs finite difference modeling of shallow water equations through periodic boundary conditions on a two dimensional grid [27]. All its running time is spent in the stencil computation. Modeling the earth's

atmosphere and surface also involves two-dimensional periodic domains [26].

Tiling for locality and parallelism [17, 37, 41] has been studied intensively for the optimization of stencils with no periodicity. Tiling for locality allows simultaneous reuse in multiple directions—the directions correspond to the loop dimensions being tiled. In particular, the loop that iterates over time in stencils carries temporal reuse while the space loops carry constant reuse as well as spatial reuse along the innermost space dimension. Tiling for parallelism allows reduction in the frequency of synchronization. Tiling for locality and parallelism together makes the computation less dependent on memory bandwidth, and in the context of multicore processors, can make it scale better.

We make a key observation about stencils with periodic boundary conditions: tiling can be enabled by first splitting the iteration domain (or index set) in a very specific way. Then, an extension of existing auto-parallelization and tiling techniques enables the necessary program transformations allowing for dramatic improvement in performance. In particular, our technique improves performance by several factors when compared to code that just tiles and parallelizes the loops that tile and iterate over the data space code.

The rest of this paper is organized as follows. Section 2 introduces the technical background. Section 3 discusses challenges and the feasibility of various approaches. Section 4 and Section 5 describe our solution. Experimental results are presented in Section 6 before concluding in Section 8.

2. BACKGROUND

In this section, we introduce notation and the mathematical background necessary for the sections that follow.

Definition 1 (Hyperplane). A hyperplane is an $n-1$ dimensional affine sub-space of an n dimensional space.

Since we are interested in integer spaces, by a hyperplane we refer to the set of all vectors $x \in \mathbb{Z}^n$ such that $\vec{h}.\vec{x} = k$, for $k \in \mathbb{Z}$. Two vectors $\vec{v_1}$ and $\vec{v_2}$ lie in the same hyperplane if $\vec{h}.\vec{v_1} = \vec{h}.\vec{v_2}$. The set of parallel *hyperplane instances* correspond to different values of k with the row vector $\vec{h}$ normal to the hyperplane.

A hyperplane divides a space into two half-spaces, the positive half-space and the negative half-space. If the coefficients of $\vec{h}$ are integers, the set of integer points are divided into a non-negative half-space ($\vec{h}.x \geq k$) and a negative half-space ($\vec{h}.x \leq k-1$).

Index sets and dependences.

Let $S_1, S_2, \ldots, S_n$ be the statements of a program. The set of all iterations $\vec{i_S}$ of S is called the index set of S and is represented by I_S. Let m_S be the dimensionality of statement S. A program parameter is a symbol that is not modified in the portion of the program being represented. Problem sizes appearing in loop bounds are typical examples of program parameters. Let m_p be the number of program parameters, and $\vec{p}$ be the vector of program parameters. Let E be the set of dependence edges. For an $e \in E$, let D_e be the dependence polyhedron. D_e is a relation between source and target iterations, represented by $\vec{s}$ and $\vec{t}$ respectively, that are in dependence. For example, the vertical dependence instances in Figure 1 and Figure 2 correspond to the dependence polyhedron:

$$D_e = \{\langle \vec{s}, \vec{t} \rangle | \vec{s} = (t,i) \wedge \vec{t} = (t',i') \wedge t' = t+1 \wedge i' = i\}$$

2.1 Tiling

Tiling is considered valid if and only if a total order can be constructed for the execution of all tiles, where each tile is being executed atomically. This implies that a tiling is valid if and only if there is no dependence cycle between the tiles. This can be very hard to check statically in general. Hence, compiler optimizers work with sufficient conditions such as that of non-negative dependence components: this was from the pioneering works of Irigoin and Triolet [17] and a large amount of literature on the validity of tiling relates to or derives from it [37, 25, 20, 1, 5]. In particular, the condition involves checking if all components corresponding to (yet unsatisfied) dependences are non-negative for the set of contiguous loop nest dimensions that are being tiled. In a more general polyhedral setting, a tiling hyperplane is an affine function of the form:

$$\phi_S\left(\vec{i_S}\right) \quad = \quad (c_1 \ldots c_{m_S}) \cdot \left(\vec{i_S}\right) + \left(d_1 \ldots d_{m_p}\right) \cdot (\vec{p}) + c_0$$

$$c_0, c_1, \ldots, c_{m_S}, d_1, \ldots, d_{m_p} \in \mathbb{Z}, \quad \vec{i_S} \in I_S$$

and a sufficient condition for ϕ_{S_i} to be a statement-wise valid tiling is written as:

$$\phi_{S_p}\left(\vec{t}\right) - \phi_{S_q}\left(\vec{s}\right) \geq 0, \quad \langle \vec{s}, \vec{t} \rangle \in D_e. \tag{1}$$

When the above condition is enforced for all edges e unsatisfied up to that depth, all linearly independent solutions for ϕ in (1) form a band of valid tiling hyperplanes at that depth.

Often, when rectangular tiling is not valid on a given iteration space, it can in many cases be transformed so that rectangular tiling is valid in the new space, i.e., by finding the right set of ϕ's. E.g., a short negative dependence component can be dealt with through loop skewing with respect to an outer loop that satisfies that dependence. However, a well-known scenario in which such a transformation is not possible is when there are long dependences in either direction corresponding to a dimension. As can be seen in Figure 2, periodic stencil computations have such dependences and cannot be tiled along all dimensions readily.

2.2 Stencils

Figure 1 and 2 show the iteration space and dependences for stencil computations without and with periodic boundary conditions, respectively. As can be seen, for non-periodic stencils, all dependences are near-neighbor while for the periodic ones, there are edges wrapping around boundaries.

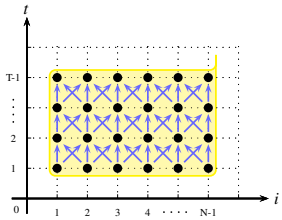


Figure 1: no periodicity

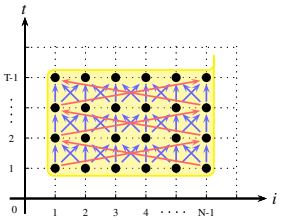


Figure 2: with periodicity

There are multiple ways one could implement the periodic boundary conditions in program code. Figure 3a and 3b show two ways of implementing a simple periodic stencil on a one-dimensional grid. Figure 3b uses a conditional to make the boundary updates access the correct values, while Figure 3a employs copies on to ghost regions to take care of the flow of values across boundaries. When the copy statements are taken into account, the data flow for both codes is equivalent. The conditional can be hoisted out to avoid overhead in the innermost loop. Also, the above code is written with modulo indexing for the time dimension instead of using two copies of the array—the latter is common practice as well. The swim SPEC benchmark in particular uses copies for boundary conditions, and

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

(a) periodic (with copies)

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

(b) periodic (with boundary conditionals)

Figure 3: Stencil: heat-1d equation

uses an old and a new copy for the array instead of indexing the time dimension with a modulo operation.[1]

Time loops and space loops.

The number of times the space grid is updated is determined by the number of iterations in the outermost loop which refer to as the time loop. The loops that update points in the grid are referred to as space loops.

In this context, time tiling refers to tiling the time loop, i.e., the outermost loop. Note that the space loops, being inner parallel loops, can be freely tiled. Time tiling allows temporal reuse to be exploited along the time dimension. This is often the source of a dramatic improvement in single-thread performance, as well as excellent scaling, as time tiled code may be either less memory bandwidth bound or no longer memory bandwidth bound.

