Superloop Scheduling: Loop Optimization via Direct Statement Instance Reordering

Cedric Bastoul Alain Ketterlin Vincent Loechner

University of Strasbourg & Inria CAMUS team

IMPACT '23, Toulouse

University of Strasbourg

Inria

Outline

University of Strasbourg

Inria

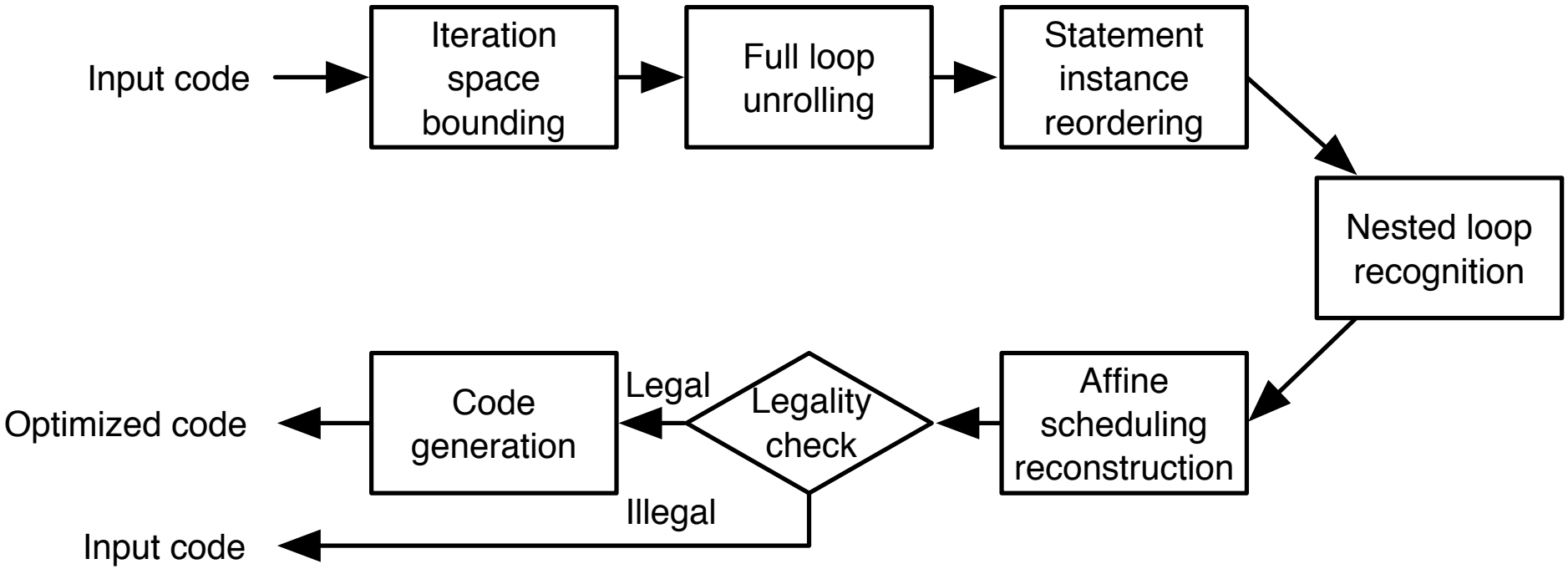
Affine Scheduling Algorithms

- By solving systems of affine constraints:
  - Proposed by Paul Feautrier (multidimensional time, 1992)
  - Improved by Uday Bondhugula (PLuTo, 2008)
  - Many variants that differ in the way the affine constraint system is built
- By composition of basic primitives:
  - Suggested by W. Kelly and W. Pugh (1993),
  - Improved by many, e.g., Baghdadi et al. (Tiramisu, 2018).

Affine Scheduling Algorithms

- By solving systems of affine constraints:
  - Limited by the affine representation
  - Struggle to find a good solution in a huge space
- By composition of basic primitives:
  - Limited by the set of considered primitives
  - Or subject to combinatorial explosion
- By loop reconstruction, from finest-grain statement instance reordering $\rightarrow$ superloop scheduling
  - Closer to a manual optimization by an expert

