Data Locality on Manycore Architectures

Duco van Amstel

Inria / Kalray – Grenoble, France

Ph.D defence – July 18th, 2016

Topics

Introduction

The Proverbial Memory Wall

Observations:

- Available computational power keeps growing exponentially
- Bandwidth of memory interfaces increases slower

Has been the case since the '90s and is called the **Memory-Wall**

Definition

For a given executable / partial code :

$$\text{Operational Intensity (OI)} = \frac{\text{Number of instructions}}{\text{Number of memory operations}}$$

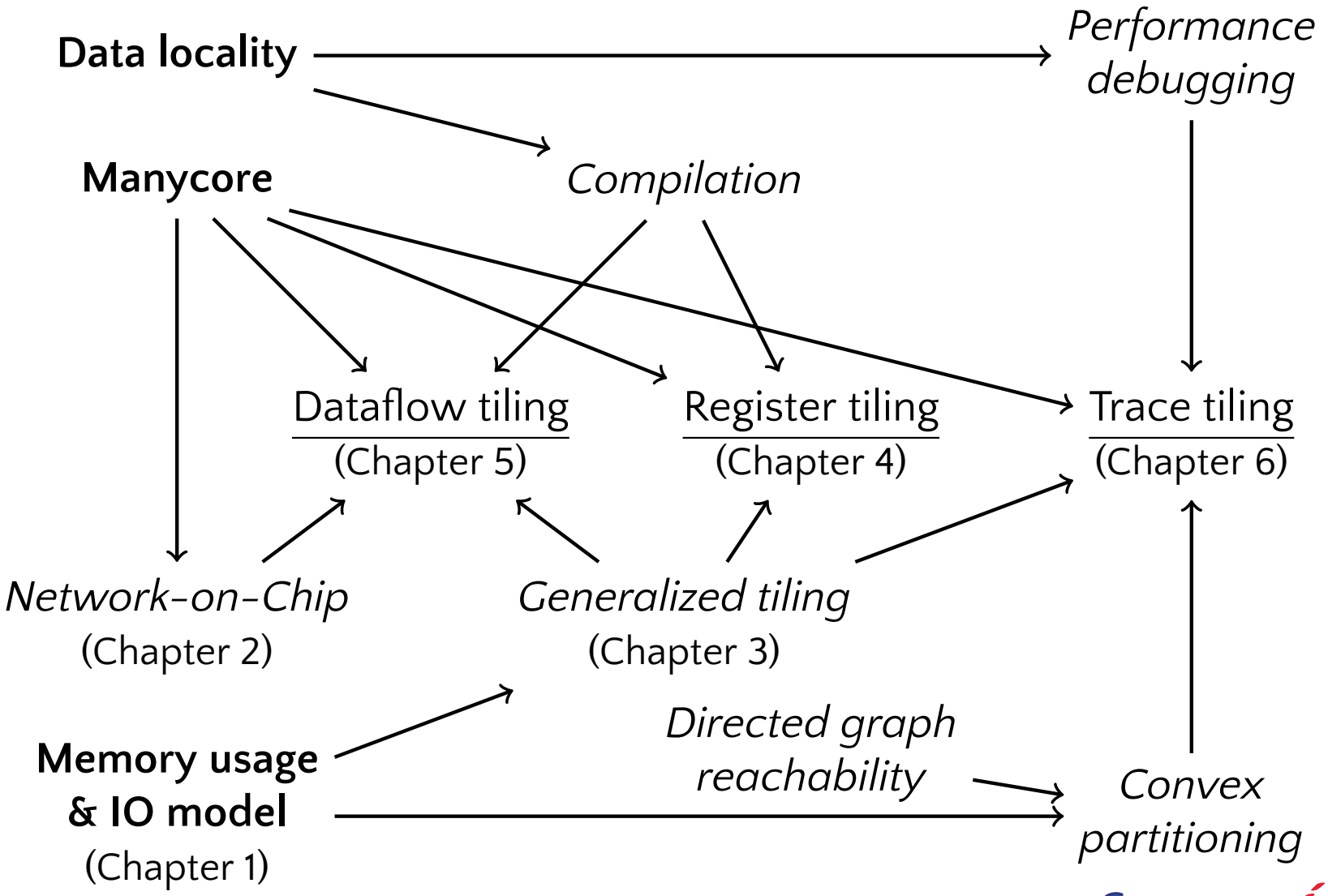
Research topics

Working hypothesis

Local target memory of limited size communicating with distant memory of infinite size through memory operations

Idea: Improve data reuses and reduce IO operations of frequently executed code
$\rightarrow$ Improve **data locality** of programs

Overview


Data locality
Performance debugging
Manycore
Compilation
Dataflow tiling
(Chapter 5)
Register tiling
(Chapter 4)
Trace tiling
(Chapter 6)
Network-on-Chip
(Chapter 2)
Generalized tiling
(Chapter 3)
Directed graph reachability
Memory usage & IO model
(Chapter 1)
Convex partitioning

Topics

1. Modeling memory usage and IO costs

Memory usage
& IO model
(Chapter 1)

Code representation

What we want:

1. a compact representation
2. data reuses and no other dependencies
3. straightforward evaluation of memory usage for any part of the representation

We refer to it as the **memory-use graph**.

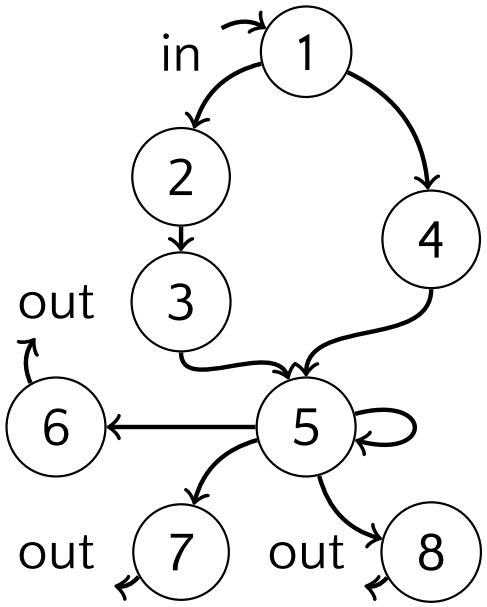


Figure: Dataflow diagram

Memory-use graph construction

```
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];
}
```

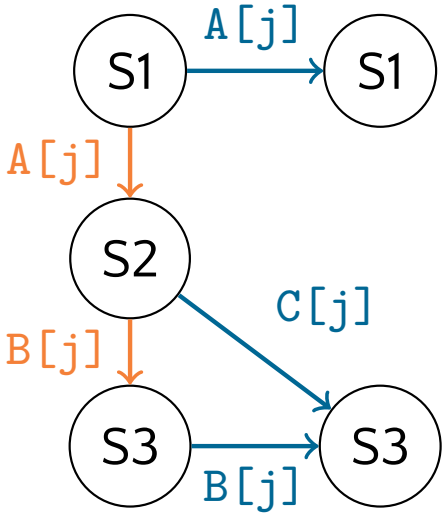
Memory-use graph construction

```
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
- intra-iteration edges

Some transformations need to be performed

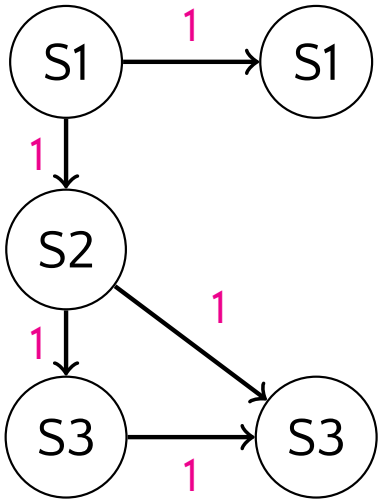


Memory-use graph construction

```
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];
}
```

Steps:

1. Indicate the size of reused data on edges

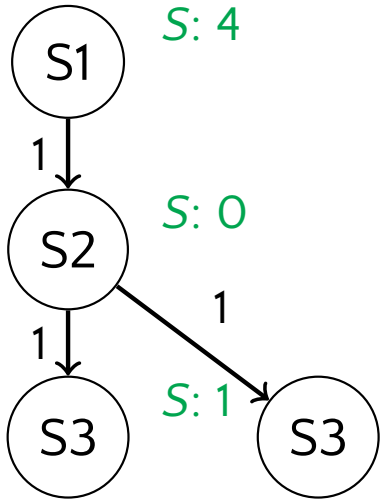


Memory-use graph construction

```
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];
}
```

Steps:

1. Indicate the size of reused data on edges
2. Transform self-reuses into **state** data

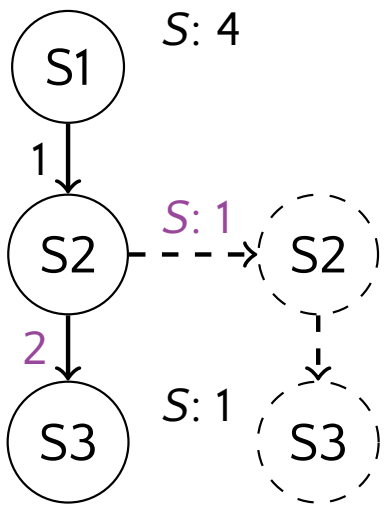


Memory-use graph construction

```
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];
}
```

Steps:

1. Indicate the size of reused data on edges
2. Transform self-reuses into **state** data
3. Transform other inter-iteration reuses into **state** and intra-iteration reuse data

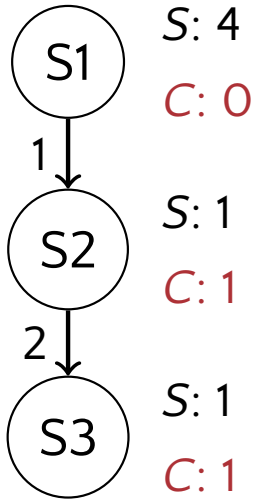


Memory-use graph construction

```
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];
}
```

Steps:

1. Indicate the size of reused data on edges
2. Transform self-reuses into **state** data
3. Transform other inter-iteration reuses into **state** and intra-iteration reuse data
4. Add **internal computation requirements** to account for hidden intermediary values

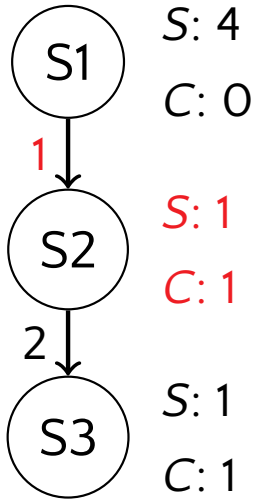


Memory-use graph construction

```
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



Evaluation of memory usage

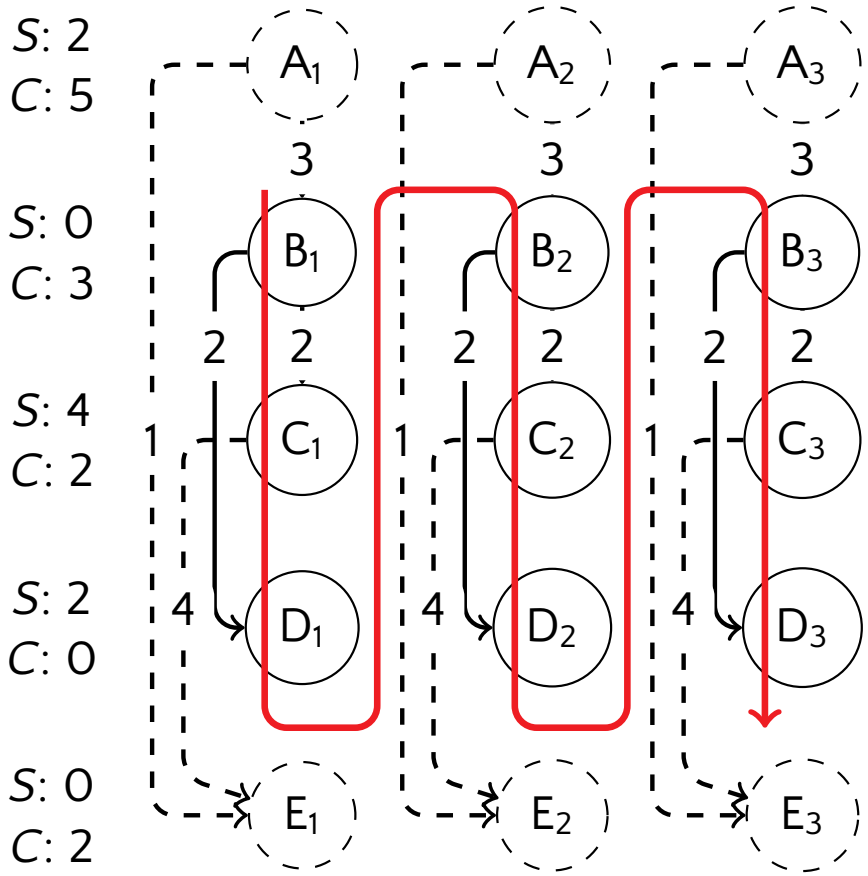


Figure: Iterated **memory-use graph**

Consider a **partial** execution:

$$B_1 \to C_1 \to D_1 \to B_2 \to C_2 \to D_2 \to B_3 \to C_3 \to D_3$$

Compute the size of data stored in memory in and between nodes:

$$3, 4, 10, \ldots, 10, 12, 8, \ldots, 12, 4, 4$$

The maximum is the **memory usage** of the partial execution, *i.e* 12

IO cost computation

For an arbitrary piece of code:

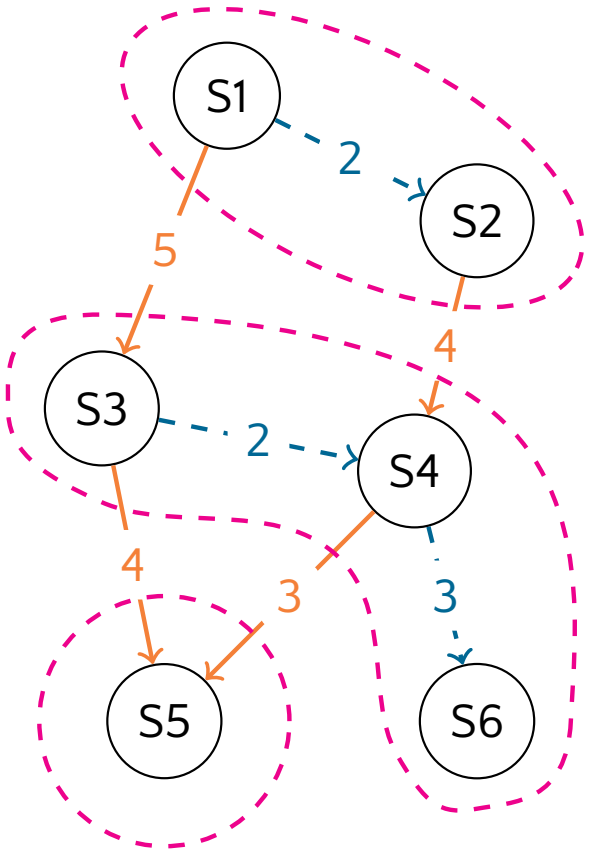
memory usage $\leq$ memory size

$\downarrow$

Internal reuses do not require IO

Divide evaluated code in such pieces

- Only external reuses generate IO
- IO cost equal to cut between pieces



Memory size 7
IO cost 16

Topics

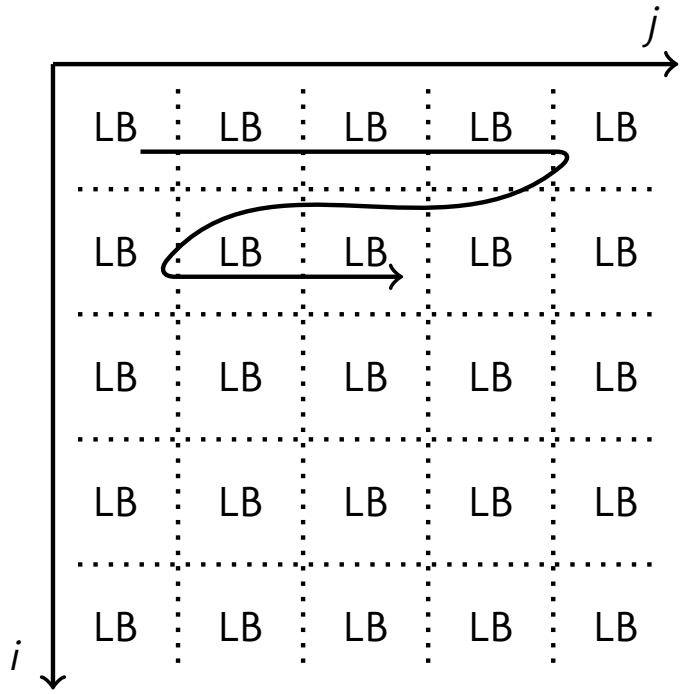
2 Generalized tiling

Data locality

Generalized tiling
(Chapter 3)