For non-periodic stencils, two existing techniques for time tiling are shown in Figure 4. The first one is classical parallelogram-shaped tiling (Figure 4a) that can be obtained by skewing the space dimension(s) with respect to the time loop. Exposing parallelism on such tiles induces a pipelined startup and drain delay, since there is no boundary along which tiles can start in parallel. The second one that we refer to as diamond tiling was recently proposed [3]. It allows concurrent start of all diamonds along the horizontal line: these tiles have no dependences among each other and can start in parallel. It leads to better load balance and maximizes the number of tiles on the wavefront without any pipeline fill-up and drain delays.

3. CHALLENGES AND APPROACHES

This section explores different approaches to the problem of tiling iterated stencil computations with periodic boundary conditions. While these approaches are generally not applicable in a compiler, and sometimes even unsuitable for manual transformation, they provide valuable insights into the challenges involved.

As mentioned earlier, for stencils on periodic domains, the wrap-around dependences at the boundaries create dependence cycles in an otherwise valid tiling. For example, there is no cyclic dependence between tiles in Figure 4a or Figure 4b. However, applying this same tiling to Figure 2 will create a cyclic dependence between tiles at either boundary.

3.1 Merging boundaries

We observe that the cyclic dependence between tiles can be broken if the tiles at either boundary for a dimension can be merged into a single special tile. If the partial tiles at each end match, as is the case in Figure 4b, they could be merged to give a full tile like those in the middle. However, this is not possible in general: depending on the alignment, the height of partial tiles at either boundary may not be the same.

[1]Note that using a modulo with respect to two or any power of two does not hurt performance since it directly translates to a bitwise binary operation as opposed to a branch. Alternatively, such modulo indexing can be eliminated through partial unrolling of the loop.

As shown in Figure 5, a proper choice of tile alignment could be found that guarantees matching height for the partial tiles by shifting the tile origin by an amount equal to half the remainder when the dimension length is divided by the tile size. Note that even with such an alignment, if the boundary tiles are not exactly half of a full tile, one would end up with either a smaller full tile or a larger non-convex tile. We will then need to alternate between the two shapes every time tile step. A roadblock to this approach is that it is not practical for compiler automation since it requires the knowledge of fixed tile shape and size, and it does not explicitly say in which cases an invalid tiling can be fixed to make it valid, and which of the tiling schemes is to be chosen. It would also miss other directs way of tiling the space, which would not require such a post-correction. In addition, a stencil in which dependences arise through multiple statements make such a trial and error approach almost infeasible.

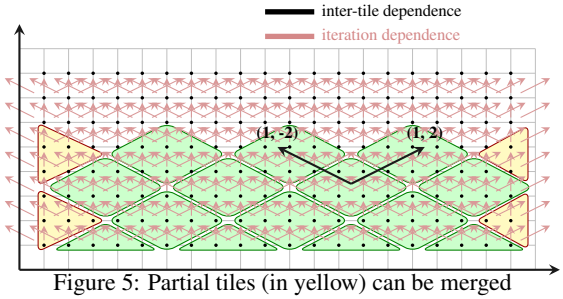


Figure 5: Partial tiles (in yellow) can be merged

3.2 Cut and paste dependent portion

Figure 6 shows an alternative approach where the cyclic dependences are broken through cutting-and-pasting the loop iterations that are transitively affected by periodic boundary conditions. This displacement effectively results in the shortening of the periodic boundary dependences. This is also equivalent to circular loop skewing [38].

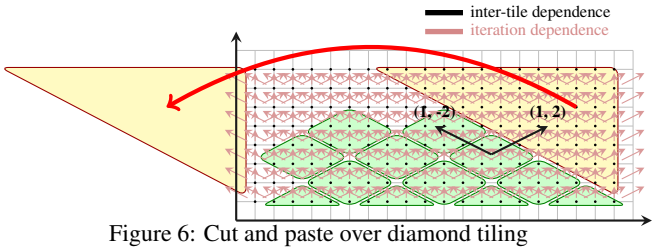
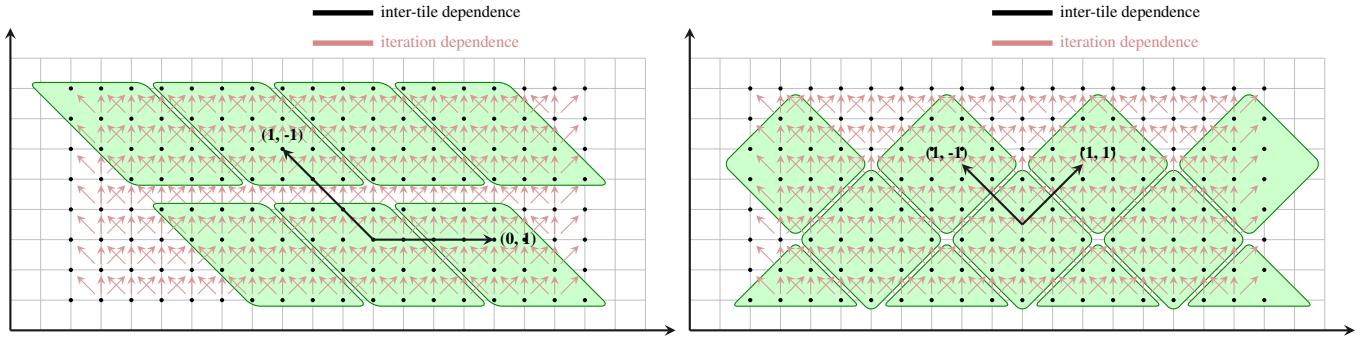


Figure 6: Cut and paste over diamond tiling

This approach requires determining the set of iterations on which another set depends. In other words, it requires computation of a transitive closure of dependences which has remained a very hard problem. Practical approximation schemes for it remain extremely complex and expensive [35]. Libraries like `isl` [35, 36] that do implement transitive closure often recommend avoiding its usage.



(a) Pipelined parallel tiling (b) Diamond tiling (concurrent start-up)

Figure 4: Two ways of tiling heat-1d (non-periodic)

In particular, to enable time tiling for a code like swim from the spec suite, one needs to compute a transitive closure over tens of dependences across multiple statements. In addition, unlike in the one-dimensional case, the backward slice that a tile depends on is not convex, i.e., it is a union of a large number of convex polyhedra. The number of polyhedra in such a union increases with the dimensionality of space, i.e., the number of space dimensions in the stencil.

3.3 Duplicating computation at boundaries

Redundant computation can be performed at each boundary to eliminate the dependence between boundary tiles resulting from periodic conditions. This is equivalent to replicating computation from the opposite boundary. Doing so would be similar to the approach used in [18] where neighboring tiles were overlapped and redundant computation performed to break a dependence for allowing concurrent start.

Formalizing and implementing this approach would again require one to determine the set of dependent iterations, i.e., to compute the transitive closure. Hence, it would suffer from the same limitations as the "cut and paste approach", and in addition, lead to redundant computation. The amount of redundant computation needed will increase with the size of the tile along the time dimension, and with the dimensionality of the data space.

3.4 Folding

The approach of folding [8, 42] used in the systolic array literature provides an interesting conceptual basis for addressing this problem. The folding approach folds the data domain along the middle of each dimension to bring the boundaries together, placing them one on top of the other. The technique of smashing as described by Osheim et al. [21] uses this idea of folding to describe time tiling for periodic stencil computations. They view smashing as "a data allocation technique rather than a loop/iteration transformation". This statement is partly inaccurate since dependence distances cannot be shortened by allocating, reordering, or laying out data in a particular way. They can only be changed by reordering iterations since the distances correspond to distances in the iteration space. Hence, allocating or storing data does not change the ability to tile unless the iterations themselves were reordered with respect to the order in which they were performed on the original data domain. It is assumed that the authors meant the execution order is also implicitly changed with the new data layout. Figure 7 illustrates the effect of folding on the 1-d heat stencil: the two horizontal halves (in the data space) are stacked on top of each other, converting the long cyclic dependences into short ones.

