Tiling and Optimizing Time-Iterated Computations over Periodic Domains

Uday Bondhugula[1], Vinayaka Bandishti[1], Albert Cohen[2,3], Guillain Potron[1,3], Nicolas Vasilache[4]

[1]Indian Institute of Science, [2]INRIA, [3]ENS, [4]Reservoir Labs

PACT 2014
Aug 24–27, 2014
Edmonton, Canada

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

Stencils: Code Structure (non-periodic)

```
for (t=1; t<=T; t++) { /* Time loop */
  for (x=1; x<=N-2; x++) { /* Space loop: x */
    for (y=1; y<=N-2; y++) { /* Space loop: y */
      for (z=1; z<=N-2; z++) { /* Space loop: z */
          d[t%2][x][y][z] = f(d[(t-1)%2][x][y][z],
                              d[(t-1)%2][x][y][z+1],
                              d[(t-1)%2][x][y][z-1],
                              d[(t-1)%2][x-1][y][z],
                              d[(t-1)%2][x+1][y][z],
                              d[(t-1)%2][x][y+1][z],
                              d[(t-1)%2][x][y-1][z]
                             );
      }
    }
  }
}
```

- 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, $N^3 * T$ 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 results 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

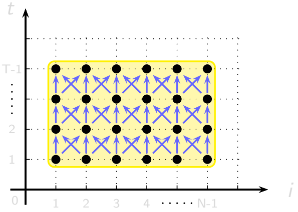


Figure: no periodicity

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 results 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

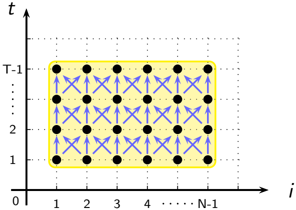


Figure: no periodicity

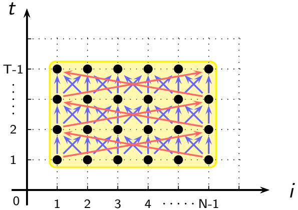


Figure: with periodicity

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 results 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

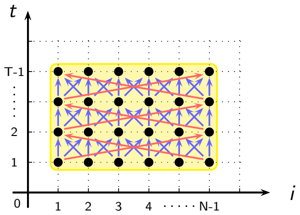
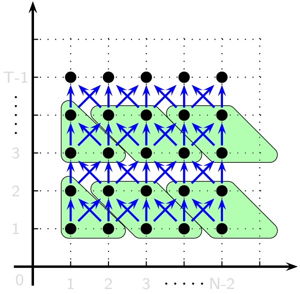


Figure: with periodicity

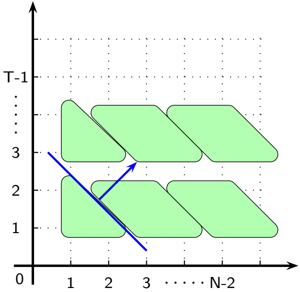
Tiling or Blocking

- 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



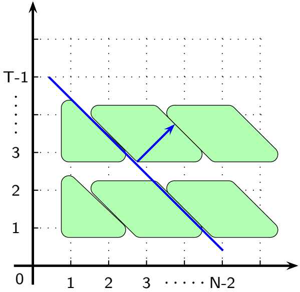
Tiling or Blocking

- 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



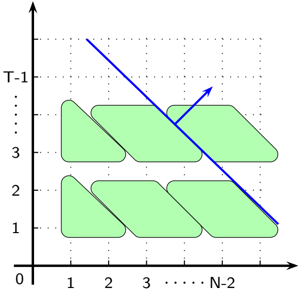
Tiling or Blocking

- 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)

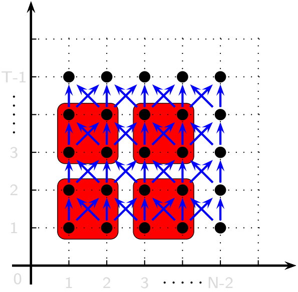


Figure: Invalid tiling

- **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)

