Tiling and Optimizing Time-Iterated Computations over Periodic Domains

Uday Bondhugula\textsuperscript{1}, Vinayaka Bandishti\textsuperscript{1}, Albert Cohen\textsuperscript{2,3}, Guillain Potron\textsuperscript{1,3}, Nicolas Vasilache\textsuperscript{4}

\textsuperscript{1}Indian Institute of Science, \textsuperscript{2}INRIA, \textsuperscript{3}ENS, \textsuperscript{4}Reservoir Labs

PACT 2014
Aug 24–27, 2014
Edmonton, Canada
1 Introduction and Motivation

2 Our Technique: Index Set Splitting

3 Dependence Shortening

4 Experimental Evaluation
Stencil Computations

- Used in iterative solution to finite element methods for partial differential equations
- Repeatedly perform computations on a data grid a certain number of times or till convergence
- Exhibit near-neighbor dependences
Run for a certain number of iterations (surrounding time loop)
Each time iteration sweeps a discretized data grid (typically 2-d or 3-d)
Computationally intensive: $\theta(N^3)$ data, $N^3 \times T$ iterations
Memory bandwidth bound as per original specification
Stencils on Periodic Domains

- The data domains can be non-periodic or **periodic**
- Periodicity arises as a result of modeling a portion of a larger domain
- ... also as a result of a flattened domain (a ring modeled in a linear array, cylinder, or sphere)
- Periodicity is a significant domain (*swim* SPECFP2000)
- Results in *long* dependences in both directions
- These long dependences create a problem for tiling and other optimizations

Figure: no periodicity
Introduction and Motivation

Stencils on Periodic Domains

- The data domains can be non-periodic or **periodic**
- Periodicity arises as a result of modeling a portion of a larger domain
- ... also as a result of a flattened domain (a ring modeled in a linear array, cylinder, or sphere)
- Periodicity is a significant domain (*swim* SPECFP2000)
- Results in *long* dependences in both directions
- These long dependences create a problem for tiling and other optimizations

**Figure:** no periodicity

**Figure:** with periodicity
The data domains can be non-periodic or **periodic**. Periodicity arises as a result of modeling a portion of a larger domain. Also, as a result of a flattened domain (a ring modeled in a linear array, cylinder, or sphere). Periodicity is a significant domain (*swim* SPECFP2000) and results in *long* dependences in both directions. These long dependences create a problem for tiling and other optimizations.

**Figure:** with periodicity
Tiling or Blocking

- Decomposes an iteration space uniformly into blocks; each block is executed atomically (no cycle between tiles)
- Proposed first by Irigoin and Triolet [POPL 1988], Wolf [SC 1989]

- [Locality] A way to exploit reuse in multiple directions
- [Parallelism] A way to reduce frequency of synchronization
Tiling or Blocking

- Decomposes an iteration space uniformly into blocks; each block is executed atomically (no cycle between tiles)
- Proposed first by Irigoin and Triolet [POPL 1988], Wolf [SC 1989]
- **Locality** A way to exploit reuse in multiple directions
- **Parallelism** A way to reduce frequency of synchronization
Tiling or Blocking

- Decomposes an iteration space uniformly into blocks; each block is executed atomically (no cycle between tiles)
- Proposed first by Irigoin and Triolet [POPL 1988], Wolf [SC 1989]
- **Locality** A way to exploit reuse in multiple directions
- **Parallelism** A way to reduce frequency of synchronization
Tiling or Blocking

- Decomposes an iteration space uniformly into blocks; each block is executed atomically (no cycle between tiles)
- Proposed first by Irigoin and Triolet [POPL 1988], Wolf [SC 1989]
- **Locality** A way to exploit reuse in multiple directions
- **Parallelism** A way to reduce frequency of synchronization
Validity of Tiling

- Non-negative dependence components along a contiguous set of dimensions
- Short negative dependences can be made non-negative via loop skewing (time skewing)

Figure: Invalid tiling

- Periodic stencils cannot be tiled this way
Validity of Tiling

- Non-negative dependence components along a contiguous set of dimensions
- Short negative dependences can be made non-negative via loop skewing (time skewing)

Figure: Invalid tiling

Figure: Valid tiling

Periodic stencils cannot be tiled this way
Validity of Tiling

- Non-negative dependence components along a contiguous set of dimensions
- Short negative dependences can be made non-negative via loop skewing (time skewing)