The folding approach is attractive in that in the folded view of the iteration space, all dependences are short and existing tiling techniques will work without any fixing, replication, or computation of transitive closure. Osheim et al. [21] present the smashing technique as a manual or semi-automatic optimization strategy: there are no heuristics to determine when and how to fold or smash. In addition, visualizing it for higher than two dimensional data grids is not straightforward, and hence, there is a need for a formalism to reason about, express, and compute transformations that achieve the proposed effect. Doing so also automatically solves the code generation problem.

Our approach is strongly influenced by folding, but it handles the periodic boundary constraints through iteration reordering transformations only. We require no changes to the data layout: user-defined data spaces remain unaffected.

4. INDEX SET SPLITTING TO CUT LONG DEPENDENCES

This section describes the first systematic method to enable tiling of stencil computations with periodic boundary conditions. The first step in our approach is to perform a preprocessing that splits the index sets of statements. The second step is to ensure that the transformation space allows the necessary transformations to be found for the new index sets and other performance enhancing transformations can be applied on it. This section deals with the first step while the next one deals with the second.

The method we propose subsumes folding techniques summarized in the previous section. The key reason that a transformation like folding cannot be performed by existing frameworks is that affine transformations typically apply the same affine schedule to the entire index set of a program statement. If the statement's index set is partitioned at compile time into a finite number of partitions, and a possibly different affine transformation may be applied to each one, folding-like transformations fall into the space of valid, *piece-wise affine* transformations. Thus, an *index set splitting* heuristic has to be devised that suits the needs of periodic stencils.

4.1 Short and long dependences

We first explain the classification of dependences as being short or long along a particular dimension. A single dependence represented by an edge e in the dependence graph can correspond to multiple dependence instances, i.e., multiple source and target iteration pairs, $\langle \vec{s}, \vec{t} \rangle$ that are in dependence. A dependence instance is *short* along a dimension if its length, i.e., the difference (or distance) between the source and target iteration along that particular dimension can be bounded by a small constant. This constant is

typically a small number and it is important that it not be comparable to loop trip counts. We see that a value of five is sufficient in practice. This value is fixed and it is used for any input program and all its dependences. A larger value such as ten could also be used as long as it is not comparable to the trip counts we expect for the problems of interest here. At the same time it should be larger than the stencil width. In practice, choosing this value to classify dependences as short is never a problem. We find that a value like five works well for the entire domain of interest. For example, for a 3-d stencil used, the grid sizes typically of interest while optimizing for execution time are at least a few hundred along each dimension.

A dependence is considered *long* if it is not *short*. Intuitively, a long dependence is one whose length is of the order of iteration space extents and any bound on its length has to involve program parameters that are symbols appearing in loop bounds, typically problem sizes. A dependence whose length varies (depending on the particular source/target instances in dependence) from a small value to a large value is also thus a long dependence. For a dependence edge to be labeled short along a dimension, all of its dependence instances should be short along that dimension; while a single dependence instance being long will label the dependence as being long. In addition, if a dependence is referred to being long in a dimension-independent manner, it implies that there was at least one dimension along which the dependence was long. The above notion of short or long dependences is only meaningful in the context of a schedule for the iterations. When referring to it without mentioning a schedule, these are implicitly assumed to be defined for the identity schedule that corresponds to the original execution order. Applying another schedule will change these dependence distances and their property of being long or short along a dimension. Note that dependence distances for inter-statement dependences are only meaningful under a schedule since the source and target statements could have different dimensionalities. Since a a statement-wise schedule maps all statements to the same set of time dimensions, the distance between the mapped points in the transformed space is meaningful.

In the examples presented so far, the dependence edge that captures the flow of values across the boundaries is a long dependence while all remaining dependences in the grid are short dependences. The length of the arrows in Figure 2 captures this in an obvious way. As an example, for the code in Figure 3b, the long dependence from the left boundary to the right one between $\vec{s} = (t, i)$ and $\vec{t} = (t', i')$ is given by:

$$t' = t + 1 \wedge i = 0 \wedge i' = N - 1 \wedge 0 \leq t \leq T - 3.$$

The above is long along the inner dimension (i/i') in the positive direction with length $N-1$ while short along the outer dimension (t/t'). The short blue arrows are short dependences with the distances being standard distance vectors (1,0), (1,1), and (1,-1): these are the well-known constant distance vectors used in compiler literature [2, 4, 38].

4.2 Details of the approach

Key idea.

The approach we describe below attempts to cut all dependences that are long along one dimension roughly at its mid-point, while not affecting how the shorter dependences will be transformed in the resulting space. A hyperplane is used to cut the statement's index set into half spaces. After this cut, a separate affine transformation can be applied to each half space. The goal is to allow transformation frameworks to make all dependences short along at least one more dimension than was previously possible. This is sufficient to enable time tiling for periodic stencils.

We first describe our approach for the case when all dependences are intra-statement. In our context, this is a stencil on a single data grid. An affine hyperplane is defined by two characteristics: its orientation given by its normal vector $\vec{h}$, and its position given by an affine function of the program parameters, $v(\vec{p})$, i.e., $v(\vec{p})$ is of the form $\vec{P} \cdot \vec{p} + r$. Finding a suitable hyperplane cut is the same as finding a suitable orientation and position. The cut itself is given by

$$\vec{h}.\vec{i_S} = v(\vec{p}), \quad \text{i.e.,} \quad \vec{h}.\vec{i_S} = \vec{P} \cdot \vec{p} + r.$$

With such a cut, the index set of S, I_S, is partitioned into two halves given by I_S^+ and I_S^-:

$$I_S^+ = I_S \wedge \{\vec{h}.\vec{i_S} \geq v(\vec{p})\} \qquad I_S^- = I_S \wedge \{\vec{h}.\vec{i_S} \leq v(\vec{p}) - 1\}.$$

For example, a possible cut is $2i = N$, cutting the i dimension in the middle; this corresponds to $\vec{h} = (0, 2)$, $\vec{i_S} = (t, i)$, and $v(\vec{p}) = N$ with $\vec{P} = (1)$, $\vec{p} = (N)$, and $r = 0$. Having two linearly independent hyperplanes ($\vec{h}$) would generate four partitions, and so on.

While trying to cut long dependences, we need to make sure the short dependences can continue to remain short, i.e., no short dependence is made long while the long ones are being reduced through a future automatic transformation algorithm. Consider separate affine transformations being applied to each half-space of some cutting hyperplane. If both ends of a short dependence lie on the same side of the hyperplane, the dependence continues to remain short because the same affine transformation is applied on it. If the ends lie on different sides, they both stay at a constant distance from where the dependence crosses the hyperplane. The crossing point thus has to be a fixed point for both affine transformations, and then the dependence will remain short. This provides the intuition that if the long dependences are all cut at their mid-points or at a fixed distance from their mid-points, the source and target iterations of the long dependences can be brought close with the new split index sets while keeping the original short dependences short. Note that it would be valid even if some long dependence instances, potentially belonging to the same dependence edge, are cut at their mid-points while others are cut close to it. We now propose a technique that automatically finds such a cut whenever possible.

