SNPD2026

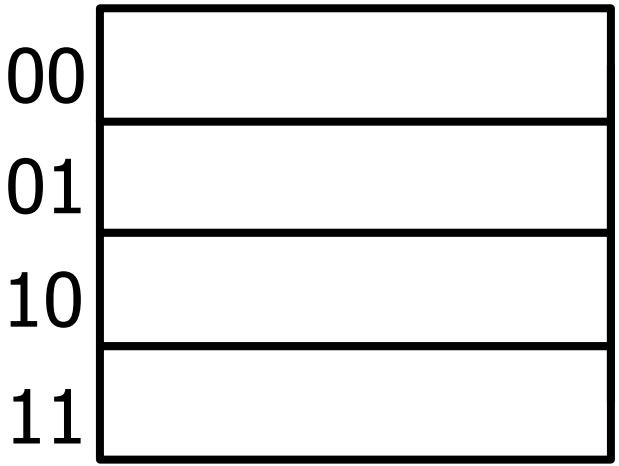
Improving Cache Locality via Multidimensional Store Reordering

Yudai Yamagishi and Yasunobu Sumikawa
Takushoku University, Japan

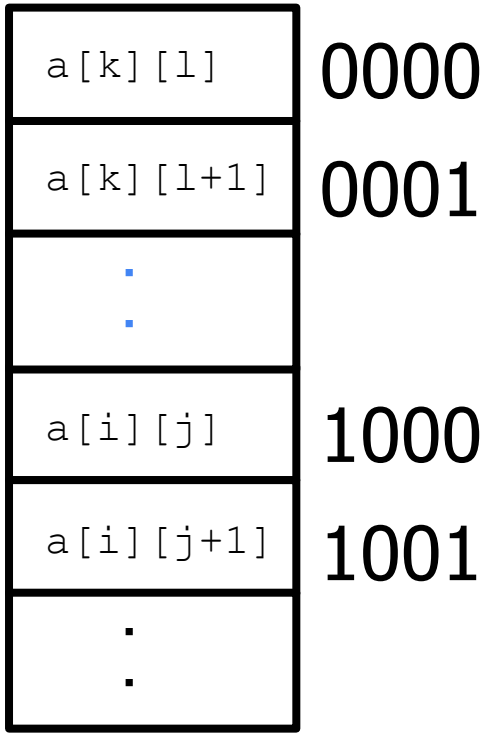
Motivation | write miss at cache memory

```
main(){
  a[i][j]=...
  a[k][l]=...
  a[i][j+1]=...
}
```