Overview of Superloop Scheduling

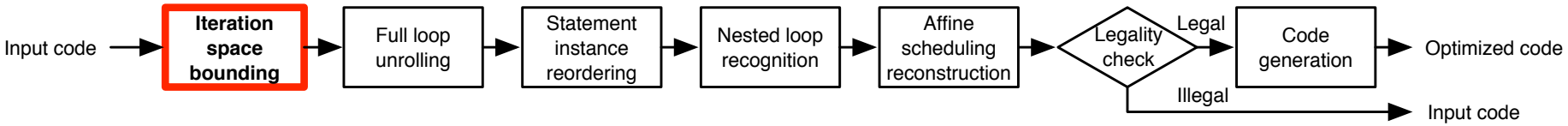


Outline

University of Strasbourg

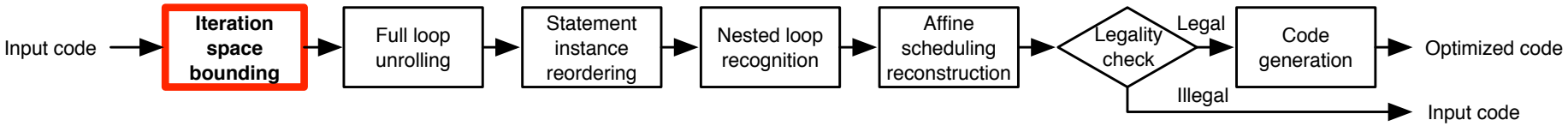
Inria

Step a. Iteration Space Bounding



- Instantiate parameters
- Artificially bound the loop to get a $block^d$ box

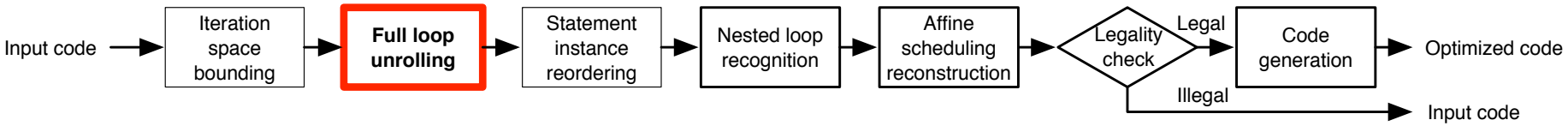
Step a. Iteration Space Bounding



Example: matrix product

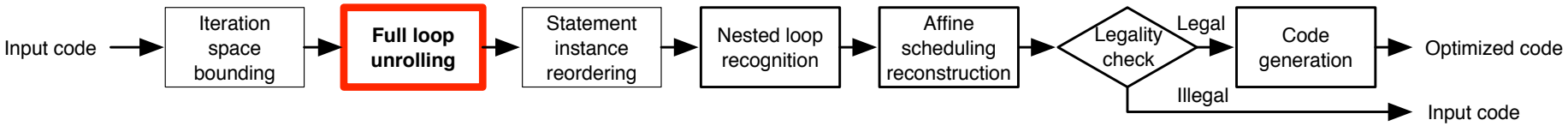
```
for (i = 0; i < 2; i++)
  for (j = 0; j < 2; j++) {
    C[i][j] = 0.;                          // S0
    for (k = 0; k < 2; k++)
      C[i][j] += A[i][k] * B[k][j]; // S1
  }
```

Step b. Full Loop Unrolling



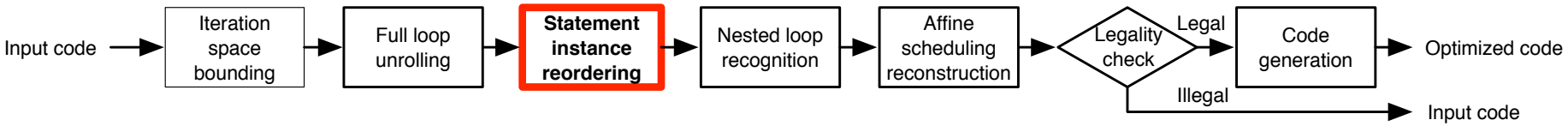
- Unroll the code to a sequence of statement instances

Step b. Full Loop Unrolling




```
C[0][0] = 0;                       // S0  0  0
C[0][0] += A[0][0] * B[0][0];      // S1  0  0  0
C[0][0] += A[0][1] * B[1][0];      // S1  0  0  1
C[0][1] = 0;                       // S0  0  1
C[0][1] += A[0][0] * B[0][1];      // S1  0  1  0
C[0][1] += A[0][1] * B[1][1];      // S1  0  1  1
C[1][0] = 0;                       // S0  1  0
C[1][0] += A[1][0] * B[0][0];      // S1  1  0  0
C[1][0] += A[1][1] * B[1][0];      // S1  1  0  1
C[1][1] = 0;                       // S0  1  1
C[1][1] += A[1][0] * B[0][1];      // S1  1  1  0
C[1][1] += A[1][1] * B[1][1];      // S1  1  1  1
```