**Figure:** Two ways of tiling heat-1d (non-periodic): parallelogram & diamond

- Periodic stencils cannot be tiled this way
Validity of Tiling

- Non-negative dependence components along a contiguous set of dimensions
- Short negative dependences can be made non-negative via loop skewing (time skewing)

No affine transformation on this domain that can make dependences non-negative along all dimensions

- Periodic stencils cannot be tiled this way
Validity of Tiling

- Non-negative dependence components along a contiguous set of dimensions
- Short negative dependences can be made non-negative via loop skewing (time skewing)

**Periodic stencils cannot be tiled this way**
- Short dependences $\Rightarrow$ Tiling is possible
- Long edges in both directions $\Rightarrow$ No tiling possible
1. Introduction and Motivation

2. Our Technique: Index Set Splitting

3. Dependence Shortening

4. Experimental Evaluation
**Key Idea**

- **Intuition**: cut the iteration space close to the mid-point of all long dependences simultaneously
- Create multiple statements out of the domains that result out of the cut
- Schedule these “sub-statements” with a dependence distance minimization objective (enlarges the space of transformations)
Key Idea

- **Intuition**: cut the iteration space close to the mid-point of all long dependences simultaneously
- Create multiple statements out of the domains that result out of the cut
- Schedule these “sub-statements” with a dependence distance minimization objective (enlarges the space of transformations)
Our Technique: Index Set Splitting

Example: a 1-d domain

- Cut it along $2i = N$
- Reverse the second domain $(t, i) \rightarrow (t, -i)$
- Shift it to the left (negative $i$ direction) by $N$ $(t, i) \rightarrow (t, N - i)$
- Now, all dependences are short and can apply time skewing
- Tiling transformations exist on the split index sets
Our Technique: Index Set Splitting

Example: a 1-d domain

- Cut it along $2i = N$
- Reverse the second domain $(t, i) \rightarrow (t, -i)$
- Shift it to the left (negative $i$ direction) by $N$ $(t, i) \rightarrow (t, N - i)$
- Now, all dependences are short and can apply time skewing
- Tiling transformations exist on the split index sets
Our Technique: Index Set Splitting

Example: a 1-d domain

- Cut it along $2i = N$
- Reverse the second domain $(t, i) \rightarrow (t, -i)$
- Shift it to the left (negative $i$ direction) by $N$
  $(t, i) \rightarrow (t, N - i)$
- Now, all dependences are short and can apply time skewing
- Tiling transformations exist on the split index sets
Our Technique: Index Set Splitting

Example: a 1-d domain

- Cut it along $2i = N$
- Reverse the second domain $(t, i) \rightarrow (t, -i)$
- Shift it to the left (negative $i$ direction) by $N$
  $(t, i) \rightarrow (t, N - i)$
- Now, all dependences are short and can apply time skewing
- Tiling transformations exist on the split index sets
How do you find such cuts in general?

How do you find the right sequence of transformations after the cuts?
Our Technique: Index Set Splitting

Finding near mid-point cuts

- A hyperplane is used to characterize a cut
- $\vec{h}$: orientation, $k$: position

Let the splitting hyperplane be $\vec{h}.\vec{i}_S = p$

With such a cut, $I_S$, is partitioned into two halves given by $I_S^+$ and $I_S^-$: $I_S^+ = I_S \cap \{\vec{h}.\vec{i}_S \geq p\}$ and $I_S^- = I_S \cap \{\vec{h}.\vec{i}_S \leq p - 1\}$
Our Technique: Index Set Splitting

Finding near mid-point cuts

- A hyperplane is used to characterize a cut
- $\vec{h}$: orientation, $k$: position

![Diagram showing a hyperplane with points $s_1$, $s_2$, $i_1$, and $i_2$.]

