Optimal ILP and Register Tiling: Analytical Model and Optimization Framework

Lakshminarayanan. Renganarayana,
Upadrasta Ramakrishna,
Sanjay Rajopadhye
Computer Science Department
Colorado State University

Overview

- ILP and register reuse
- Execution time and register pressure functions
- Optimal ILP and register tiling problem
- Optimal tiling problem as convex opt. problem
- Validation
- Related work
- Conclusions & Future work

ILP and Register Reuse

- Loop programs
  - dominate application execution time
  - main sources of ILP and register reuse
- Transformations
  - expose / exploit ILP
  - enable register reuse
- These transformations interact in subtle ways
- ILP - Register Reuse tradeoff?

ILP - Register Reuse Tradeoff

- Optimal combination of transformations
- Quantification of interactions
- A mathematical model
  - to study the interactions
  - to choose the optimal trans. parameters
- TTBOOK: no such model has been studied

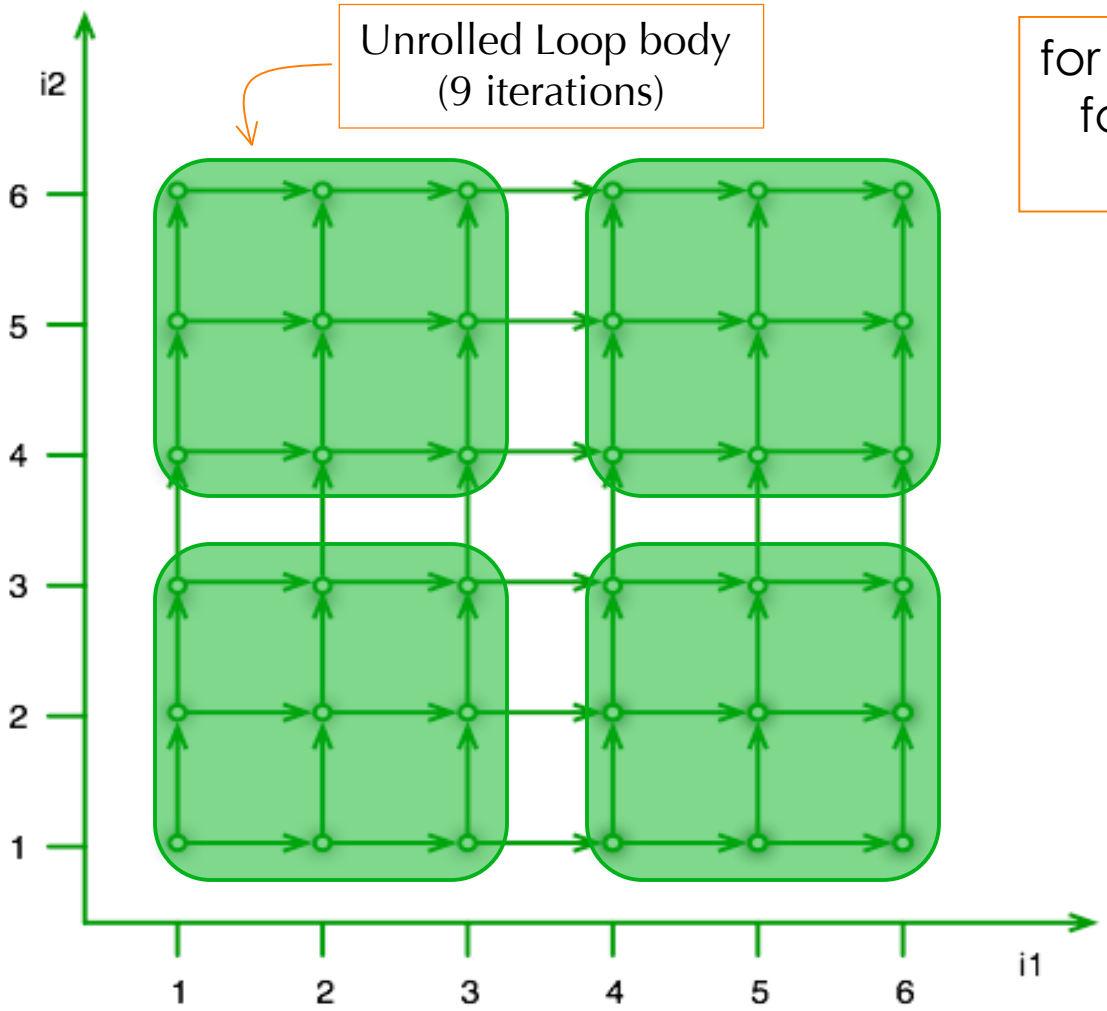
Contributions

- Cost model with trans. params. as variables
  - closed forms: execution time & register pressure
- Convex optimization problem formulation
- A globally optimal solution
- First such formulation & optimal solution