C program



Cache memory

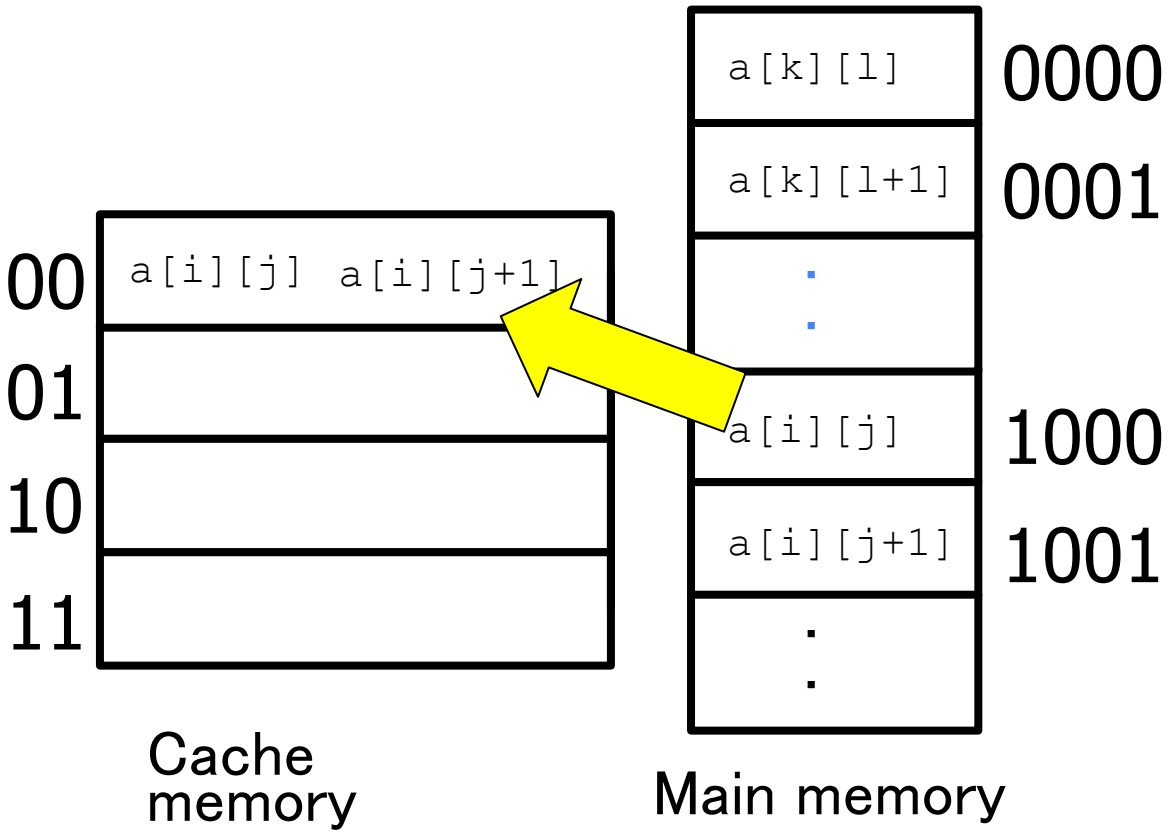


Main memory

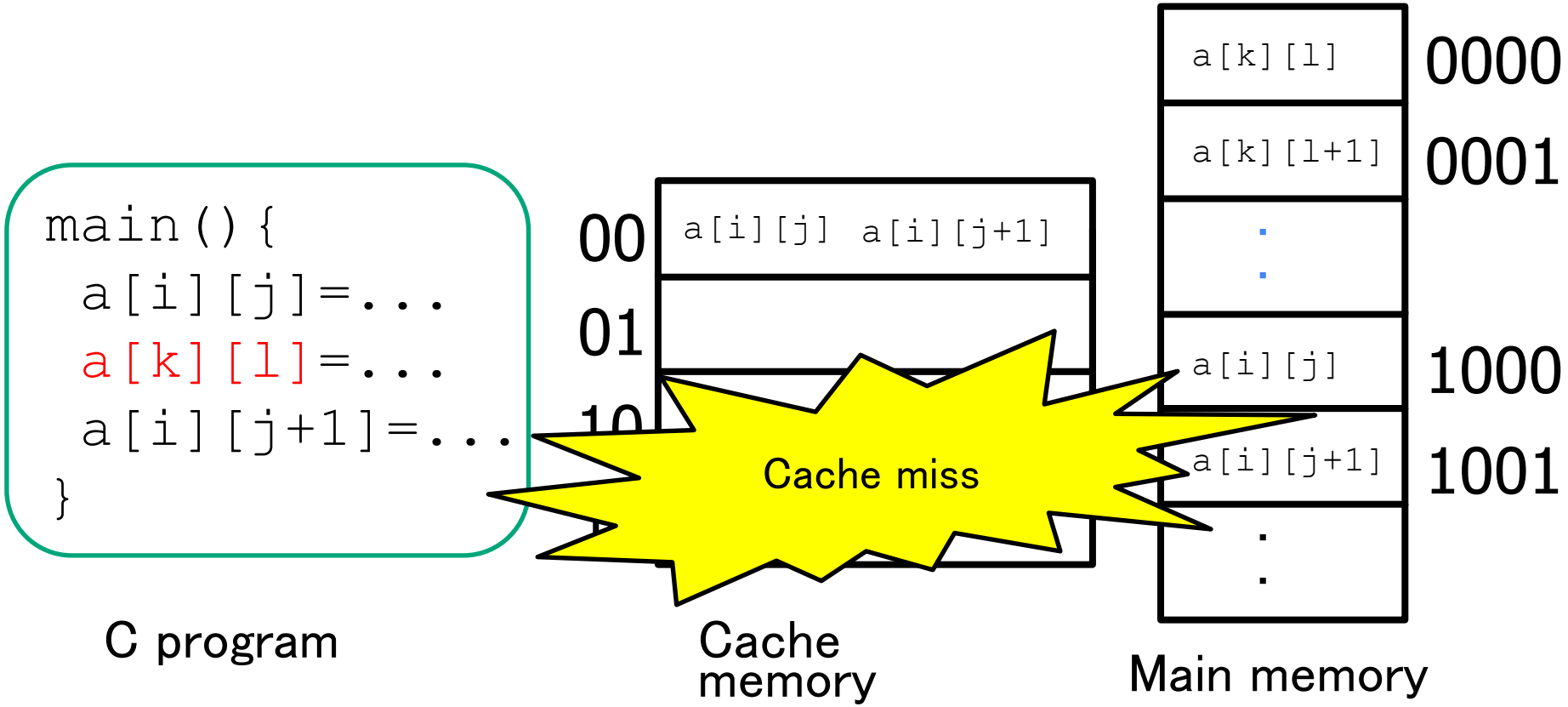
Motivation | write miss at cache memory

```
main(){
  a[i][j]=...
  a[k][l]=...
  a[i][j+1]=...
}
```