The following approach is taken for each dimension along which there are long dependences in *both* directions, positive and negative—since this is what prevents tiling. Let $\vec{s}$ and $\vec{t}$ be the source and target iterations corresponding to a dependence edge e, characterized by dependence polyhedron D_e, that is long along a dimension. In order to cut all dependences within a bounded constant distance from their mid-points, the cutting hyperplane $\vec{h}$ has to satisfy the following:

$$-m \leq (v(\vec{p}) - \vec{h} \cdot \vec{s}) - (\vec{h} \cdot \vec{t} - v(\vec{p})) \leq m$$

for some $m \in \mathrm{Z}^+$ that we will minimize later. We thus have

$$2v(\vec{p}) - m \leq \vec{h} \cdot \vec{s} + \vec{h} \cdot \vec{t} \leq 2v(\vec{p}) + m, \quad \langle \vec{s}, \vec{t} \rangle \in D_e. \tag{2}$$

Note that $\vec{h}$ is unknown while $\vec{s}$, $\vec{t}$ are related through the dependence polyhedron. The above can be linearized with the affine form of the Farkas lemma [28, 11], i.e., if $\vec{f_0}$, $\vec{f_1}$, ..., $\vec{f_m}$ are the faces of D_e, then there exist $\lambda_i \geq 0$ such that

$$2v(\vec{p}) + m - \vec{h} \cdot \vec{s} - \vec{h} \cdot \vec{t} \equiv \lambda_0 + \lambda_1 \vec{f_1} + \cdots + \lambda_m \vec{f_m} \tag{3}$$

$$\vec{h} \cdot \vec{s} + \vec{h} \cdot \vec{t} - 2v(\vec{p}) + m \equiv \lambda'_0 + \lambda'_1 \vec{f_1} + \cdots + \lambda'_m \vec{f_m}. \tag{4}$$

The coefficients of iterators in $\vec{s}$ and $\vec{t}$ from the LHS and RHS can be equated to obtain constraints linear in $\vec{h}$'s coefficients, $\vec{P}$'s coefficients, r, and m. The λ_is, also called Farkas multipliers, can be eliminated locally for each of the long dependences along that dimension, and the constraints aggregated. The constraints are now solved with *the objective to minimize m*. If a solution is found, a hyperplane orientation and position is obtained that cuts within a non-parametric or *constant* distance from the mid-points of all dependence instances in question, and we succeed in finding a split that in turn will allow distinct affine transformations on the split sets that shorten all of these dependences. m is that constant distance since it is free of program parameters ($\vec{p}$). $m = 0$ implies that all mid-points lie on $\vec{h}.\vec{i_S} = v(\vec{p})$. If a solution is not found, there exists no hyperplane that cuts all dependences at a bounded distance from their respective mid-points, and no index set splitting is applied for that dimension. This approach is repeated along all canonical dimensions along which dependences are long in both directions.

Figure 8 shows two other synthetic examples where the cut will lead to better transformations. In general, our technique is robust and resilient to variation in the boundary dependences, width and pattern of the stencils. This is because we minimize the upper bound on the distance of the mid-points of the dependences from the splitting hyperplane, m, as opposed to looking for solutions with $m = 0$. It thus clearly works for the entire domain of interest. Comments on its applicability beyond this domain are made towards the end of the next section.

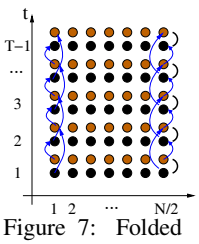


Figure 7: Folded periodic 1-d heat

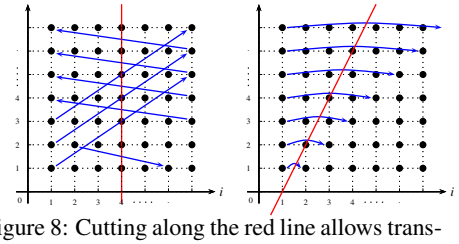


Figure 8: Cutting along the red line allows transformations shortening all dependences

As an example, for the periodic 1-d heat stencil from Figure 2, the split index sets are given by:

$$I_{S^+} = I_S \wedge \{2 * i \geq N\} \qquad I_{S^-} = I_S \wedge \{2 * i \leq N-1\}.$$

I_S is thus replaced with two statements, with index set I_{S^+} and I_{S^-}. We will show in Section 5 how profitable transformations can be applied automatically with the new statements.

4.3 Multi-statement stencils

In the case of multiple statements, long self-dependences could be hidden since they could be implied transitively through other inter-statement dependences which cannot themselves be classified as long or short. This is the case for stencils written with copies at the boundaries for every time step. If the approach described in the previous section is applied just for the intra-statement dependences in the case of multiple statements, it will not enable tiling even if it succeeds in finding a cut.

This problem can be addressed by computing transitive dependences with respect to a set of dependences. A full transitive closure is not needed: one may only compute transitive dependences for paths leading back to the same statement. Once this is done, the approach described in the previous sub-section is applied to determine the index set splitting. In the case of periodic stencils, such transitivity is over a path of length two. However, if the code is not written with copies but with conditionals (Figure 3b), the need for computing transitive dependences does not arise even with multiple statements. This is the case we encounter for all experimental evaluations.

5. POST-ISS SHORTENING AND TRANSFORMATIONS

In the previous section, we showed that index set splitting *opens the possibility* for dependences being shortened. We now argue that the Pluto framework, that shortens dependences, naturally finds the tiling transformation on the split index sets.

5.1 Pluto scheduling algorithm

We first provide some background on the Pluto scheduling algorithm. Consider a one-dimensional affine transformation for S:

$$\phi_S\left(\vec{i_S}\right) = (c_1 \ldots c_{m_S}) \cdot \left(\vec{i_S}\right) + (d_1 \ldots d_{m_p}) \cdot (\vec{p}) + c_0,$$
$$c_0, c_1, \ldots, c_{m_S}, d_1, \ldots, d_{m_p} \in \mathbb{Z}$$

where $\vec{i_S}$ is an iteration vector of S, m_S is the dimensionality of statement S, m_p is the number of program parameters, i.e., symbols appearing in the program (typically representing problem sizes), and $\vec{p}$ is the vector of those program parameters. Each statement has its own set of c_i and d_i coefficients: c_i correspond to the index set dimensions while d_i correspond to parameters and model parametric shifts. For convenience, the notation we use does not involve a superscript specific to S, i.e., c_i^S, d_i^S.

The Pluto algorithm [5] finds such one-dimensional affine transformations, iterating from the outermost inwards while looking for tilable bands, i.e., for ϕs satisfying

$$\phi_{S_j}(\vec{t}) - \phi_{S_i}(\vec{s}) \geq 0, \quad \langle \vec{s}, \vec{t} \rangle \in D_e. \tag{5}$$

The objective function it uses is that of reducing dependence distances using a parametric upper bounding function that was first proposed as a technique by Feautrier [11].

$$\vec{u} \cdot \vec{p} + w - \left(\phi_{s_j}(\vec{t}) - \phi_{s_i}(\vec{s})\right) \;\geq\; 0, \;\; \langle \vec{s}, \vec{t} \rangle \in D_e \tag{6}$$

$\vec{u}$ and w are then minimized, in order of decreasing priority, using the lexicographic minimum as

$$\text{lexmin}\left(\vec{u}, w, \ldots, c_i^{S_j}, d_i^{S_j}, \ldots\right). \tag{7}$$