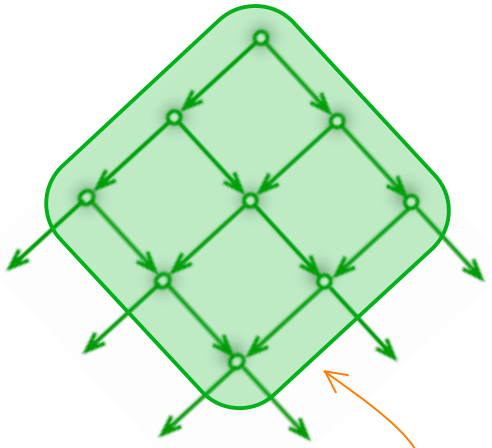
Exposing and Exploiting ILP

- Exposing ILP
  - Unroll and Jam
  - Loop permutation or skewing
  - Multi-dimensional scheduling
- Exploiting ILP
  - DAG schedulers
  - Software pipelining

Exposing ILP with Unroll and Jam



for i_1 = 1 to 6
  for i_2 = 1 to 6
    $A[i_1,i_2] = A[i_1-1,i_2]+A[i_1,i_2-1]$

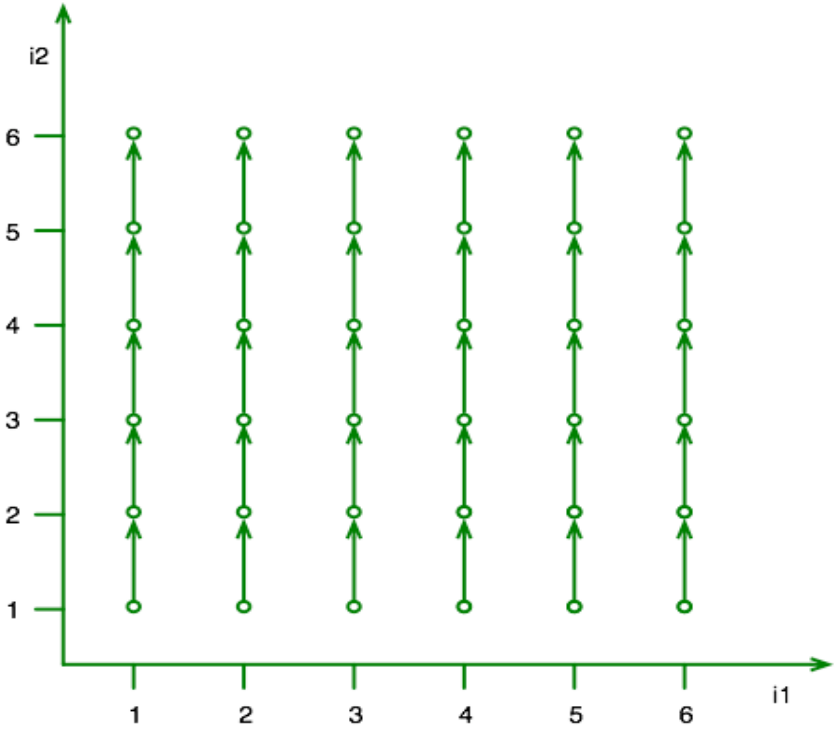