C program



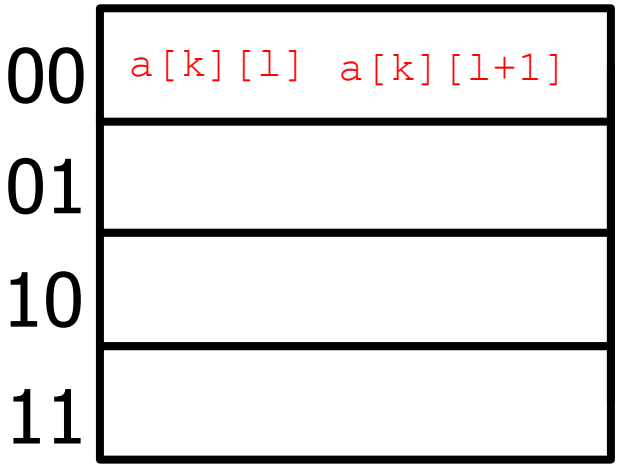
Motivation | write miss at cache memory



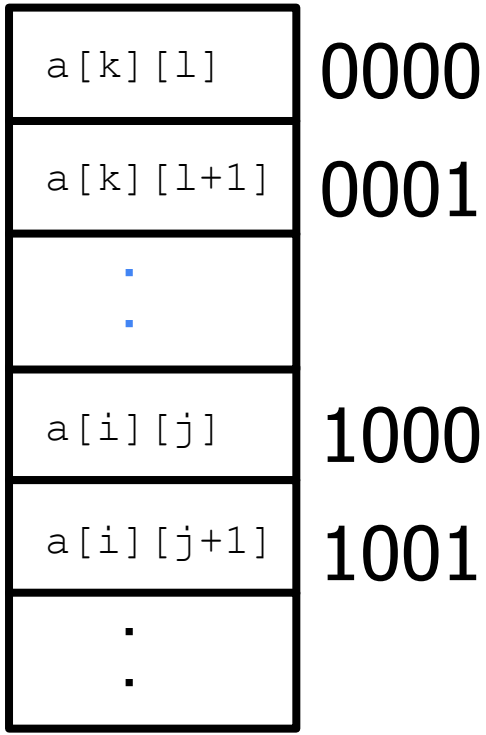
Motivation | write miss at cache memory

```
main(){
  a[i][j]=...
  a[k][l]=...
  a[i][j+1]=...
}
```