Memory usage
& IO model
(Chapter 1)

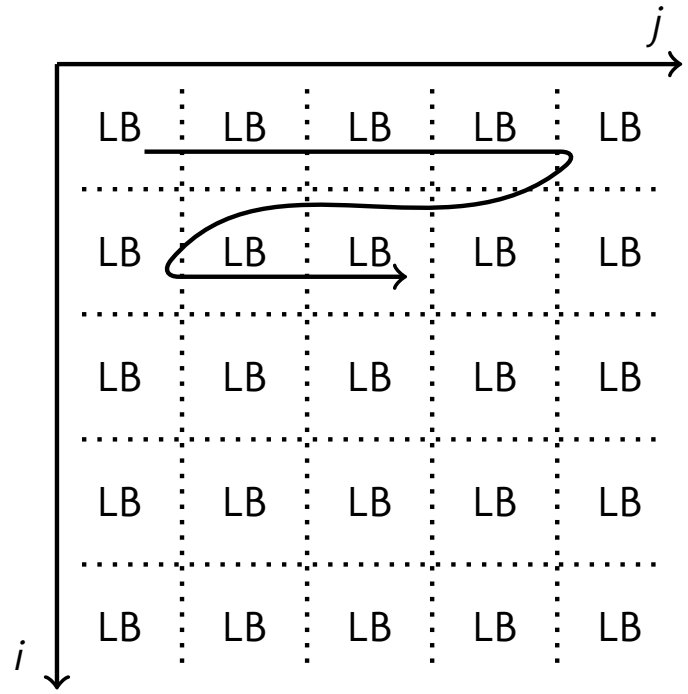
Representing the iteration space




```
for (int i = 1; i < N; i++)
 for (int j = 1; j < M; j++)
  S1: A[i,j] = B[i] + C[j];
  S2: A[i,j-1] = A[i,j-1] * A[i,j];
  S3: A[i-1,j] = A[i-1,j] * A[i,j];
  S4: A[i,j] = k * A[i,j];
```

LB (Loop-body) = {S1,S2,S3,S4}

Representing the iteration space

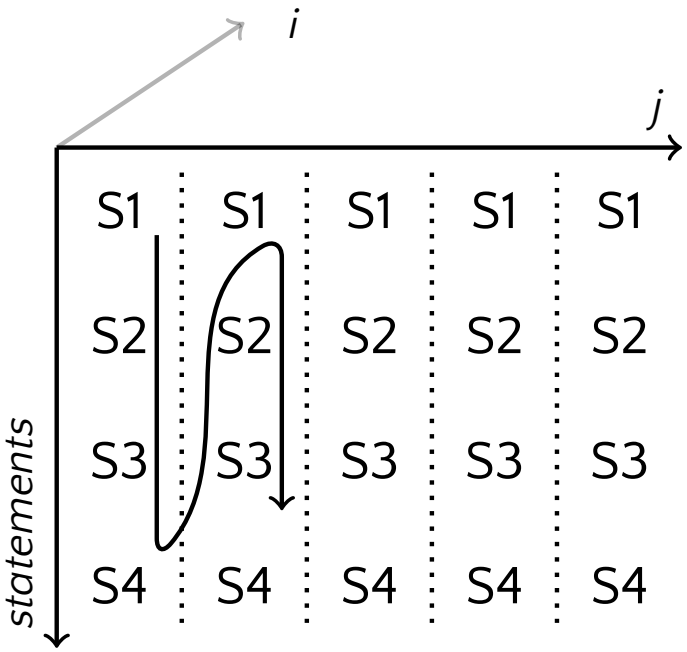



```
for (int i = 1; i < N; i++)
 for (int j = 1; j < M; j++)
  S1: A[i,j] = B[i] + C[j];
  S2: A[i,j-1] = A[i,j-1] * A[i,j];
  S3: A[i-1,j] = A[i-1,j] * A[i,j];
  S4: A[i,j] = k * A[i,j];
```

LB (Loop-body) = {S1,S2,S3,S4}

Two new dimensions:

1. Statements / actors
2. Innermost iteration / inputs

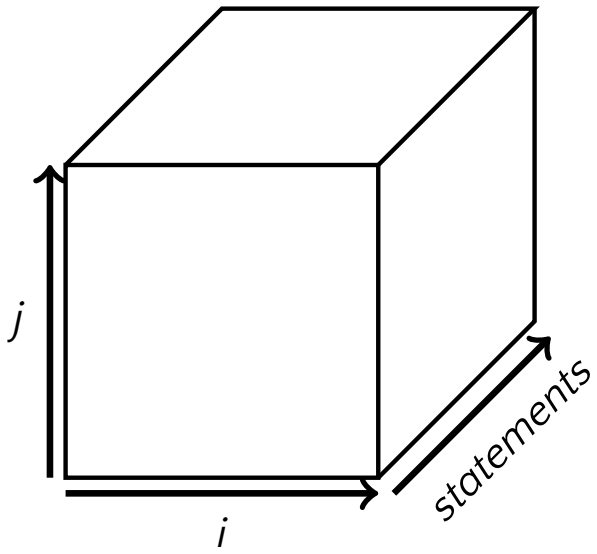


Data locality optimizations

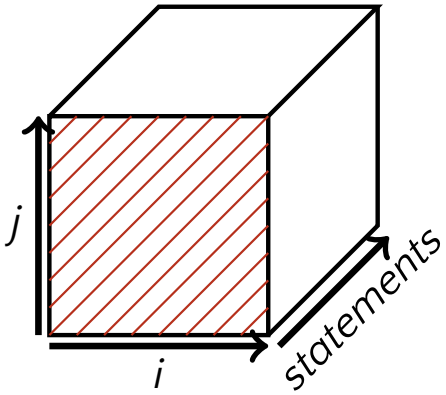
Code example:

```
for (int i = 1; i < N; i++)
 for (int j = 1; j < M; j++)
  S1: A[i,j] = B[i] + C[j];
  S2: A[i,j-1] = A[i,j-1] * A[i,j];
  S3: A[i-1,j] = A[i-1,j] * A[i,j];
  S4: A[i,j] = k * A[i,j];
```

Space representation

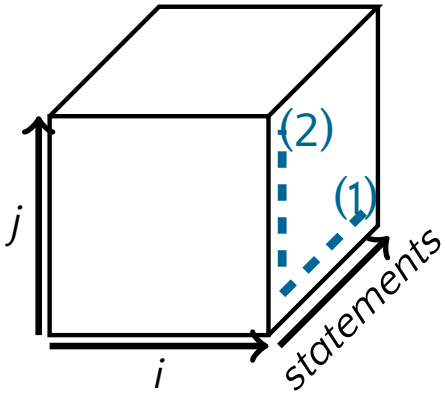


A missing piece



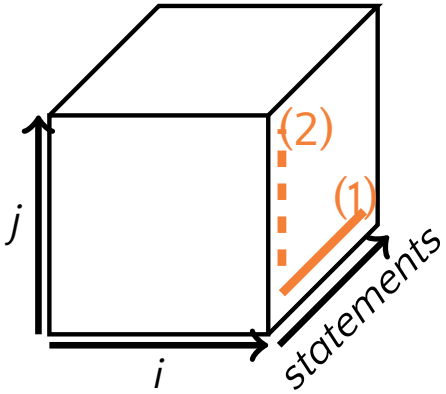
Register tiling

Loop-nest reuses



Scalar promotion

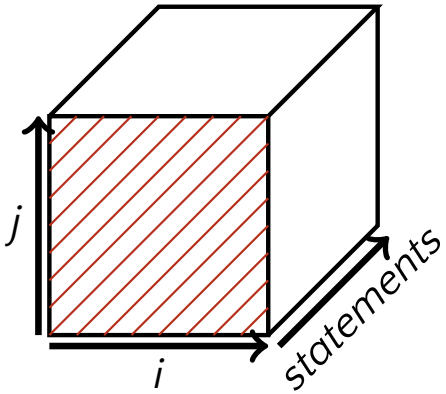
Intra & cross-iteration greedy reuses



Register allocation

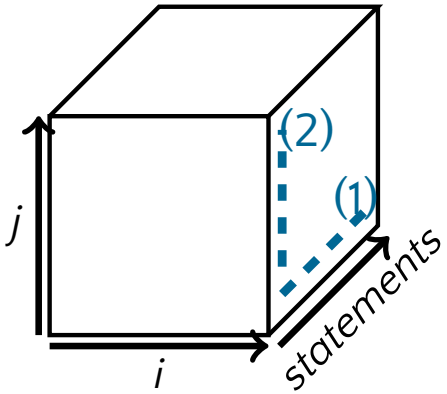
Intra-iteration optimal reuses

A missing piece



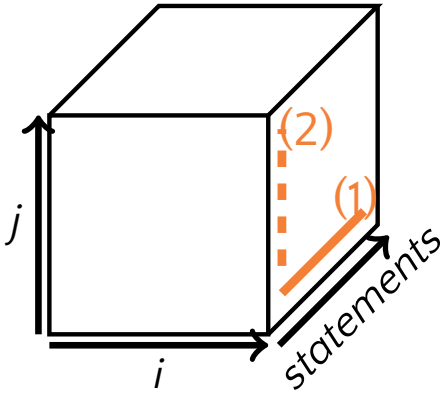
Register tiling

Loop-nest reuses



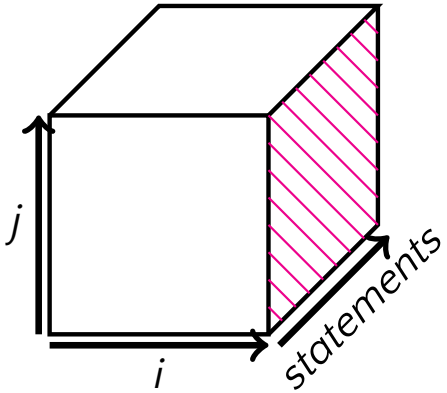
Scalar promotion

Intra & cross-iteration greedy reuses



Register allocation

Intra-iteration optimal reuses



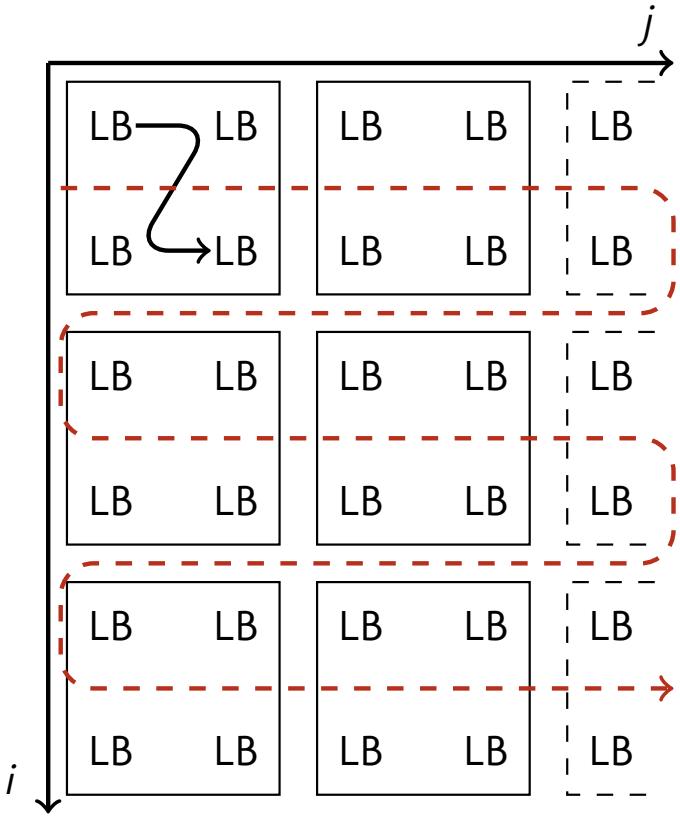
Generalized register tiling

Inter & intra-iteration reuses

Comparison of tiling methods

Tile regularity

classical tiling



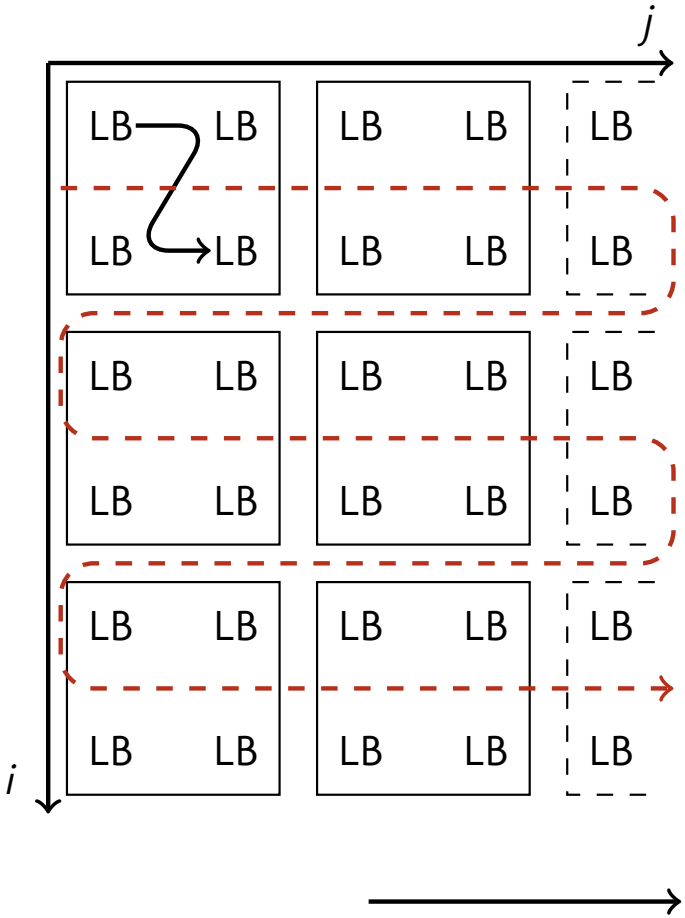
Schedule within a tile

Ordering of tiles

Comparison of tiling methods

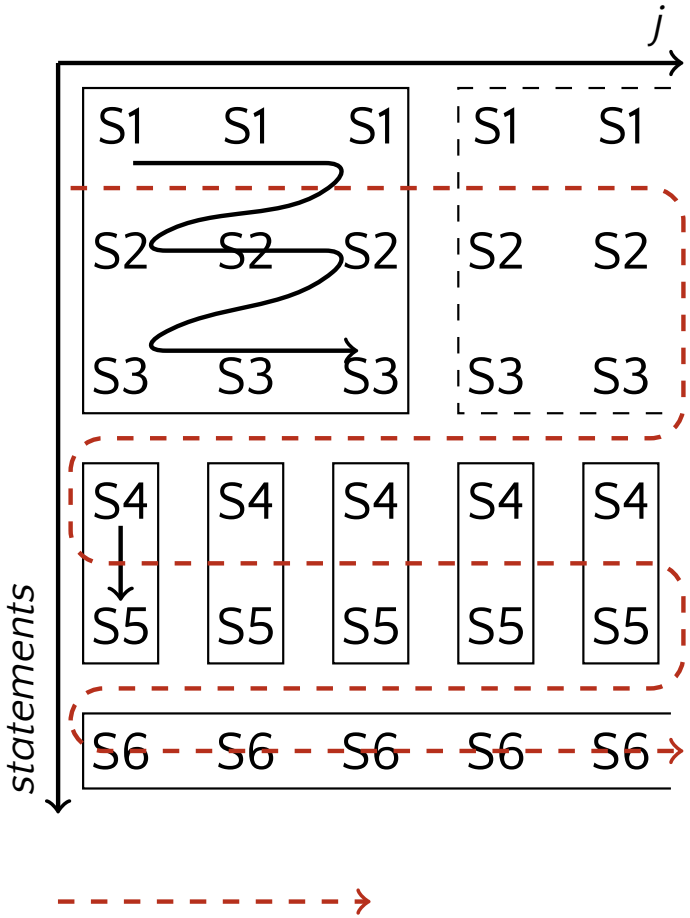
Tile regularity

classical tiling



Tile semi-regularity

generalized tiling



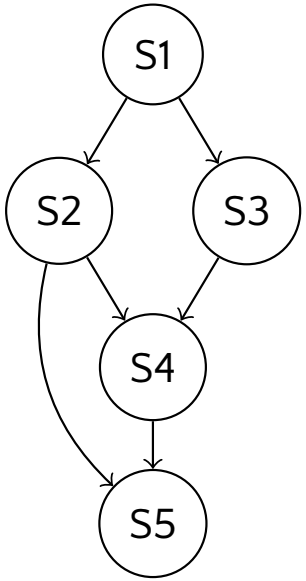
Schedule within a tile

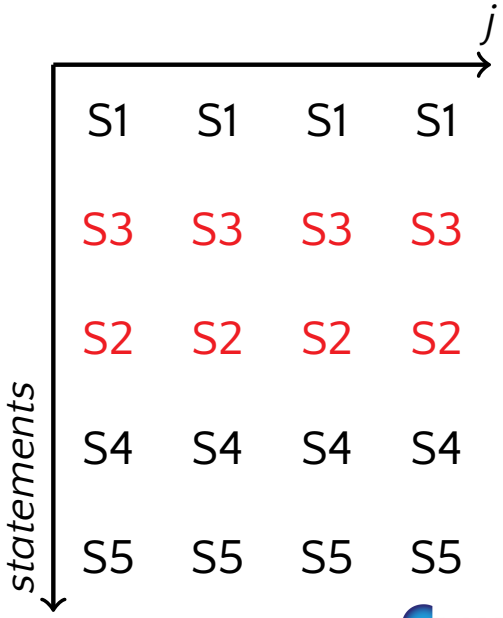
Ordering of tiles

Optimization problem

Defining a tiling solution means:

1. Linearization / schedule the memory-use graph

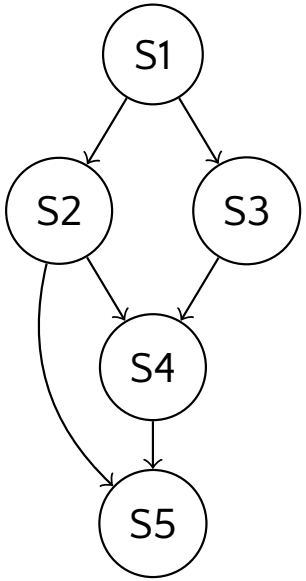


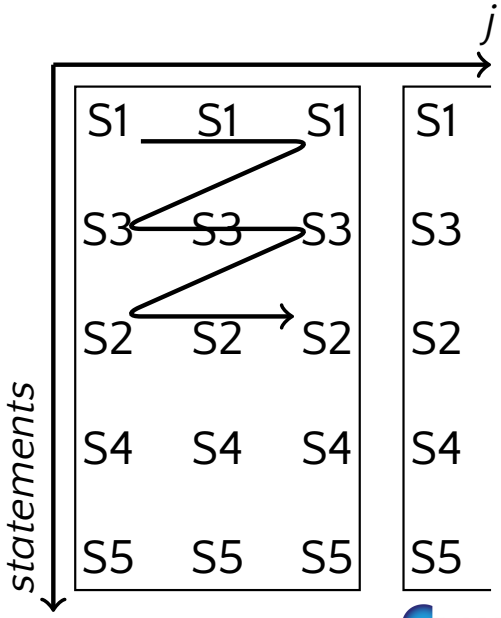


Optimization problem

Defining a tiling solution means:

1. Linearization / schedule the memory-use graph
2. Choosing points in the schedule where to place tile limits
3. Specifying a width for each tile



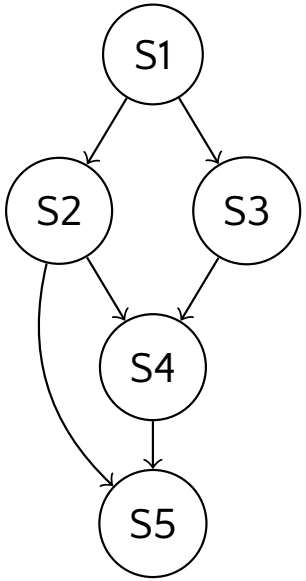


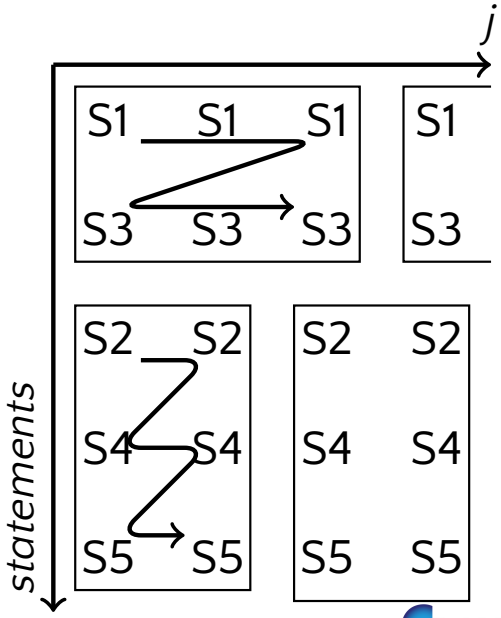
Optimization problem

Defining a tiling solution means:

1. Linearization / schedule the memory-use graph
2. Choosing points in the schedule where to place tile limits
3. Specifying a width for each tile

...and there are multiple solutions



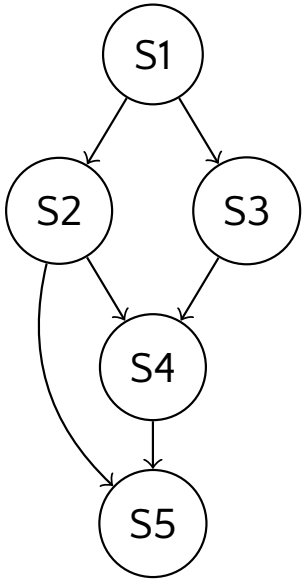


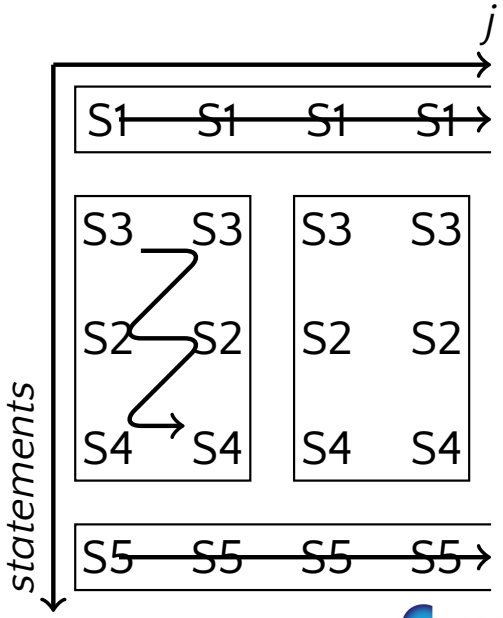
Optimization problem

Defining a tiling solution means:

1. Linearization / schedule the memory-use graph
2. Choosing points in the schedule where to place tile limits
3. Specifying a width for each tile

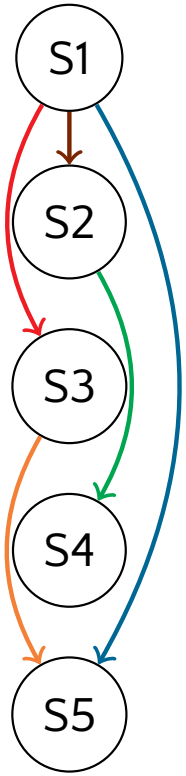
...and there are multiple solutions





Heavy-edge scheduling

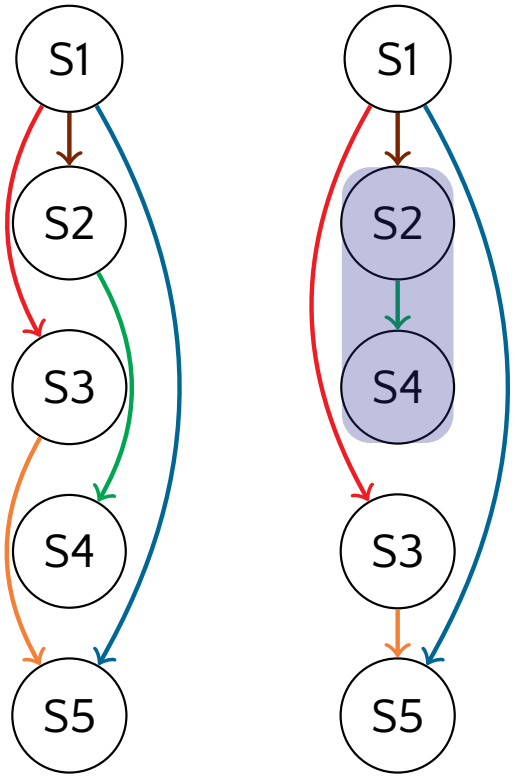
Objective Reducing the reach of reuses



Edges ordered by descending weight: green, red, orange, blue, brown

Heavy-edge scheduling

Objective Reducing the reach of reuses

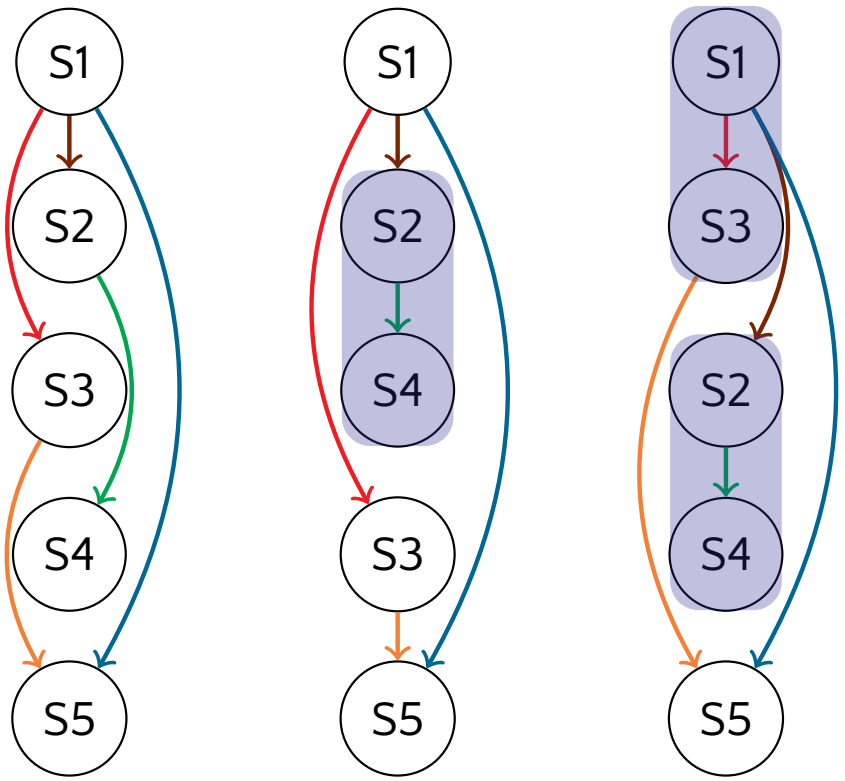


Edges ordered by descending weight: green, red, orange, blue, brown

Blue zones have been frozen

Heavy-edge scheduling

Objective Reducing the reach of reuses

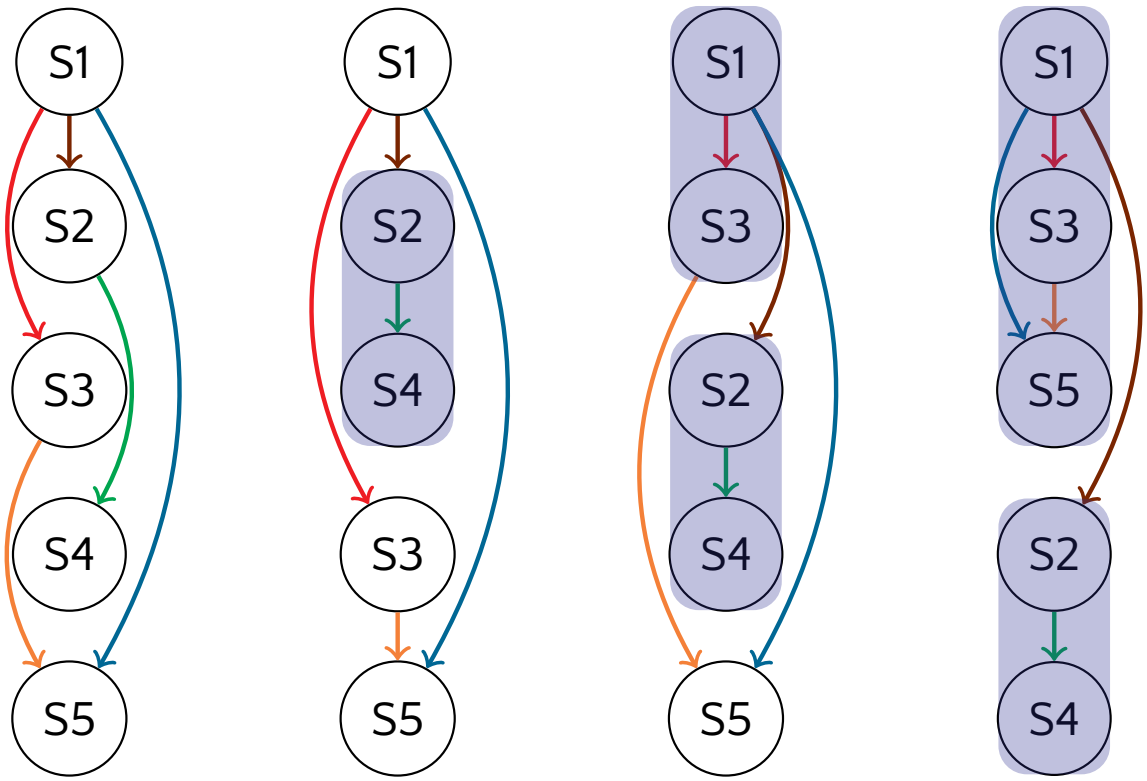


Edges ordered by descending weight: green, red, orange, blue, brown

Blue zones have been frozen

Heavy-edge scheduling

Objective Reducing the reach of reuses



Edges ordered by descending weight: green, red, orange, blue, brown

Blue zones have been frozen

Heuristical solver methods

Memory-use graph

↓

Schedule

↓

Tile

↓

Tiling solution

Heuristical solver methods

Heavy-edge / Greedy

Perform heavy-edge scheduling and greedily compute locally optimal tiles

Memory-use graph

↓

Schedule

↓

Tile

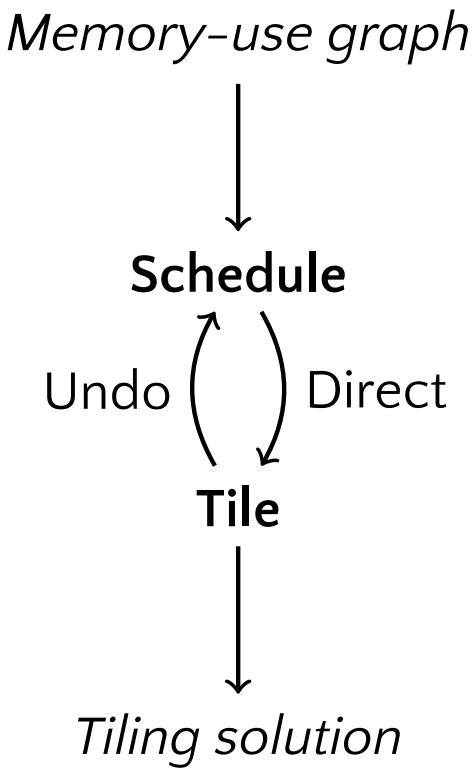
↓

Tiling solution

Heuristical solver methods

Heavy-edge / Greedy

Perform heavy-edge scheduling and greedily compute locally optimal tiles



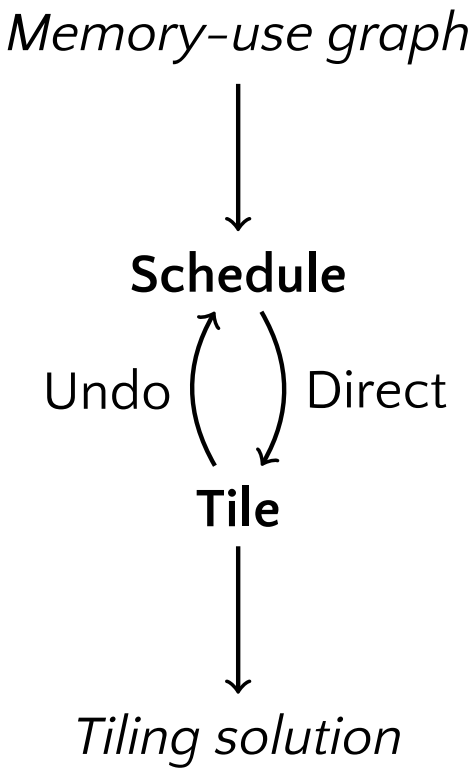
Heuristical solver methods

Heavy-edge / Greedy

Perform heavy-edge scheduling and greedily compute locally optimal tiles

Tile-aware / Heavy-edge

Contract edges one-by-one and create or expand tiles to include them; Roll back contraction if not possible