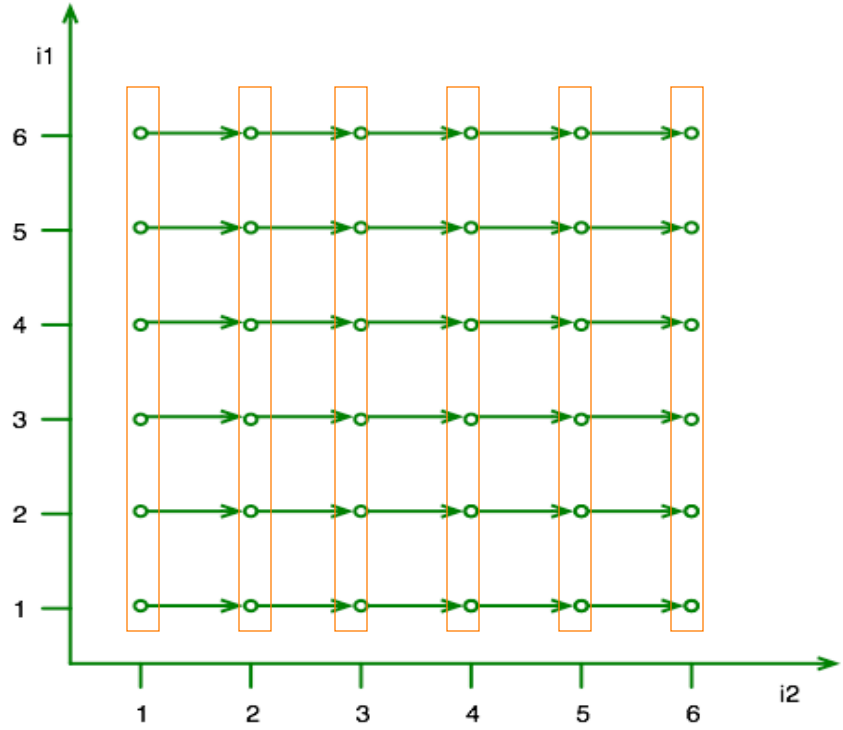
DAG exposes parallelism

Exposing ILP with Permutation

```
for i1 = 1 to 6
  for i2 = 1 to 6
    A[i1,i2] = 3.23 * A[i1,i2-1]
```

```
for i2 = 1 to 6
  for i1 = 1 to 6
    A[i1,i2] = 3.23 * A[i1,i2-1]
```

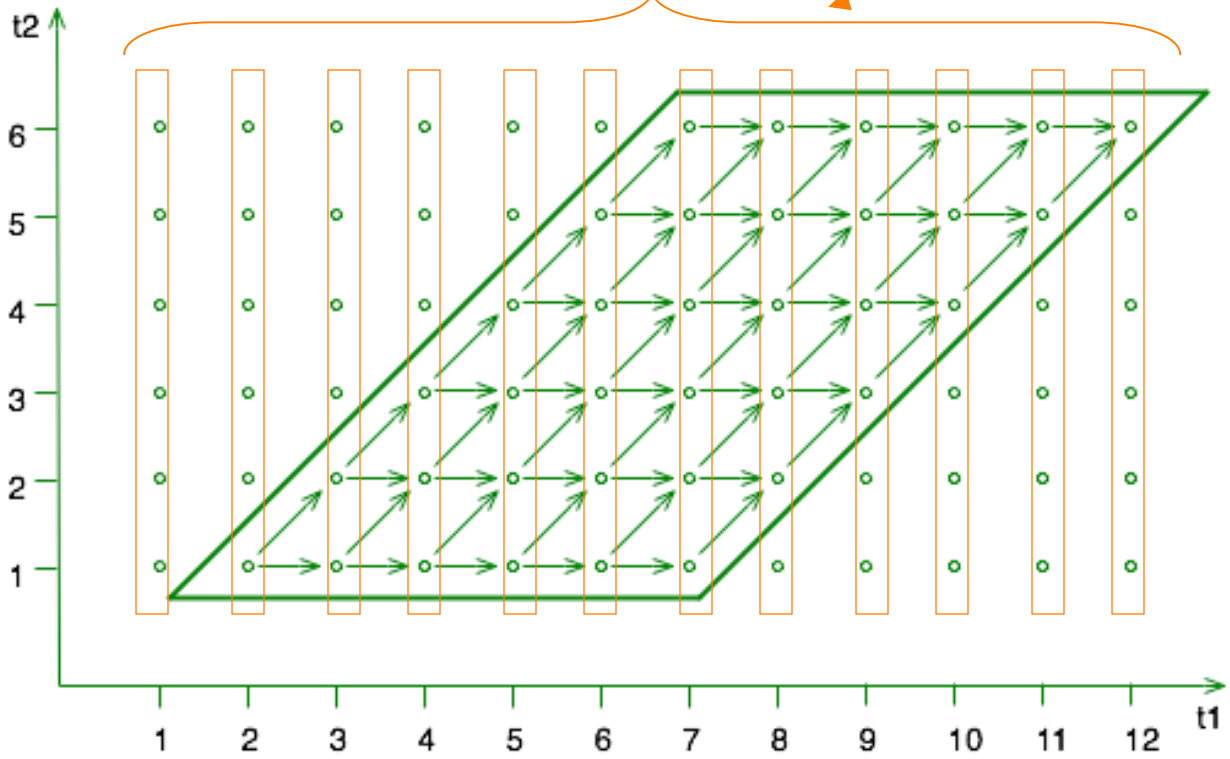




Exposing ILP with Skewing

```
for i1 = 1 to 6
  for i2 = 1 to 6
    A[i1,i2] = A[i1-1,i2]+A[i1,i2-1]
```