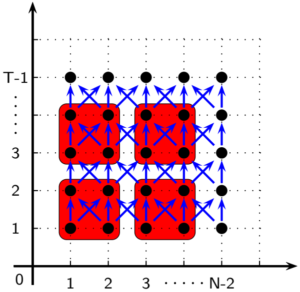


Figure: Invalid tiling

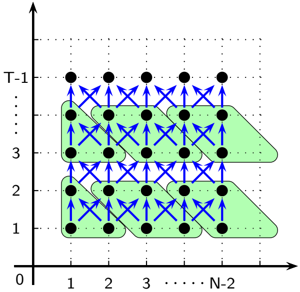


Figure: Valid tiling

- 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)

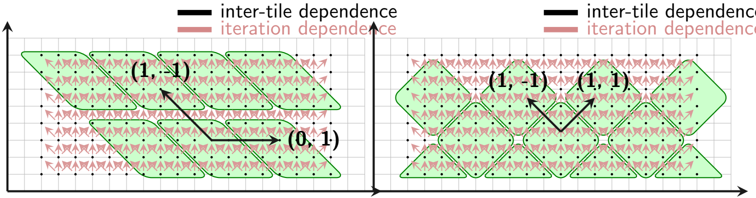
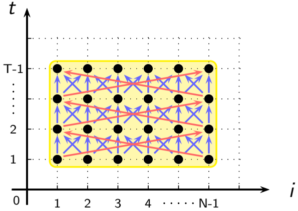


Figure: Two ways of tiling heat-1d (non-periodic): parallelogram & diamond

- 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

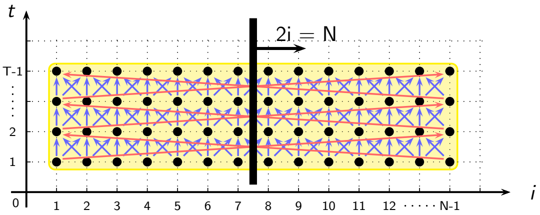
Key Idea

- **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

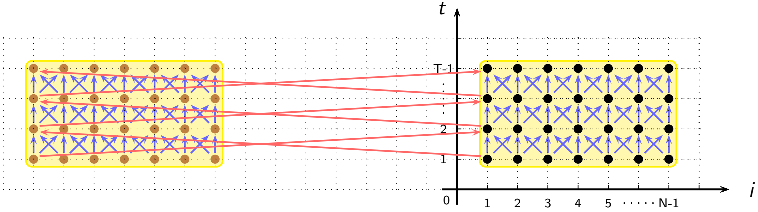
- **Intuition**: cut the iteration space close to the mid-point of all long dependences simultaneously
- Create multiple statements out of the domains that result out of the cut
- Schedule these "sub-statements" with a dependence distance minimization objective (enlarges the space of transformations)

Example: a 1-d domain



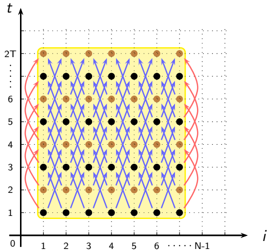
- Cut it along $2i = N$
- Reverse the second domain $(t, i) \longrightarrow (t, -i)$
- Shift it to the left (negative i direction) by N $(t, i) \longrightarrow (t, N - i)$
- Now, all dependences are short and can apply time skewing
- Tiling transformations exist on the split index sets

Example: a 1-d domain



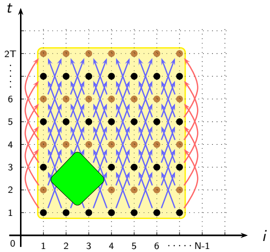
- Cut it along $2i = N$
- Reverse the second domain $(t, i) \longrightarrow (t, -i)$
- Shift it to the left (negative i direction) by N $(t, i) \longrightarrow (t, N - i)$
- Now, all dependences are short and can apply time skewing
- Tiling transformations exist on the split index sets

Example: a 1-d domain



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

Example: a 1-d domain

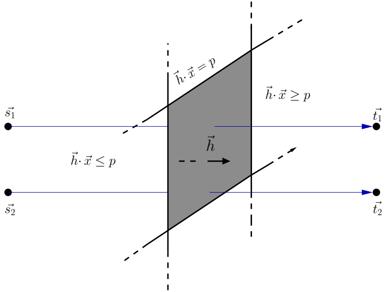


- Cut it along $2i = N$
- Reverse the second domain $(t, i) \longrightarrow (t, -i)$
- Shift it to the left (negative i direction) by N
  $(t, i) \longrightarrow (t, N - i)$
- Now, all dependences are short and can apply time skewing
- Tiling transformations exist on the split index sets

- How do you find such cuts in general?
- How do you find the right sequence of transformations after the cuts?