Let the splitting hyperplane be $\vec{h} \cdot \vec{i}_S = p$

With such a cut, $I_S$, is partitioned into two halves given by $I^+_S$ and $I^-_S$:

- $I^+_S = I_S \cap \{\vec{h} \cdot \vec{i}_S \geq p\}$
- $I^-_S = I_S \cap \{\vec{h} \cdot \vec{i}_S \leq p - 1\}$
Near mid-point cuts - more examples

- Cutting along the red line allows transformations shortening all dependences.
- Multiple cuts may be needed when dependences have long components in multiple dimensions.
Our Technique: Index Set Splitting

Finding the Split: Linear-Algebraically

- Recall: source iteration: \( \vec{s} \), target iteration: \( \vec{t} \), \( \vec{h} \) is hyperplane orientation, position of the hyperplane: \( p \)

- \( p \) should be at fixed distance from mid-point of \( \vec{h} \cdot \vec{s} \) and \( \vec{h} \cdot \vec{t} \):

\[
p - m \leq \frac{\vec{h} \cdot \vec{s} + \vec{h} \cdot \vec{t}}{2} \leq p + m, \quad \forall \langle \vec{s}, \vec{t} \rangle \in P_e
\]

- \( m \in \mathbb{Z}^+ \) does not depend on problem sizes
- There is a way to linearize these constraints (Farkas lemma)
- Solve an LP

Unknowns: \( \vec{h}, p, m \)
Our Technique: Index Set Splitting

Finding the Split: Linear-Algebraically

- Recall: source iteration: $\vec{s}$, target iteration: $\vec{t}$, $\vec{h}$ is hyperplane orientation, position of the hyperplane: $p$

- $p$ should be at fixed distance from mid-point of $\vec{h} \cdot \vec{s}$ and $\vec{h} \cdot \vec{t}$:

$$p - m \leq \frac{\vec{h} \cdot \vec{s} + \vec{h} \cdot \vec{t}}{2} \leq p + m, \quad \forall \langle \vec{s}, \vec{t} \rangle \in P_e$$

- $m \in \mathbb{Z}^+$ does not depend on problem sizes
- There is a way to linearize these constraints (Farkas lemma)
- Solve an LP
Our Technique: Index Set Splitting

Finding the Split: Linear-Algebraically

- Recall: source iteration: \( \vec{s} \), target iteration: \( \vec{t} \), \( \vec{h} \) is hyperplane orientation, position of the hyperplane: \( p \)
- \( p \) should be at fixed distance from mid-point of \( \vec{h} \cdot \vec{s} \) and \( \vec{h} \cdot \vec{t} \):

\[
p - m \leq \frac{\vec{h} \cdot \vec{s} + \vec{h} \cdot \vec{t}}{2} \leq p + m, \quad \forall \langle \vec{s}, \vec{t} \rangle \in P_e
\]

- \( m \in \mathbb{Z}^+ \) does not depend on problem sizes
- There is a way to linearize these constraints (Farkas lemma)
- Solve an LP
- Unknowns: \( \vec{h}, p, m \)
- Objective: minimize \( m \) to obtain \( \vec{h} \) and \( p \)
Our Technique: Index Set Splitting

Finding the Split: Linear-Algebraically

- Recall: source iteration: \( \vec{s} \), target iteration: \( \vec{t} \), \( \vec{h} \) is hyperplane orientation, position of the hyperplane: \( p \)
- \( p \) should be at fixed distance from mid-point of \( \vec{h} \cdot \vec{s} \) and \( \vec{h} \cdot \vec{t} \):
  \[
  p - m \leq \frac{\vec{h} \cdot \vec{s} + \vec{h} \cdot \vec{t}}{2} \leq p + m, \quad \forall \langle \vec{s}, \vec{t} \rangle \in P_e
  \]
- \( m \in \mathbb{Z}^+ \) does not depend on problem sizes
- There is a way to linearize these constraints (Farkas lemma)
- Solve an LP
- **Unknowns**: \( \vec{h}, p, m \)
- **Objective**: minimize \( m \) to obtain \( \vec{h} \) and \( p \)
If such an $m$ exists, a split hyperplane exists that cuts all long dependences within a fixed distance from their mid-points.

Otherwise, no such splitting hyperplane exists.