Heuristical solver methods

Heavy-edge / Greedy

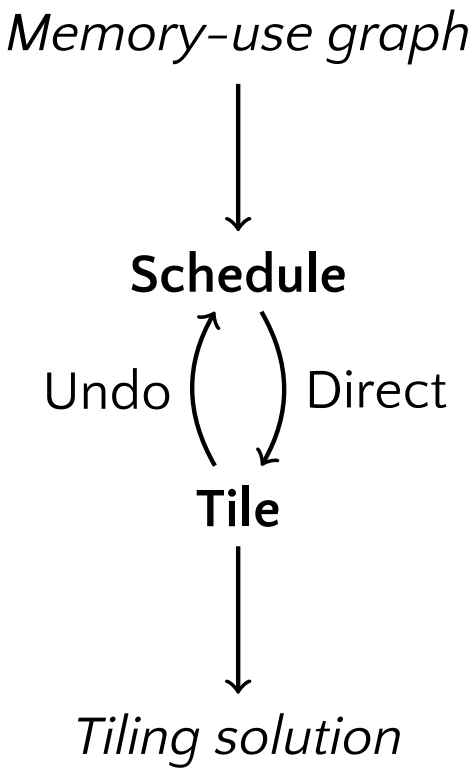
Perform heavy-edge scheduling and greedily compute locally optimal tiles

Tile-aware / Heavy-edge

Contract edges one-by-one and create or expand tiles to include them; Roll back contraction if not possible

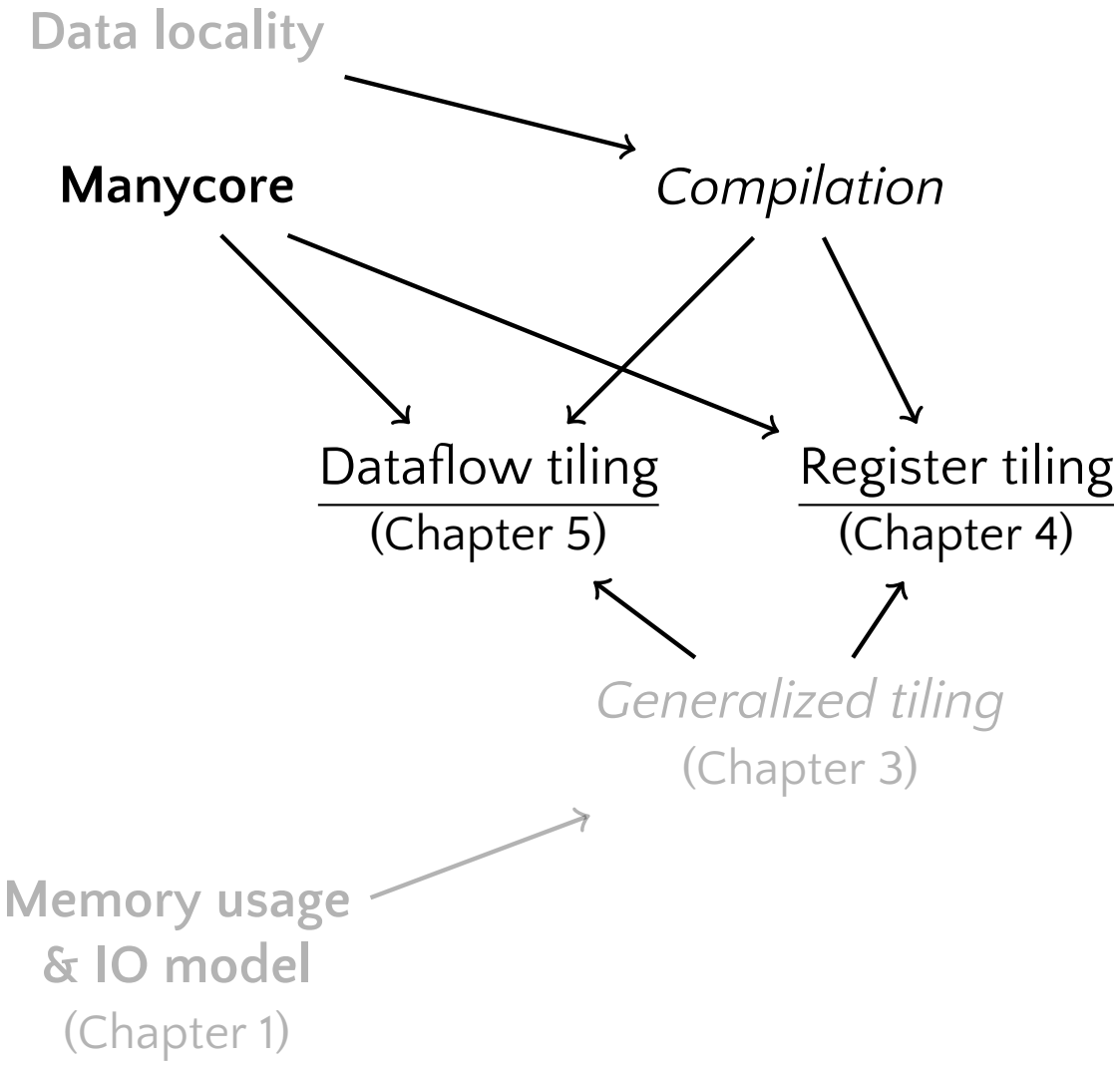
Tile-aware / Conservative

Contract edges one-by-one and create tiles to include them; Roll back contraction if not possible



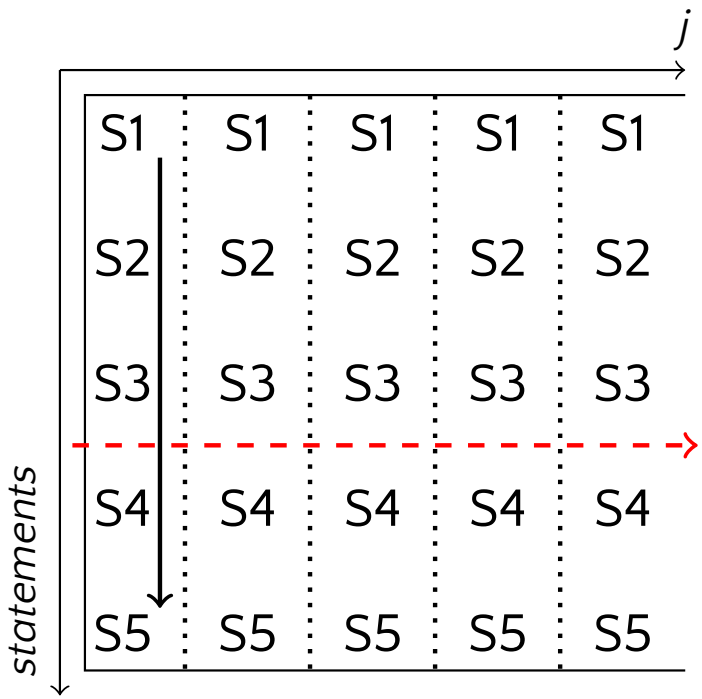
Topics

3 Applications in loop and dataflow optimization



Generalized register tiling

Code generation



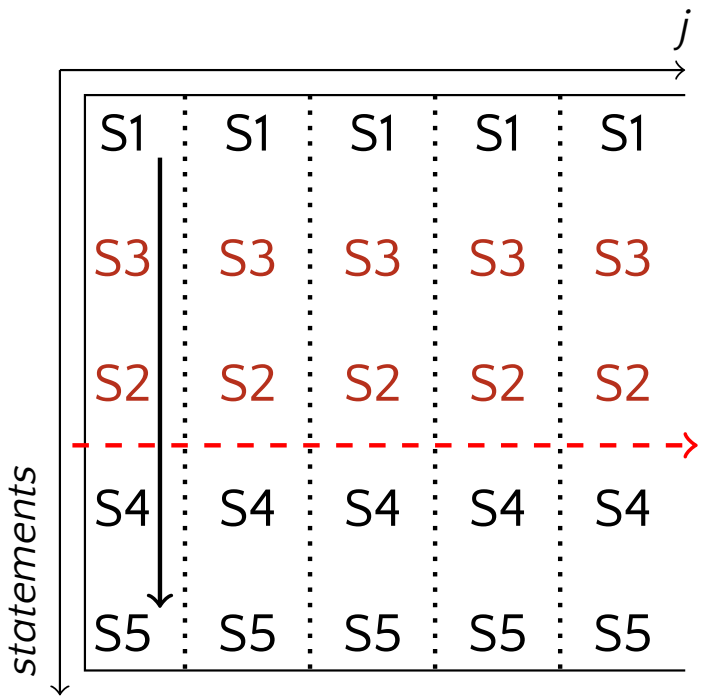
Target tiling:

Schedule	S1, S3, S2, S4, S5
Tile 1	{S1, S3} – width 3
Tile 2	{S2, S4, S5} – width 2

Transformations to be applied:

Generalized register tiling

Code generation



Target tiling:

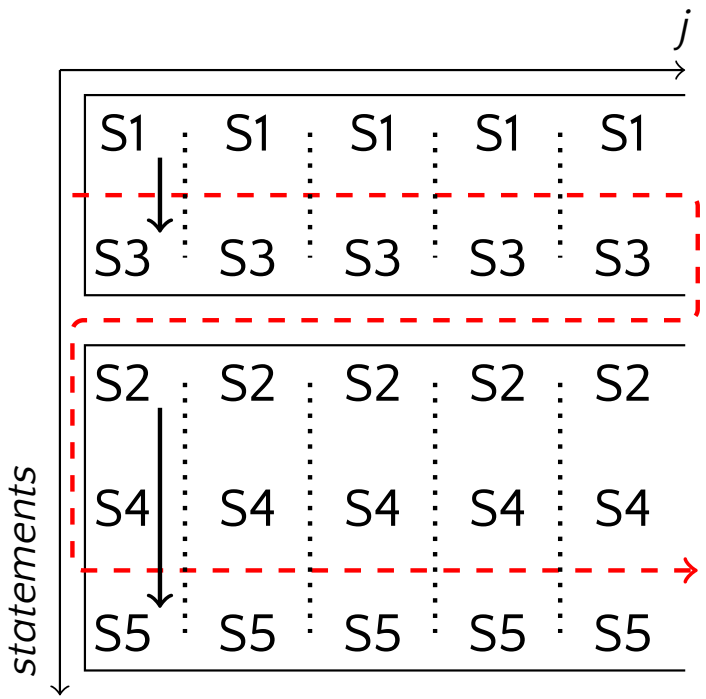
Schedule	S1, S3, S2, S4, S5
Tile 1	{S1, S3} – width 3
Tile 2	{S2, S4, S5} – width 2

Transformations to be applied:

1. Rescheduling

Generalized register tiling

Code generation



Target tiling:

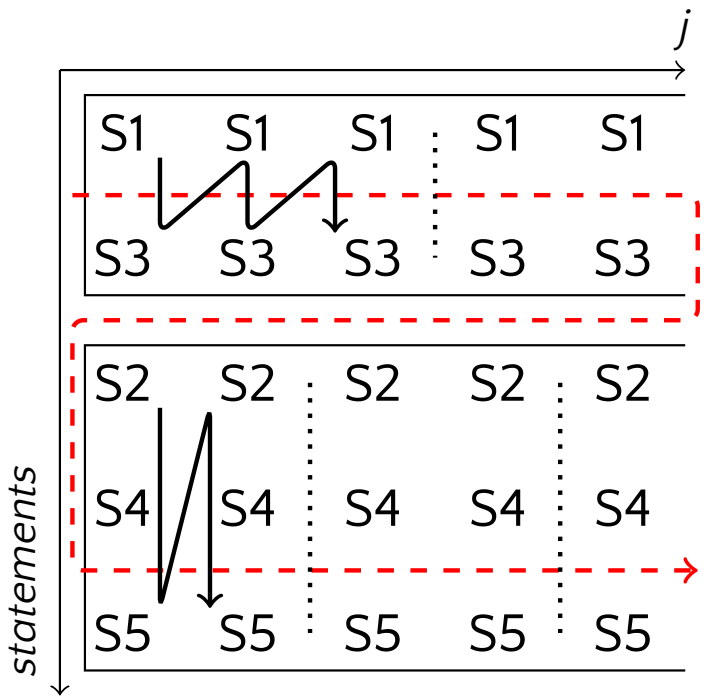
Schedule	S1, S3, S2, S4, S5
Tile 1	{S1, S3} – width 3
Tile 2	{S2, S4, S5} – width 2

Transformations to be applied:

1. Rescheduling
2. Loop fission

Generalized register tiling

Code generation



Target tiling:

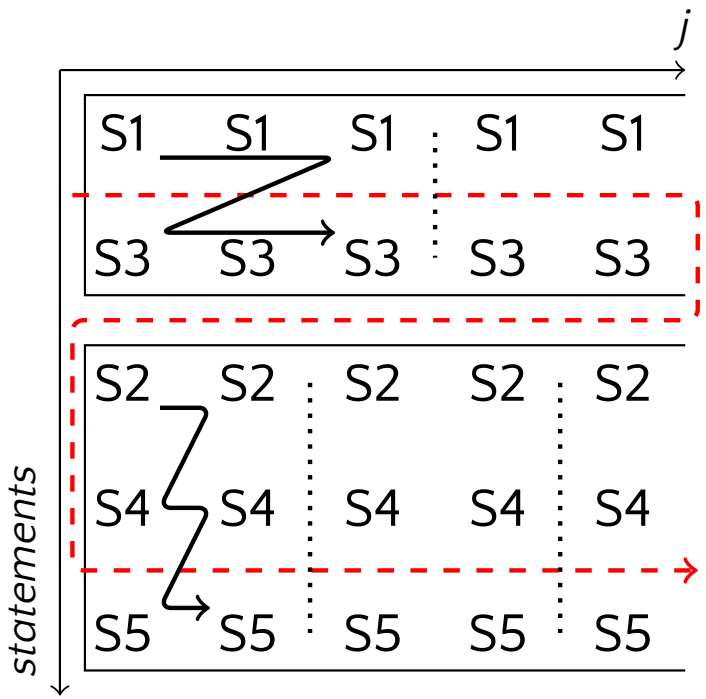
Schedule	S1, S3, S2, S4, S5
Tile 1	{S1, S3} – width 3
Tile 2	{S2, S4, S5} – width 2

Transformations to be applied:

1. Rescheduling
2. Loop fission
3. Loop unrolling

Generalized register tiling

Code generation



Target tiling:

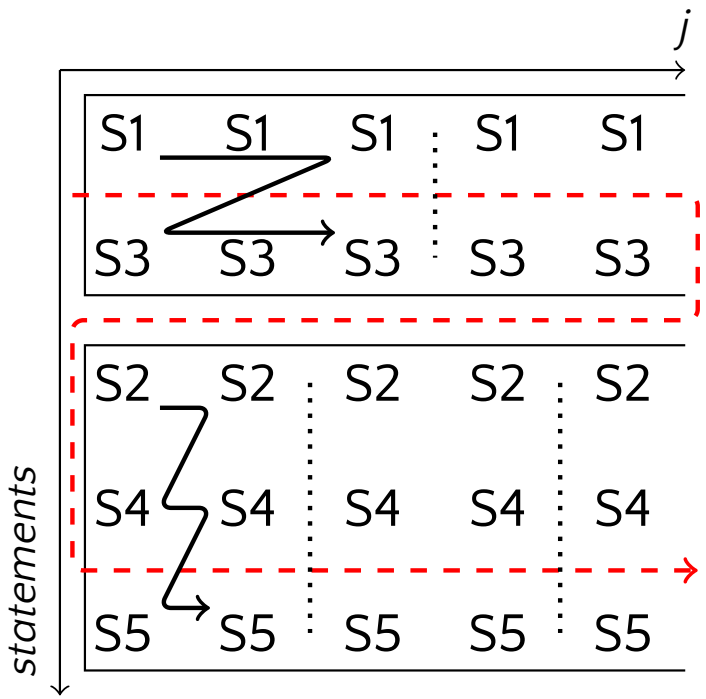
Schedule	S1, S3, S2, S4, S5
Tile 1	{S1, S3} – width 3
Tile 2	{S2, S4, S5} – width 2

Transformations to be applied:

1. Rescheduling
2. Loop fission
3. Loop unrolling
4. Rescheduling

Generalized register tiling

Code generation



Target tiling:

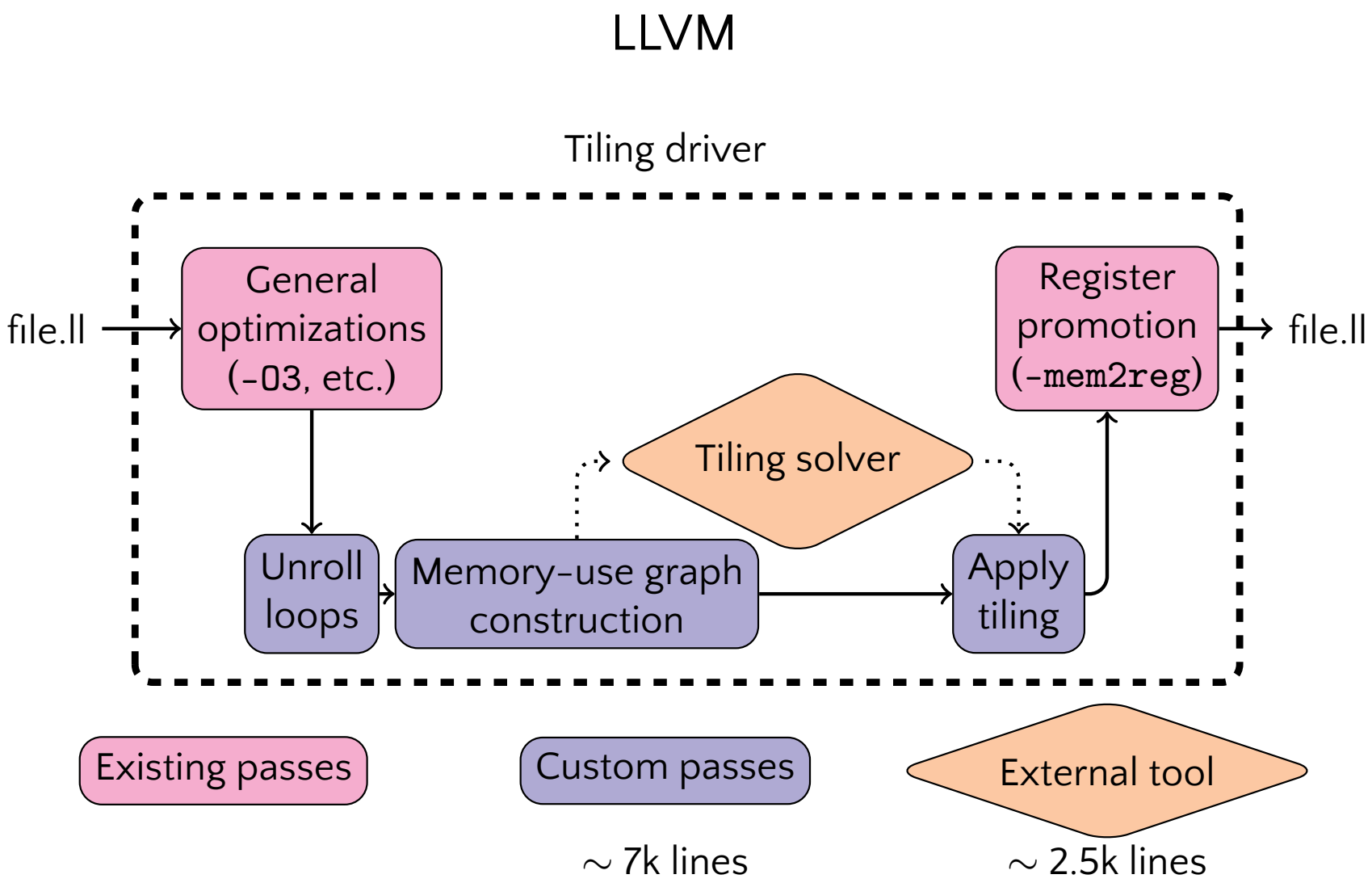
Schedule	S1, S3, S2, S4, S5
Tile 1	{S1, S3} – width 3
Tile 2	{S2, S4, S5} – width 2

Transformations to be applied:

1. Rescheduling
2. Loop fission
3. Loop unrolling
4. Rescheduling

Prototyping in the Open64 compiler suggests the impact of generalized tiling on load/store operations could be significant.

Implementation



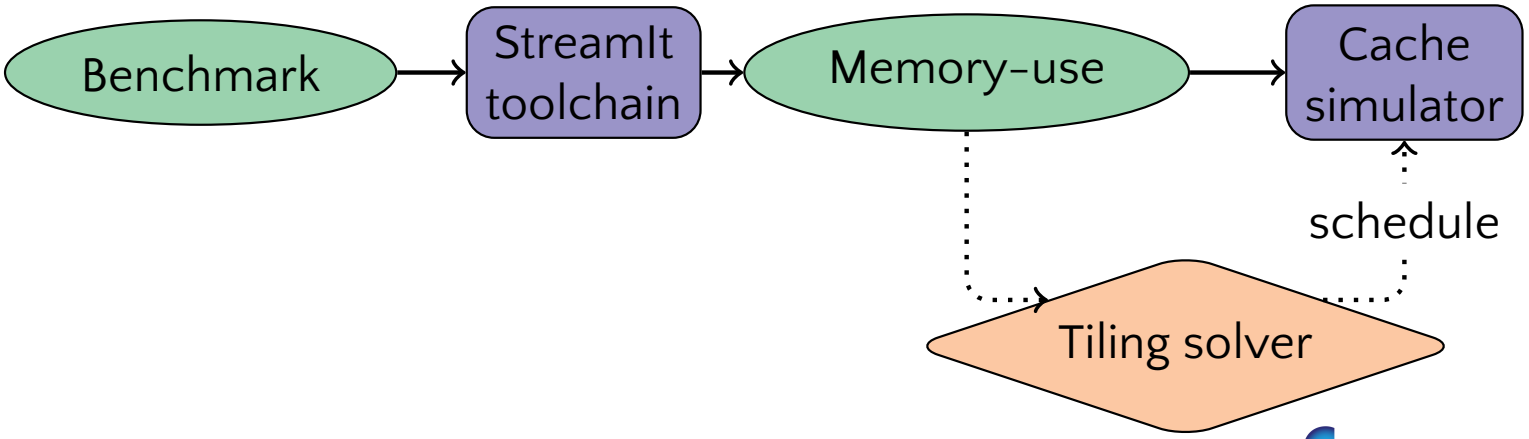
Generalized dataflow tiling

Target

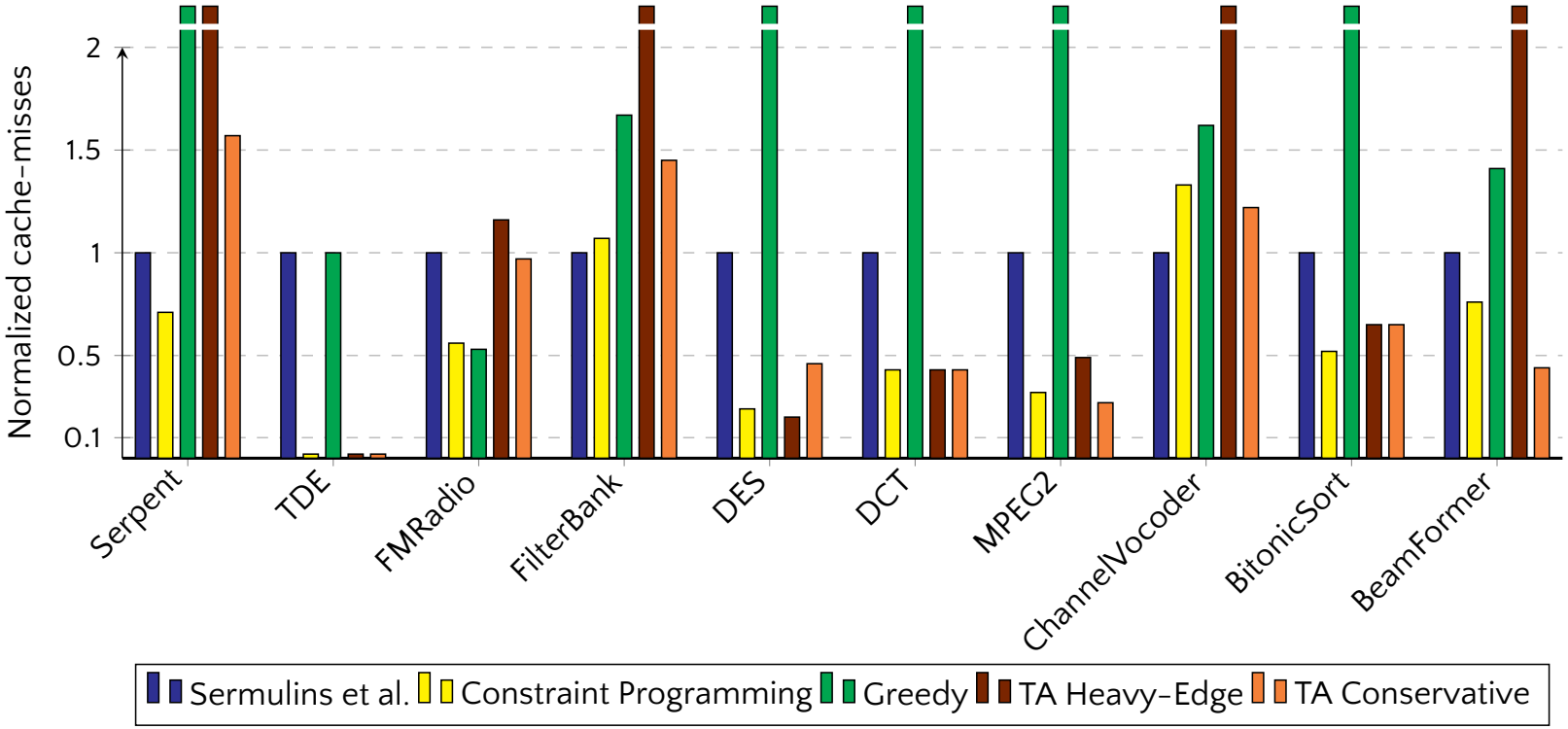
Static & cyclo-static dataflow languages

Prototyping was done in **StreamIt**:

- Usage of preprocessed StreamIt benchmarks
- Rescheduling of actors & execution scaling
- Simulation of cache-behaviour / evaluation of cache-misses