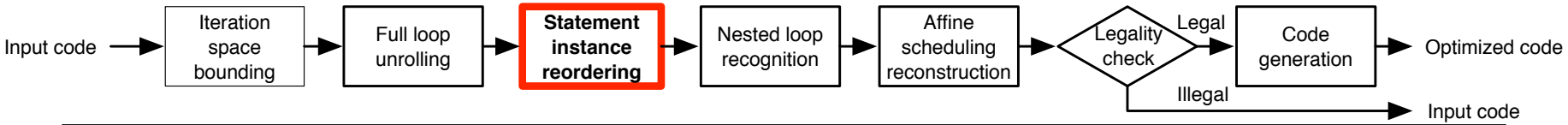
Step c. Statement Instance Reordering



- Low-level statement instance reordering
- Trivial parallelism extraction
- Superword level parallelization [Mendis and Amarasinghe, 2018]

\+ Enforce regularity

Step c. Statement Instance Reordering




```
//thread 0
C[0][0] = 0;                          // S0  0  0
C[0][1] = 0;                          // S0  0  1
C[0][0] += A[0][0] * B[0][0]; // S1  0  0  0
C[0][1] += A[0][0] * B[0][1]; // S1  0  1  0
C[0][0] += A[0][1] * B[1][0]; // S1  0  0  1
C[0][1] += A[0][1] * B[1][1]; // S1  0  1  1
//thread 1
C[1][0] = 0;                          // S0  1  0
C[1][1] = 0;                          // S0  1  1
C[1][0] += A[1][0] * B[0][0]; // S1  1  0  0
C[1][1] += A[1][0] * B[0][1]; // S1  1  1  0
C[1][0] += A[1][1] * B[1][0]; // S1  1  0  1
C[1][1] += A[1][1] * B[1][1]; // S1  1  1  1
```

University of Strasbourg

Inria

Step d. Nested Loop Recognition

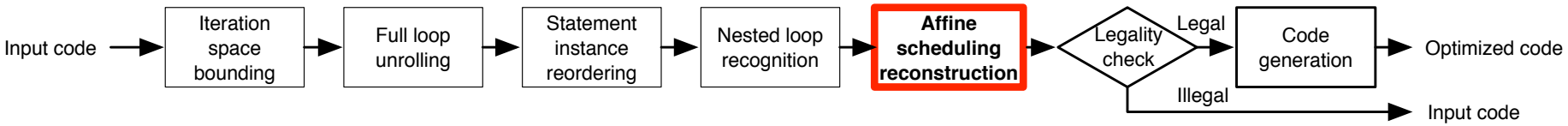
- Recover loops
- Using NLR[1], a fast and incremental algorithm [Ketterlin and Clauss, 2008]
- With the help of *tags*
  statement + iteration vector, as shown on the previous codes

[1] https://icps.u-strasbg.fr/nlr/

Step d. Nested Loop Recognition

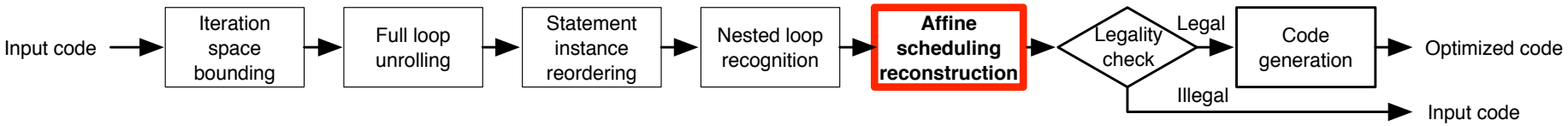

```
for i0 = 0 to 1
   for i1 = 0 to 1
     val S0 , 1*i0 , 1*i1
   for i1 = 0 to 1
     for i2 = 0 to 1
       val S1 , 1*i0 , 1*i2 , 1*i1
```

Step e. Affine Scheduling Reconstruction



- Build an affine scheduling expression from:
  - The loop structure
  - The mapping information present in NLR's output
- Map the original iteration vectors using the NLR output (without parameters, instructions reordered, ...)

Step e. Affine Scheduling Reconstruction

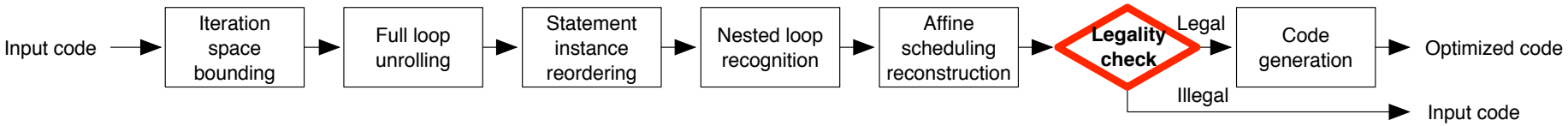



```
for i0 = 0 to 1
   for i1 = 0 to 1
     val S0 , 1*i0 , 1*i1    // <- S0 i j
   for i1 = 0 to 1
     for i2 = 0 to 1
       val S1 , 1*i0 , 1*i2 , 1*i1  // <- S1 i j k
```