C program

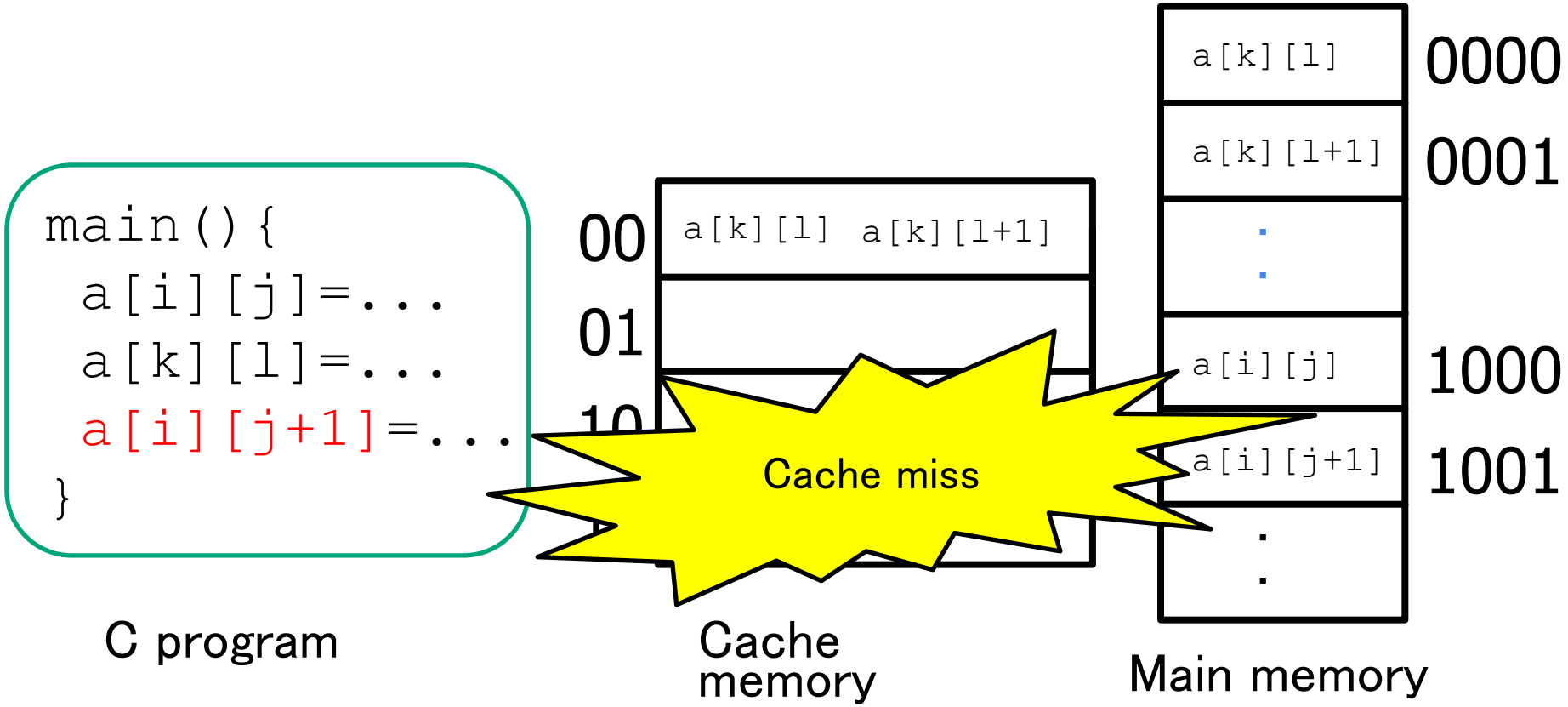


Cache memory



Main memory

Motivation | write miss at cache memory

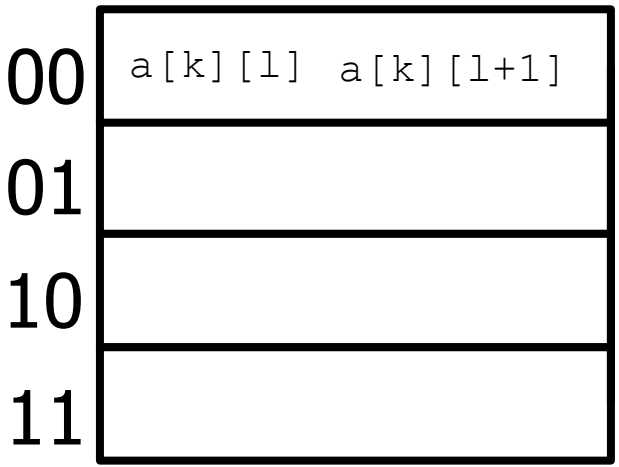


Motivation | write miss at cache memory

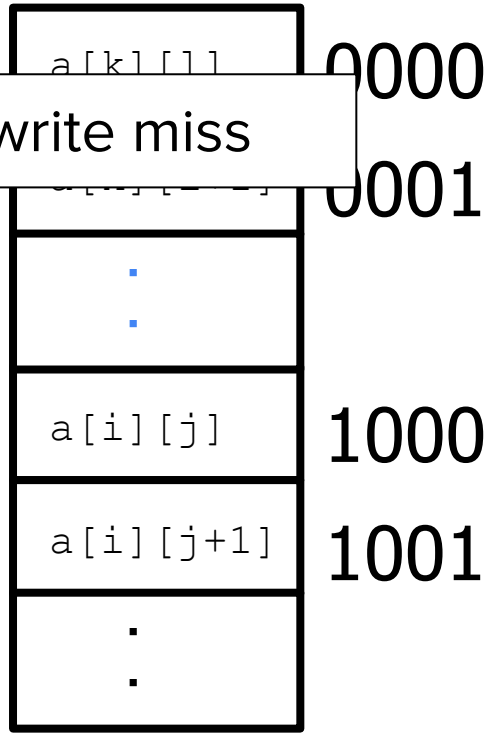
Reordering the executions may prevent the write miss

```
main(){
  a[i][j]=...
  a[k][l]=...
  a[i][j+1]=...
}
```

C program



Cache memory



Main memory

Algorithm

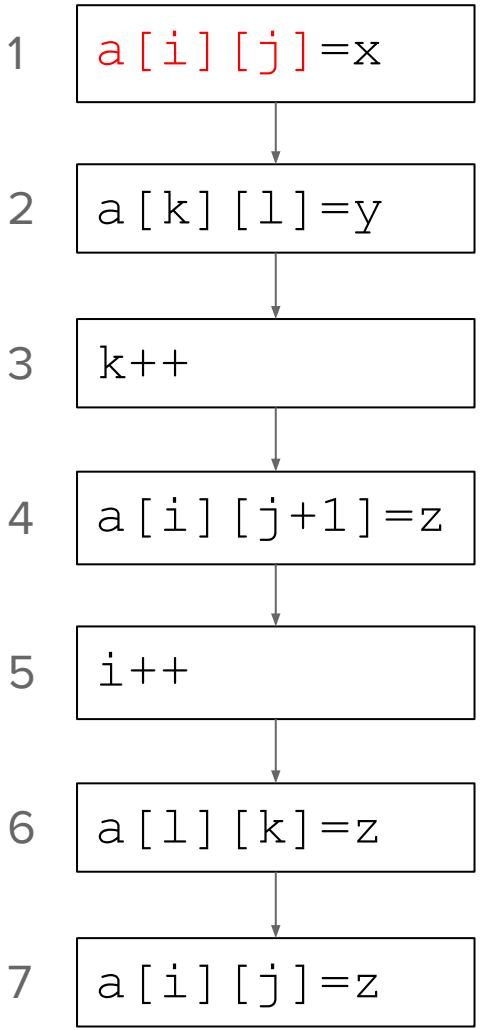
- We propose a code sinking algorithm to arrange array access
  - Reorders store statements to maintain consecutive accesses to
    - the same array, while
    - maximizing the number of consecutive matching higher-order indexes

