Generalized tiling

Locality optimizations for irregular dependency graphs

Duco van Amstel

Inria, Équipe Corse - Kalray

September 21-23, 2015

Introduction

PhD / Cifre contract between Kalray S.A & Inria with Fabrice Rastello

Academic work within the Inria team Corse

Running April 2013 – Spring 2016 on the subject of

Data Locality and Optimizations for Dataflow Languages on Manycore Architectures

Presentation outlay

Problem statement

Generalized tiling

Solving the tiling problem

Beyond semi-regularity

Resource constraints

Conclusion & Future work

Presentation progress

Problem statement

Generalized tiling

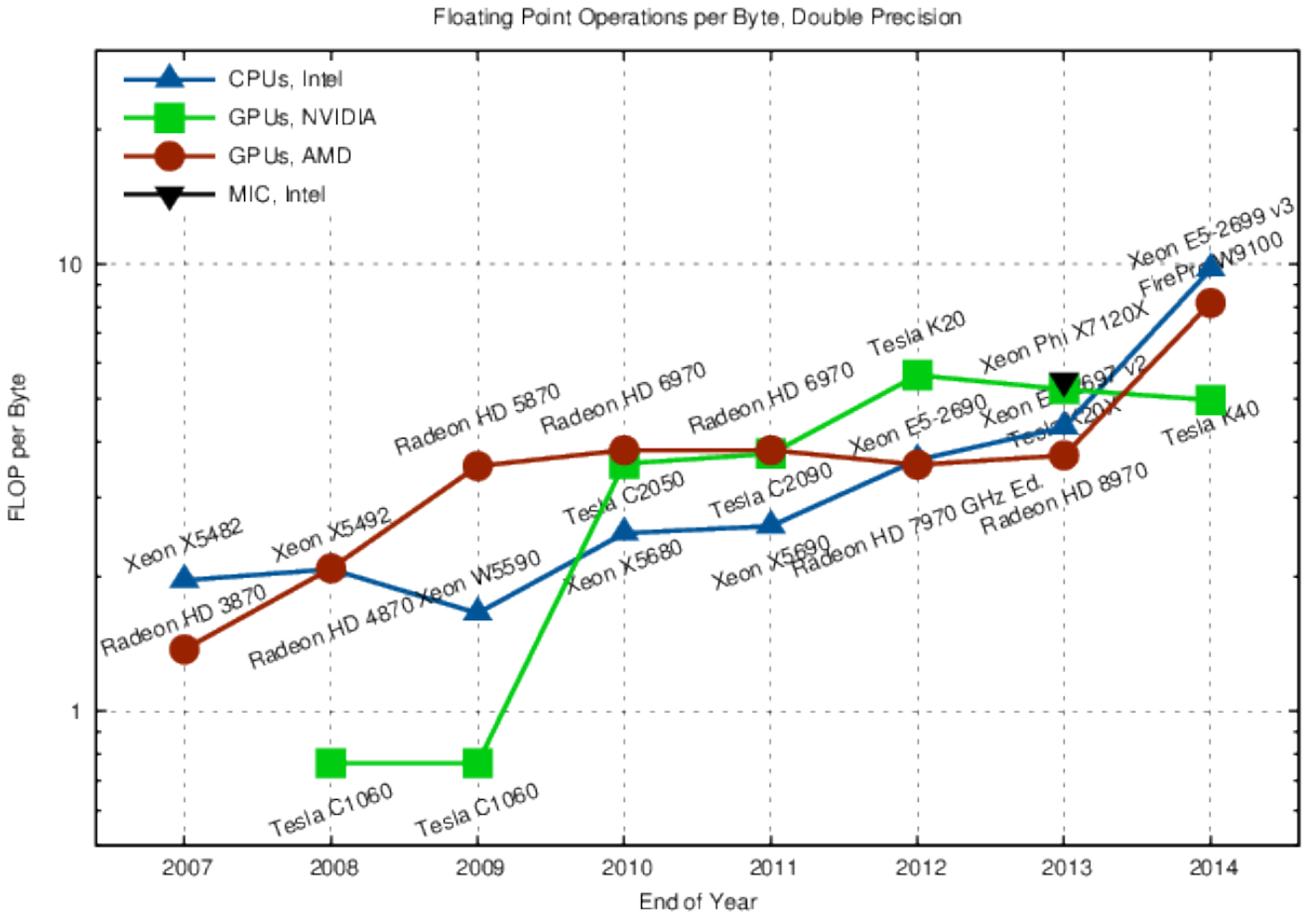
Solving the tiling problem

Beyond semi-regularity

Resource constraints

Conclusion & Future work

The Proverbial Memory Wall



Extracted from www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardward-characteristics-over-time/

Possible directions for mitigation:

- New algorithms with a better inherent computational intensity
- Increase the size of cache-memory and register file
- Increase memory-bandwidth via new interfaces
- Develop alternative architectures
- Improve software IO efficiency

Possible directions for mitigation:

- New algorithms with a better inherent computational intensity
- Increase the size of cache-memory and register file
- Increase memory-bandwidth via new interfaces
- Develop alternative architectures
- Improve software IO efficiency

Idea: Improve data reuse $\rightarrow$ Reduce amount of necessary IO

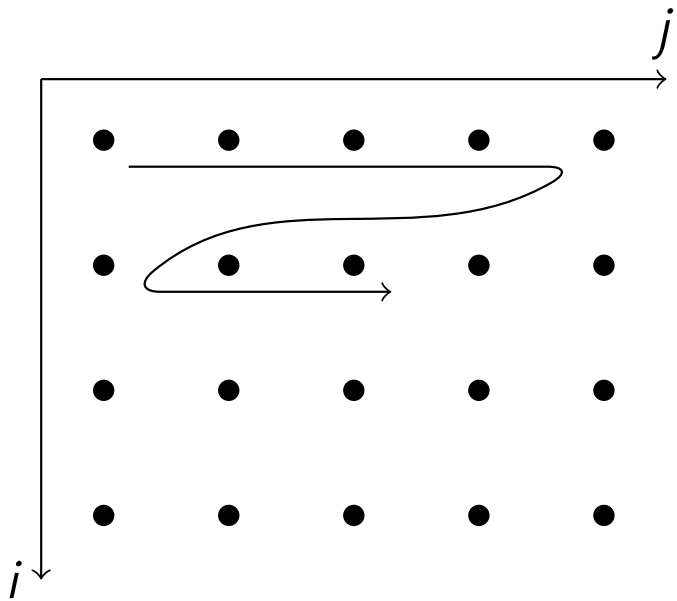
Good Ol' Loop Tiling

Optimizations have biggest effect on frequently executed code
$\rightarrow$ Loop optimizations for inter-iteration data-reuse

Good Ol' Loop Tiling

Optimizations have biggest effect on frequently executed code
→ Loop optimizations for inter-iteration data-reuse

With the canonical loop example:

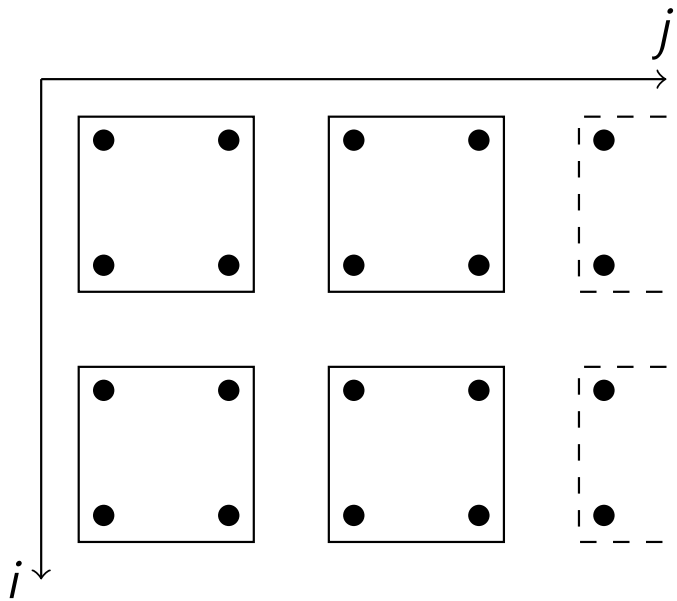



```
for (int i = 0; i < n; i++)
  for (int j = 0; j < m; j++)
    <Loop content>
```

Good Ol' Loop Tiling

Optimizations have biggest effect on frequently executed code
→ Loop optimizations for inter-iteration data-reuse

With the canonical loop example:

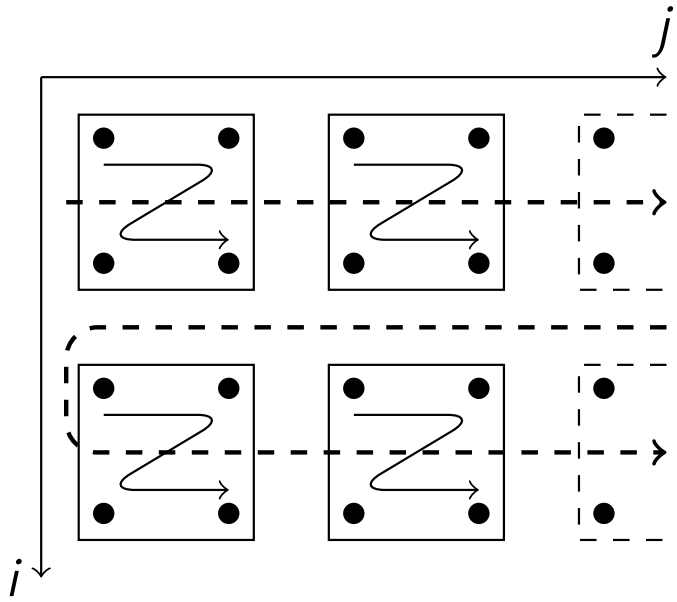



```
for (int i1 = 0; i1 < n/2; i1++)
  for (int j1 = 0; j1 < m/2; j1++)
    for (int i2 = 0; i2 < 2; i2++)
      for (int j2 = 0; j2 < 2; j2++) {
        int i = i1 * 2 + i2;
        int j = j1 * 2 + j2;
        <loop content>
      }
```

Good Ol' Loop Tiling

Optimizations have biggest effect on frequently executed code
→ Loop optimizations for inter-iteration data-reuse

With the canonical loop example:




```
for (int i1 = 0; i1 < n/2; i1++)
  for (int j1 = 0; j1 < m/2; j1++)
    for (int i2 = 0; i2 < 2; i2++)
      for (int j2 = 0; j2 < 2; j2++) {
        int i = i1 * 2 + i2;
        int j = j1 * 2 + j2;
        <loop content>
      }
```

What if there is only a single non-nested loop ?

```
for (int j = 1; j < N - 1; j++) {
  A[j] = (A[j-2] + A[j-1] + A[j] + A[j+1] + A[j+2]) / 5;
  B[j] = B[j] + A[j] * C[j];
  B[j] = B[j] - (B[j-1] - B[j]) * C[j-1];
}
```

What if there is only a single non-nested loop ?

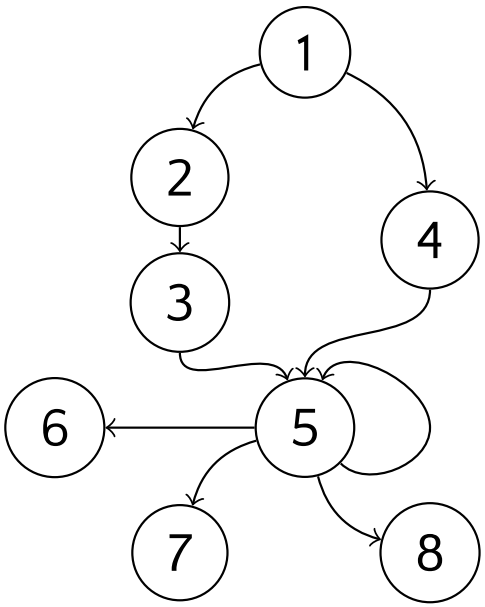
```
for (int j = 1; j < N - 1; j++) {
  A[j] = (A[j-2] + A[j-1] + A[j] + A[j+1] + A[j+2]) / 5;
  B[j] = B[j] + A[j] * C[j];
  B[j] = B[j] - (B[j-1] - B[j]) * C[j-1];
}
```

Only an illustration but:

- many occasions for intra- and inter-iteration reuse
- single iteration will not fit in small number of registers
- scalar promotion will not help
- cache & register tiling can not be applied
- loop fission or fusion may not help for larger real loops