Finding near mid-point cuts

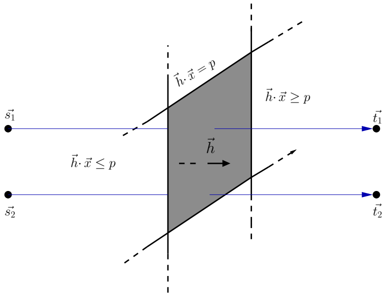
- A hyperplane is used to characterize a cut
- $\vec{h}$: orientation, k: position



- 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^-: $I_S^+ = I_S \cap \{\vec{h}.\vec{i_S} \geq p\}$ $\quad I_S^- = I_S \cap \{\vec{h}.\vec{i_S} \leq p-1\}$

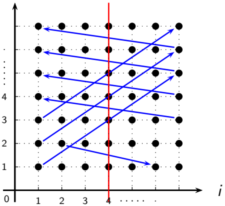
Finding near mid-point cuts

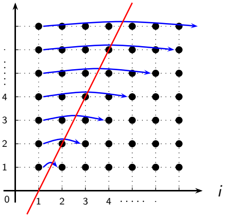
- A hyperplane is used to characterize a cut
- $\vec{h}$: orientation, k: position



- 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^-: $I_S^+ = I_S \cap \{\vec{h}.\vec{i_S} \geq p\}$ $\quad I_S^- = I_S \cap \{\vec{h}.\vec{i_S} \leq p-1\}$

Near mid-point cuts - more examples

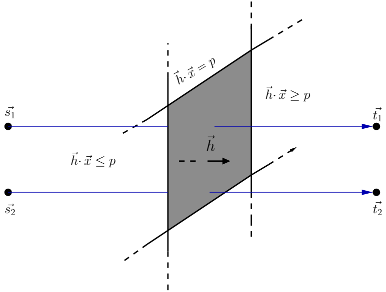




- Cutting along the red line allows transformations shortening all dependences
- Multiple cuts may be needed when dependences have long components in multiple dimensions

Finding the Split: Linear-Algebraically

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



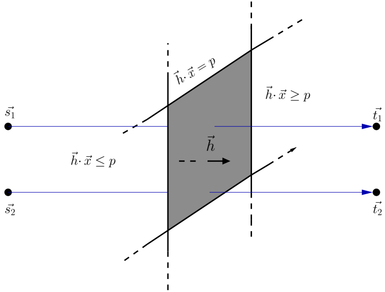
- p should be at fixed distance from mid-point of $\vec{h}.\vec{s}$ and $\vec{h}.\vec{t}$:

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

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

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 $\vec{h}.\vec{s}$ and $\vec{h}.\vec{t}$:

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

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

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 $\vec{h}.\vec{s}$ and $\vec{h}.\vec{t}$:

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

- $m \in \mathrm{Z}^+$ 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

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 $\vec{h}.\vec{s}$ and $\vec{h}.\vec{t}$:

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

- $m \in \mathbb{Z}^+$ 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

Finding the Cut: Linear Algebraically

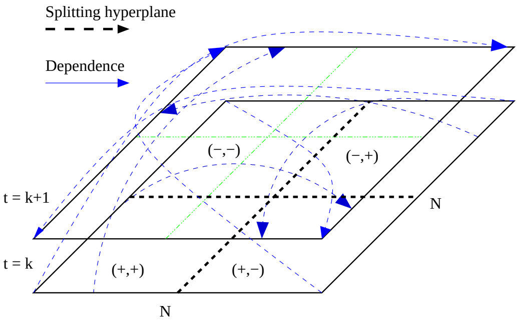
- If such an m exists, a split hyperplane exists that cuts all long dependences within a fixed distance from their mid-points
- Otherwise, no such splitting hyperplane exists
- Perform such a procedure for each dimension along which dependences are long