- Designed as a compiler code optimization
  - Multi Dimensional Global Store statement Aggregation (MDGSA)

Algorithm

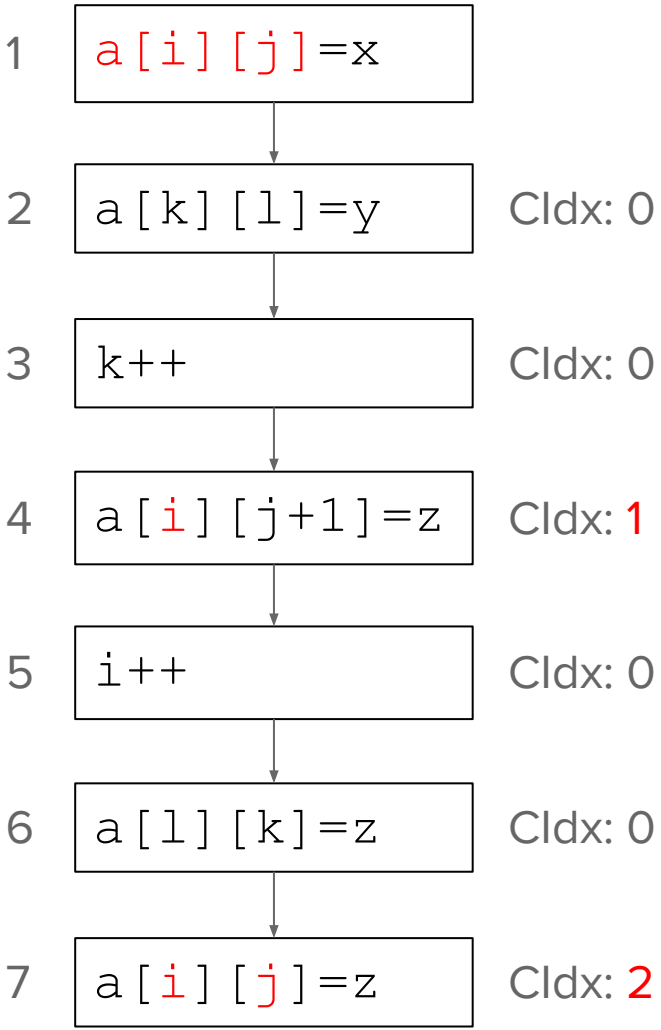
Assumption: a CFG has been created

1. Traverse the CFG in reverse topological order
2. For each node containing a store statement:
    a. Analyze local and global predicates
    b. Move the store statement toward the end node if necessary



Algorithm | local predicates

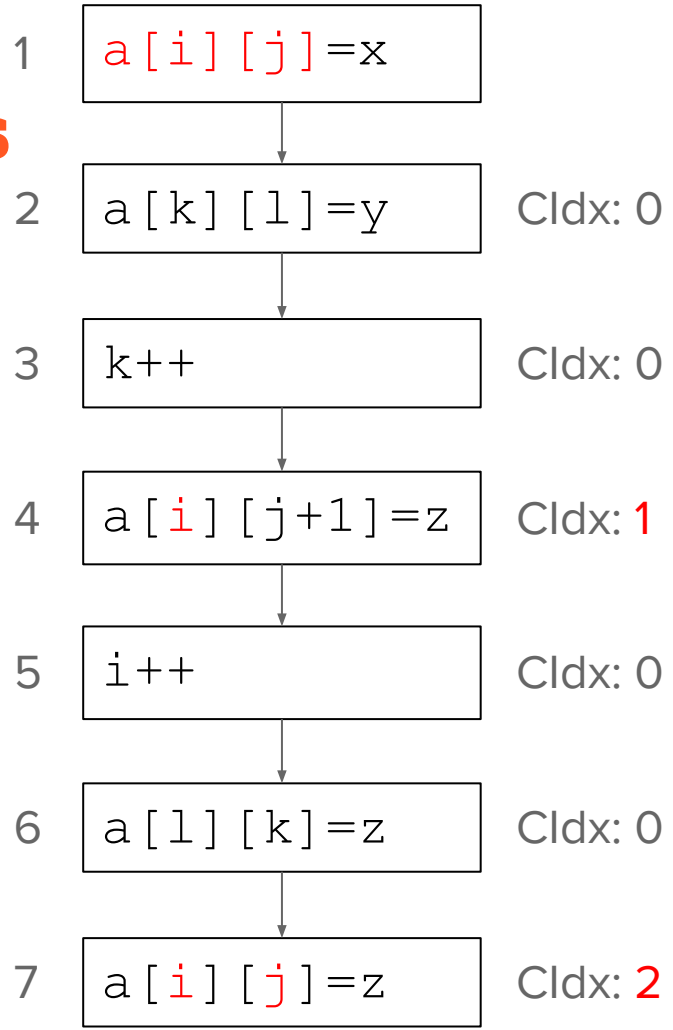
1. Calculate *Cldx*
    a. Count consecutive higher-order index matches between the array to be moved and other arrays