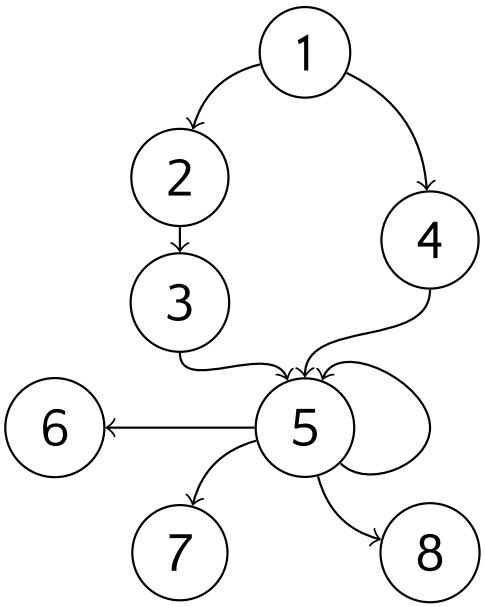
Scaling up

Another form of frequently executed code: a dataflow program



Scaling up

Another form of frequently executed code: a dataflow program

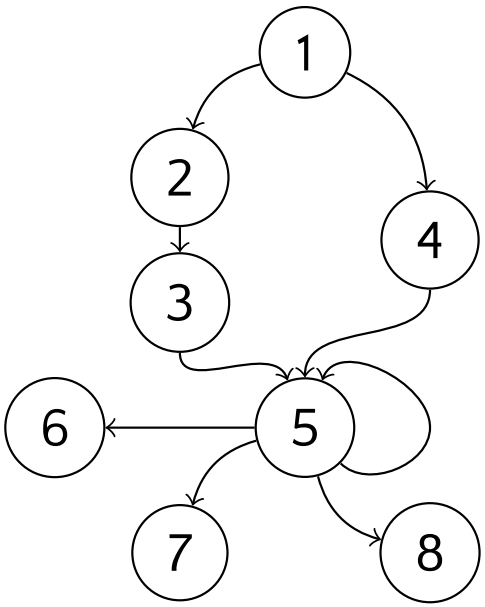


Interesting points:

- Nodes may have an internal state
- Nodes and edges have weights
- Scheduling
- Resource allocation (multi-core)

Scaling up

Another form of frequently executed code: a dataflow program



Interesting points:

- Nodes may have an internal state
- Nodes and edges have weights
- Scheduling
- Resource allocation (multi-core)

Strong ressemblance with the reuses of a single loop iteration

Presentation progress

Problem statement

Generalized tiling

Solving the tiling problem

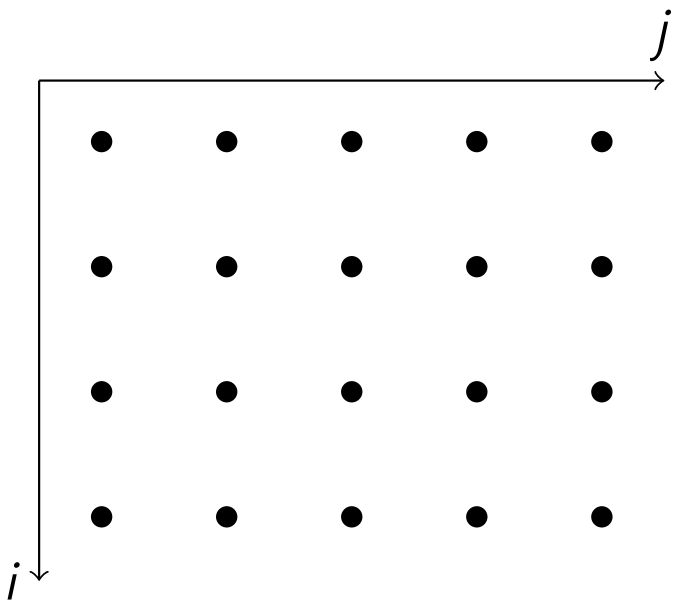
Beyond semi-regularity

Resource constraints

Conclusion & Future work

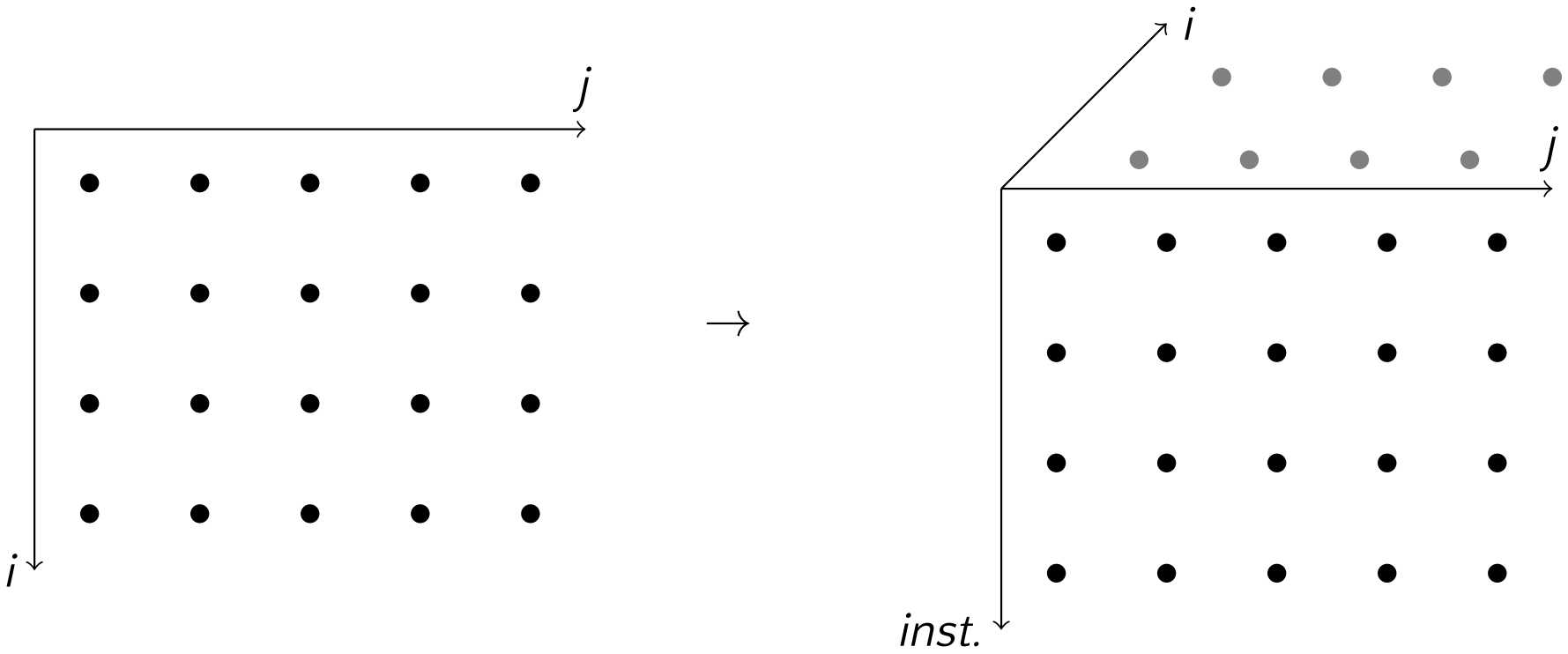
Paving Irregular Dimensions

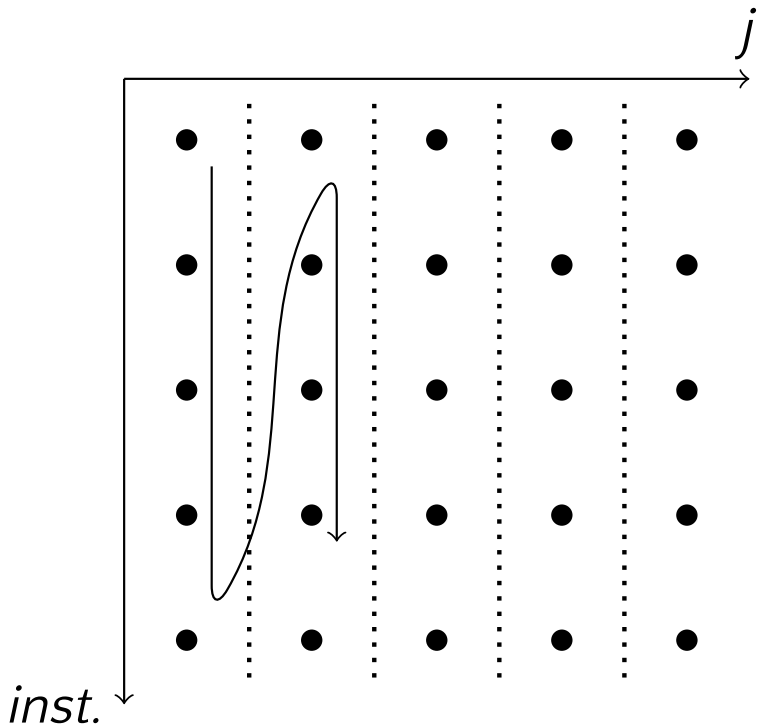
Idea: Consider the instructions of a loop iteration as an extra dimension



Paving Irregular Dimensions

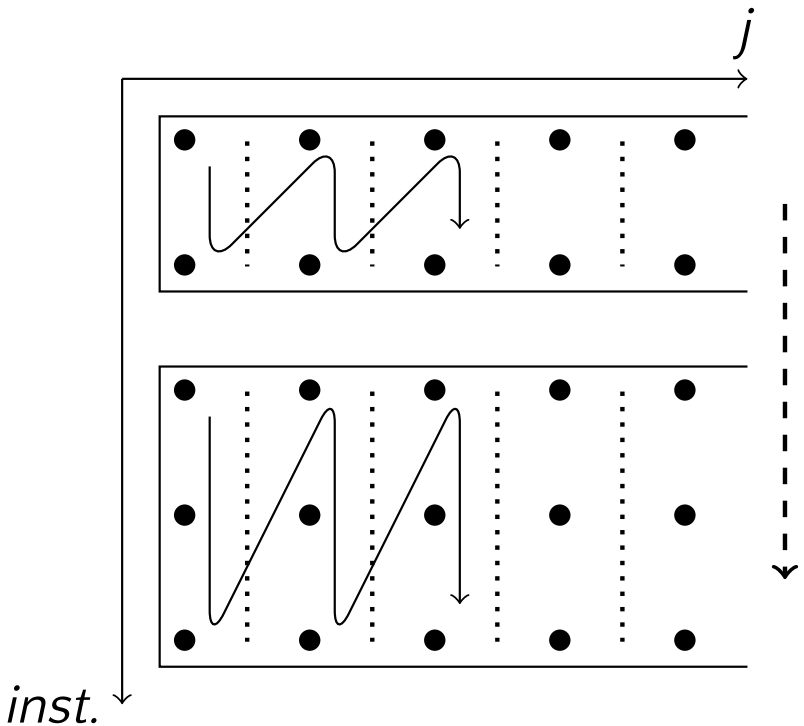
Idea: Consider the instructions of a loop iteration as an extra dimension





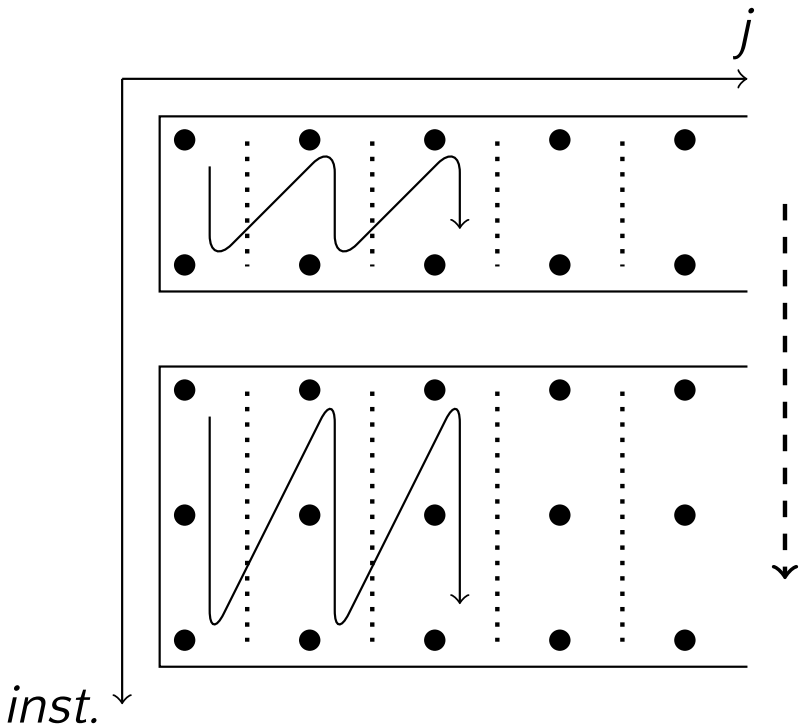
Transformations to be applied:

1. Loop fission



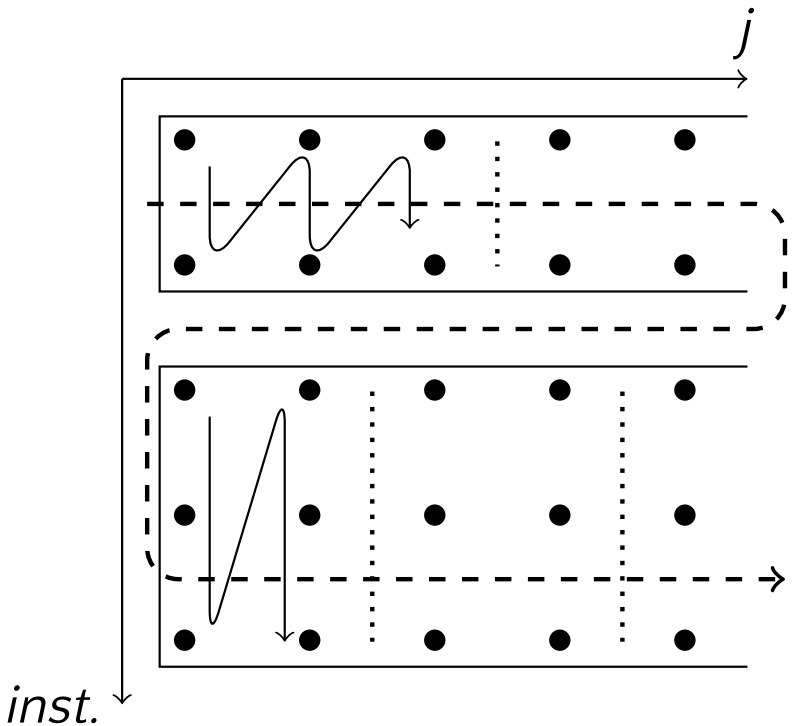
Transformations to be applied:

1. Loop fission



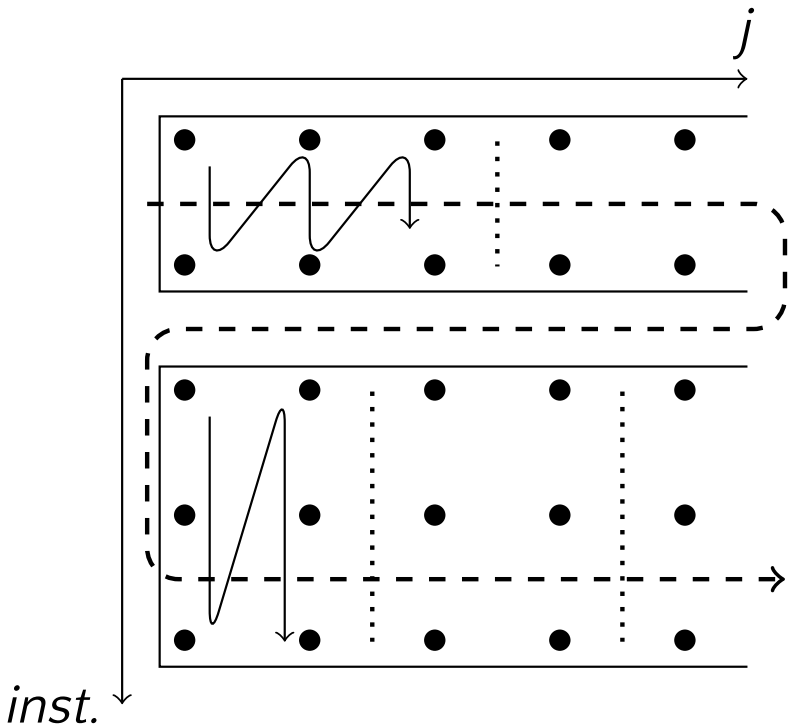
Transformations to be applied:

1. Loop fission
2. Loop unrolling



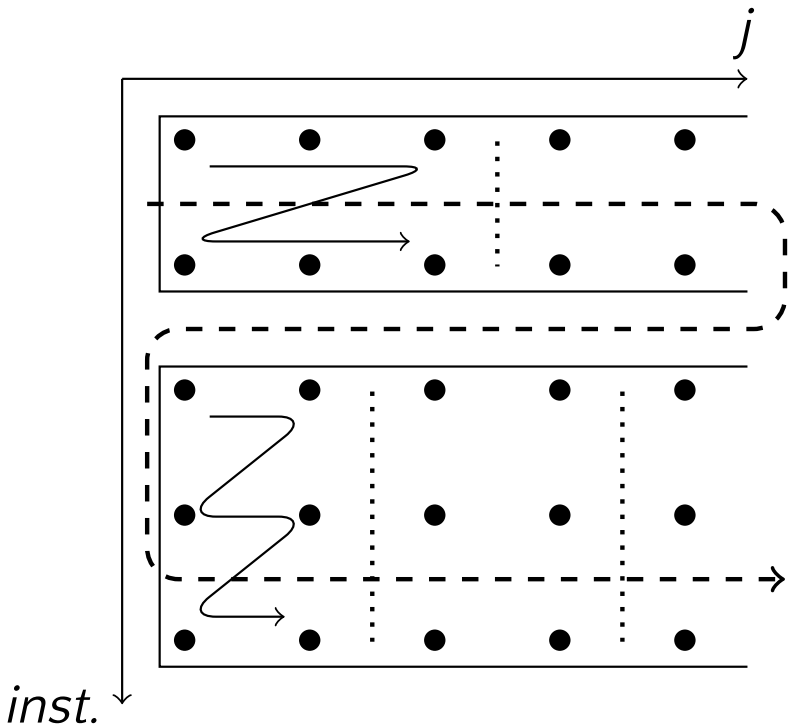
Transformations to be applied:

1. Loop fission
2. Loop unrolling



Transformations to be applied:

1. Loop fission
2. Loop unrolling
3. Rescheduling



Transformations to be applied:

1. Loop fission
2. Loop unrolling
3. Rescheduling

Down the Rabbit Hole

In order to compute the cost of a specific tiling solution we need:

1. The memory requirements of a tile

Memory model

A node in the *reuse graph* has two weights (possibly 0):

- An *Internal Computation* requirement
- An *Internal State* requirement

Edges are intra-iteration reuses, weight expresses the amount of data

Together this defines the *Memory-Use Graph* of our tiling problem

$$\text{Tile memory footprint} = f(\text{scheduling}, \text{tile width})$$

In order to compute the cost of a specific tiling solution we need:

2. The number of IO operations required by the tiling solution

IO model

Loads can be one of two types:

- *Internal state* loads for the graph nodes with an internal state
- *Reuse* loads for data from another tile carried by an edge

Stores are not taken into account as:

$$\#\text{stores} \quad \leq \quad \#\text{data points}$$

```
for (int j = 1; j < N - 1; j++) {
  /* S1 */ A[j] = (A[j-2] + A[j-1] + A[j] + A[j+1] + A[j+2]) / 5;
  /* S2 */ B[j] = B[j] + A[j] * C[j];
  /* S3 */ B[j] = B[j] - (B[j-1] - B[j]) * C[j-1];
}
```

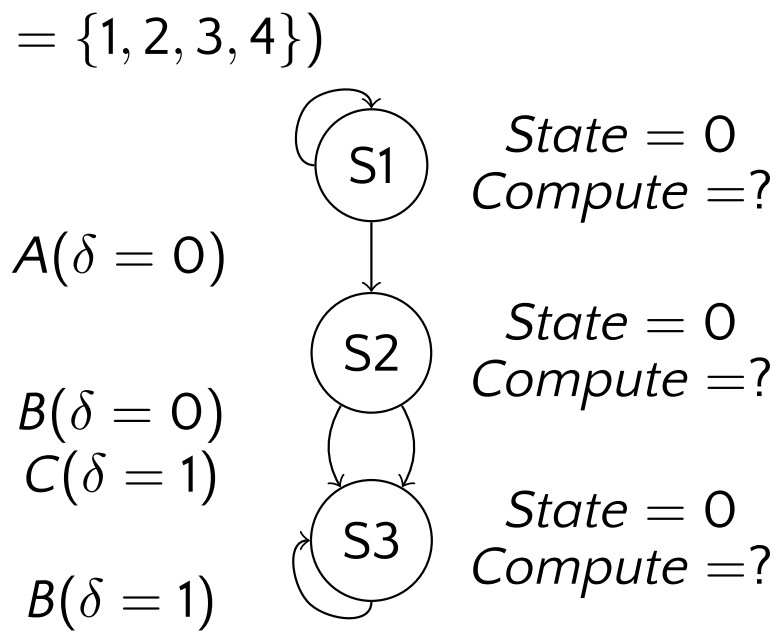
```
for (int j = 1; j < N - 1; j++) {
  /* S1 */ A[j] = (A[j-2] + A[j-1] + A[j] + A[j+1] + A[j+2]) / 5;
  /* S2 */ B[j] = B[j] + A[j] * C[j];
  /* S3 */ B[j] = B[j] - (B[j-1] - B[j]) * C[j-1];
}
```

We observe:

- inter-iteration edges (*i.e* $d > 0$)
- intra-iteration edges (*i.e* $d = 0$)

But we need:

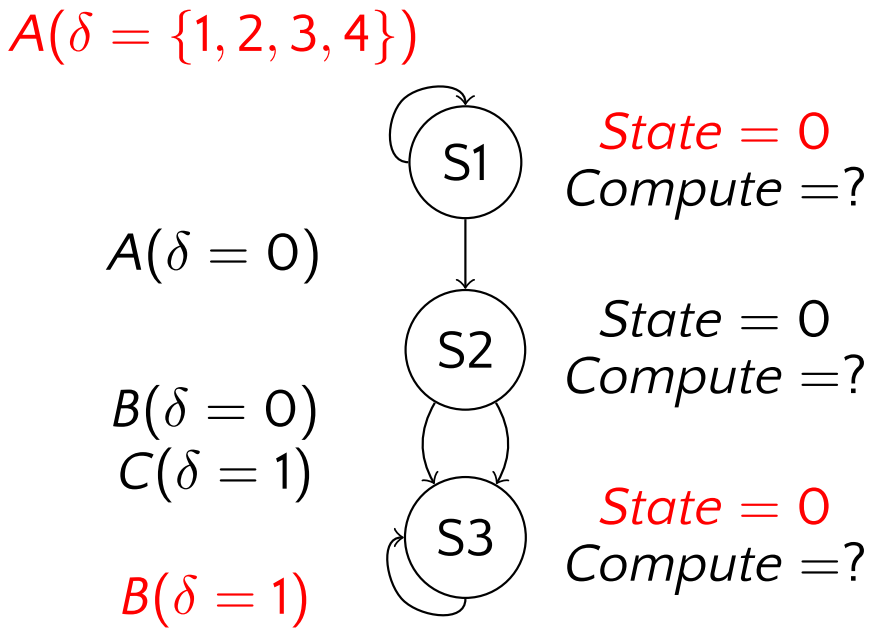
- only intra-iteration edges
- potential non-empty states



```
for (int j = 1; j < N - 1; j++) {
  /* S1 */ A[j] = (A[j-2] + A[j-1] + A[j] + A[j+1] + A[j+2]) / 5;
  /* S2 */ B[j] = B[j] + A[j] * C[j];
  /* S3 */ B[j] = B[j] - (B[j-1] - B[j]) * C[j-1];
}
```

Actions:

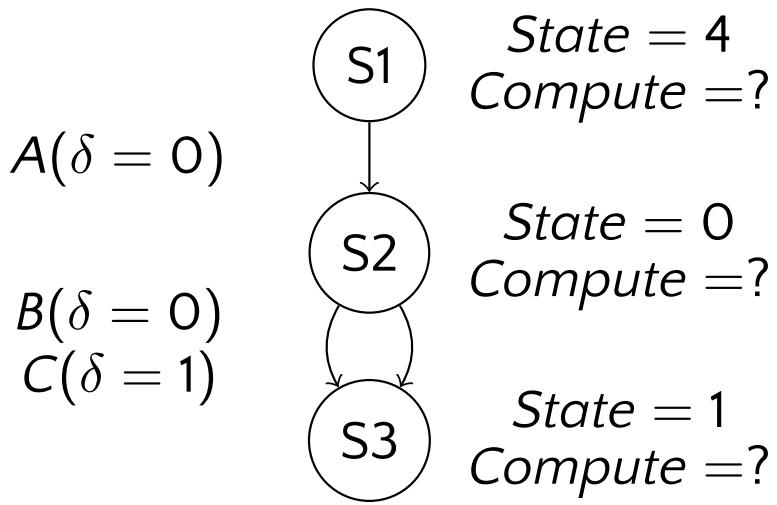
1. Transform inter-iteration self-reuses into an internal state



```
for (int j = 1; j < N - 1; j++) {
  /* S1 */ A[j] = (A[j-2] + A[j-1] + A[j] + A[j+1] + A[j+2]) / 5;
  /* S2 */ B[j] = B[j] + A[j] * C[j];
  /* S3 */ B[j] = B[j] - (B[j-1] - B[j]) * C[j-1];
}
```

Actions:

1. Transform inter-iteration self-reuses into an internal state



```
for (int j = 1; j < N - 1; j++) {
  /* S1 */ A[j] = (A[j-2] + A[j-1] + A[j] + A[j+1] + A[j+2]) / 5;
  /* S2 */ B[j] = B[j] + A[j] * C[j];
  /* S3 */ B[j] = B[j] - (B[j-1] - B[j]) * C[j-1];
}
```