5.2 Dependence shortening

Once an index set splitting is performed, the long dependence is still long since a new execution order has still not been specified. The split index sets obtain their schedules from the unsplit index set. For example, the long dependence in Figure 2 (code in Figure 3b that goes from the left boundary to the right one is given by:

$$t' = t + 1 \wedge i = 0 \wedge i' = N - 1 \wedge 0 \leq t \leq T - 3 \tag{8}$$

has the dependence distance along its two dimensions given by the vector $[1, N-1]^T$ and this is long along the second dimension as per the original execution order. We now show that the objective function (6) is well-suited to enable tiling for periodic stencils as well. Note that a solution that corresponds to $\vec{u} = \vec{0}$ is preferred over a solution with $\vec{u} \succ \vec{0}$ since the former would have a better objective function value as per (7). Importantly, $\vec{u} = \vec{0}$ corresponds to a transformation that *shortens all dependence distances* to a constant (due to (6)), the constant itself being given by w that is also minimized as part of (7). Hence, transformations that shorten

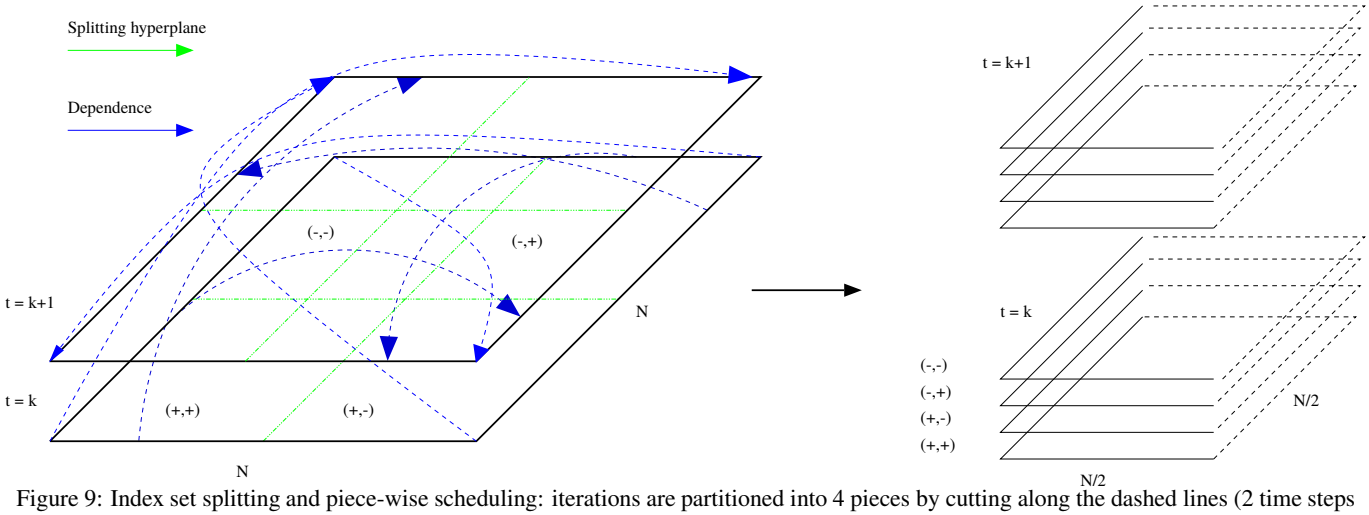


Figure 9: Index set splitting and piece-wise scheduling: iterations are partitioned into 4 pieces by cutting along the dashed lines (2 time steps shown); interleaving the pieces, (shown on the right) results in a space with short dependences only

all dependences to within a fixed constant, which would be w, have a better objective function value than those that do not.

For the dependence in (8), with index set splitting, $\vec{s} = (t,i)$ and $\vec{t} = (t',i')$ are placed into two different index sets, S^+ and S^-. Consider the following transformation on S^+ and S^-:

$$T_{I_S^+}(\vec{i_{S^+}}) = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} t \\ i \end{bmatrix} \quad (9)$$

$$T_{I_S^-}(\vec{i_{S^-}}) = \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} t \\ i \end{bmatrix} + \begin{bmatrix} 1 \\ -1 \end{bmatrix} [N] \quad (10)$$

For S^-, the above can be seen as a composition of $(t,i) \rightarrow (t,N-i)$ with the diamond tiling transformation: $(t,i) \rightarrow (t+i,t-i)$, resulting in transformation $(t-i+N,t+i-N)$: this is the same as (10) written concisely. With the transformation in (9) (10), we get the new dependence distance for dependence (8) as:

$$T_{I_S^-}(\vec{t}) - T_{I_S^+}(\vec{s}) = \begin{bmatrix} (t+1)-(0+N-1)+N \\ (t+1)+(0+N-1)-N \end{bmatrix} - \begin{bmatrix} t+0 \\ t-0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Hence, the dependence is made short along both dimensions by T: this is implied by $\vec{u} = \vec{0}$ at both levels. The other long dependence is also shortened similarly by this transformation. Though there are other transformations that also enable such a shortening, this transformation, in addition, also enables concurrent start leading to tiles of shape shown in Figure 4b [3]. However, as long as dependences can be shortened, one can exploit temporal locality along the time dimension and reduce frequency of synchronization with tiling. Thus, the objective is still well-suited for tiling periodic stencils. Another time tiling transformation that will lead to parallelogram tiles of shape in Figure 4a (still with $\vec{u} = 0$ at both levels) is:

$$T(S^+) = (t,t+i) \qquad T(S^-) = (t,t-i+N)$$

As can be viewed geometrically and as a direct fall-out of the index-set splitting proposed in Section 4.2, the transformations that allow shortening of dependences, once index sets have been split, include *reversals* as well as *negative* (backward) *parametric shifts*. In particular, for a 2-d grid in Figure 9, three of four stacked split sets have to be reversed and shifted backwards along one or two dimensions in order to be aligned as depicted. Consequently, negative coefficients are needed in the statement-wise affine transformation functions.

The Pluto algorithm [5] does not allow negative coefficients in its transformations. This is primarily due to the combinatorial difficulty in avoiding the trivial zero solution for ϕ's coefficients as well as in modeling the space of solutions representing linear independent sub-spaces [7]. This trade-off between expressiveness and computational complexity has worked well in practice for many affine loop nests in which reversals are not a prerequisite to enable efficient parallelization. For the important class of computations we consider here, this trade-off is a limitation: the required reversals are not part of the space of valid transformations. Other transformation algorithms like those of Feautrier [11, 12] also avoid negative values in their coefficients. Such algorithms are also designed to extract the maximal amount of fine-grained parallelism, by greedily satisfying dependences as early as possible. This design is incompatible with time tiling. This limitation in Pluto was recently addressed by Bondhugula and Cohen [6], thereby extending it to include transformations that allow reversal and negative parametric shifts in conjunction with other transformations. Since the objective function itself is untouched, index set splitting only enlarges the space of transformations with transformations that were originally in unsplit space still included.

Overall impact.

Note that our technique kicks in only when there are long dependences in both directions along a dimension. We make three observations here: (1) this is sufficient to deal with all stencils on periodic domains, (2) there is obviously no loss of good transformations in cases where the index set splitting does not succeed, and (3) all transformations that were valid on the unsplit index set are also naturally valid on the split index sets. We thus conclude that the approach has no detrimental impact on cases that lie outside the domain for which this technique has been developed. If our technique is applied even when there are long dependences in one direction, the index set splitting may still lead to better parallelization even though tiling was already valid. Evaluating this is out of the scope of this work and is left for future.

5.3 Complementary transformations

Vectorization is key to obtain good single thread performance for stencils [15, 16]. We rely on the native compiler to perform it. To

this end, we only make sure that the generated code is preconditioned for good automatic vectorization.