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

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

Dependence Shortening

- 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)

Dependence Shortening

- 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)

Dependence Shortening

- 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)

Dependence Distance 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

$$\begin{aligned} T_{I_S^+}(t,i) &= (t,i) \\ T_{I_S^-}(t,i) &= (t,-i) \end{aligned}$$

- 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 *short*

Dependence Distance 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:

  Parametric shift

$$\begin{aligned} T_{I_S^+}(t,i) &= (t,i) \\ T_{I_S^-}(t,i) &= (t,-i+N) \end{aligned}$$

- 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 *short*

Dependence Distance 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:

  Diamond tiling

$$\begin{aligned} T_{I_S^+}(t, i) &= (t + i, t - i) \\ T_{I_S^-}(t, i) &= (t - i + N, t + i - N) \end{aligned}$$

- 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 *short*

Dependence Distance 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:
  Diamond tiling

$$
\begin{aligned}
T_{I_S^+}(t,i) &= (t+i, t-i) \\
T_{I_S^-}(t,i) &= (t-i+N, t+i-N)
\end{aligned}
$$

- 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:

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

and this distance is *short*

Dependence Distance Shortening

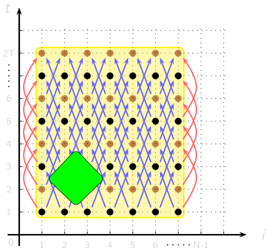
- Original domain: I_S, original schedule: $T_{I_S} = (t, i)$
- Split with $2i = N$ to obtain S^+ and S^-, consider this sequence of transformations:
- Consider a dependence from $(t, 0)$ to $(t+1, N-1)$ – it is originally long (distance is $(1, N-1)$)
- The new dependence distance becomes:

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

and this distance is *short*

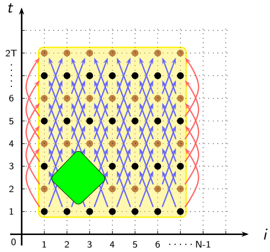
Regaining lost single thread performance

- 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



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



Summary of Approach

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

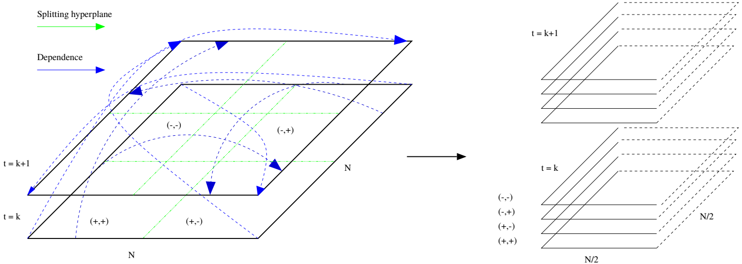
- Cut close to mid-points of long dependences
- Reduce dependence distances on the new pieces created
- Perform complementary optimizations on a single tile
- [+] Allows seamless application of existing techniques for non-periodic case
- [+] Transparent to code generation

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

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

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)

Swim benchmark (SPEC2000fp)

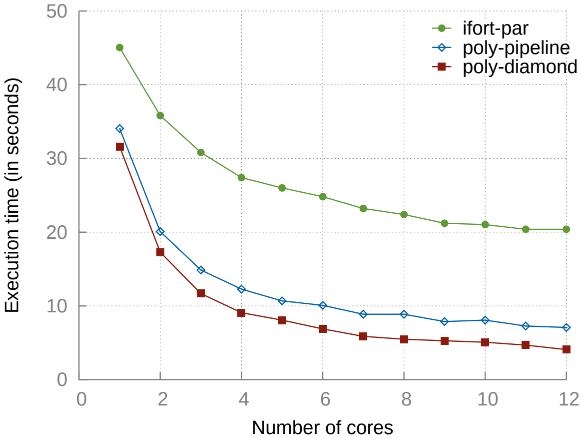


Figure: on 2-way SMP Intel Xeon E5645 (12 cores)

- A speedup of 5× with *poly-diamond* over *icc -parallel* on 12-cores (1.5× on single core)

Performance Counters

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: 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