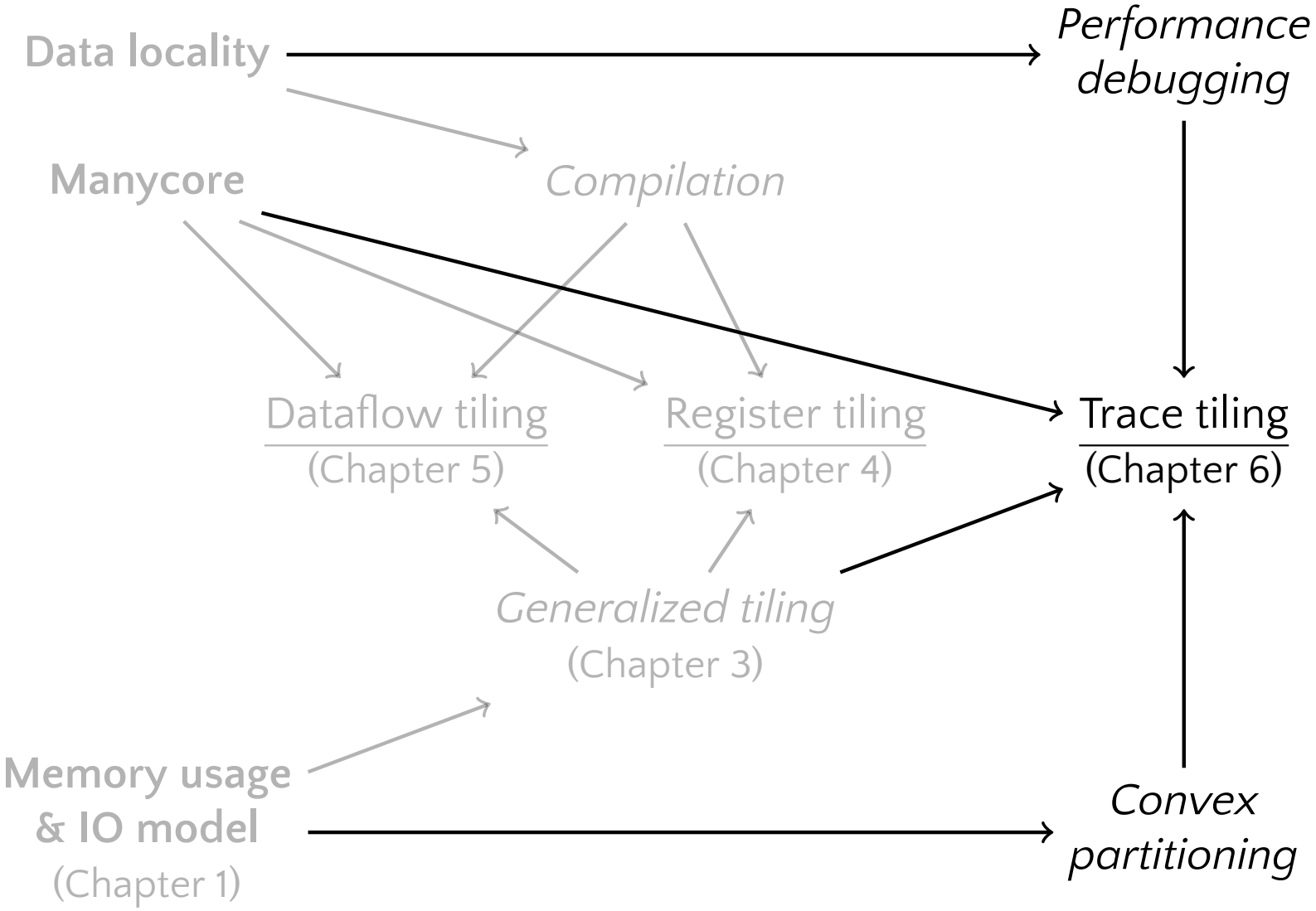


Prototype results



Topics

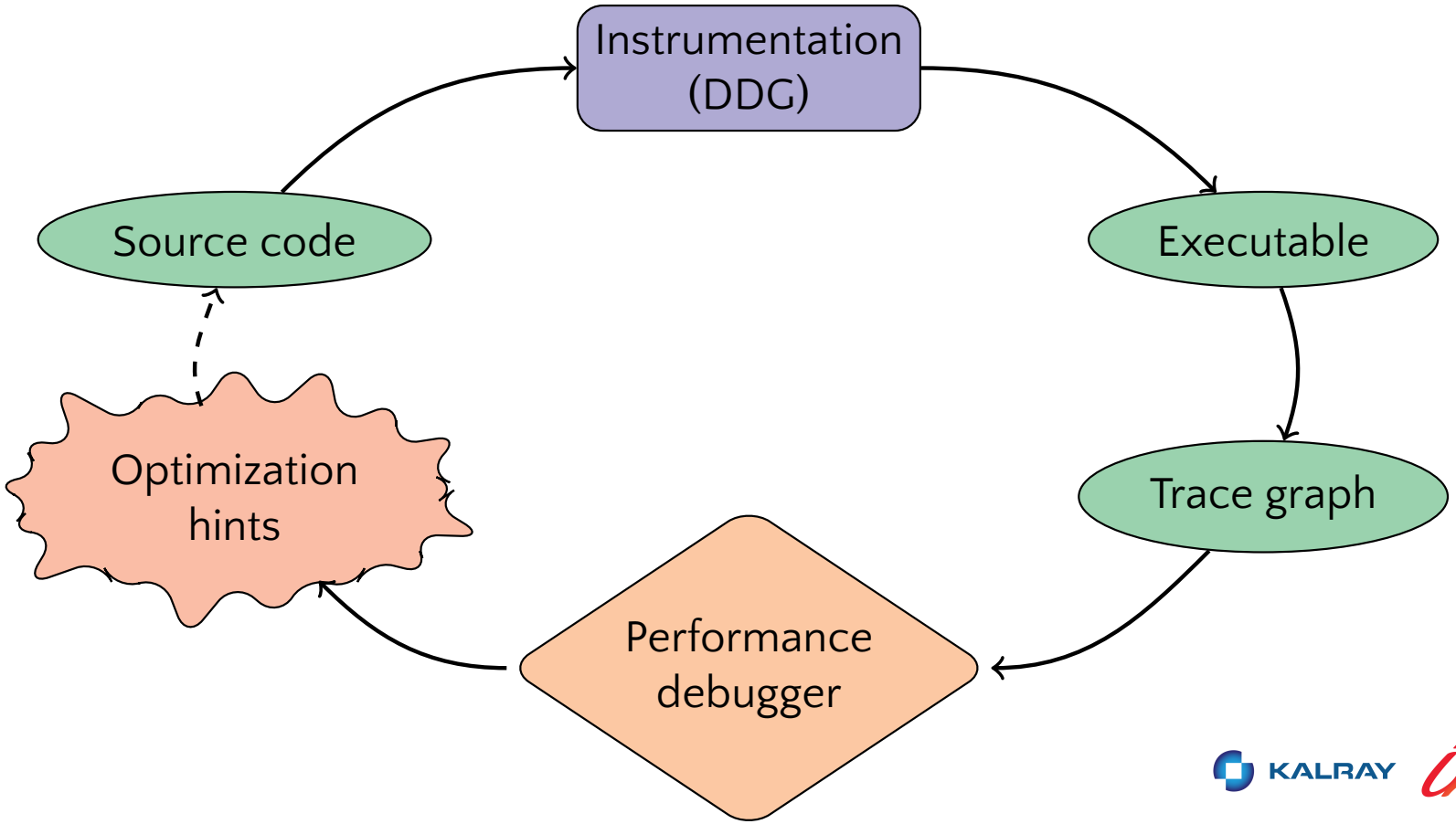
4 Performance debugging with convex graph partitioning



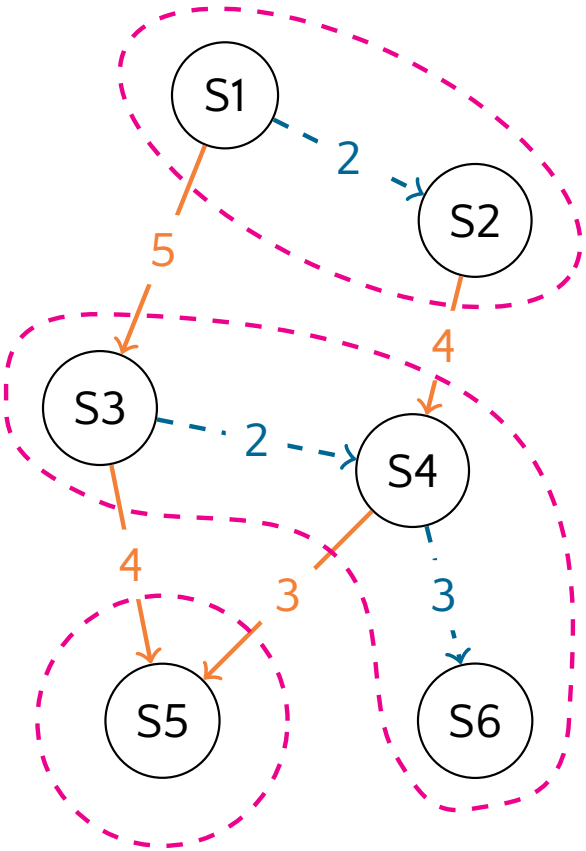
Overview

Performance debugging

Analysis of the execution of code in order to identify performance bottle-necks

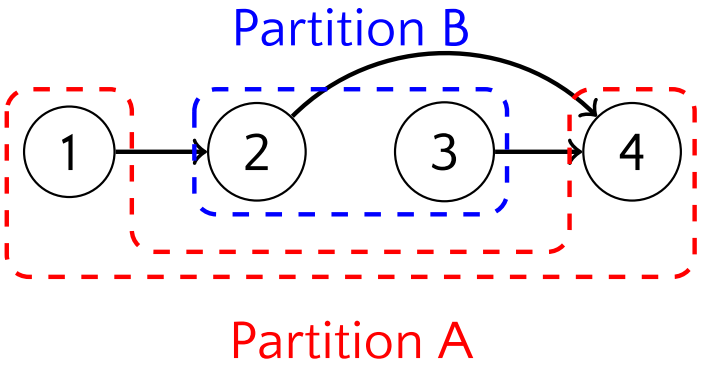


Convex partitioning



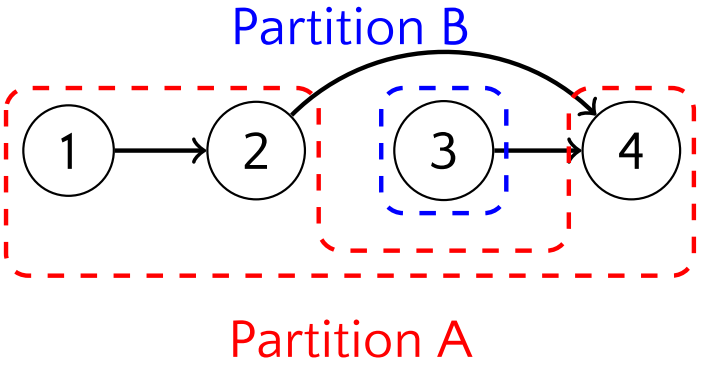
...is a convex partitioning

Non-convex



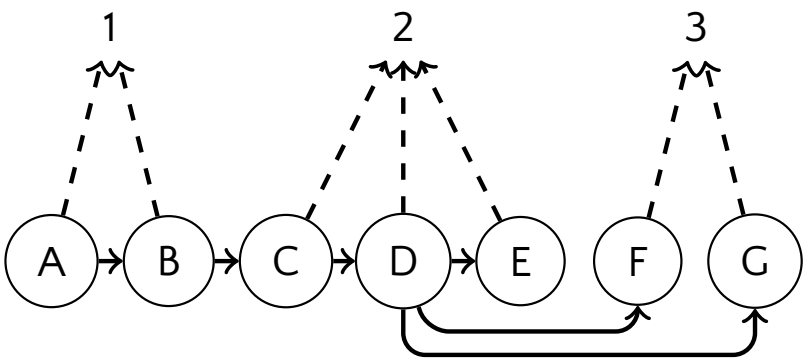
Offending path: 1 (A) → 2 (B) → 3 (A)

Convex

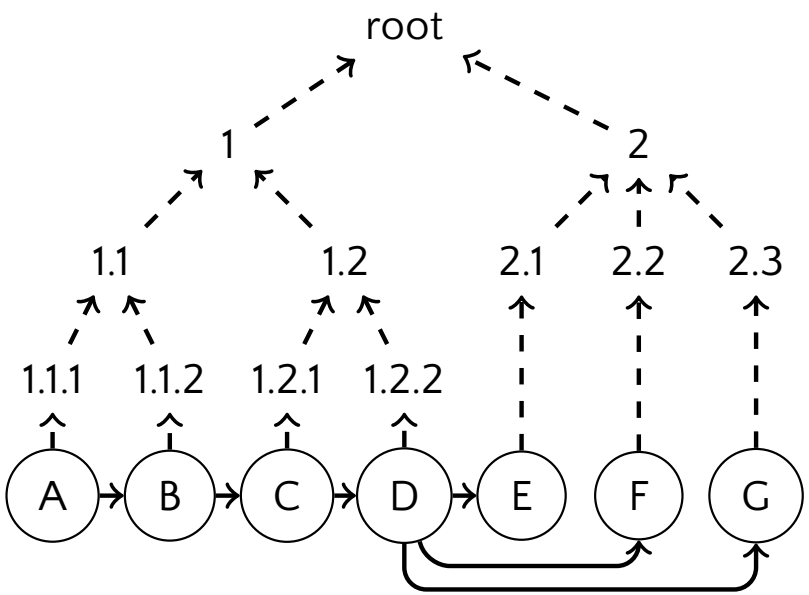


Partition schemes

k-way partition scheme



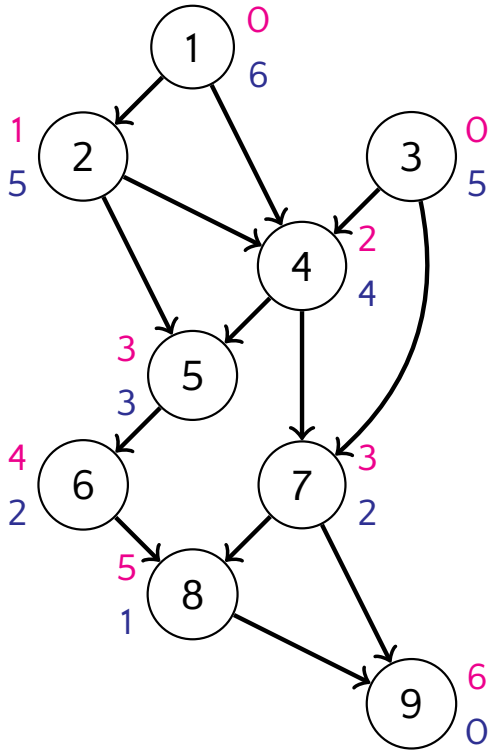
Hierarchical partition scheme



Max-distance criterion

Idea: For each vertex compute longest path originating at a source and terminating at a sink

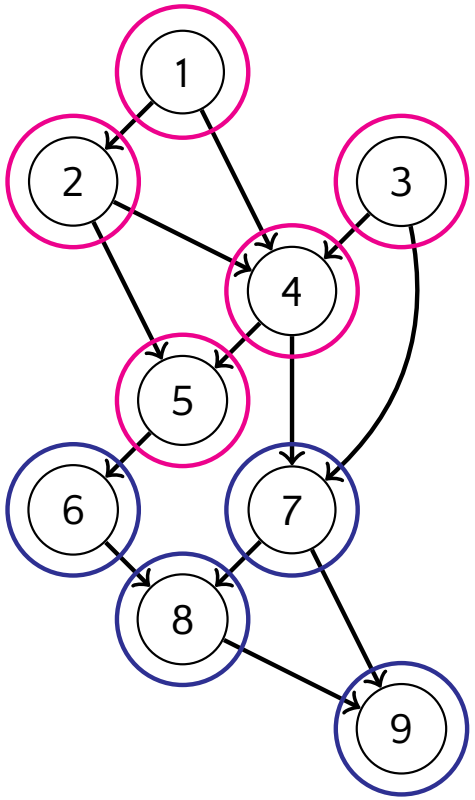
1. Source path $\leq$ Sink path
   $\rightarrow$ source partition
2. Source path $>$ Sink path
   $\rightarrow$ sink partition



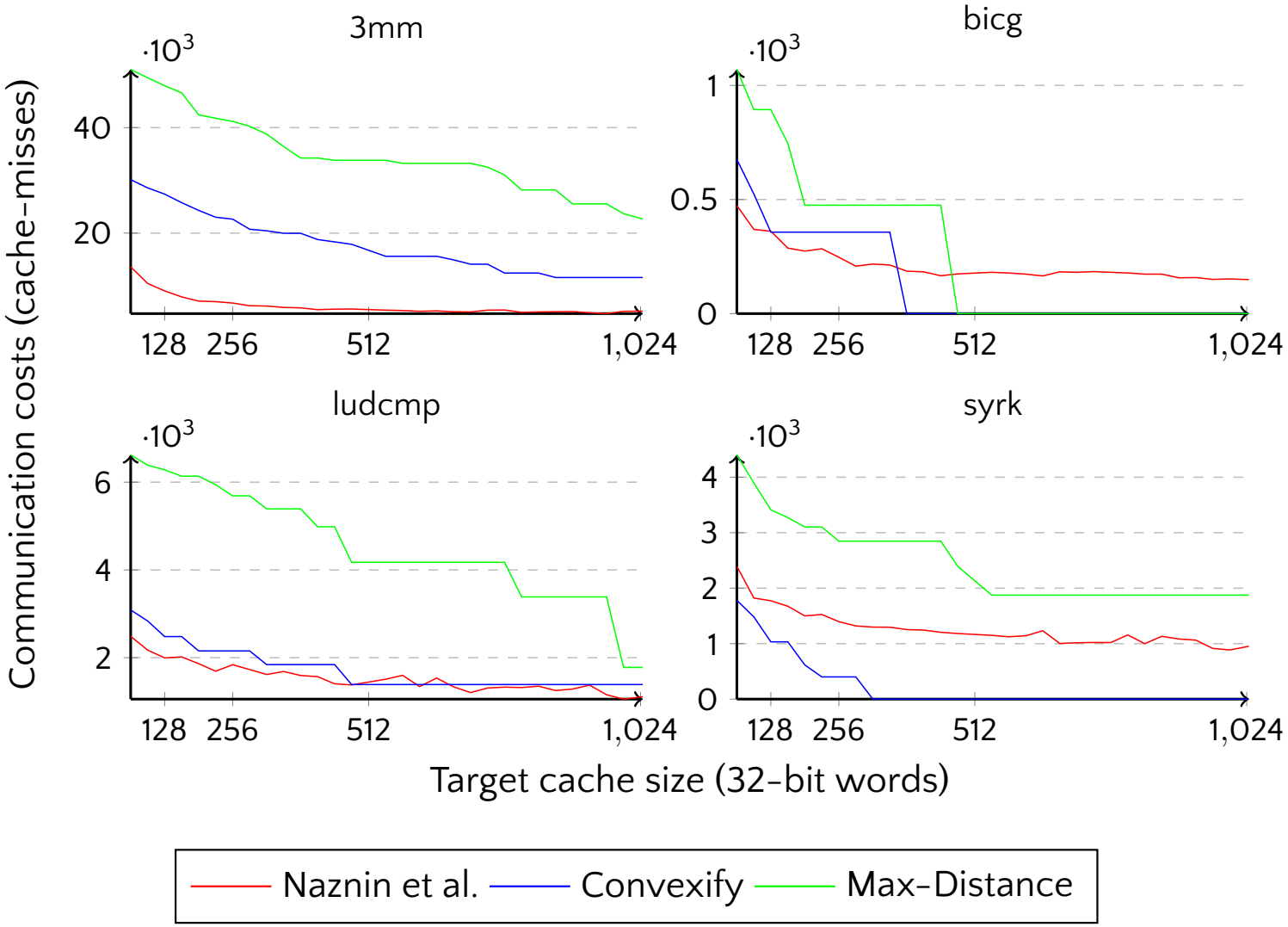
Max-distance criterion

Idea: For each vertex compute longest path originating at a source and terminating at a sink

1. Source path $\leq$ Sink path
   $\rightarrow$ source partition
2. Source path $>$ Sink path
   $\rightarrow$ sink partition