Once dependences are shortened and the code tiled, one need not maintain the execution order implied within a tile, i.e., the split sets can be freely reordered within a tile even if it makes the dependences longer again. This is because the dependence would only be longer inside the tile, preserving the validity of the tiling. Such reordering is helpful for vectorization, prefetching, better cache capacity use within a tile, and register tiling.

Note that the split index sets all use different data for the most part except at the boundaries. Hence, we make the following changes to the schedule:

1. Reverse the reversed split index sets back so that we always have a positive stride. This helps vectorization as well as prefetching.

2. Separate out the split index sets at the tile level so that the entire cache capacity is used for each split index set independently, without interleaving. This ensures we do not artificially mix working sets, and that we keep tile sizes as large as possible. This also reduces cache conflicts and pollution among the split index sets.

Both of the above optimizations significantly improve single thread performance while preserving the benefits of tiling and parallel scaling.

6. EXPERIMENTS

	Intel Xeon E5645	*AMD Opteron 6136*
Microarch	Westmere-EP	Magny-Cours
Clock speed	2.4 GHz	2.4 GHz
Cores / socket	6	8
Total cores	12	16
L1 cache / core	32 KB	128 KB
L2 cache / core	512 KB	512 KB
L3 cache / socket	12 MB	12 MB
Peak GFLOPs	115.2	153.6
Compiler	icc/ifort 12.1.3	icc/ifort 12.1.3
Compiler flags	-O3 -fp-model precise	-O3 -fp-model precise
Linux kernel	2.6.32	2.6.35

Table 1: Details of architectures used for experiments

Benchmark	Problem size
heat-1dp	$1.6\text{x}10^6 \times 1000$
heat-2dp	$16000^2 \times 500$
heat-3dp	$300^3 \times 200$
swim	$1335^2 \times 800$

Table 2: Problem sizes for benchmarks (grid $\times$ time steps)

Techniques we described have been implemented into Pluto [22]. Experimental evaluation was performed on two different multi-way SMP multicore setups: an Intel Xeon SMP system and an Opteron SMP one. Table 1 lists their hardware specification. Intel's C, C++, and Fortran compilers version 12.1.3 were used for all experiments, including for compiling codes we automatically generated.

The SPEC CPU2000fp swim benchmark (171.swim) is a weather prediction application that performs finite difference modeling of shallow water equations. It involves periodic boundary conditions on a two dimensional grid. Given that swim is part of the SPEC benchmarks, performance of code generated by a production compiler like *icc* is expected to be highly competitive and a strong reference point. There is no hand-optimized time-tiled code available for swim from prior art. Compiler flags used with *ifort* were "-O3 -ipo". We experimented with other combinations and found these to be the best. Most scores reported on `spec.org` for swim also use these flags for both base and peak tuning configurations. The Pochoir domain-specific compiler could not be used to specify such computation as explained in the next section.

Besides swim, we use three other representative periodic stencil benchmarks, heat-1dp, heat-2dp, and heat-3dp, from the Pochoir suite [33]. Problem sizes used are provided in Table 2. For swim, the reference input that the benchmark is required to be reported with was used—it specifies a 2-d grid of size 1335×1335 with an outer time loop of 800 iterations. For the heat benchmarks, problem sizes used are from the Pochoir suite and are meaningful for the respective computations. Performance for all is compared with Intel's compiler as well as the Pochoir stencil compiler [33] (version 0.5) that is publicly available.

Choice of benchmarks and coverage.

We argue that these benchmarks indeed comprehensively cover the domain of interest. Firstly, all realistic grid dimensionalities are covered. Other variations in input in this domain could come from a different width for the stencil, i.e., a different number of neighbors. However, this only affects the skewing factor needed to perform the tiling. For example, in Figure 4b, the skewing factor is one. The structure of the code and all other transformations and their effects remain the same. In addition, we did not find using different problem sizes or a different computation for the actual point update providing any additional insights. All data sets are significantly larger than the L3 cache.

icc-par or *ifort-par* refers to code auto-parallelized with Intel's C or Fortran compiler respectively, using the '-parallel' flag in addition to the flags specified in Table 1, while *icc-seq* refers to the same without auto-parallelization. *poly-diamond* refers to code we generate that is time tiled using diamond tiling while *poly-pipeline* is tiled with parallelogram shaped tiles. *poly-pipeline* suffers from pipelined startup and drain and thus load imbalance, while diamond tiling allows concurrent startup enabling maximal parallelism; both enable reuse along the time dimension. The tile sizes for *poly-diamond* are set to maximize locality and single thread performance. However, those for *poly-pipeline* are set to guarantee a sufficient number of tiles on the wavefront in the steady-state of the pipeline to keep all processors busy.

Table 3 and Table 4 show the performance of different tiled versions, and compare them with pochoir and the native compiler's auto-parallelization. Table 11 shows scaling for heat-2d periodic on the AMD Opteron. Overall, a very big improvement is seen over *icc-par* as the latter is not expected to time tile such stencils. Lack of time tiling makes the code memory bandwidth-bound yielding no or limited speedup in spite of parallelization. Due to better locality from time tiling, *poly-diamond* code incurs less memory-bandwidth per core, and the improvement with it increases with the number of cores. In some cases, the scaling with *poly-diamond* is not close to ideal since all implementations tend to get memory-bandwidth-bound for a large number of cores. However, the improvement is still very significant. Improvements are higher for lower dimensional stencils than for higher ones as the spatial reuse is lower for the former.

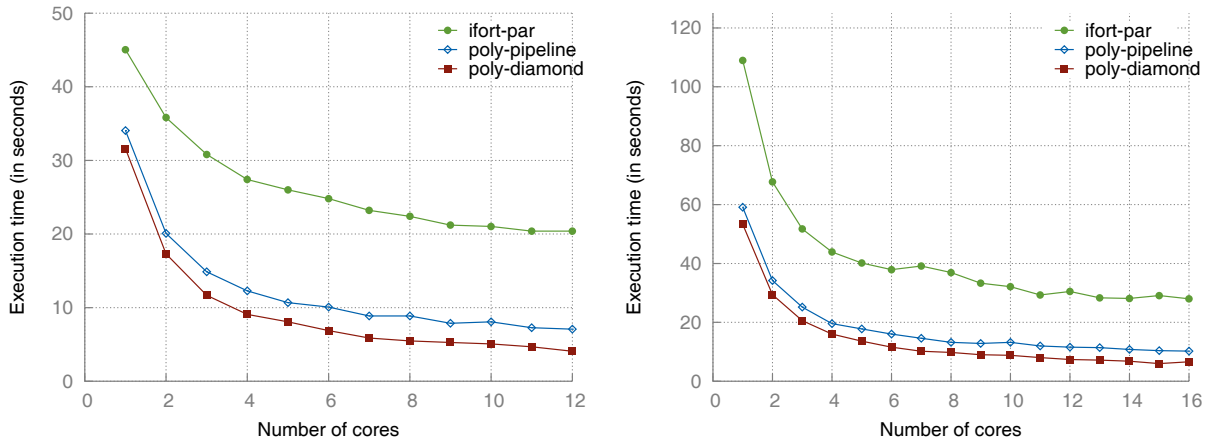
Except in one case, an improvement of 6% to 4$\times$ is seen over the Pochoir stencil compiler that is able to tile in the presence of periodicity, though with a different tiling strategy. The mean (geometric) speedup over it on the Intel and Opteron systems is 1.42$\times$ and 1.5$\times$ respectively. These performance improvements over Pochoir are similar to those observed for non-periodic stencils by Bandishti et al. [3].