Heat-2d on the Opteron

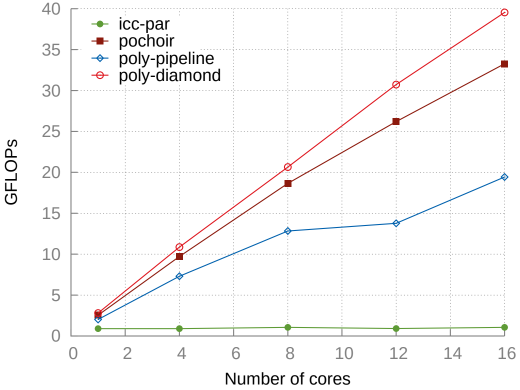
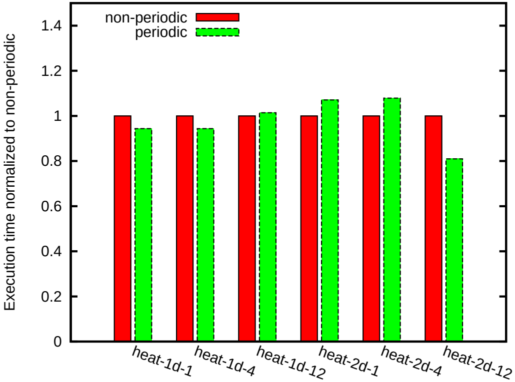


Figure: Periodic heat-2d scaling on the Opteron system

- Speedup of 3.1× on 1 core and of 37× on 16 cores over *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

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	-

- 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

Summary of Performance

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	-

- 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

Summary of Performance

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	-

- 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/

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])

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])

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])

Related Work: Past approaches conceptually applicable

Past approaches conceptually applicable for periodic stencils

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

1. Cut and Paste

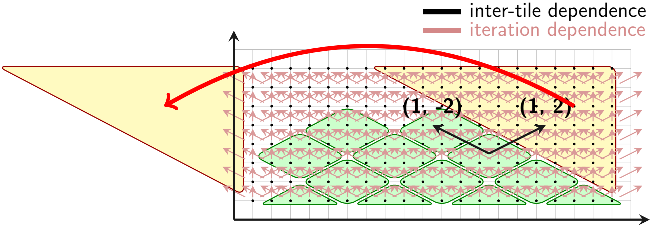


Figure: Cut and paste over diamond tiling

- 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

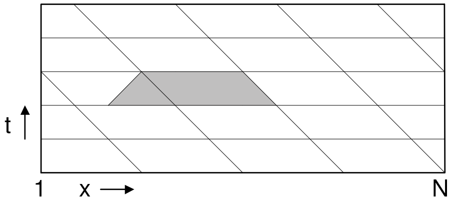
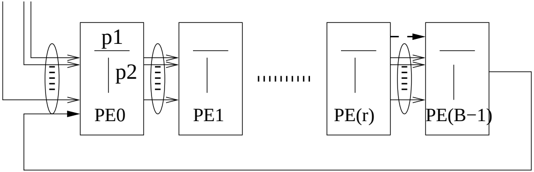


Figure: Overlapped tiling for 1-D Jacobi.

- 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

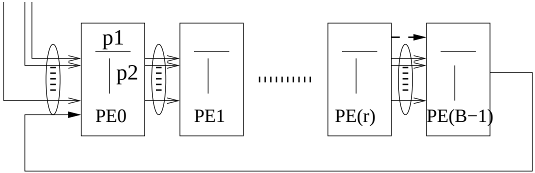
- [Choffrut and Ciulik 1983, Yaacoby and Cappello 1995] Used in 1-d systolic arrays to reduce long wires at boundaries



- 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

Thank you

Acknowledgments

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

Questions?

Problem sizes

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	$1335^2 \times 800$

Periodic Stencils: The code

- There are different ways to write periodic boundary conditions
  1. Conditionals
  2. Copies (Swim SPEC FP 2000 code)

```
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;
    }
}
```

Periodic Stencils: The code

- There are different ways to write periodic boundary conditions
  1. Conditionals
  2. Copies (Swim SPEC FP 2000 code)

```
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;
    }
}
```