All the iterations of inner-most loop are parallel



☺ Sufficient ILP

☺ Performance limited only by the available execution resources

Register Reuse

- Unrol and Jam → Scalar replacement
  - scalar replacement enables register placement
  - classic register allocators are sufficient
- Loop tiling → array register allocation
  - registers allocated to array values
  - no code size increase
  - 😐 requires an array register allocator

Scalar Replacement

```
for i1 = 1 to 6
  for i2 = 1 to 6
    A[i1,i2] = A[i1-1,i2]+A[i1,i2-1]
```

unroll 2 x 2

```
for i1 = 1 to 6 step 2
  for i2 = 1 to 6 step 2
    A[i1,i2]     = A[i1-1,i2]+A[i1,i2-1]
    A[i1+1,i2]   = A[i1,i2]+A[i1+1,i2-1]
    A[i1,i2+1]   = A[i1-1, i2+1]+A[i1,i2]
    A[i1+1,i2+1] = A[i1,i2+1]+A[i1+1,i2]
```

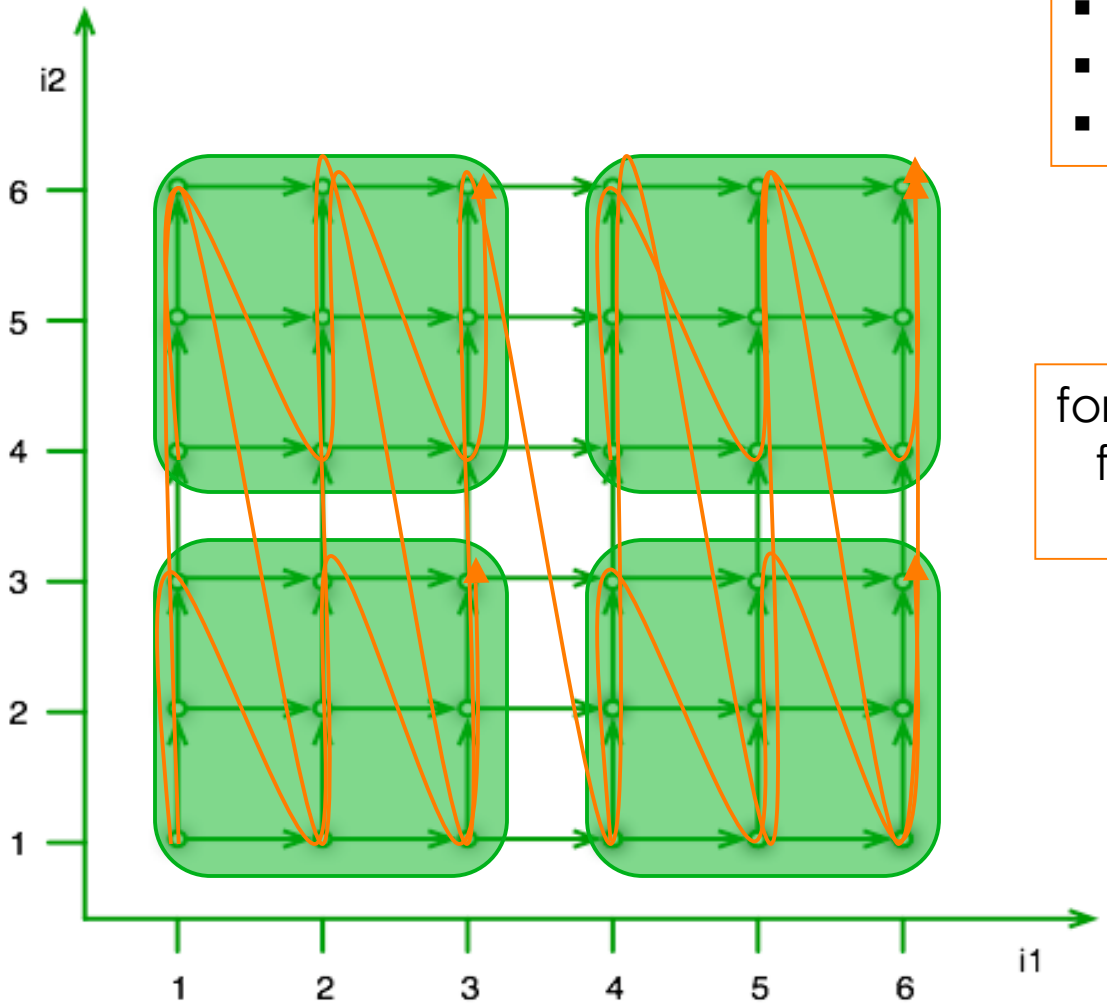
replace $A[i_1,i_2]$ by a scalar T

```
for i1 = 1 to 6 step 2
  T = A[i1,0]
  for i2 = 1 to 6 step 2
    T            = A[i1-1,i2]+ T
    A[i1+1,i2]   = T +A[i1+1,i2-1]
    A[i1,i2+1]   = A[i1-1, i2+1]+T
    A[i1+1,i2+1] = A[i1,i2+1]+A[i1+1,i2]
    A[i1,i2]     = T
```

- Saves 2 loop independent loads plus 1 loop carried load
- T can be allocated to a register

Which array references to scalar replace?