Perform such a procedure for each dimension along which dependences are long.
Our Technique: Index Set Splitting

Finding the Cut: Linear Algebraically

- If such an $m$ exists, a split hyperplane exists that cuts all long dependences within a fixed distance from their mid-points.
- Otherwise, no such splitting hyperplane exists.
- Perform such a procedure for each dimension along which dependences are long.
Our Technique: Index Set Splitting

Splitting a 2-d periodic domain with periodic conditions

2-d data grid and 1 time dimension

- Requires two cuts: \( \{2i = N\}, \{2j = N\} \)
- Iterations are partitioned into 4 pieces
1 Introduction and Motivation

2 Our Technique: Index Set Splitting

3 Dependence Shortening

4 Experimental Evaluation
Index set splitting allows the possibility of such a minimization on the split domains

Pluto’s [CC 2008, PLDI 2008] cost functions transforms to minimize dependence distances with a linear objective function

All techniques valid on non-periodic stencils can now be applied here (parallelograms, trapezoids, diamond, overlapped, split)
Index set splitting allows the possibility of such a minimization on the split domains.


All techniques valid on non-periodic stencils can now be applied here (parallelograms, trapezoids, diamond, overlapped, split).
Index set splitting allows the possibility of such a minimization on the split domains.


All techniques valid on non-periodic stencils can now be applied here (parallelograms, trapezoids, diamond, overlapped, split).
Dependence Shortening

- Original domain: \( I_S \), original schedule: \( T_{I_S} = (t, i) \)
- Split with \( 2i = N \) to obtain \( S^+ \) and \( S^- \), consider this sequence of transformations:
  - Reversal

\[
\begin{align*}
T_{I_S^+}(t, i) &= (t, i) \\
T_{I_S^-}(t, i) &= (t, -i)
\end{align*}
\]

- The new dependence distance becomes:

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

and this distance is short
Original domain: $I_S$, original schedule: $T_{I_S} = (t, i)$

Split with $2i = N$ to obtain $S^+$ and $S^-$, consider this sequence of transformations:

Parametric shift

$$T_{I_S^+}(t, i) = (t, i)$$
$$T_{I_S^-}(t, i) = (t, -i + N)$$

The new dependence distance becomes:

$$T_{I_S^-}(t) - T_{I_S^+}(s) = \left[ \begin{array}{c} (t + 1) - (0 + N - 1) + N \\ (t + 1) + (0 + N - 1) - N \\ t + 0 \\ t - 0 \end{array} \right] = \left[ \begin{array}{c} 1 \\ 0 \end{array} \right]$$

and this distance is short.
Original domain: $I_S$, original schedule: $T_{I_S} = (t, i)$

Split with $2i = N$ to obtain $S^+$ and $S^-$, consider this sequence of transformations:

Diamond tiling

$$T_{I_S^+}(t, i) = (t + i, t - i)$$
$$T_{I_S^-}(t, i) = (t - i + N, t + i - N)$$

The new dependence distance becomes:

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

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

and this distance is short.
Original domain: $I_S$, original schedule: $T_{I_S} = (t, i)$

Split with $2i = N$ to obtain $S^+$ and $S^-$, consider this sequence of transformations:

Diamond tiling

$$T_{I_S}^+(t, i) = (t + i, t - i)$$

$$T_{I_S}^-(t, i) = (t - i + N, t + i - N)$$

Consider a dependence from $(t, 0)$ to $(t + 1, N - 1)$ – it is originally long (distance is $(1, N - 1)$)

The new dependence distance becomes:

$$T_{I_S}^-(\vec{t}) - T_{I_S}^+(\vec{s}) = \left[ \begin{array}{c} (t + 1) - (0 + N - 1) + N \\ (t + 1) + (0 + N - 1) - N \end{array} \right] - \left[ \begin{array}{c} t + 0 \\ t - 0 \end{array} \right] = \left[ \begin{array}{c} 1 \\ 0 \end{array} \right]$$

and this distance is short
Original domain: \( I_S \), original schedule: \( T_{I_S} = (t, i) \)

Split with \( 2i = N \) to obtain \( S^+ \) and \( S^- \), consider this sequence of transformations:

Consider a dependence from \((t, 0)\) to \((t + 1, N - 1)\) – it is originally long (distance is \((1, N - 1)\))

The new dependence distance becomes:

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

and this distance is \textit{short}
Inside a tile, reverse the backward half-domains back since, once tiled, the dependences can be made long inside.

- Distribute the half-domains at the innermost level.
- Allows better locality, prefetching, and vectorization.
Regaining lost single thread performance

- Inside a tile, reverse the backward half-domains back since, once tiled, the dependences can be made long inside.
- Distribute the half-domains at the innermost level.
- Allows better locality, prefetching, and vectorization.