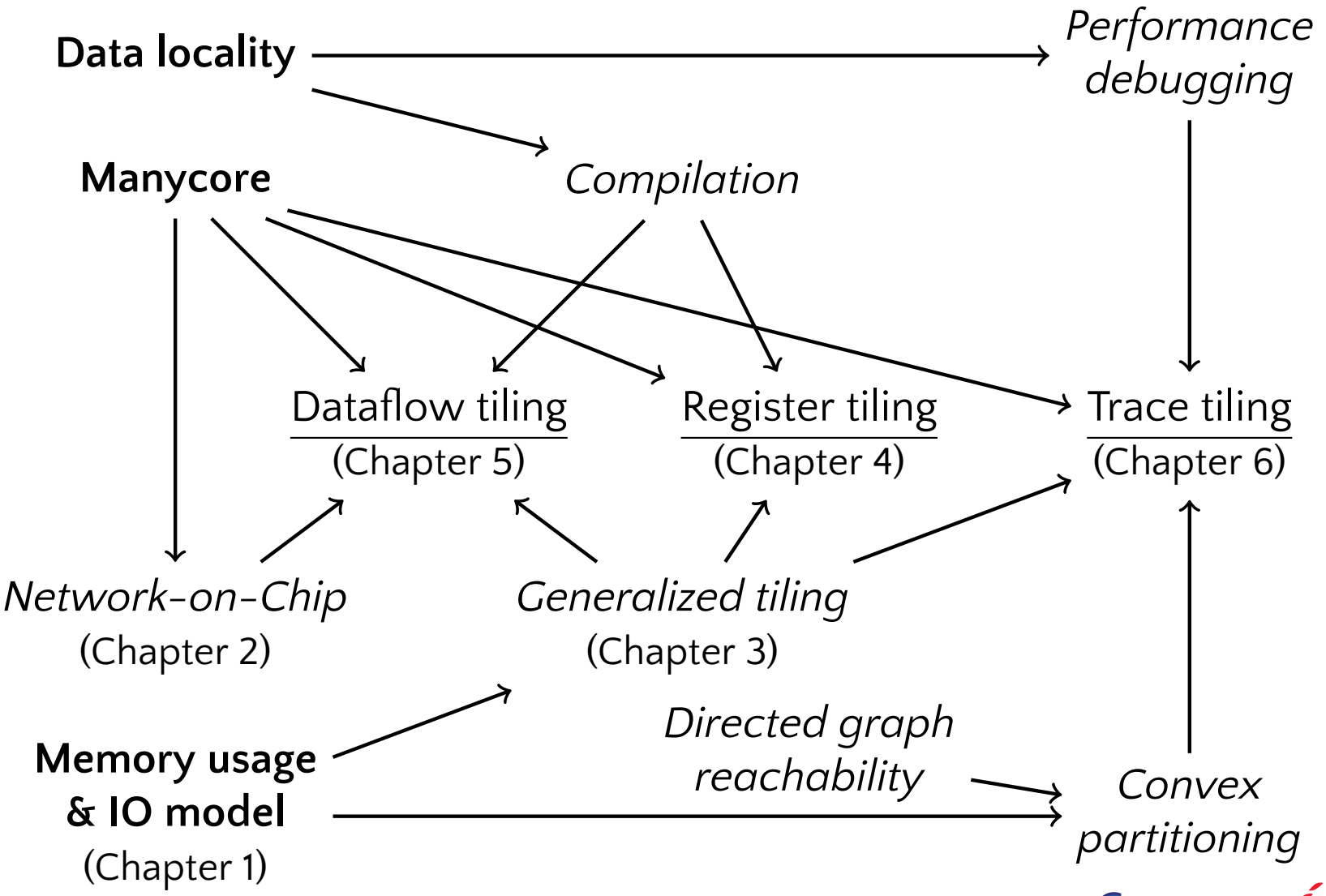


Experimental results



Conclusion & perspectives

Overview


Data locality
Performance debugging
Manycore
Compilation
Dataflow tiling
(Chapter 5)
Register tiling
(Chapter 4)
Trace tiling
(Chapter 6)
Network-on-Chip
(Chapter 2)
Generalized tiling
(Chapter 3)
Directed graph reachability
Memory usage & IO model
(Chapter 1)
Convex partitioning

Contributions

Operational intensity

Model to compute a close approximation of its value for arbitrary code

Generalized tiling

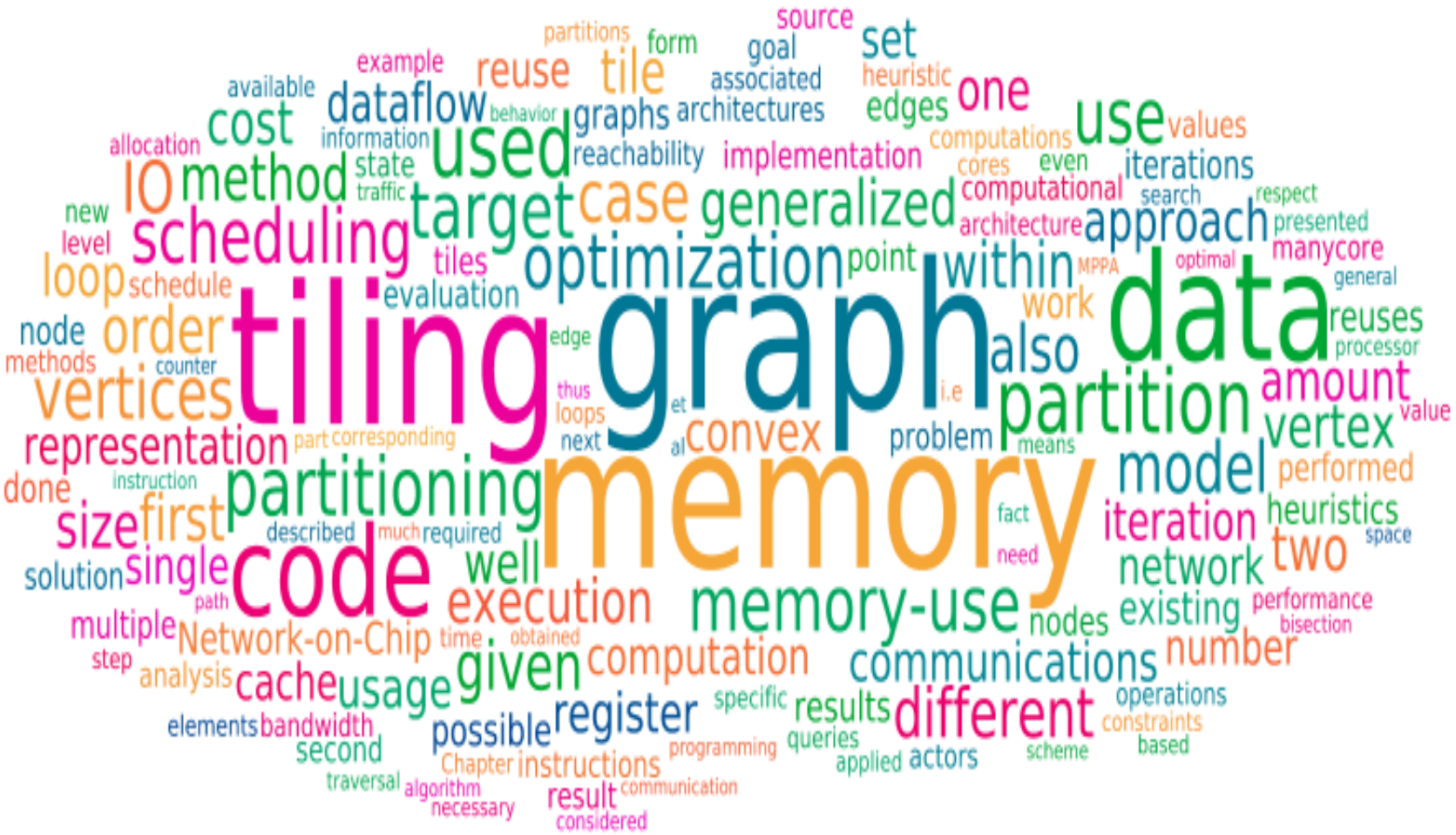
Formalization of an optimization problem and proposal of several heuristics to compute solutions

GraphUtilities library

Multi-threaded library for directed graph manipulation:
state-of-the-art reachability queries, convex partitioning & associated metrics ($\sim$ 5k lines)

Future work

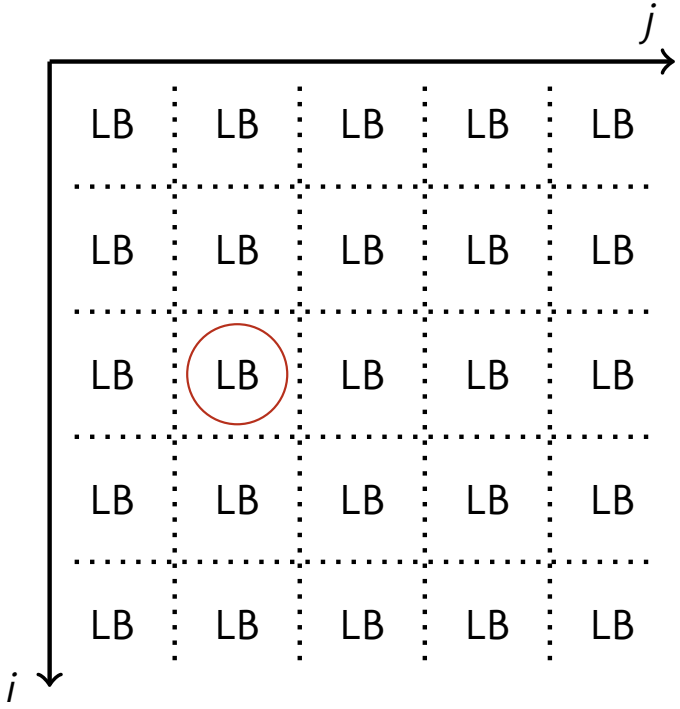
- Stabilize / rewrite LLVM implementation of generalized tiling
- Pursue implementation of dataflow tiling
- Expand generalized tiling formalization for more flexibility
- Consider parallel execution of tiles
- Continue improving hierarachical partition schemes
- ...


tiling
graph
memory
data
code
scheduling
optimization
partitioning
partition
generalized
target
case
used
method
IO
cost
dataflow
reuse
tile
set
one
use
values
order
vertices
loop
schedule
node
methods
counter
representation
part
corresponding
size
first
single
solution
multiple
Network-on-Chip
step
analysis
cache
usage
given
execution
well
convex
memory-use
computation
communications
number
different
results
register
possible
second
elements
bandwidth
Chapter
instructions
traversal
algorithm
necessary
result
communication
considered
programming
queries
applied
actors
scheme
based
constraints
operations
specific
nodes
existing
performance
bisection
network
two
iteration
heuristics
space
model
performed
amount
vertex
value
reuses
processor
manycore
general
presented
respect
approach
architecture
computational
search
iterations
even
cores
computations
implementation
reachability
graphs
architectures
edges
heuristic
associated
goal
form
partitions
source
example
available
allocation
information
behavior
state
traffic
new
level
tiles
evaluation
point
within
MPPA
optimal
work
also
edge
thus
loops
next
et
al
i.e
problem
means
fact
need
described
much
required
path
time
obtained
done
instruction
counter

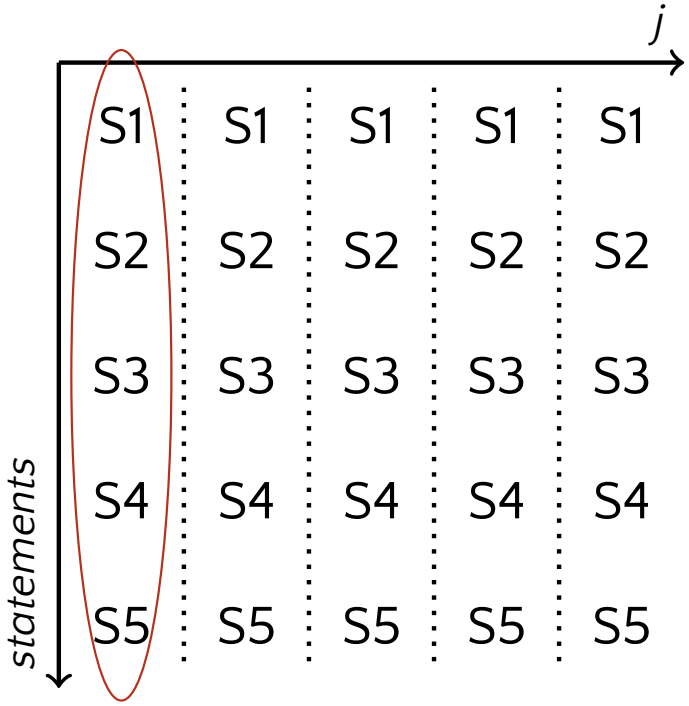
Backup slides

Generalized tiling in the polyhedral model?

Polyhedral model

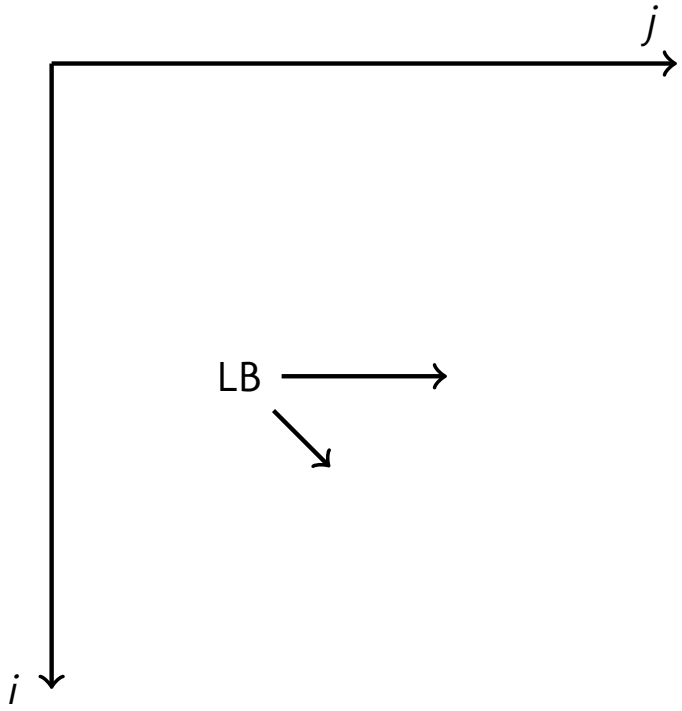


Memory-use graph

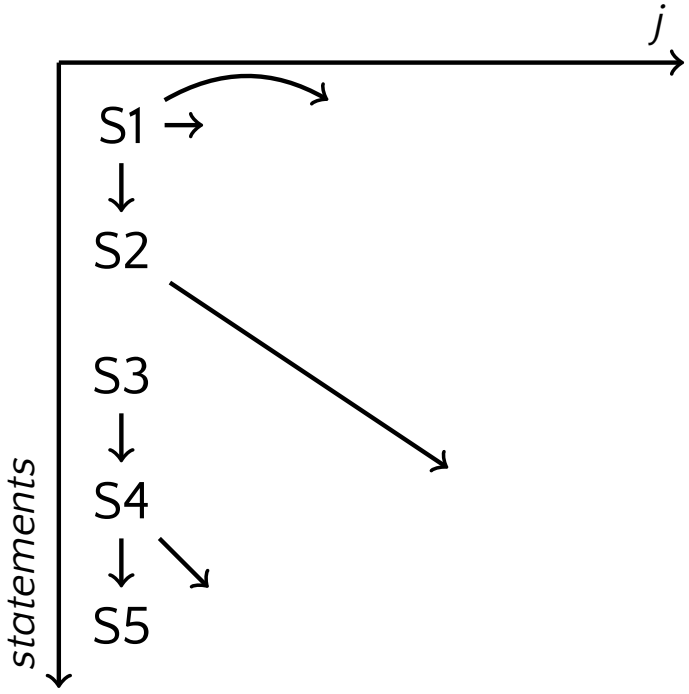


Generalized tiling in the polyhedral model?