Benchmark	1 core				12 cores				Speedup over	
	icc-seq	pochoir	pipeline	diamond	icc-par	pochoir	pipeline	diamond	icc-par	pochoir
heat-1dp	4.50s	2.09s	4.41s	1.66s	0.583s	195ms	2.5s	162.4ms	26.50	1.20
heat-2dp	517.9s	304.1s	459s	305.8s	570s	26.7s	65.5s	25.1s	22.70	1.06
heat-3dp	39.17s	50.27s	41.3s	36.81s	38.19s	11.5s	10.78s	5.07s	7.53	2.26
swim	45.04s	-	34.05s	31.6s	20.4s	-	7.07s	4.07s	5.00	-

Table 3: Running times and speedup with *poly-diamond* on Intel Xeon multicore SMP

Benchmark	1 core				16 cores				Speedup over	
	icc-seq	pochoir	pipeline	diamond	icc-par	pochoir	pipeline	diamond	icc-par	pochoir
heat-1dp	7.16s	3.36s	5.69s	1.88s	9.25s	0.235s	0.789s	0.318s	29.10	0.74
heat-2dp	1424s	504.5s	632.6s	454.8s	1211s	38.47s	65.9s	32.37s	4.12	1.19
heat-3dp	88.5s	93.86s	51.1s	67.55s	38.19s	32.81s	15.95s	8.41s	4.54	3.90
swim	109s	-	59.08s	53.29s	28.00s	-	10.21s	6.61s	4.24	-

Table 4: Running times and speedup with *poly-diamond* on AMD Opteron multicore SMP



(a) on 2-way SMP Intel Xeon E5645 (12 cores) (b) 2-way SMP AMD Opteron (16 cores)

Figure 10: Performance on Swim benchmark from SPEC2000fp

Figures 10a and 10b show improvement on the full swim benchmark. Our approach splits the data domain into four partitions as shown in Figure 9 before applying reversal, shifts, and skewing transformations. Time tiling allows nearly ideal scaling in contrast to *ifort-par* which scales poorly even when the number of cores is low. This behavior has indeed been expected without techniques to exploit reuse along the time dimension. Table 5 supports the claim that improved locality leads to higher performance and better scaling. With inner space loop tiling and parallelization alone, the computation incurs significantly higher number of cache misses and is memory bandwidth-bound. Both *ifort-par* and *poly-diamond* utilize all cores as was reflected from the CPU utilization. *poly-pipeline* suffers from load imbalance due to a pipelined startup and drain phase.

Hardware event	Count (in billions)	
	ifort-par	poly-diamond
L2_RQSTS.LD_HIT	1.23	0.731
L2_RQSTS.LD_MISS	1.74	0.238
L2_RQSTS.LOADS	2.97	0.977
L2_RQSTS.MISS	5.73	0.635
L2 prefetch requests	4.15	0.400
L2 prefetch hits	0.63	0.070
L2 prefetch misses	3.52	0.322

Table 5: Performance counters comparing *ifort-par* with *poly-diamond* for swim on 12 cores on the Intel multicore

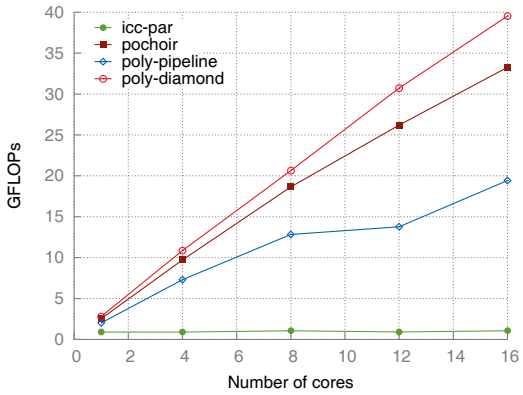


Figure 11: Periodic heat-2d scaling on the Opteron system

The running times of our generated diamond-tiled code for swim on the Intel system and the resulting SPEC rate of 761.67 that we achieve are also better than the highest ever publicly reported on `spec.org`—across all machines (as of 2013). A direct comparison with any of those numbers is however not possible since the machines for which the numbers were reported are different from ours.

Figure 12 shows that time-tiled code for the periodic case provides roughly the same performance as the non-periodic ones. Note that the amount of computation for both periodic and non-periodic,

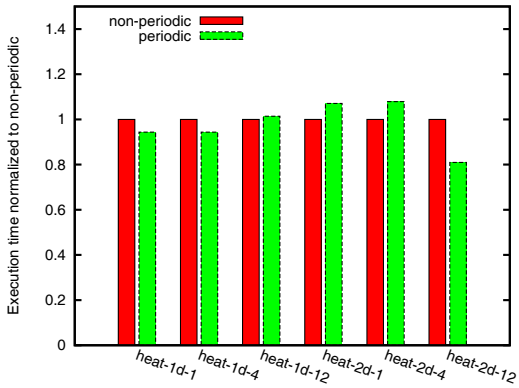


Figure 12: Non-periodic vs periodic stencil performance with time tiling (poly-diamond) on the Intel system (1, 4, 12 cores)

given a particular grid size and number of time iterations, is the same. In one case, surprisingly, the periodic version performs better than the non-periodic. This clearly shows that the non-periodic one could have been optimized better. In general, the periodic stencil code has a more complex structure and is expected to only perform at most as well as the non-periodic one. More optimizations in the polyhedral code generator Cloog could simplify it for better optimization by the native compiler. Overall, these interactions have not yet been studied fully, and this not being the main focus of this paper, are planned for future.

7. RELATED WORK

Recent stencil optimization works that include some domain-specific ones [10, 30, 31, 34, 16] and compiler-based ones [29, 39, 18, 7, 3] do not optimize those with periodic boundary conditions. The Pochoir [33] stencil compiler is the only one, to the best of our knowledge, that supports periodic conditions while applying the optimizations within the scope of this paper. Results indicate that Pochoir is able to perform time tiling via trapezoids regardless of the presence of periodic conditions, but the generated code is not as efficient as with our technique, as discussed in Section 6. We could not find a way to write multiple inter-related stencil computations with Pochoir, and hence the SPEC benchmark swim could not be expressed with it. However, ours being a general-purpose compiler approach driven by data dependences naturally handles such code. Of course, domain-specific optimization efforts have an opportunity to generate better code due to the greater amount of information they have about the problem, and our framework is suitable for integration into domain-specific stencil compilers.

Index-set splitting [14] and iteration space slicing [23] are transformations that partitions iteration domains into smaller sub-domains. This in turn allows different scheduling functions for different pieces of the program and results in more freedom. These seminal works focus on minimizing the dimensionality and latency of admissible schedules. In this work we exploit the degrees of freedom offered by index-set splitting as well as the expressiveness of linear transformations to reduce folding to an index-set splitting problem followed by a dependence shortening transformation problem.

Multiple tiling strategies have been devised to optimize stencil computations for shared and distributed memories. Originally, spatial decomposition through rectangular tiles is applied to the spatial dimensions. Spatial decomposition has the advantage of being simple to achieve but does not exhibit temporal reuse. The 171.swim SPEC CPU2000fp benchmark implements a well-known shallow water simulation model [32, 27]; an earlier version with a smaller data set was already included in the SPEC CPU1995fp suite. It lends itself well to spatial decomposition. However, spatial decomposition alone is not sufficient to reduce the memory-bandwidth consumption of the simulation model. As shown in an earlier work on semi-automatic loop nest optimization, the swim benchmark is amenable to loop fusion across one iteration of the time loop. Such polyhedral-enhanced fusion improves temporal locality and achieved 34% speedup on single-threaded execution [9, 13]. But despite much progress in production and research compilers since 1995, and despite the promises of a boost in the overall SPEC CPU score, time tiling remained inaccessible for the swim benchmark.