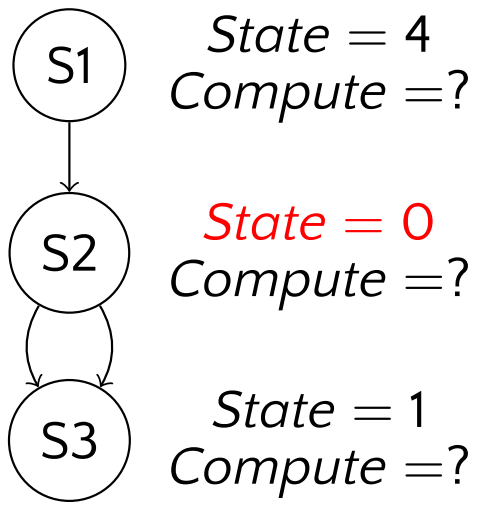
Actions:

1. Transform inter-iteration self-reuses into an internal state
2. Transform inter-iteration reuses into an internal state and an intra-iteration edge

$A(\delta = 0)$

$B(\delta = 0)$

$C(\delta = 1)$



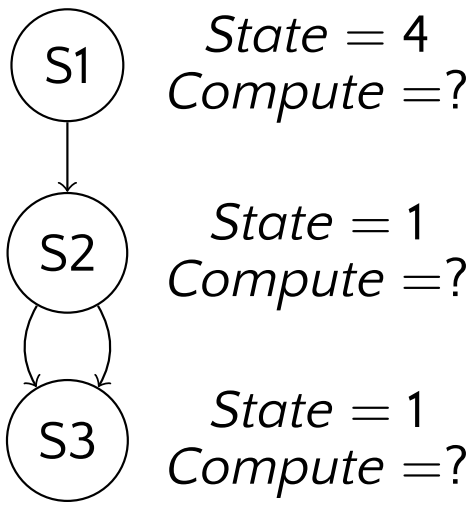
```
for (int j = 1; j < N - 1; j++) {
  /* S1 */ A[j] = (A[j-2] + A[j-1] + A[j] + A[j+1] + A[j+2]) / 5;
  /* S2 */ B[j] = B[j] + A[j] * C[j];
  /* S3 */ B[j] = B[j] - (B[j-1] - B[j]) * C[j-1];
}
```

Actions:

1. Transform inter-iteration self-reuses into an internal state
2. Transform inter-iteration reuses into an internal state and an intra-iteration edge

$A(\delta = 0)$

$B(\delta = 0)$
$C(\delta = 0)$



```
for (int j = 1; j < N - 1; j++) {
  /* S1 */ A[j] = (A[j-2] + A[j-1] + A[j] + A[j+1] + A[j+2]) / 5;
  /* S2 */ B[j] = B[j] + A[j] * C[j];
  /* S3 */ B[j] = B[j] - (B[j-1] - B[j]) * C[j-1];
}
```

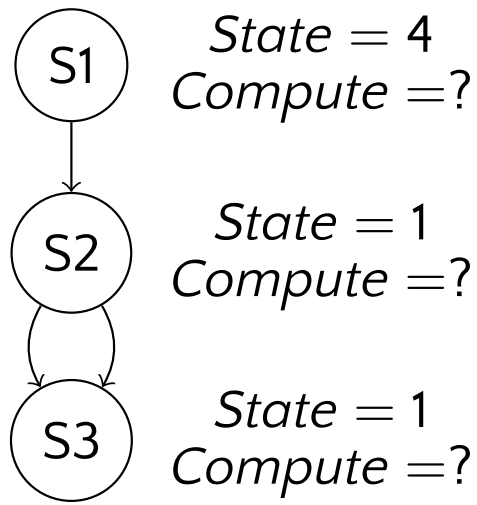
For the last step we retrieve the internal memory requirements for each node.

Base-pointers and iterators are in dedicated registers:

- `A` $\rightarrow$ `R13`
- `B` $\rightarrow$ `R14`
- `C` $\rightarrow$ `R15`
- `j` $\rightarrow$ `RSI`

$A(\delta = 0)$

$B(\delta = 0)$
$C(\delta = 0)$



```
for (int j = 1; j < N - 1; j++) {
  /* S1 */ A[j] = (A[j-2] + A[j-1] + A[j] + A[j+1] + A[j+2]) / 5;
  /* S2 */ B[j] = B[j] + A[j] * C[j];
  /* S3 */ B[j] = B[j] - (B[j-1] - B[j]) * C[j-1];
}
```

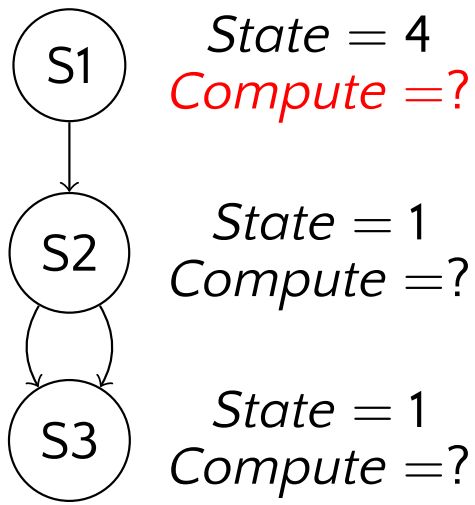
S1:

```
add R10D, R8D ;Retire A[j-2]
mov R8D, R9D ;Shift A memory
mov R9D, R10D ;Shift A memory
mov R10D, R11D ;Shift A memory
mov R11D, [R13 + RSI*4 + 8] ;Load A[j+2]
add R9D, R8D ;Compute on R9
add R9D, R10D ;Compute on R9
add R9D, R11D ;Compute on R9
div R9D, 5 ;Compute on R9
mov [R13 + RSI*4], R9D ;Save B[j]
```

Registers used: 4

$A(\delta = 0)$

$B(\delta = 0)$
$C(\delta = 0)$



```
for (int j = 1; j < N - 1; j++) {
  /* S1 */ A[j] = (A[j-2] + A[j-1] + A[j] + A[j+1] + A[j+2]) / 5;
  /* S2 */ B[j] = B[j] + A[j] * C[j];
  /* S3 */ B[j] = B[j] - (B[j-1] - B[j]) * C[j-1];
}
```

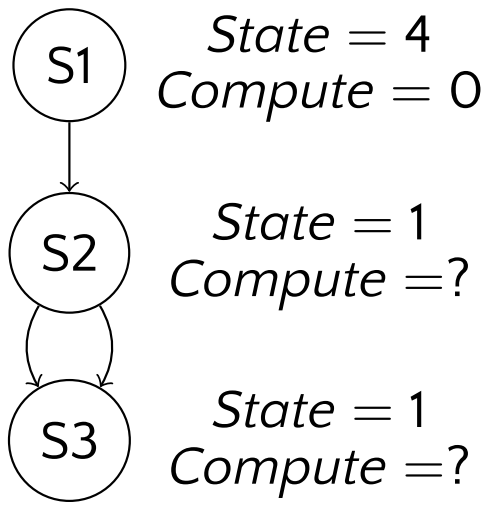
S1:

```
add R10D, R8D ;Retire A[j-2]
mov R8D, R9D ;Shift A memory
mov R9D, R10D ;Shift A memory
mov R10D, R11D ;Shift A memory
mov R11D, [R13 + RSI*4 + 8] ;Load A[j+2]
add R9D, R8D ;Compute on R9
add R9D, R10D ;Compute on R9
add R9D, R11D ;Compute on R9
div R9D, 5 ;Compute on R9
mov [R13 + RSI*4], R9D ;Save B[j]
```

Registers used: 4

$A(\delta = 0)$

$B(\delta = 0)$
$C(\delta = 0)$



```
for (int j = 1; j < N - 1; j++) {
  /* S1 */ A[j] = (A[j-2] + A[j-1] + A[j] + A[j+1] + A[j+2]) / 5;
  /* S2 */ B[j] = B[j] + A[j] * C[j];
  /* S3 */ B[j] = B[j] - (B[j-1] - B[j]) * C[j-1];
}
```

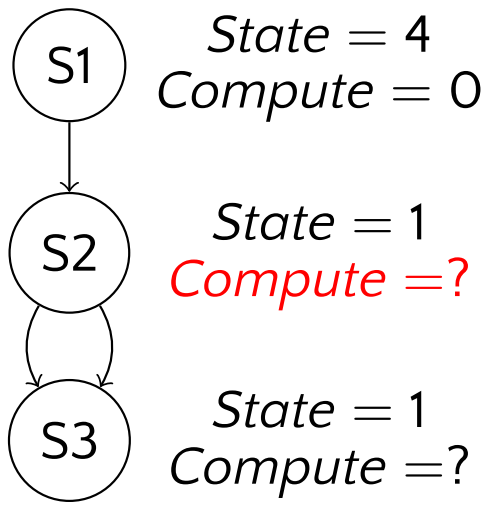
S2:

```
mov [R15 + RSI*4 - 4], R8D ;Save C[j-1]
mov R8D, [R15 + RSI*4] ;Load C[j]
mov EBX, [R14 + RSI*4] ;Load B[j]
mov EAX, [R13 + RSI*4] ;Load A[j]
mul EAX, R8D ;Compute on RA
add EBX, EAX ;Compute on RB
mov [R14 + RSI*4], EBX ;Save B[j]
```

Registers used: 3

$A(\delta = 0)$

$B(\delta = 0)$
$C(\delta = 0)$



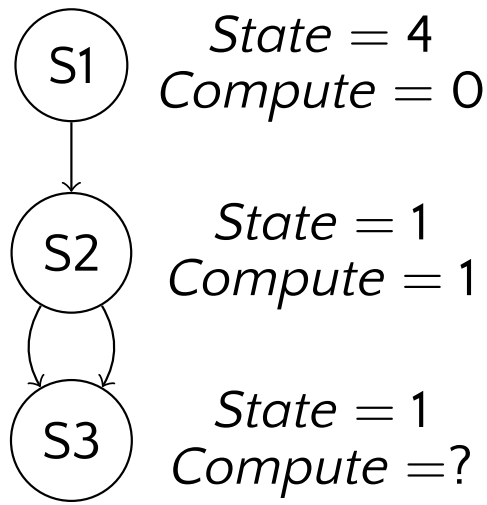
```
for (int j = 1; j < N - 1; j++) {
  /* S1 */ A[j] = (A[j-2] + A[j-1] + A[j] + A[j+1] + A[j+2]) / 5;
  /* S2 */ B[j] = B[j] + A[j] * C[j];
  /* S3 */ B[j] = B[j] - (B[j-1] - B[j]) * C[j-1];
}
```

```
S2:
  mov [R15 + RSI*4 - 4], R8D ;Save C[j-1]
  mov R8D, [R15 + RSI*4] ;Load C[j]
  mov EBX, [R14 + RSI*4] ;Load B[j]
  mov EAX, [R13 + RSI*4] ;Load A[j]
  mul EAX, R8D ;Compute on RA
  add EBX, EAX ;Compute on RB
  mov [R14 + RSI*4], EBX ;Save B[j]
```

Registers used: 3

$A(\delta = 0)$

$B(\delta = 0)$
$C(\delta = 0)$



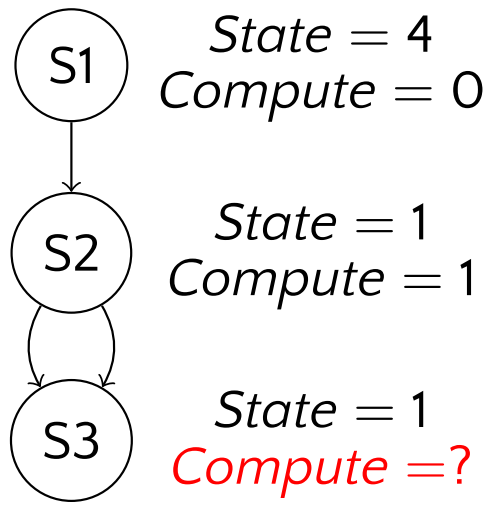
```
for (int j = 1; j < N - 1; j++) {
  /* S1 */ A[j] = (A[j-2] + A[j-1] + A[j] + A[j+1] + A[j+2]) / 5;
  /* S2 */ B[j] = B[j] + A[j] * C[j];
  /* S3 */ B[j] = B[j] - (B[j-1] - B[j]) * C[j-1];
}
```

```
S3:
  mov EBX, R8D ;Retire B[j-1]
  mov EAX, [R15 + RSI*4 - 4] ;Load C[j-1]
  mov R8D, [R14 + RSI*4] ;Load B[j]
  mul EBX, EAX ;Compute on RB
  add EAX, 1 ;Compute on RA
  mul R8D, EAX ;Compute on R8
  sub R8D, EBX ;Compute on R8
  mov [R14 + RSI*4], R8D ;Save B[j]
```