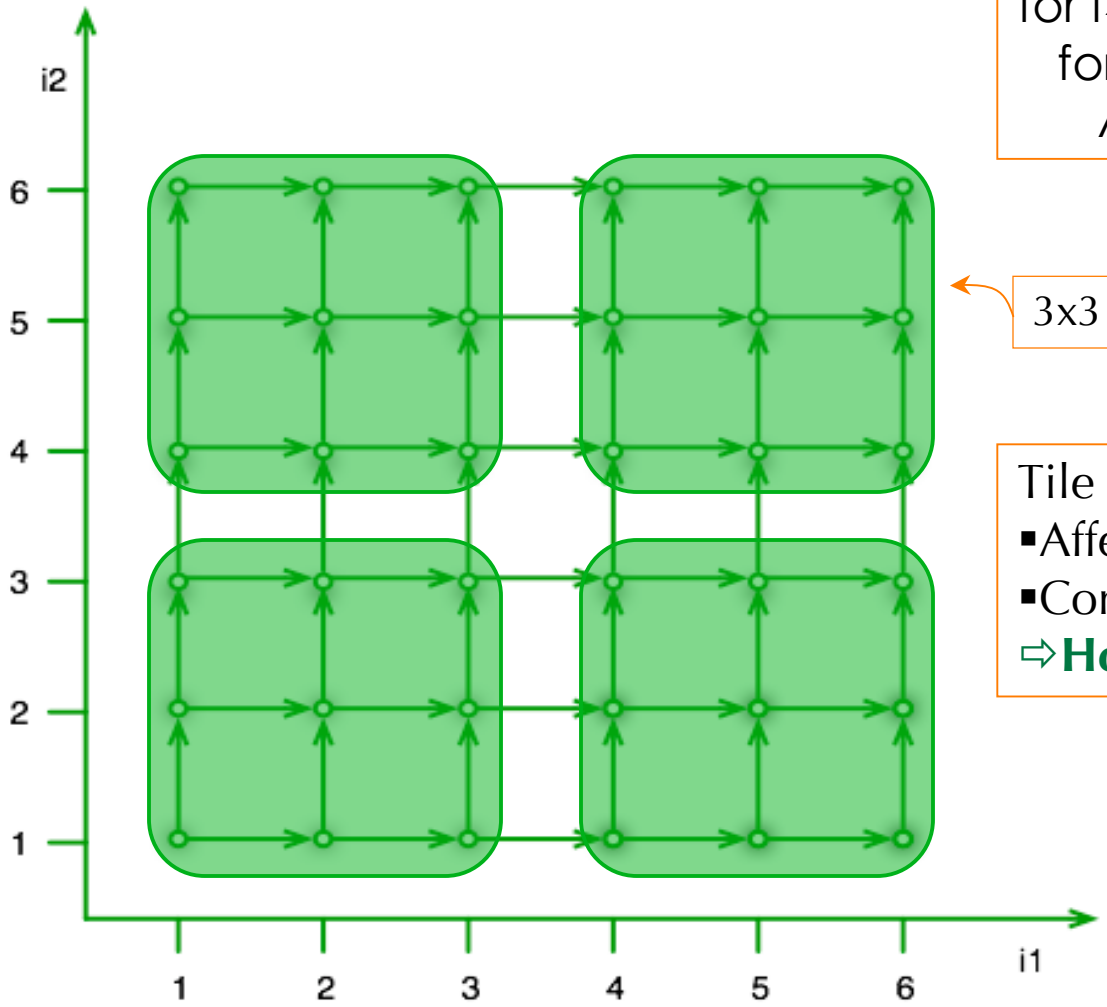
Tiling



Tiling
- Similar to Unroll and Jam
- Decreases life time of values
- Limits MAXLIVE