![Diagram showing the distribution of half-domains within a tile.](image-url)
Summary of Approach

- Use the polyhedral framework to represent computation (domains, schedules, and dependences)

- Cut close to mid-points of long dependences
- Reduce dependence distances on the new pieces created
- Perform complementary optimizations on a single tile
  - [+] Allows seamless application of existing techniques for non-periodic case
  - [+] Transparent to code generation
• Use the polyhedral framework to represent computation (domains, schedules, and dependences)

• Cut close to mid-points of long dependences
• Reduce dependence distances on the new pieces created
• Perform complementary optimizations on a single tile
• [+] Allows seamless application of existing techniques for non-periodic case
• [+] Transparent to code generation
Dependence Shortening

Splitting a 2-d periodic domain with periodic conditions

- Requires two cuts: \( \{2i = N\}, \{2j = N\} \)
- Iterations are partitioned into 4 pieces
- Stacking them as shown on the right results in a space with short dependences only
1. Introduction and Motivation

2. Our Technique: Index Set Splitting

3. Dependence Shortening

4. Experimental Evaluation
Experimental Setup

- On a 12-core Intel Xeon E5645 (Westmere) (2 x 6)
- On a 16-core AMD Opteron (Magny-cours) (2 x 8)
- All codes compiled with icc 12.1.3 with “-O3 -fp-model precise” on Linux (64-bit kernel 2.6. *)
  - heat-1d, heat-2d, heat-3d with periodic conditions, swim from SPECFP2000
  - Selected benchmarks cover the domain quite well
  - Comparison with pochoir [Tang et al SPAA’2011], icc -parallel, our approach with parallelogram poly-pipeline and diamond tiling poly-diamond after index set splitting
  - Implemented with open-source polyhedral libraries (pluto, cloog, pet, isl)
Experimental Setup

- On a 12-core Intel Xeon E5645 (Westmere) (2 x 6)
- On a 16-core AMD Opteron (Magny-cours) (2 x 8)
- All codes compiled with icc 12.1.3 with “-O3 -fp-model precise” on Linux (64-bit kernel 2.6.*)
- heat-1d, heat-2d, heat-3d with periodic conditions, swim from SPECFP2000
- Selected benchmarks cover the domain quite well
- Comparison with pochoir [Tang et al SPAA’2011], icc -parallel, our approach with parallelogram poly-pipeline and diamond tiling poly-diamond after index set splitting
- Implemented with open-source polyhedral libraries (pluto, cloog, pet, isl)
Experimental Evaluation

Swim benchmark (SPEC2000fp)

Figure: on 2-way SMP Intel Xeon E5645 (12 cores)

- A speedup of $5 \times$ with poly-diamond over `icc -parallel` on 12-cores ($1.5 \times$ on single core)
Performance Counters

<table>
<thead>
<tr>
<th>Hardware event</th>
<th>Count (in billions)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>L2.RQSTS.LD_HIT</td>
<td>1.23</td>
<td>0.731</td>
</tr>
<tr>
<td>L2.RQSTS.LD_MISS</td>
<td>1.74</td>
<td>0.238</td>
</tr>
<tr>
<td>L2.RQSTS LOADS</td>
<td>2.97</td>
<td>0.977</td>
</tr>
<tr>
<td>L2.RQSTS.MISS</td>
<td>5.73</td>
<td>0.635</td>
</tr>
<tr>
<td>L2 prefetch requests</td>
<td>4.15</td>
<td>0.400</td>
</tr>
<tr>
<td>L2 prefetch hits</td>
<td>0.63</td>
<td>0.070</td>
</tr>
<tr>
<td>L2 prefetch misses</td>
<td>3.52</td>
<td>0.322</td>
</tr>
</tbody>
</table>

Table: Performance counters comparing Intel Fortran compiler with poly-diamond for swim on 12 cores on the Intel multicore

- The number of L2 misses reduce by almost 8 times
Experimental Evaluation

Heat-2d on the Opteron

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

- Speedup of $3.1 \times$ on 1 core and of $37 \times$ on 16 cores over \textit{icc-par}
Non-periodic vs Periodic

Non-periodic vs periodic stencil performance with time tiling (*poly-diamond*).