Registers used: 3

$A(\delta = 0)$

$B(\delta = 0)$
$C(\delta = 0)$



```
for (int j = 1; j < N - 1; j++) {
  /* S1 */ A[j] = (A[j-2] + A[j-1] + A[j] + A[j+1] + A[j+2]) / 5;
  /* S2 */ B[j] = B[j] + A[j] * C[j];
  /* S3 */ B[j] = B[j] - (B[j-1] - B[j]) * C[j-1];
}
```

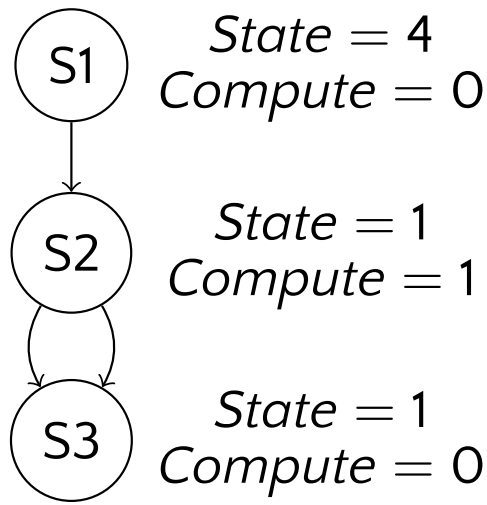
S3:

```
mov EBX, R8D ;Retire B[j-1]
mov EAX, [R15 + RSI*4 - 4] ;Load C[j-1]
mov R8D, [R14 + RSI*4] ;Load B[j]
mul EBX, EAX ;Compute on RB
add EAX, 1 ;Compute on RA
mul R8D, EAX ;Compute on R8
sub R8D, EBX ;Compute on R8
mov [R14 + RSI*4], R8D ;Save B[j]
```

Registers used: 3

$A(\delta = 0)$

$B(\delta = 0)$
$C(\delta = 0)$



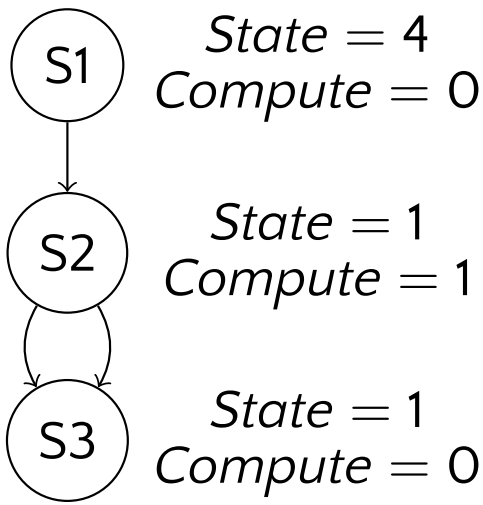
```
for (int j = 1; j < N - 1; j++) {
  /* S1 */ A[j] = (A[j-2] + A[j-1] + A[j] + A[j+1] + A[j+2]) / 5;
  /* S2 */ B[j] = B[j] + A[j] * C[j];
  /* S3 */ B[j] = B[j] - (B[j-1] - B[j]) * C[j-1];
}
```

Finish by:

1. Merge edges with same source & destination nodes
2. Anonymize edges
3. Shorten notations

$A(\delta = 0)$

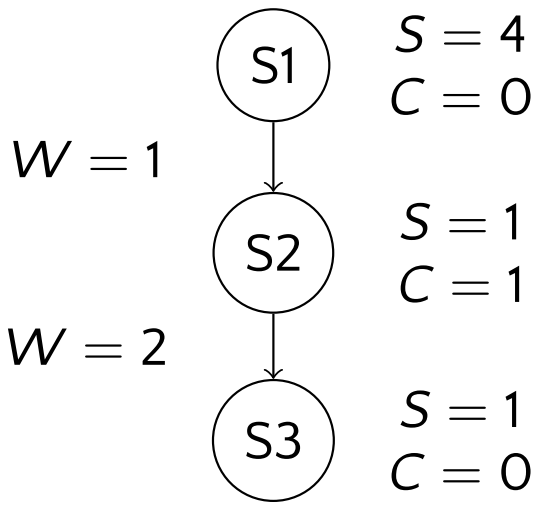
$B(\delta = 0)$
$C(\delta = 0)$



```
for (int j = 1; j < N - 1; j++) {
  /* S1 */ A[j] = (A[j-2] + A[j-1] + A[j] + A[j+1] + A[j+2]) / 5;
  /* S2 */ B[j] = B[j] + A[j] * C[j];
  /* S3 */ B[j] = B[j] - (B[j-1] - B[j]) * C[j-1];
}
```

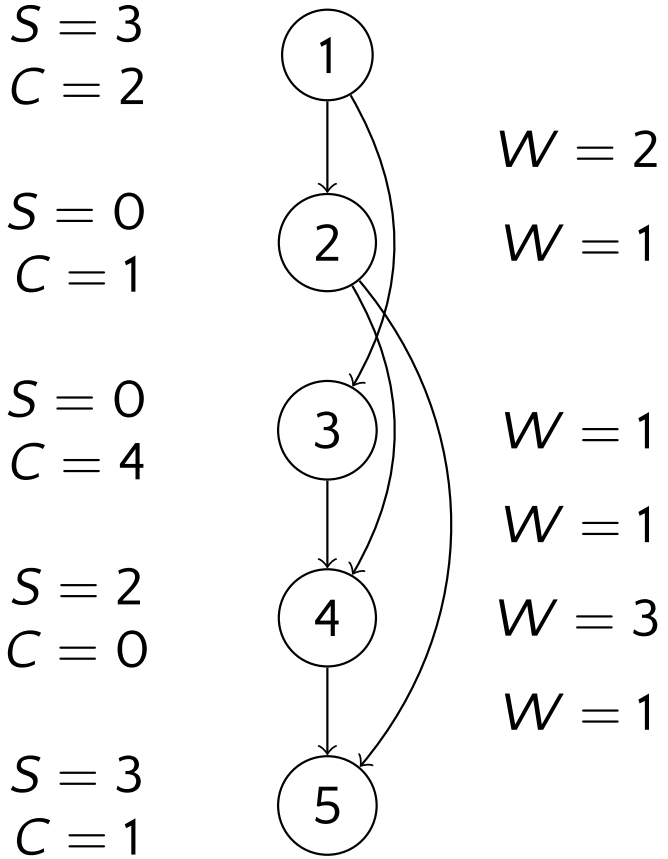
Graph that is exactly the same as a dataflow graph...

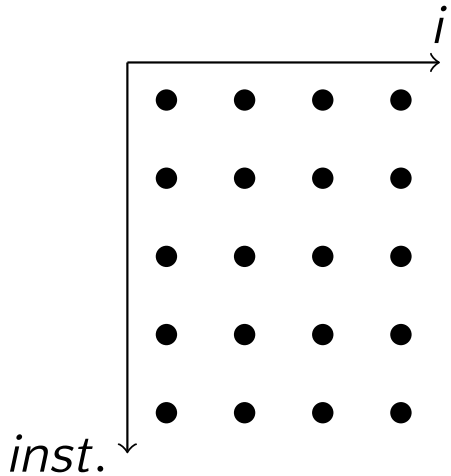
... and corresponds to the definition of a *Memory-Use graph*



There's Always a Price to Pay

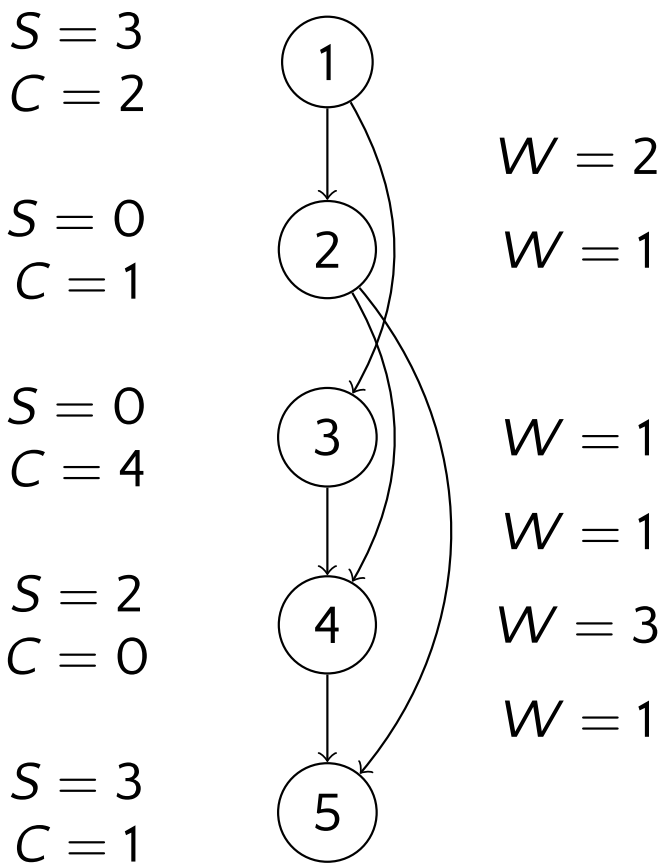
Another *Memory-Use Graph*:

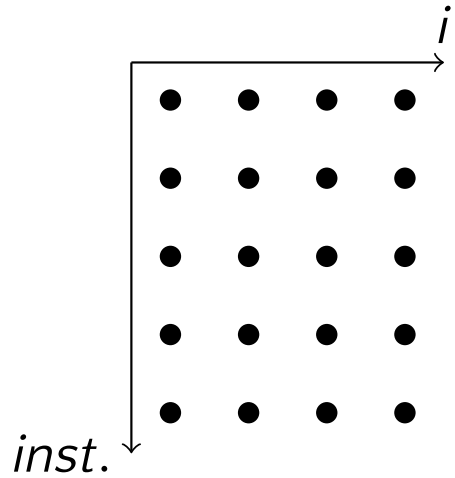




There's Always a Price to Pay

Another *Memory-Use Graph*:



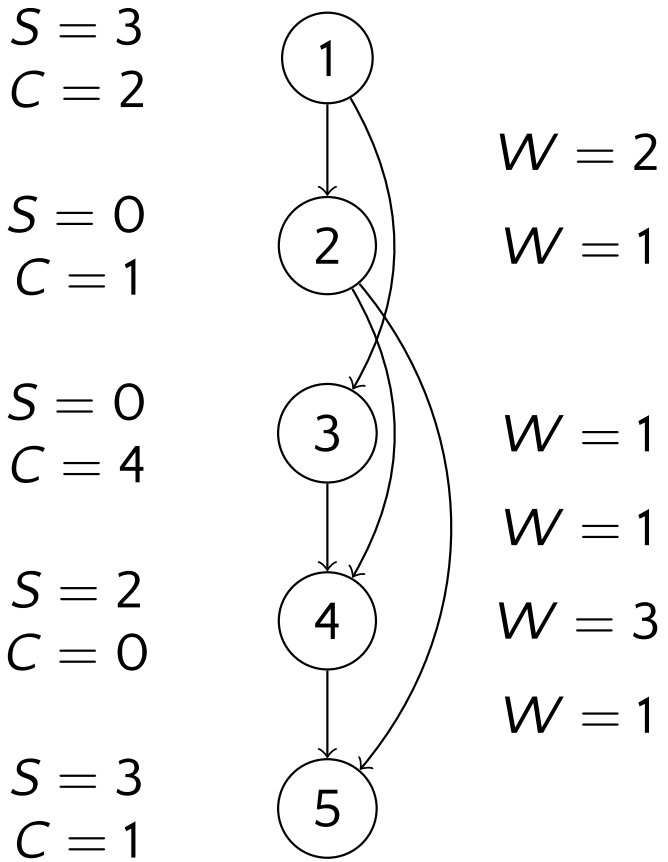


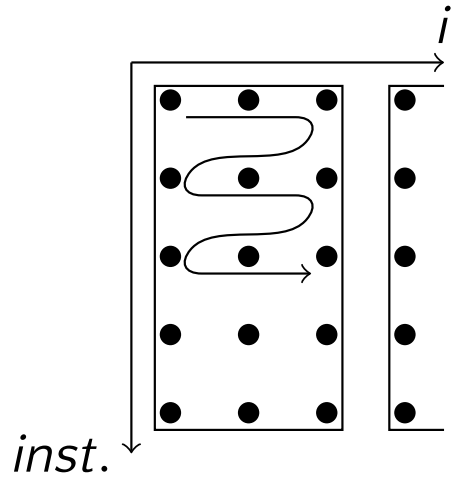
Assume canonical tile scheduling:
horizontal $\rightarrow$ vertical

More on this later...