Algorithm | global predicates

1. Calculate *CIF*
   a. Propagate the number of matches calculated by *Cldx* to the succeeding node
   b. Repeat for each store statement accessing the array

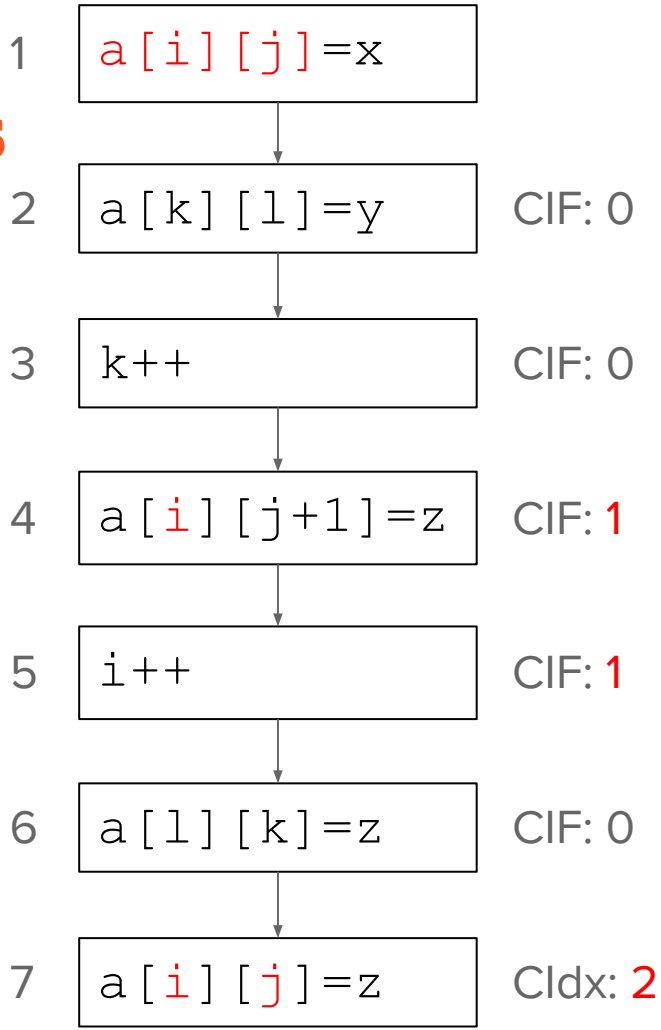
2. Define *CDim*
   a. CDim identifies the range over which CIF remains non-decreasing



Algorithm | global predicates

1. Calculate *CIF*
   a. Propagate the number of matches calculated by *CIdx* to the succeeding node
   b. Repeat for each store statement accessing the array

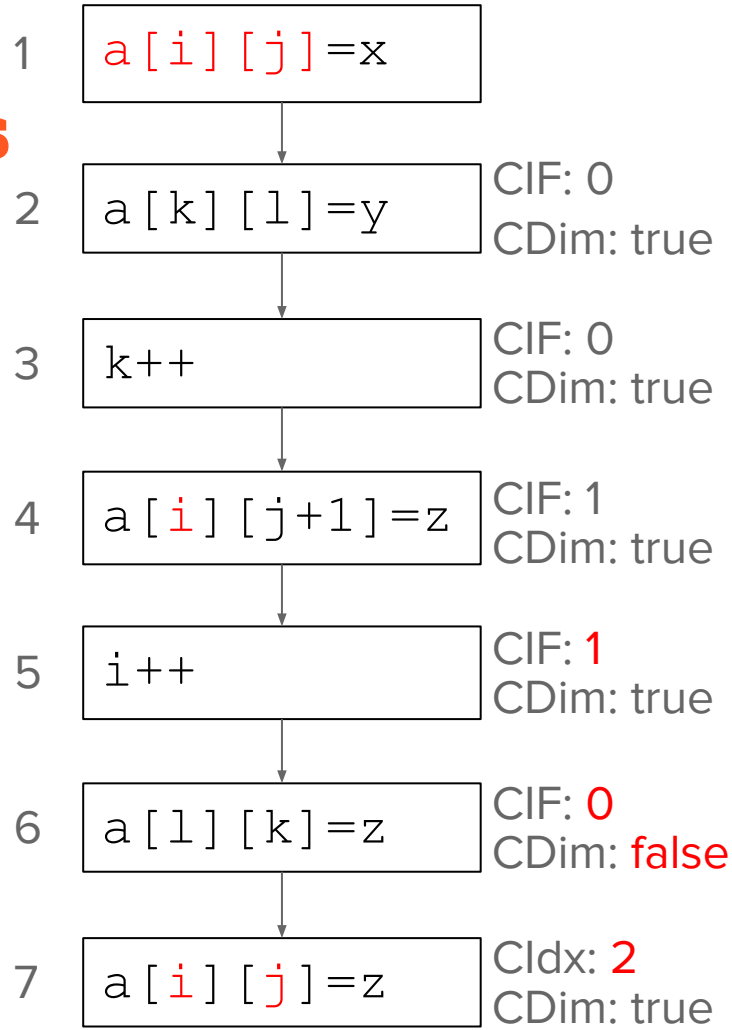
2. Define *CDim*
   a. CDim identifies the range over which CIF remains non-decreasing