Time tiling was proposed to aggregate multiple time iterations and increase temporal reuse compared to tiling only in the data space [39]. Time tiling has roots in Lamport's hyperplane method [19] and is the most widely implemented technique within polyhedral transformation tools and compilers. Due to its reliance on loop skewing to extract parallel wavefronts of tiles, traditional time tiling suffered from two problems: (1) pipelined startup and shutdown phases in which some processors do not have work, and (2) load-imbalance due to insufficient number of tiles along each wavefront. For stencils implementing an explicit residual smoothing scheme such as Jacobi iterations, concurrent startup is possible [18] and results in asymptotically more parallelism than available with the traditional form of skewing-enabled time tiling. A successful tiling scheme which systematically exploits available parallelism is based on diamond tiling [3]. Our contribution builds on these insights, and extends them to stencils with periodic boundary conditions. This results in asymptotically more parallelism and locality on stencils with boundary conditions than was previously available [40].

Choffrut and Culik [8] perform folding on two-dimensional systolic arrays eliminating long wires for connections between elements that are related by reflections and/or rotations. [24] hints at using reflections to find piece-wise linear schedules as opposed to schedules for tiling: however, we found the approach proposed to determine splits itself to be incomplete and preliminary in its description, and very limited in its applicability. Yaacoby et al. [42] presents an algorithm on "uniformizing" dependences in affine recurrence equations in the context of systolic array synthesis through generalized folding. Though the method is unique because of its use of images of dependences and the characterization of affine recurrence equations which can be uniformized, its practical application and subsequent scalability is limited by its reliance on closures of dependence maps, eigenvalues and cycles in the dependence graph. Also, the formalism as described does not capture long dependences across boundaries—this is needed to derive folding for periodic stencils. Overall, our approach is inspired by folding, but, for the problem of tiling and parallelization for the domain of interest here, is more general and made possible by reasoning through index set splitting for dependence shortening. It is also far more robust and resilient to variations in dependence patterns, as argued towards the end of Section 4: it was made possible by minimizing the upper bound on the distance of the splitting hyperplane from the mid-points of long dependences. It thus subsumes reflections. Our approach can also seamlessly deal with any grid dimensionality as opposed to only up to two-dimensional as in the case of [8].

8. CONCLUSIONS

We introduced an automatic method to optimizing time-iterated computations on periodic domains. Our method relies on an original index set splitting scheme. The scheme allowed us to transparently apply tiling transformations with the existing objective function used in Pluto. Experimental results on the swim SPEC

CPU2000fp benchmark showed a speedup of nearly 5× over the highest performance achieved by a highly tuned commercial production compiler. We are not aware of any SPEC numbers for swim that come close to this result, obtained through either manual or automatic means. On other representative stencil computations, our scheme provides performance similar to that achieved with no periodicity. In addition, our technique always matches or outperforms—by up to 4×—a domain-specific stencil capable of handling periodicity in simpler cases. Our method is implemented in an open source research compiler and is available [22].

These results are not only interesting for computational sciences, but also excellent news for programming language and compiler designers. We conclude that it is practically infeasible to manually reproduce the optimizations we performed on swim or any other periodic stencil, especially on a two-dimensional or higher data grid. On the other hand, advanced tools can deal with this complexity, opening dimensions of program optimization that have so far been practically out of the reach of domain experts.

Acknowledgments.

This work was partially funded as part of the ARTEMIS COPCAMS project id. 332913, by the FP7 project CARP id. 287767, and by the INRIA-IISc PolyFlow associate team. We would like to thank P. Sadayappan and Tobias Grosser for their important role during the early stages of this work. We are also thankful to the reviewers of PACT 2014 for their detailed comments.

9. REFERENCES

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

[2] A. V. Aho, R. Sethi, J. D. Ullman, and M. S. Lam. *Compilers: Principles, Techniques, and Tools Second Edition*. Prentice Hall, 2006.

[3] V. Bandishti, I. Pananilath, and U. Bondhugula. Tiling stencil computations to maximize parallelism. In *SC*, pages 40:1–40:11, 2012.

[4] U. K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[5] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *ETAPS CC*, 2008.

[6] U. Bondhugula and A. Cohen. Handling negative coefficients in Pluto. Technical Report 1, Indian Institute of Science, Feb. 2014.

[7] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI*, pages 101–113, 2008.

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

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

[10] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. A. Patterson, J. Shalf, and K. A. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Supercomputing*, page 4, 2008.

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

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

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

[14] M. Griebl, P. Feautrier, and C. Lengauer. Index set splitting. *International Journal of Parallel Programming*, 28(6):607–631, 2000.

[15] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short simd architectures. In *ETAPS International Conference on Compiler Construction (CC'11)*, pages 225–245, Saarbrucken, Germany, Mar. 2011.

[16] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. In *ACM ICS*, 2013.

[17] F. Irigoin and R. Triolet. Supernode partitioning. In *ACM SIGPLAN Principles of Programming Languages*, pages 319–329, 1988.

[18] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective Automatic Parallelization of Stencil Computations. In *ACM SIGPLAN PLDI*, July 2007.

[19] L. Lamport. The hyperplane method for an array computer. In *Proceedings of the Sagamore Computer Conference on Parallel Processing*, pages 113–131, London, UK, 1975. Springer-Verlag.

[20] A. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 201–214, 1997.

[21] N. Osheim, M. M. Strout, D. Rostron, and S. Rajopadhye. Smashing: Folding space to tile through time. In J. N. Amaral, editor, *LCPC*, pages 80–93. Springer-Verlag, 2008.

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

[23] W. Pugh and E. Rosser. Iteration space slicing and its application to communication optimization. In *International Conference on Supercomputing*, pages 221–228, 1997.

[24] S. Rajopadhye, L. Mui, and S. Kiaei. Piecewise linear schedules for recurrence equations. In *VLSI Signal Processing V, IEEE Press*, pages 375–384, Oct. 1992.

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

[26] D. A. Randall, T. D. Ringler, R. P. Heikes, P. Jones, and J. Baumgardner. Climate modeling with spherical geodesic grids. *Computing in Science and Engg.*, 4(5):32–41, Sept. 2002.

[27] R. Sadourny. The dynamics of finite-difference models of the shallow-water equations. *J. atm. sciences*, 32(4), Apr. 1975.

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

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

[30] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel. Cache oblivious parallelograms in iterative stencil computations. In *ACM ICS*, pages 49–59, 2010.

[31] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel. Cache accurate time skewing in iterative stencil computations. In *ICPP*, pages 571–581, 2011.

[32] P. N. Swarztrauber. 171.swim spec cpu2000 benchmark description file. Standard Performance Evaluation Corporation. `http://www.spec.org/cpu2000/CFP2000/171.swim/docs/171.swim.html`, 2000.

[33] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *SPAA*, pages 117–128, 2011.

[34] J. Treibig, G. Wellein, and G. Hager. Efficient multicore-aware parallelization strategies for iterative stencil computations. *Journal of Computational Science*, 2(2):130 – 137, 2011.

[35] S. Verdoolaege. isl: An integer set library for the polyhedral model. In K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327, pages 299–302. Springer, 2010.

[36] S. Verdoolaege. Integer Set Library, 2013. An integer set library for program analysis.

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

[38] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., 1995.

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

[40] D. Wonnacott and M. Strout. On the scalability of loop tiling techniques. In *International workshop on Polyhedral compilation techniques*, 2013.

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

[42] Y. Yaacoby and P. R. Cappello. Converting affine recurrence equations to quasi-uniform recurrence equations. *VLSI Signal Processing*, 11(1-2):113–131, 1995.