There's Always a Price to Pay

Another *Memory-Use Graph*:

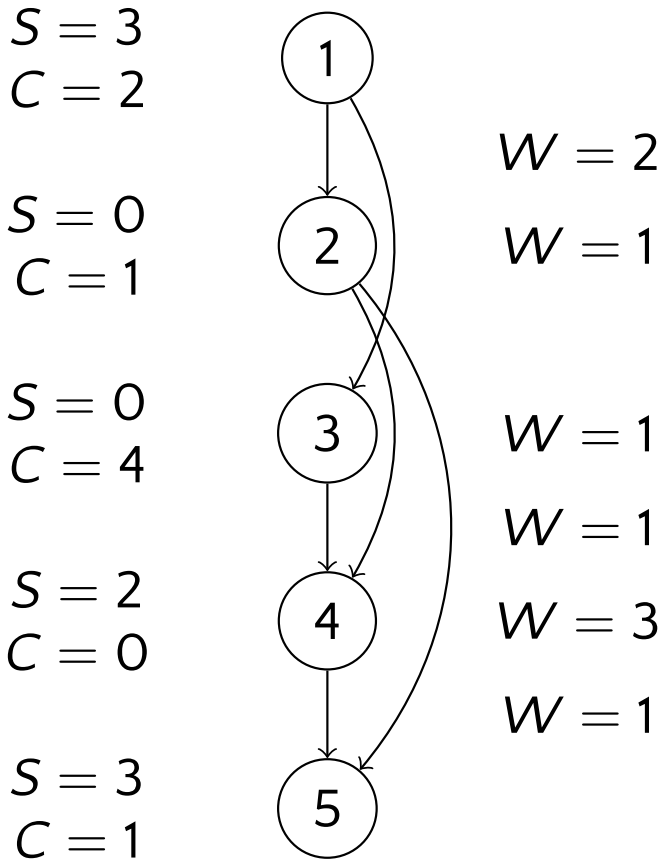


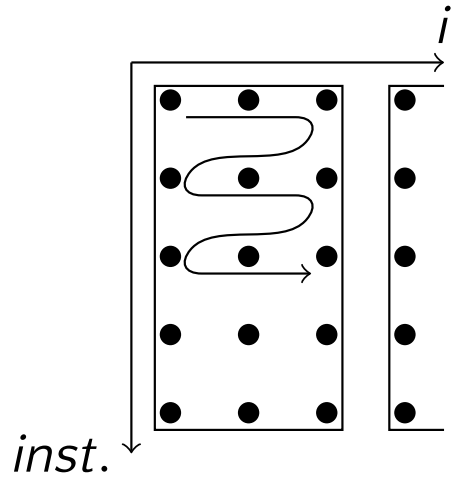


Tiling: $[1; 2; 3; 4; 5]^3$

There's Always a Price to Pay

Another *Memory-Use Graph*:





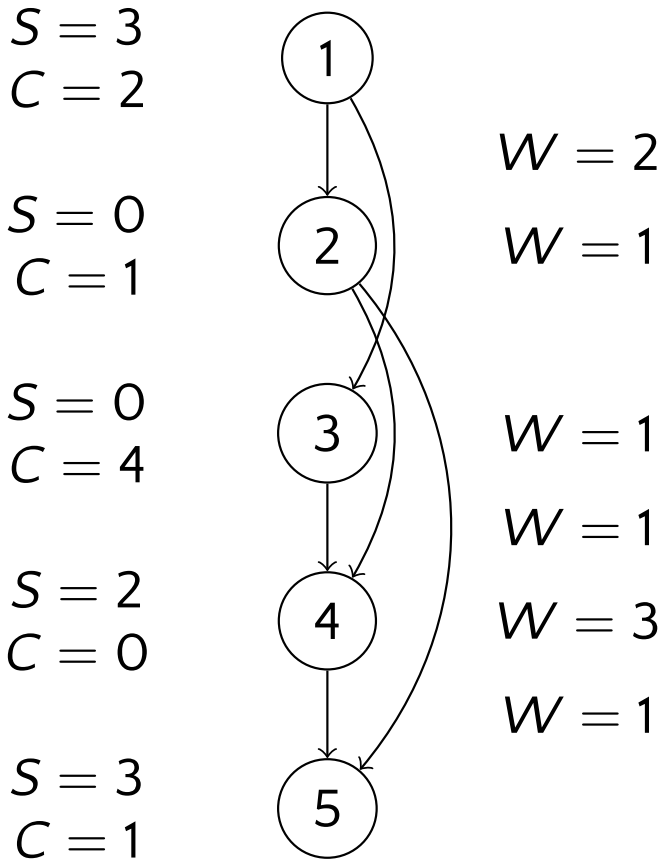
Tiling: $[1; 2; 3; 4; 5]^3$
Memory costs:

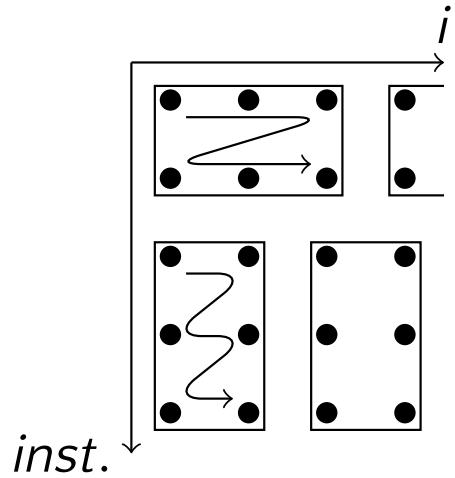
- $[1; 2; 3; 4; 5]^3 \rightarrow 20$

IO-cost: 2, 67 per iteration

There's Always a Price to Pay

Another *Memory-Use Graph*:

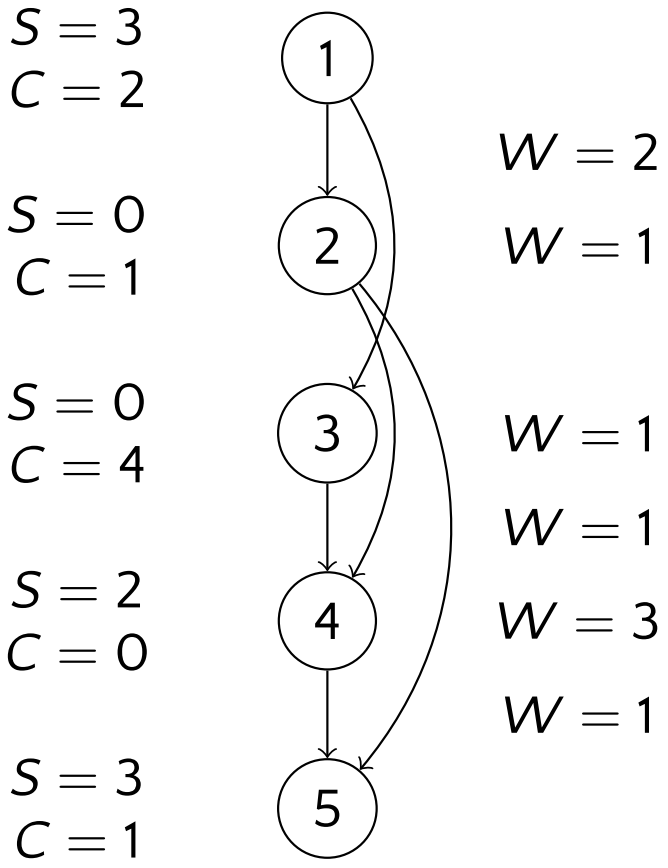


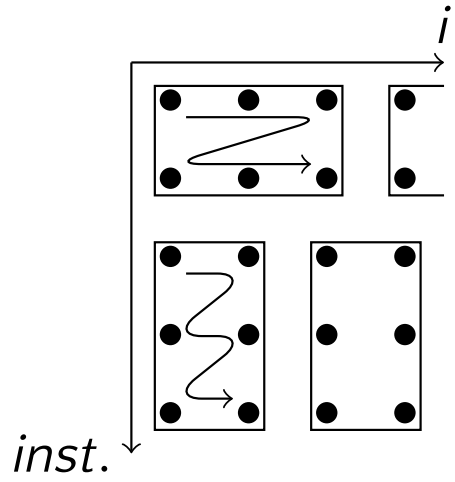


Tiling: $[1;2]^3[3;4;5]^2$

There's Always a Price to Pay

Another *Memory-Use Graph*:





Tiling: $[1;2]^3[3;4;5]^2$

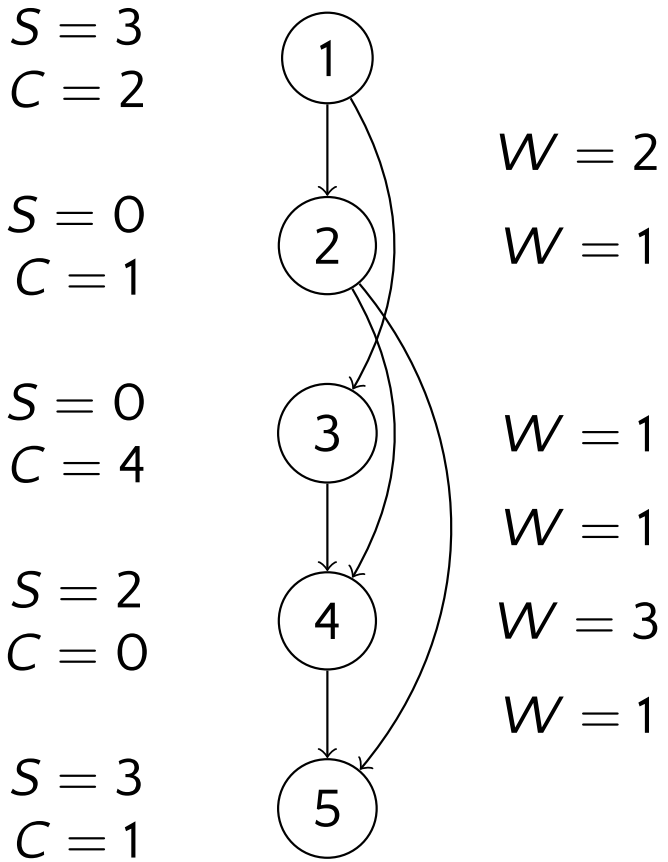
Memory costs:

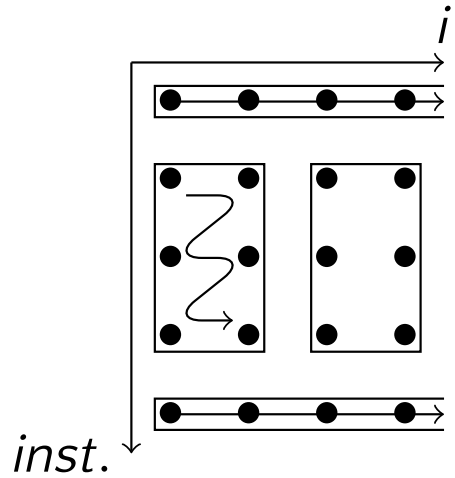
- $[1;2]^3 \rightarrow 8$
- $[3;4;5]^2 \rightarrow 9$

IO-cost: 9, 5 per iteration

There's Always a Price to Pay

Another *Memory-Use Graph*:

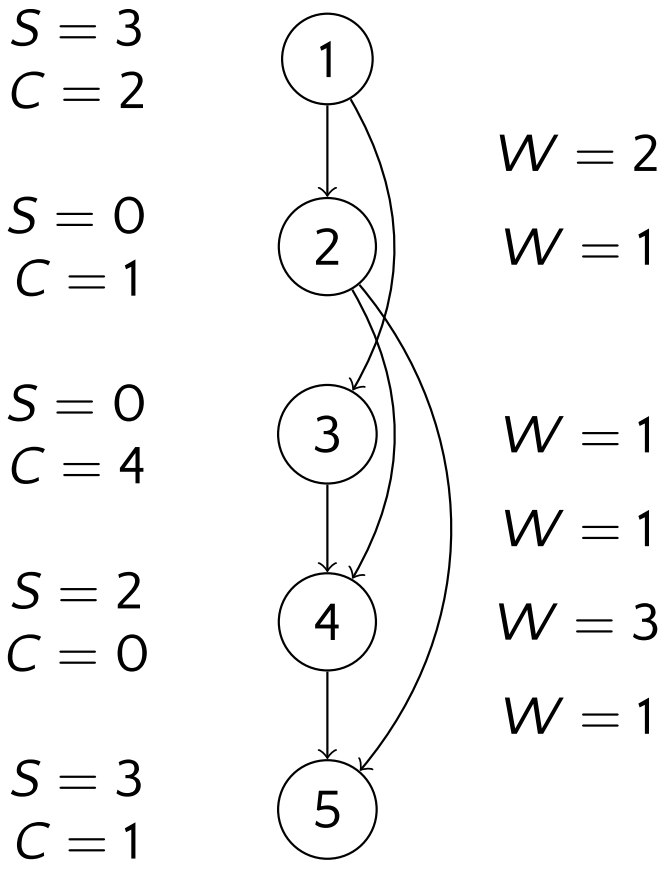


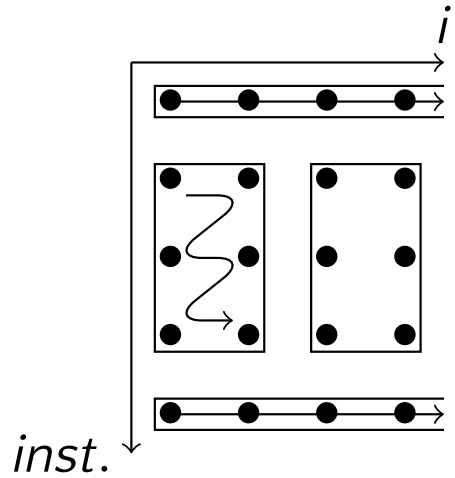


Tiling: $[1]^{\infty}[2;3;4]^{2}[5]^{\infty}$

There's Always a Price to Pay

Another *Memory-Use Graph*:





Tiling: $[1]^{\infty}[2;3;4]^{2}[5]^{\infty}$
Memory costs:

- $[1]^{\infty} \rightarrow 5$
- $[2;3;4]^{2} \rightarrow 8$
- $[5]^{\infty} \rightarrow 8$

IO-cost: 11 per iteration

Presentation progress

Problem statement

Generalized tiling

Solving the tiling problem

Beyond semi-regularity

Resource constraints

Conclusion & Future work

Science Is All About Finding Shortcuts

Regular reuse graphs (*e.g* loop dependencies, cyclo-static dataflow graphs)

$\rightarrow$ Use the graph of a single loop as the full memory-use graph

Problem statement

Entries:

- Regular memory-use graph G
- All legal schedulings S
- Target memory size C
- All possible tilings T

Find $(s_{\text{opt}}, t_{\text{opt}}) \in (S, T)$ such that:

1. $\text{MEM}(s_{\text{opt}}, t_{\text{opt}}) \leq C$
2. $\forall (s, t) \in (S, T) \quad \text{MEM}(s, t) \leq C \implies \text{IO}(s, t) \geq \text{IO}(s_{\text{opt}}, t_{\text{opt}})$

A solution can be found using Constraint Programming (CP)

- Gives an optimal solution
- Can use a significant amount of time
  (*e.g* 30s for small ($\sim$ 50 nodes) graph)

A solution can be found using Constraint Programming (CP)

- Gives an optimal solution
- Can use a significant amount of time
  (*e.g* 30s for small ($\sim$ 50 nodes) graph)

Some ideas for scheduling heuristics?

- Choose an arbitrary scheduling
- Perform code motion in order to ”shorten” heavy edges
- ...

A solution can be found using Constraint Programming (CP)

- Gives an optimal solution
- Can use a significant amount of time
  (*e.g* 30s for small ($\sim$ 50 nodes) graph)

Some ideas for scheduling heuristics?

- Choose an arbitrary scheduling
- Perform code motion in order to "shorten" heavy edges
- ...

And for tiling?

- Greedy tiling: repeatedly choose most IO-efficient tile
- Try to include heaviest edges in a single tile
- ...

Heuristics perform reasonably well for dataflow programs (StreamIt)

$\rightarrow$ StreamIt has a very strong bias towards internal-state

Heuristics perform reasonably well for dataflow programs (StreamIt)
$\rightarrow$ StreamIt has a very strong bias towards internal-state

In the general case (*e.g* loops) the origin of spill is of crucial importance:

- Intra-iteration reuses $\rightarrow$ prefer "tall" tiles with "vertical" scheduling
- Internal-state data $\rightarrow$ prefer "wide" tiles with "horizontal" scheduling

Heuristics perform reasonably well for dataflow programs (StreamIt)
$\rightarrow$ StreamIt has a very strong bias towards internal-state

In the general case (*e.g* loops) the origin of spill is of crucial importance:

- Intra-iteration reuses $\rightarrow$ prefer "tall" tiles with "vertical" scheduling
- Internal-state data $\rightarrow$ prefer "wide" tiles with "horizontal" scheduling

Our global optimization problem becomes a local one:
$\rightarrow$ Locally choose scheduling orientation before forming tiles

But this is work-in-progress...

If You Know It Works It's Not an Experiment

First attempt:

$$\texttt{GCC} \rightarrow \texttt{Tirex} \rightarrow \texttt{Optimizer} \rightarrow \texttt{Assembly}$$

Failing point: Exporting dependencies from `GCC` in the back-end

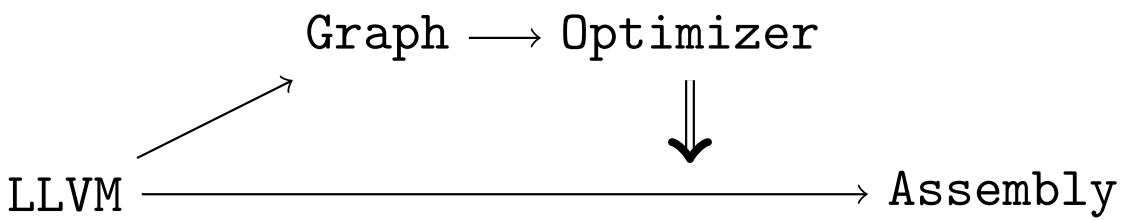
If You Know It Works It's Not an Experiment

First attempt:

$$\texttt{GCC} \rightarrow \texttt{Tirex} \rightarrow \texttt{Optimizer} \rightarrow \texttt{Assembly}$$

Failing point: Exporting dependencies from `GCC` in the back-end

Second attempt:



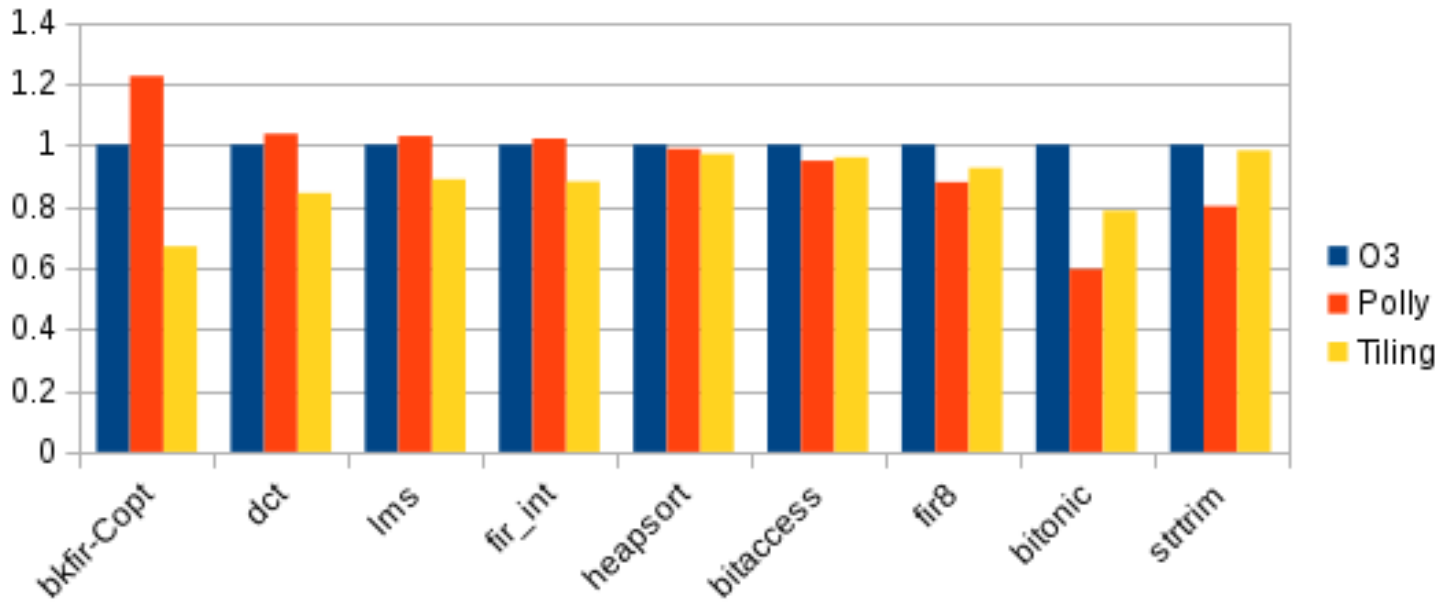
Current implementation for register tiling in loops:

- LLVM 3.6
- Custom preparation pass (uses part of Polly toolbox)
- Custom unroll pass
- Custom memory-use graph generation pass
- Custom rescheduling pass
- Constraint-Programming solver
- Heuristical solver

Some experimental results for register tiling on the Kalray MPPA:

Bench	FFT	FIR	Vecprod
LLVM -O3	32644	9325	506
Tiling	36031 ($\times$ **0.9**)	5830 ($\times$**1.6**)	370 ($\times$**1.37**)

And on x86:



Work-in-Progress for dataflow cache tiling:

- SigmaC toolchain
- Memory-Use Graph extractor (SigmaC IR)
- Heuristical solver for SigmaC & StreamIt
- SigmaC IR rescheduling & allocation

Presentation progress

Problem statement

Generalized tiling

Solving the tiling problem

Beyond semi-regularity

Resource constraints

Conclusion & Future work

Mille viae ducunt Romam

The encountered *memory-use graphs* all exhibited regularity:

$\rightarrow$ They are identically replacted over their iteration domain

Mille viae ducunt Romam

The encountered *memory-use graphs* all exhibited regularity:
→ They are identically replacted over their iteration domain

Two ways towards data-reuse optimization:

1. Static analysis, profiling and compile-time optimization
2. Trace feedback optimization (*e.g* DDG tool)

Mille viae ducunt Romam

The encountered *memory-use graphs* all exhibited regularity:
$\rightarrow$ They are identically replacted over their iteration domain

Two ways towards data-reuse optimization:

1. Static analysis, profiling and compile-time optimization
2. Trace feedback optimization (*e.g* DDG tool)

Execution traces with loads & stores or even cache-hits & misses:

- Do not exhibit regularity
- Do not have explicit iteration domains

Tiling the traced code is equivalent to partioning the trace graph

Entry: Directed acyclic graph of data dependencies and reuses
Goal: Create a tool for performance debugging

Entry: Directed acyclic graph of data dependencies and reuses
Goal: Create a tool for performance debugging

Constraints:

1. Partitioning must be convex (*i.e* graph of partitions is acyclic)
   $\rightarrow$ Implicit assumption in the loop / dataflow case
2. Communication costs between partitions should be minimized
   $\rightarrow$ Objective function for the partitioner

Constraint for automatic optimization as tracepoints relate to code:

3. Partitions must be regular *w.r.t* nodes related to identical code

Splitting Hairs

Partitioning is a well-researched topic. Convex partitioning is not.

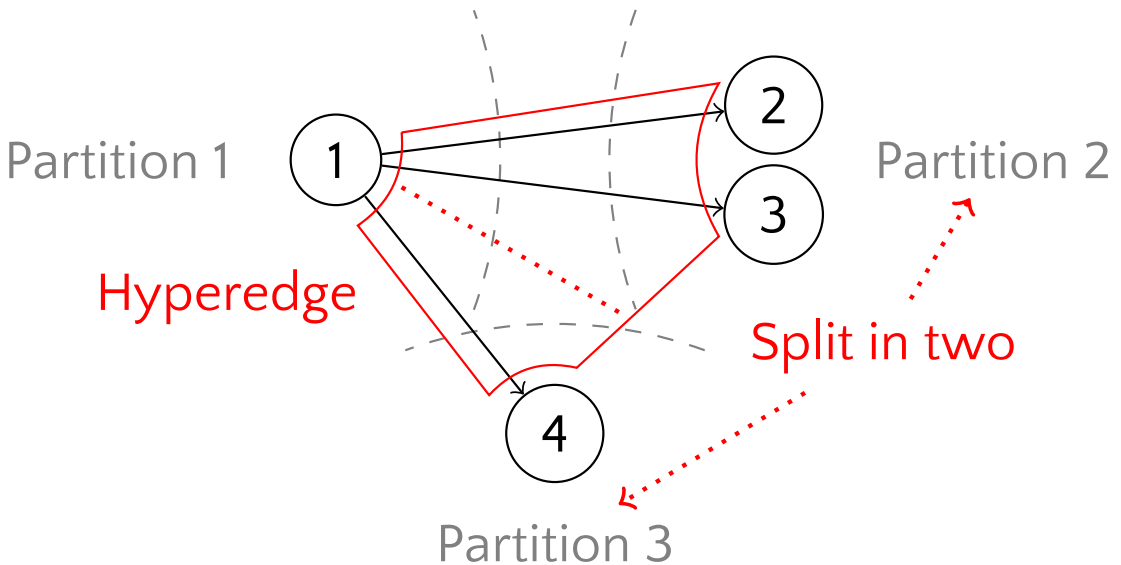
$\rightarrow$ Use regular partitioning and then convexify the result

Splitting Hairs

Partitioning is a well-researched topic. Convex partitioning is not.
$\rightarrow$ Use regular partitioning and then convexify the result

Some specifics to keep in mind:

- IO-cost of a tile $\propto$ number of loads $\rightarrow$ use hypergraphs



- Memory footprint of a partition varies with the internal scheduling
  $\rightarrow$ A conservative estimate should be used
- Speed-up partitioning of large graphs by a coarsening stage
- Hierarchical partitioning schemes can tile for multiple memory-levels simultaneously

- Memory footprint of a partition varies with the internal scheduling
  $\rightarrow$ A conservative estimate should be used
- Speed-up partitioning of large graphs by a coarsening stage
- Hierarchical partitioning schemes can tile for multiple memory-levels simultaneously

Some trace-partitioning results for a 20 kB cache:

Bench	Seidel-2D	ADI	FFT
Original $-miss	82 000	53 000	$< 5\,000$
Partitioned $-miss	19 500	24 000	42 000

Presentation progress

Problem statement

Generalized tiling

Solving the tiling problem

Beyond semi-regularity

Resource constraints

Conclusion & Future work

Tetris: If It Doesn't Fit, Flip It Over

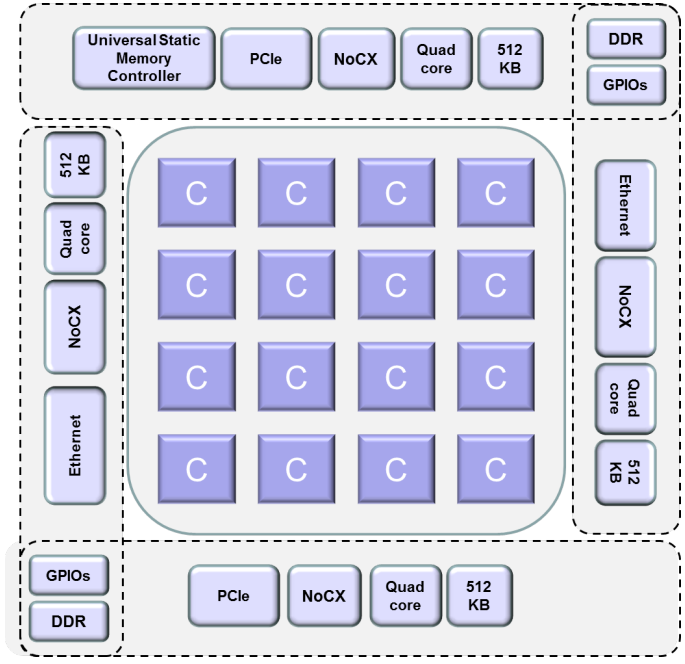
Dataflow tiling has multiple objectives:

- Register tiling (within dataflow nodes)
- Cache tiling for L1 & L2 caches

Tetris: If It Doesn't Fit, Flip It Over

Dataflow tiling has multiple objectives:

- Register tiling (within dataflow nodes)
- Cache tiling for L1 & L2 caches
- Tiling for computational resources



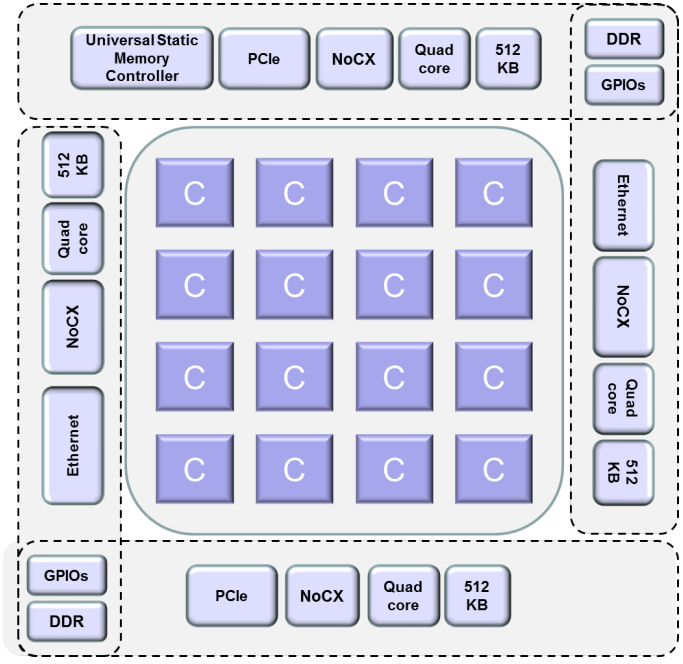
Tetris: If It Doesn't Fit, Flip It Over

Dataflow tiling has multiple objectives:

- Register tiling (within dataflow nodes)
- Cache tiling for L1 & L2 caches
- Tiling for computational resources

The correspondance with our tiling model?

1. Clusters with limited local memory
   $\rightarrow$ Extra cache-level
2. Multiple execution units
   $\rightarrow$ Profile timings for scheduling
3. Communications are explicit
   $\rightarrow$ Evaluate latencies for scheduling



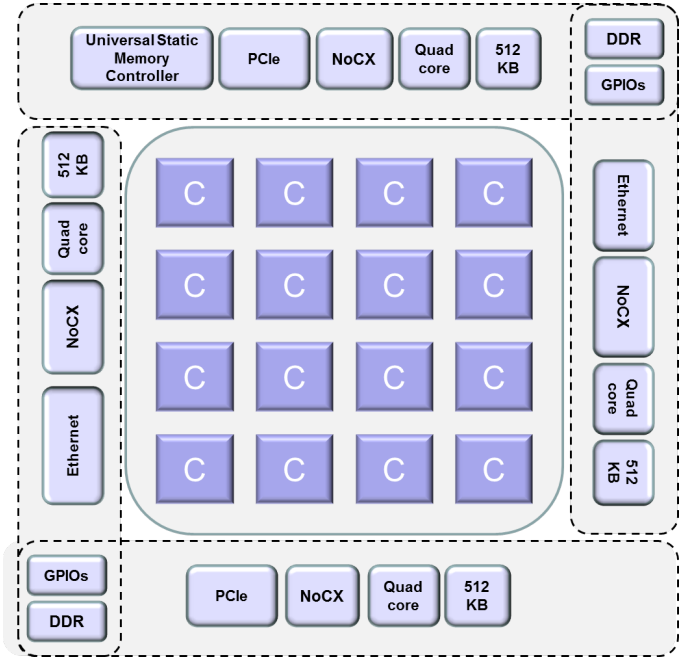
Tetris: If It Doesn't Fit, Flip It Over

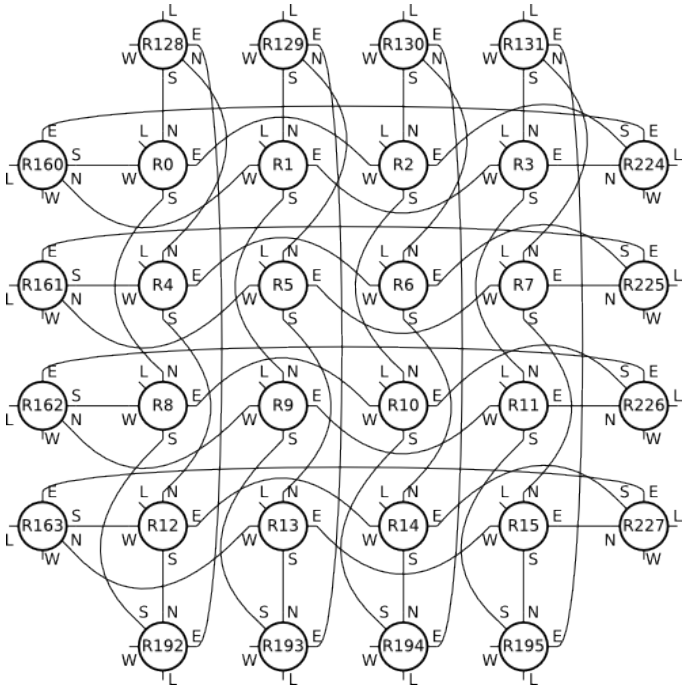
Dataflow tiling has multiple objectives:

- Register tiling (within dataflow nodes)
- Cache tiling for L1 & L2 caches
- Tiling for computational resources

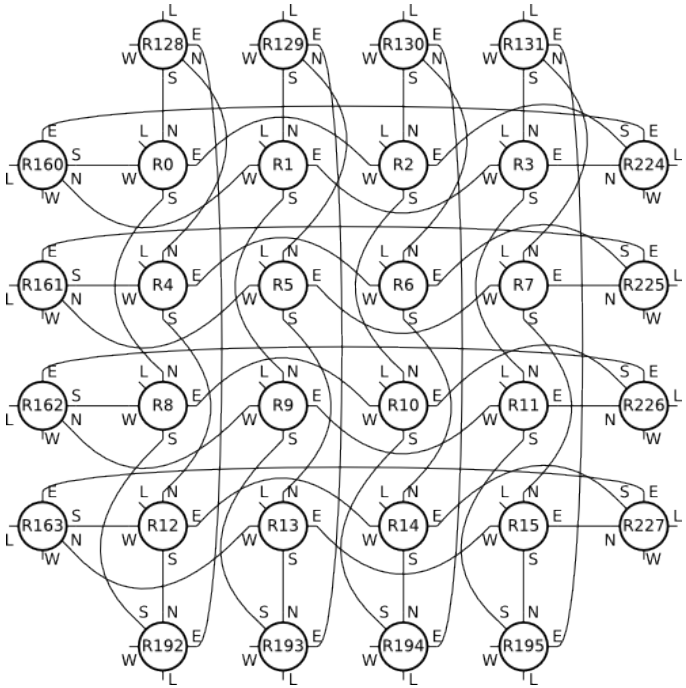
The correspondance with our tiling model?

1. Clusters with limited local memory
   $\rightarrow$ Extra cache-level
2. Multiple execution units
   $\rightarrow$ Profile timings for scheduling
3. Communications are explicit
   $\rightarrow$ Evaluate latencies for scheduling





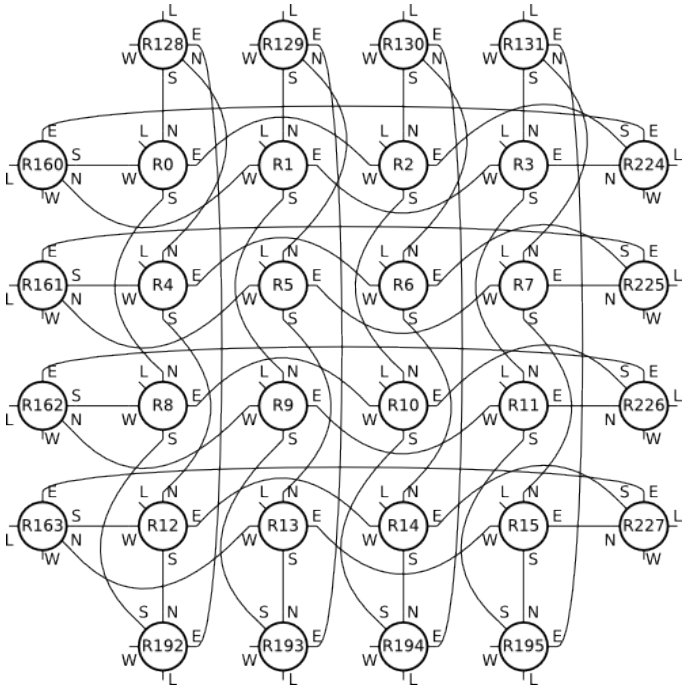
Communications use a Network-on-Chip
$\rightarrow$ Network-related issues (congestion, ...)



Communications use a Network-on-Chip
$\rightarrow$ Network-related issues (congestion, ...)

But:

- Hardware has Quality-of-Service features
- Behaviour of network is deterministic



Communications use a Network-on-Chip
$\rightarrow$ Network-related issues (congestion, ...)

But:

- Hardware has Quality-of-Service features
- Behaviour of network is deterministic

Model & restrict flows with (σ, ρ)

σ "Burstiness" of a flow

ρ Average data-rate of the flow

1. QoS usage prevents congestion $\rightarrow$ no deadlock either
2. Sticking to the model allows for computation of maximum delay

Formulate linear constraints on flows on the NoC $\rightarrow$ solve by ILP

Applied and used in the Kalray MPPA architecture

Presentation progress

Problem statement

Generalized tiling

Solving the tiling problem

Beyond semi-regularity

Resource constraints

Conclusion & Future work

Straight Ahead

Work done and obtained results:

- Memory-use and communication model for tiling
- Novel tiling method applied to various situations (loops, dataflow)
- Graph-based trace analysis for performance debugging
- Contributions to convex graph partitioning
- Modelisation of a full Network-on-Chip and implementation of Quality-of-Service

Next steps:

- Finish the implementation of a full tiling stack for SigmaC
- Improve register tiling and extend benchmarks
- Implement 1 / 2 extra convex partioning heuristics

...and future directions:

- Allow dynamic reconfiguration of tiling for dynamic dataflow
- Refine register tiling & scheduling heuristics
- Include scheduling heuristics in graph partioning to better evaluate memory requirements of potential tiles

So Long, and Thanks for All the Fish

Questions? Just ask!

... the answer is 42 anyway