Polyhedral model

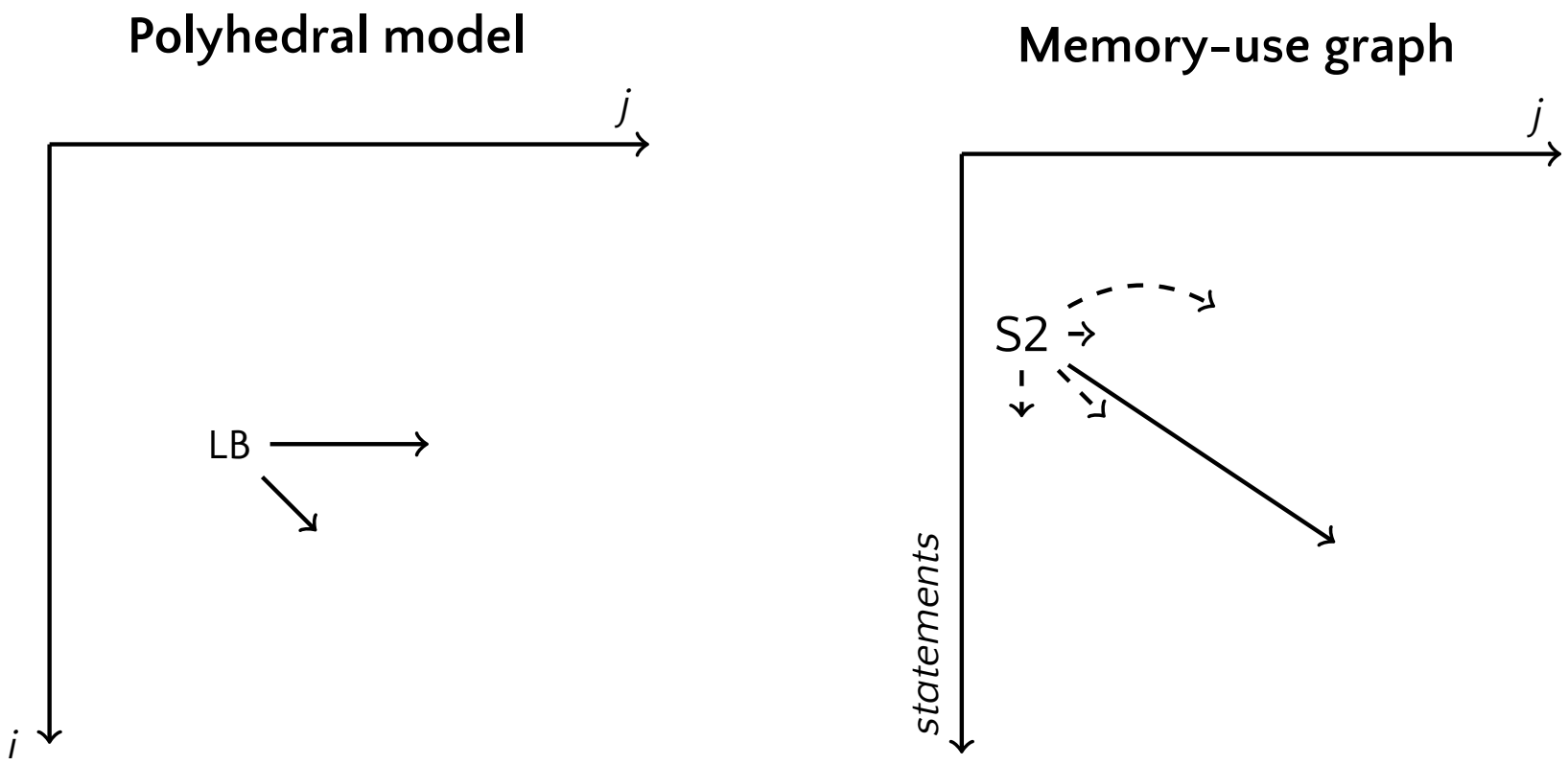


Memory-use graph



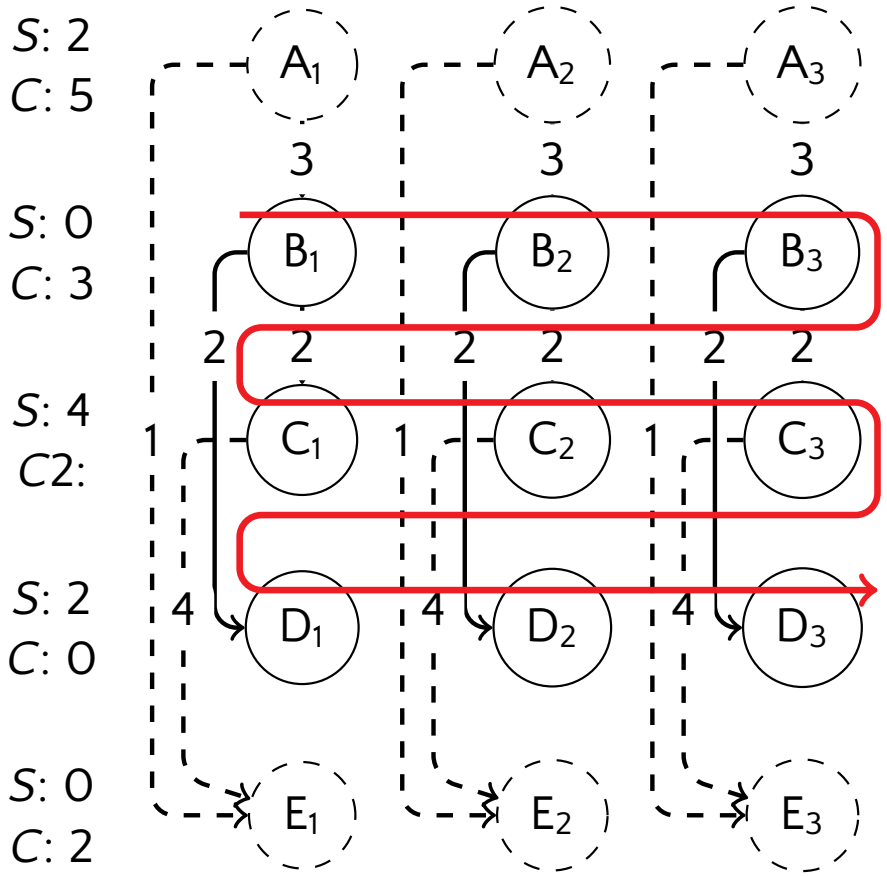
Need to create uniform reuse pattern for whole iteration space.
→ Duplicate all existing reuse edges for each point in iteration space?

Generalized tiling in the polyhedral model?



Theoretically possible but... more false reuse edges then true ones.
Very constrained optimization opportunities.

The influence of scheduling



What if we consider another schedule?

$$B_1 \to B_2 \to B_3 \to C_1 \to C_2 \to C_3 \to D_1 \to B_3 \to D_3$$

Amount of data stored in and between nodes changes...

7, 4, 11, ..., 16, 14, 16, ..., 6, 4, 4

...and so does the resulting **memory usage**: 16

Figure: Iterated **memory-use graph**

Transition to Σ-C

Context

Dataflow languages on a manycore processor

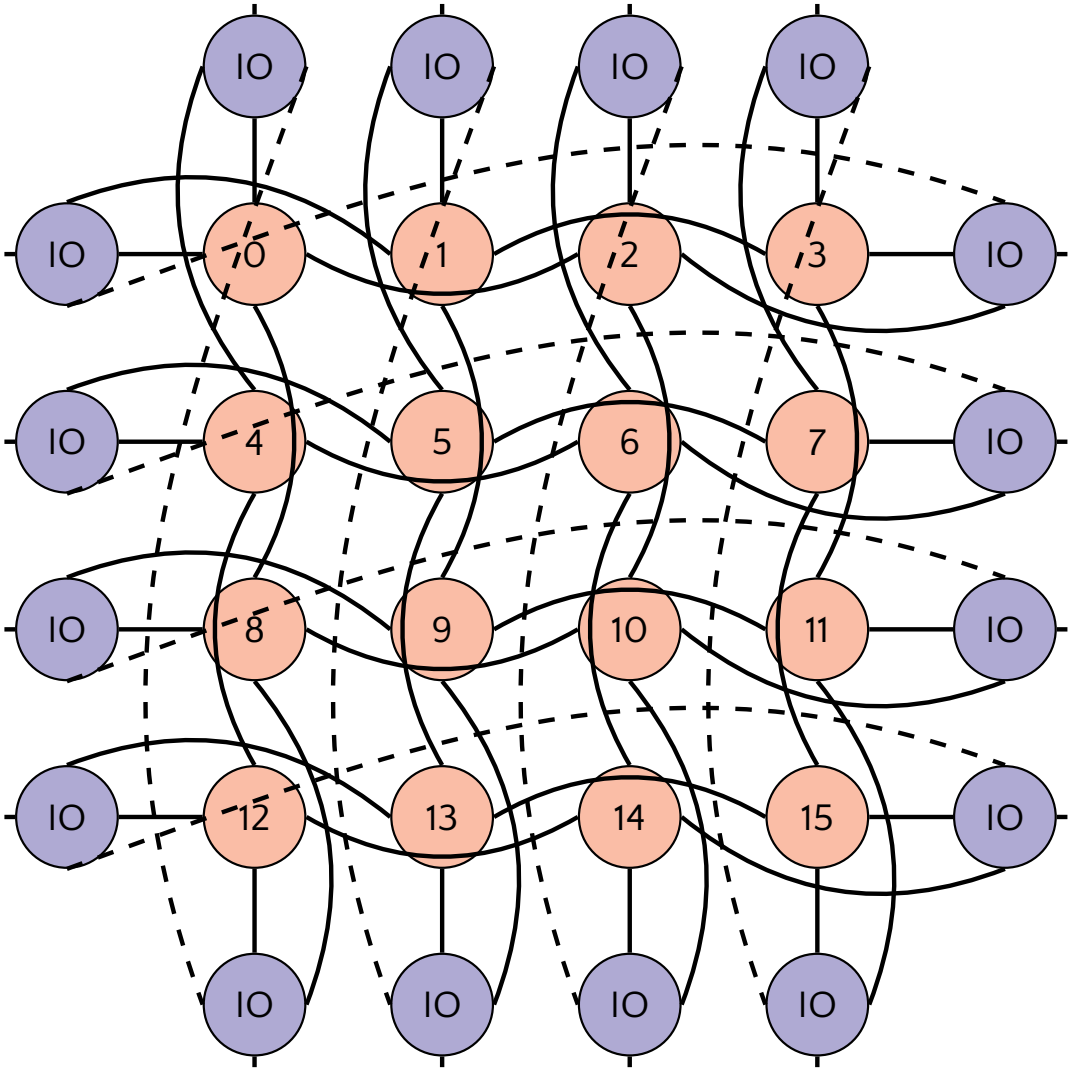
Illustration with the MPPA processor family by Kalray:

- 256 computational cores, 32 management cores
- 16 computation clusters of 17 cores each
- 4 IO clusters of 4 cores each
- Clusters each have own local memory
- Clusters linked through a Network-on-Chip

Distributed architecture

→ Dataflow programs must be mapped to ressources

Communications over the Network-on-Chip



Contributions:

- Network traffic model
- Redesign of bandwidth limiter (patent filed)
- NoC simulator ($\sim$ 1.5k lines)

Remaining challenges:

- Map actors to clusters with size constraints
- Combine memory and network models

Reachability querying

Goal

For a directed-acyclic graph and two of its vertices decide whether there exists a path from one to the other.

Reachability querying

Goal

For a directed-acyclic graph and two of its vertices decide whether there exists a path from one to the other.

Existing approaches:

- Naïve depth-first search (prohibitive for many queries)
- Compute full transitive-closure (prohibitive for large graphs)
- Pre-compute information reducing search time (many variants)

Reachability querying

Goal

For a directed-acyclic graph and two of its vertices decide whether there exists a path from one to the other.

Existing approaches:

- Naïve depth-first search (prohibitive for many queries)
- Compute full transitive-closure (prohibitive for large graphs)
- Pre-compute information reducing search time (many variants)

Contribution

1. Pre-indexation method based on reverse post-order traversals
2. Experimental results improving on state-of-the-art methods

Drawback: Index is very hard to maintain during graph partitioning

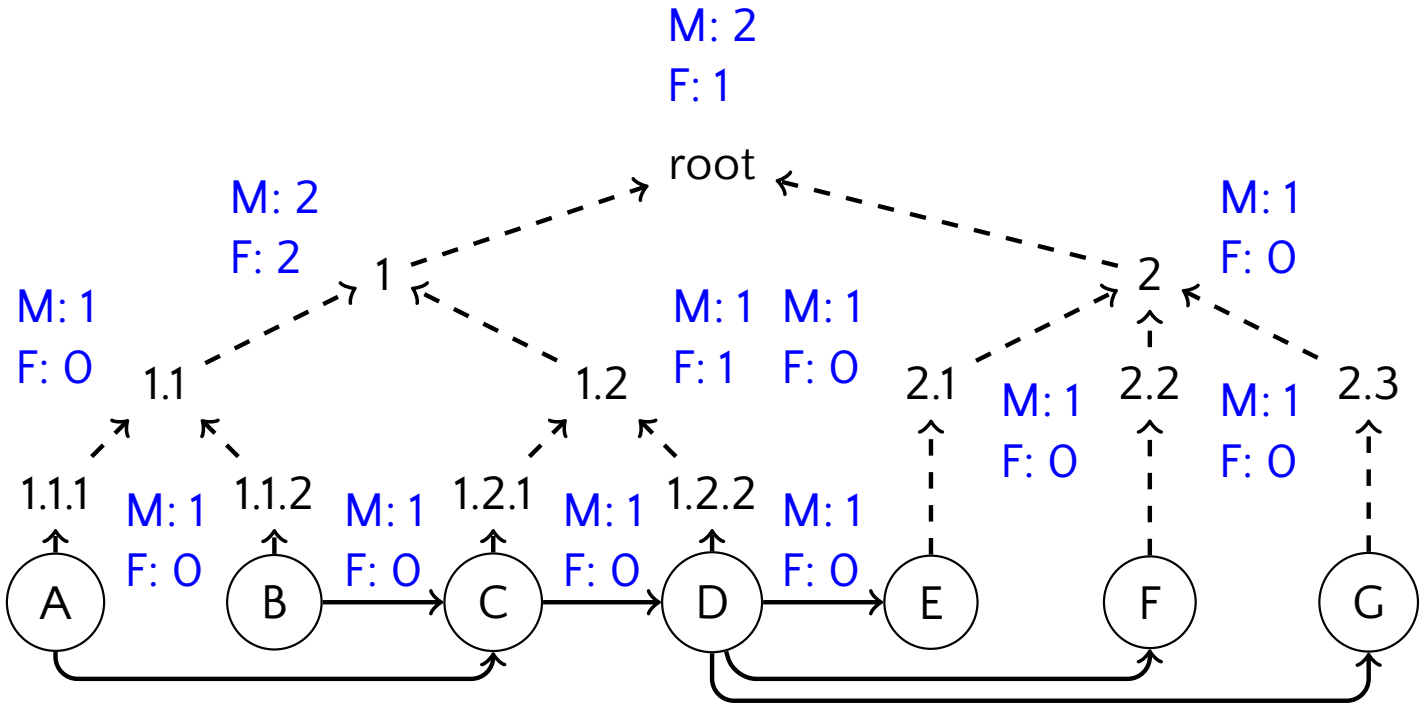
Memory usage & IO cost

k**-way partition scheme** – Hypergraph cut, like tiling

Hierarchical partition scheme – FrontierLive = hypergraph cut

$$\text{MemoryUse} = \max\left(\max_{\text{children}}\left(\text{MemoryUse}\right), \text{FrontierLive}\right)$$

M: MemoryUse F: FrontierLive



Partition tree balance

Convexify partition trees

Benchmark	Min	Max	Std.Dev
adi	15	19	0.19
bicg	9	13	0.17
durbin	10	12	0.24
fdtd-2d	15	20	0.20
gemm	12	17	0.27
gesummv	11	13	0.13
jacobi-2d	15	20	0.26
ludcmp	11	16	0.15
mvt	12	14	0.15
seidel-2d	12	17	0.27
syr2k	13	17	0.24
syrk	12	15	0.12
trisolv	9	11	0.25
trmm	14	17	0.22

Max-distance partition trees

Benchmark	Min	Max	Std.Dev
adi	4	30	13.71
bicg	3	21	10.60
durbin	2	29	19.97
fdtd-2d	2	40	22.89
gemm	3	27	15.17
gesummv	4	23	11.06
jacobi-2d	2	35	18.74
ludcmp	2	42	23.71
mvt	4	27	13.56
seidel-2d	4	46	39.66
syr2k	3	23	14.05
syrk	2	23	12.58
trisolv	2	26	19.90
trmm	3	29	17.91