↓

```
S0(i, j)    = (i, 0, j)
S1(i, j, k) = (i, 1, k, j)
```

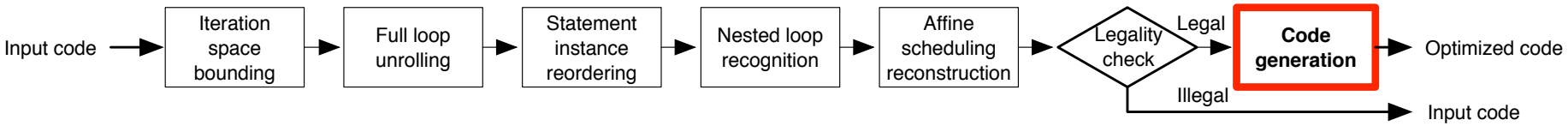
Step f. Legality Check



- Check the correctness of the schedule
  using, for example, Candl [Bastoul&Pouchet, 2008]
- Check loop properties (parallel, vector)

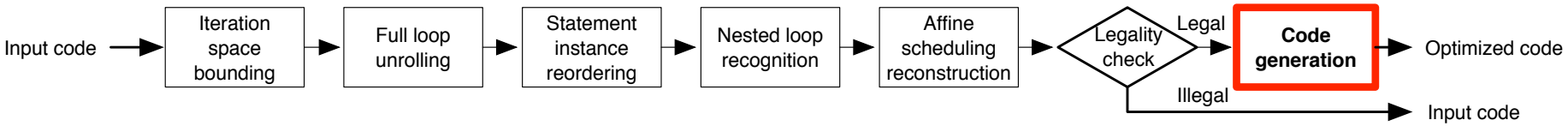
- If any of the previous steps fail, fall back to the original code

Step g. Code Generation



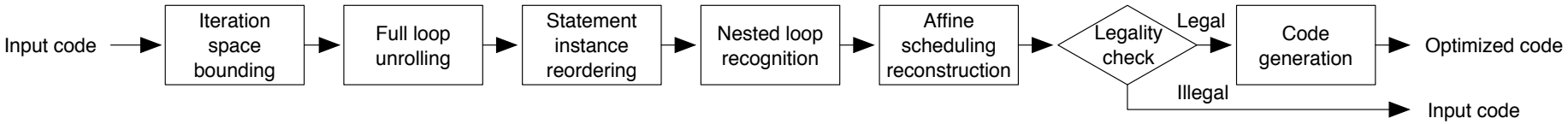
- Produce the optimized code that implements the scheduling
- Using, for example, CLooG [Bastoul, 2004]

Step g. Code Generation




```
#pragma omp parallel for private(j, k)
for (i = 0; i < M; i++) {
  #pragma vector always
  for (j = 0; j < N; j++)
    C[i][j] = 0.;                    // S0
  for (k = 0; k < P; k++)
    #pragma vector always
    for (j = 0; j < N; j++)
      C[i][j] += A[i][k] * B[k][j]; // S1
```

Conclusion



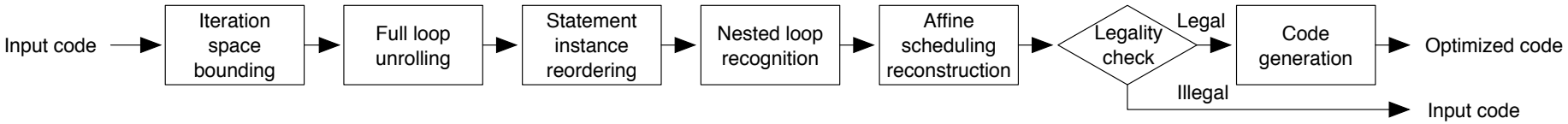
- Summary
  - A new scheduling approach and algorithm
  - Main seven steps identified and validated on a sample code (matrix product)
  - Synergistic fusion/fission - tiling - thread/vector parallelization
- Outlook
  - Method to be validated and tested
  - And applied to general codes

Thank you!

University of Strasbourg

Inria

Conclusion



- Summary
  - A new scheduling approach and algorithm
  - Main seven steps identified and validated on a sample code (matrix product)
  - Synergistic fusion/fission - tiling - thread/vector parallelization
- Outlook
  - Method to be validated and tested
  - And applied to general codes

Thank you!

University of Strasbourg

Inria