for i_1 = 1 to 6
  for i_2 = 1 to 6
    $A[i_1,i_2] = A[i_1-1,i_2]+A[i_1,i_2-1]$

Register tiling



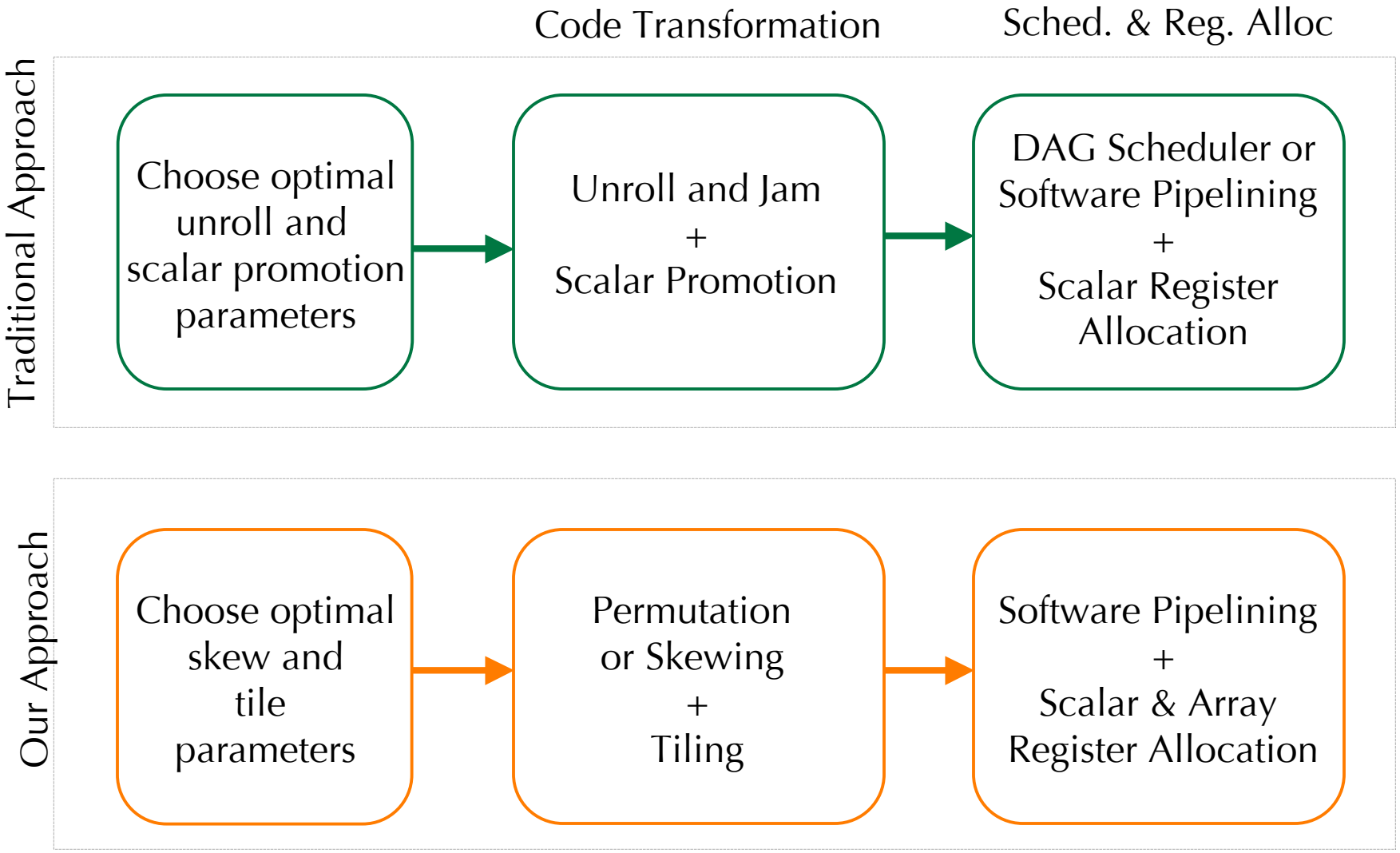

```
for i1 = 1 to 6
  for i2 = 1 to 6
    A[i1,i2] = A[i1-1,i2]+A[i1,i2-1]
```