Algorithm | global predicates

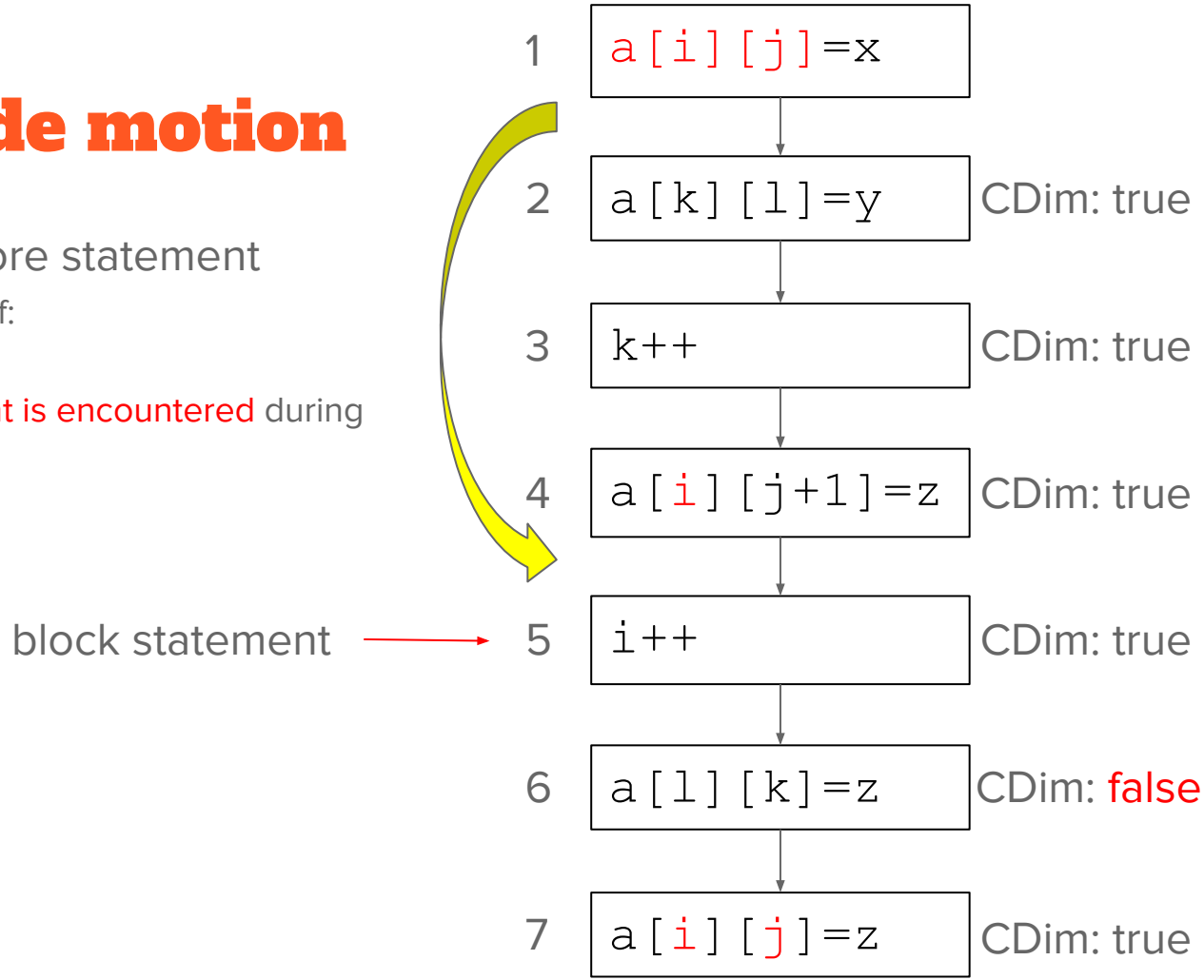
1. Calculate *CIF*
   a. Propagate the number of matches calculated by *CIdx* to the succeeding node
   b. Repeat for each store statement accessing the array