- We obtain the same level of performance as for stencils with periodic tiling.
Summary of Performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>1 core</th>
<th>12 cores</th>
<th>Speedup over</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>icc-seq</td>
<td>pochoir</td>
<td>pipeline</td>
</tr>
<tr>
<td>heat-1dp</td>
<td>4.50s</td>
<td>2.09s</td>
<td>4.41s</td>
</tr>
<tr>
<td></td>
<td>517.9s</td>
<td>304.1s</td>
<td>459s</td>
</tr>
<tr>
<td>heat-2dp</td>
<td>39.17s</td>
<td>50.27s</td>
<td>41.3s</td>
</tr>
<tr>
<td></td>
<td>45.04s</td>
<td>-</td>
<td>34.05s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Mean speedups of 12.3x and 12.0x over *icc-par* on Xeon and Opteron systems respectively
- Mean speedup of 1.5× over a state-of-the-art domain-specific stencil compiler (Pochoir)
- Speedup of 5× and 4.2× over the highest SPEC performance achieved by native compilers on Intel Xeon and AMD Opteron multicore SMP systems
Mean speedups of 12.3x and 12.0x over *icc-par* on Xeon and Opteron systems respectively

Mean speedup of 1.5× over a state-of-the-art domain-specific stencil compiler (Pochoir)

Speedup of 5× and 4.2× over the highest SPEC performance achieved by native compilers on Intel Xeon and AMD Opteron multicore SMP systems

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>1 core</th>
<th>12 cores</th>
<th>Speedup over</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>icc-seq</td>
<td>pochoir</td>
<td>pipeline</td>
</tr>
<tr>
<td>heat-1dp</td>
<td>4.50s</td>
<td>2.09s</td>
<td>4.41s</td>
</tr>
<tr>
<td>heat-2dp</td>
<td>517.9s</td>
<td>304.1s</td>
<td>459s</td>
</tr>
<tr>
<td>heat-3dp</td>
<td>39.17s</td>
<td>50.27s</td>
<td>41.3s</td>
</tr>
<tr>
<td>swim</td>
<td>45.04s</td>
<td>-</td>
<td>34.05s</td>
</tr>
</tbody>
</table>
Experimental Evaluation

Summary of Performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>1 core</th>
<th>12 cores</th>
<th>Speedup over</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>icc-seq</td>
<td>pochoir</td>
<td>diamond</td>
</tr>
<tr>
<td>heat-1dp</td>
<td>4.50s</td>
<td>2.09s</td>
<td>4.41s</td>
</tr>
<tr>
<td>heat-2dp</td>
<td>517.9s</td>
<td>304.1s</td>
<td>459s</td>
</tr>
<tr>
<td>heat-3dp</td>
<td>39.17s</td>
<td>50.27s</td>
<td>41.3s</td>
</tr>
<tr>
<td>swim</td>
<td>45.04s</td>
<td>-</td>
<td>34.05s</td>
</tr>
</tbody>
</table>

- Mean speedups of 12.3x and 12.0x over icc-par on Xeon and Opteron systems respectively
- Mean speedup of 1.5× over a state-of-the-art domain-specific stencil compiler (Pochoir)
- Speedup of 5× and 4.2× over the highest SPEC performance achieved by native compilers on Intel Xeon and AMD Opteron multicore SMP systems
Availability

- The code generator at http://pluto-compiler.sourceforge.net
- Codes are available at http://mcl.csa.iisc.ernet.in/
No prior compiler-based automatic approach to time tiling in the presence of periodicity

Most prior work: CATS [Strzodka et al.], PATUS [Univ of Basel], Pluto (prior) do not handle / cannot tile in the presence of periodicity

Smashing [Ossheim et al. LCPC 2008] work uses the folding concept but no automatic way to perform transformation or code generation

Pochoir [SPAA 2011, MIT] - only tool to perform time tiling with periodicity (domain-specific)
  - Uses cache oblivious trapezoids
  - A domain-specific as opposed to a dependence-driven compiler approach – cannot handle swim

In comparison, we use diamond tiling (shown to be better even for the non-periodic case [Bandishti SC’12])
Related Work