3x3 register tile

Tile sizes:
- Affects load/store savings
- Constrained by number of registers

⇨**How to choose the tile sizes?**

Traditional vs. Our Approach

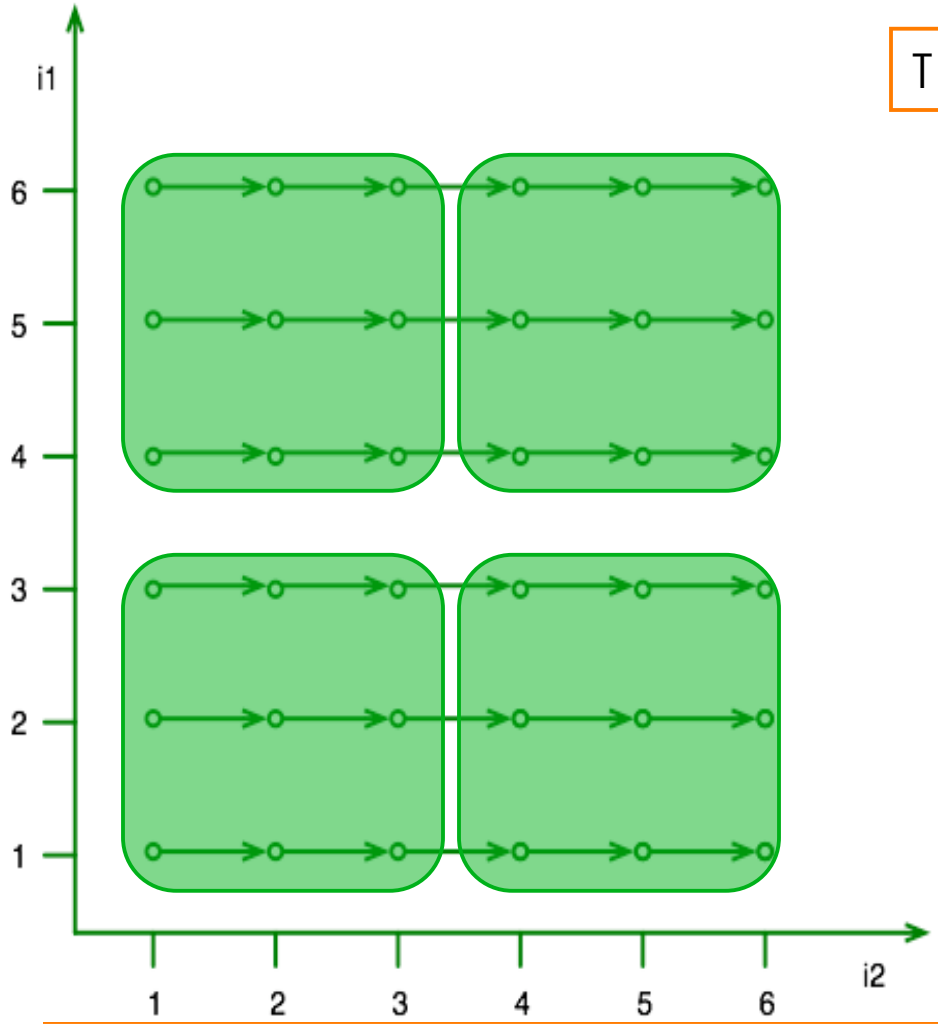


Program, Tiling, and Architecture Class

- Input loops:
  - perfectly nested, rectangular loops
  - uniform dependence bodies
- Rectangular tiling
  - we assume: input loop nest admits rectangular tiling
- ILP-exposed by: permutation or skewing
- Architectures: superscalar or VLIW

Execution Time

(When permutation exposes ILP)



T = (ntiles * tile_cost) + loop_overhead

tile_cost = max(comp_cost,load_store_cost)

comp_cost = α * tile_vol

load_store_cost = β * LS(t,D)

loop_overhead = η * LO(t,N)