2. Define *CDim*
   a. CDim identifies the range over which CIF remains non-decreasing

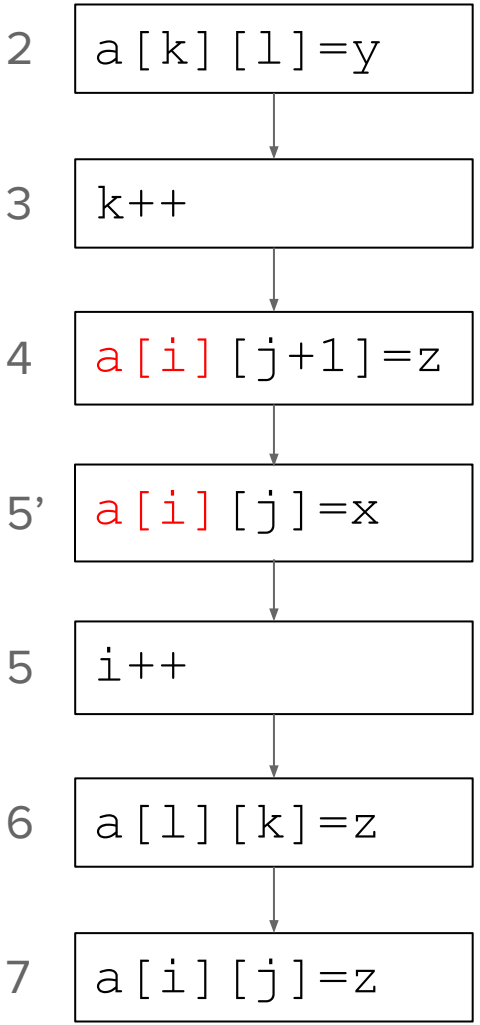


Algorithm | code motion

1. Delaying the target store statement
    a. Stop the code motion if:
        i. CDim is false, or
        ii. a block statement is encountered during the delay



Algorithm | result



Experimental Evaluation | Settings

- Implementation: COINS compiler
- Baselines
  - Partial Dead code Elimination (PDE): eliminating unused variables by delaying their definisions
  - Global Store statement Aggregation (GSA): extending PDE to delay array accesses to prevent write miss by making the same array accesses continuously
- Benchmark programs
  - **Matrix:** computes the product of two matrices.
  - **Vector**: computes the sum of two 3-dimensional vectors.
  - **Tensor**: computes the sum of two 3-dimensional tensors.
    - We applied loop unrolling to ensure that four store statements appeared

Experimental Evaluation | Data Collection

- Machine
  - OS: Ubuntu 64-bit
  - CPU: Intel Core i7-11700 2.50GHz
  - Cache:
    - L1D: 384 KiB
    - L1I: 256 KiB
    - L2: 4 MiB
    - L3: 16 MiB
- Cache-miss measurement
  - Using the perf command
    - A performance analysis tool for Linux
- Measurement procedure
  - Execute each program 10 times
  - Collect the number of cache misses and the execution times of the objective codes
  - Report the average values in the evaluation results

Experimental Evaluation | Total cache miss

Program	A. PDE	B.GSA	C.**MDGSA**	(A-C)/A	(B-C)/B
Matrix	103,638	101,913	101,096	2.45%	0.80%
Vector	108,517	106,786	99,975	7.87%	6.38%
Tensor	75,173	73,533	72,840	3.10%	0.94%

MDGSA is the best for the three programs

Experimental Evaluation | LLC store miss

Program	A. PDE	B.GSA	C.**MDGSA**	(A-C)/A	(B-C)/B
Matrix	82,281	82,307	82,362	-0.10%	-0.07%
Vector	83,078	83,345	82,317	0.92%	1.23%
Tensor	59,015	58,152	58,169	1.43%	-0.03%

Experimental Evaluation | Execution times (sec)

Program	A. PDE	B.GSA	C.**MDGSA**	(A-C)/A	(B-C)/B
Matrix	465,437	466,091	459,501	1.28%	1.41%
Vector	649.5	643.4	610.1	6.07%	5.18%
Tensor	654.9	657.8	641.8	2.00%	2.43%

MDGSA is the best for the three programs

Discussion

- MDGSA reduced cache misses compared with PDE and GSA in all programs
  - Total cache misses: 6–8% reduction for **Vector**
  - Store misses: ~1% reduction in **Vector** and **Tensor**
- MDGSA improved execution time compared with PDE and GSA in all programs

- These results indicate that code aggregation for multidimensional arrays with MDGSA improves cache utilization and execution performance

Conclusions and Future works

- Proposed MDGSA, a cache optimization algorithm for multidimensional arrays
  - Moves store statements to maximize matching of higher-dimensional indexes
  - Considers array indexes, in addition to array names, unlike existing methods
- Evaluation
  - Compared MDGSA with PDE and GSA on three programs
  - MDGSA reduced cache misses across all cache levels
  - Improved execution performance compared with existing methods
- Future work
  - Jointly move load and store statements
  - Integrate hardware-based profiling information on actual write misses

Thank you for your attention