- No prior compiler-based automatic approach to time tiling in the presence of periodicity
- Most prior work: CATS [Strzodka et al.], PATUS [Univ of Basel], Pluto (prior) do not handle / cannot tile in the presence of periodicity
- Smashing [Ossheim et al LCPC 2008] work uses the folding concept but no automatic way to perform transformation or code generation
- Pochoir [SPAA 2011, MIT] - only tool to perform time tiling with periodicity (domain-specific)
  - Uses cache oblivious trapezoids
  - A domain-specific as opposed to a dependence-driven compiler approach – cannot handle swim
- In comparison, we use diamond tiling (shown to be better even for the non-periodic case [Bandishti SC’12])
No prior compiler-based automatic approach to time tiling in the presence of periodicity

Most prior work: CATS [Strzodka et al.], PATUS [Univ of Basel], Pluto (prior) do not handle / cannot tile in the presence of periodicity

Smashing [Ossheim et al LCPC 2008] work uses the folding concept but no automatic way to perform transformation or code generation

Pochoir [SPAA 2011, MIT] - only tool to perform time tiling with periodicity (domain-specific)
  - Uses cache oblivious trapezoids
  - A domain-specific as opposed to a dependence-driven compiler approach – cannot handle *swim*
  - In comparison, we use diamond tiling (shown to be better even for the non-periodic case [Bandishti SC’12])
Related Work: Past approaches conceptually applicable for periodic stencils

- Cut and Paste / Circular skewing
- Overlapped tiling
- Folding
1. Cut and Paste

**Figure:** Cut and paste over diamond tiling

- Break cycle by cutting and pasting dependent portion
- Similar to circular loop skewing
- [-] Very hard to impractical to determine what portion at the boundaries should be cut
- [-] Code generation will be very hard
2. Overlapped Tiling

[Figure: Overlapped tiling for 1-D Jacobi.]

- Break cycle by doing redundant computation [Krishnamoorthy et al. PLDI’07]
- Performs redundant computations (higher for higher dimensional data grids)
- Determining overlapping regions at boundaries is very hard
- Code generation is very hard (esp. shared memory)
3. Folding

- [Choffrut and Ciulik 1983, Yaacoby and Cappello 1995] Used in 1-d systolic arrays to reduce long wires at boundaries

Folding it at middle point brings the boundaries together

Smashing approach [Ossheim et al. 2010] motivated by folding

Our approach is motivated by folding but much more general and powerful
3. Folding

- [Choffrut and Ciulik 1983, Yaacoby and Cappello 1995] Used in 1-d systolic arrays to reduce long wires at boundaries

- Folding it at middle point brings the boundaries together

- Smashing approach [Ossheim et al. 2010] motivated by folding

- Our approach is motivated by folding but much more general and powerful
Conclusions

- Proposed a technique to allow tiling of stencil computations over periodic domains
- Viewed the problem as index set splitting + dependence shortening
- Index set splitting was driven by *close to mid-point cutting*
- The resulting time tiling leads to dramatic speedups over non-time-tiled code (5x on *swim*)
- A fully automatic end-to-end tool to do this – can be used in domain-specific compilers too
Conclusions

- Proposed a technique to allow tiling of stencil computations over periodic domains
- Viewed the problem as index set splitting + dependence shortening
- Index set splitting was driven by *close to mid-point cutting*
- The resulting time tiling leads to dramatic speedups over non-time-tiled code (5x on *swim*)
- A fully automatic end-to-end tool to do this – can be used in domain-specific compilers too
Acknowledgments

- Reviewers of PACT 2014
- INRIA for an Associate Team grant

Questions?
## Problem sizes

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Problem size</th>
</tr>
</thead>
<tbody>
<tr>
<td>heat-1dp</td>
<td>$1.6 \times 10^6 \times 1000$</td>
</tr>
<tr>
<td>heat-2dp</td>
<td>$16000^2 \times 500$</td>
</tr>
<tr>
<td>heat-3dp</td>
<td>$300^3 \times 200$</td>
</tr>
<tr>
<td>swim</td>
<td>$1335^2 \times 800$</td>
</tr>
</tbody>
</table>
There are different ways to write periodic boundary conditions:

1. **Conditionals**

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

2. **Copies (Swim SPEC FP 2000 code)**
Periodic Stencils: The code

There are different ways to write periodic boundary conditions

1. **Conditionals**

2. **Copies (Swim SPEC FP 2000 code)**

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