ntiles = $N_1/t_1 * \ldots * N_n/t_n$

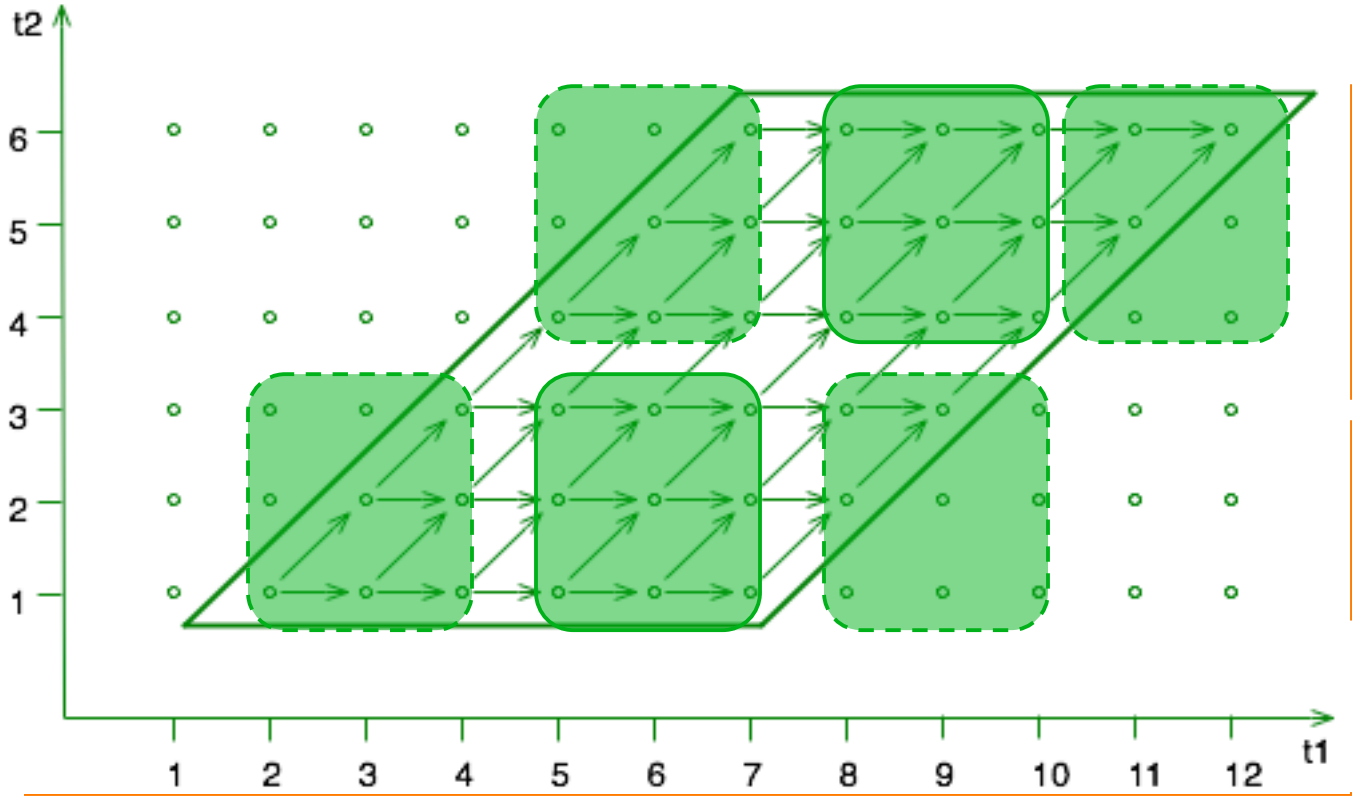
t = vector of tile sizes
N = vector of iter. space sizes
D = dependence matrix

Execution Time Model

(when permutation cannot expose ILP: skew)

Skewing affects

- iteration space shape -- makes counting of partial, full, and no. of tiles hard.
- dependence lengths -- affects the amount of data loaded / stored in a tile.



- Partial tiles treated as full tiles.
- Number of tiles approximated by $N_1/t_1 * \ldots * N_n/t_n$

- Dep. matrix = SD
- LS(t,SD) is the load store volume

Optimal ILP and Register Tiling: Optimization Problem Formulation

minimize TotalExecutionTime(t,S)
subject to LoadStoreVolume(t,S) ≤ Registers

For a fixed skew S

- ✓ **t is the only variable**
- ✓ **opt. prob. reduces to an integer convex opt. prob.**

Solution Steps

Can permutation expose a parallel loop?

- Yes!
- No skewing, only tiling
  - Fix $S=I$ in opt. prob.
- Solve for optimal tile sizes
- Single integer convex opt. problem.

- No!
- Construct set (Γ)of valid skews
- For each element in Γ solve the fixed skew optimization problem
- Pick the best
- Only *d(d-1)* problems

Solving for Optimal Tile Sizes

- Opt. Prob. for tile sizes is a Integer Geometric Program (à la Integer Linear Programs)
- GPs can be transformed into convex opt. probs.
- Standard solvers are available
- Running time:
  - depends on #vars & #constraints
  - few seconds (< 10 secs.)

Validation

- Experimental validation requires
  - array register allocator
  - architectural support (like rotating registers)
- Similar model used for finding optimal unroll factor
  - optimal unroll factors can be found with small tweaks
- In tiling for memory hierarchy
  - we have successfully used a similar model
  - almost all the cost models used by other researchers can be cast into our GP framework [RR-SC04]

Related Work

- Unroll and Jam approach
  - [Callhan et al.-90], [Carr-Kennedy-94], [Sarkar-01]
- Hierarchical tiling
  - [Carter et al.-95], [Mitchell et al.-98]
- Software pipelining of loop nests
  - [Ramanujam-94], [Rong et al. 04], [Rong et al. 05]
- Code generation for register tiling
  - [Jiminez et al.-02], [Sarkar-01]

Conclusions & Future Work

- A mathematical formulation of the combined ILP and register tiling problem.
- A globally optimal solution.
- Future work:
  - adapting modulo schedulers to pipeline skewed loops
  - developing an array register allocator
  - experimental validation on benchmarks