CS614: Advanced Compilers

Loop Transformations

Manas Thakur
CSE, IIT Bombay

Spring 2025

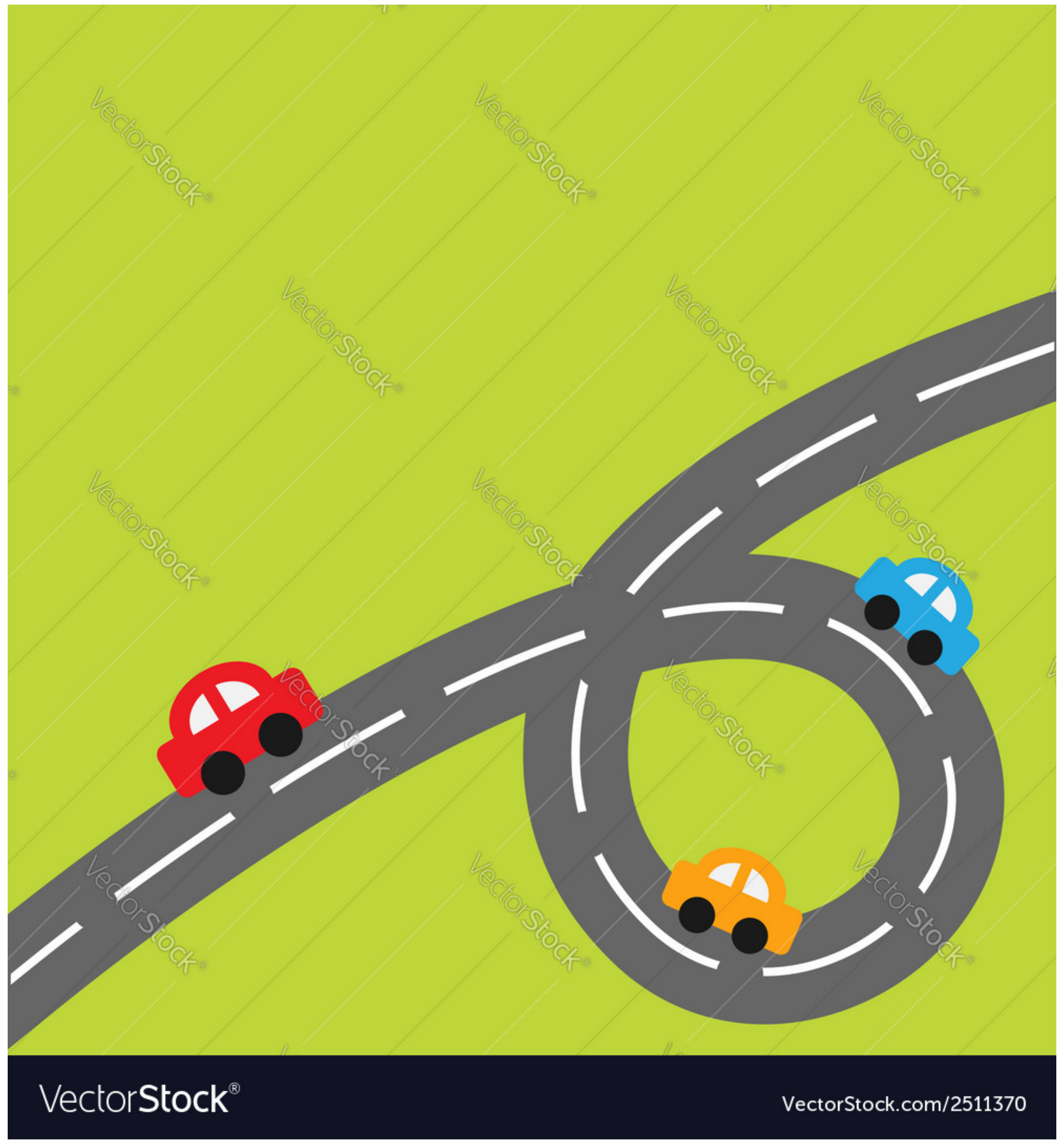
Why focus on loops?

- Form a significant portion of the time spent in executing programs.
  - If N is just 10000 (not uncommon), we have too many instructions!
    - How many in this loop?
      - What if S1/S2 is/are function calls?
      - What if they themselves are loops?
- Involves costly instructions in each iteration:
  - Comparisons
  - Jumps
- Worth spending high efforts in optimizing loops.

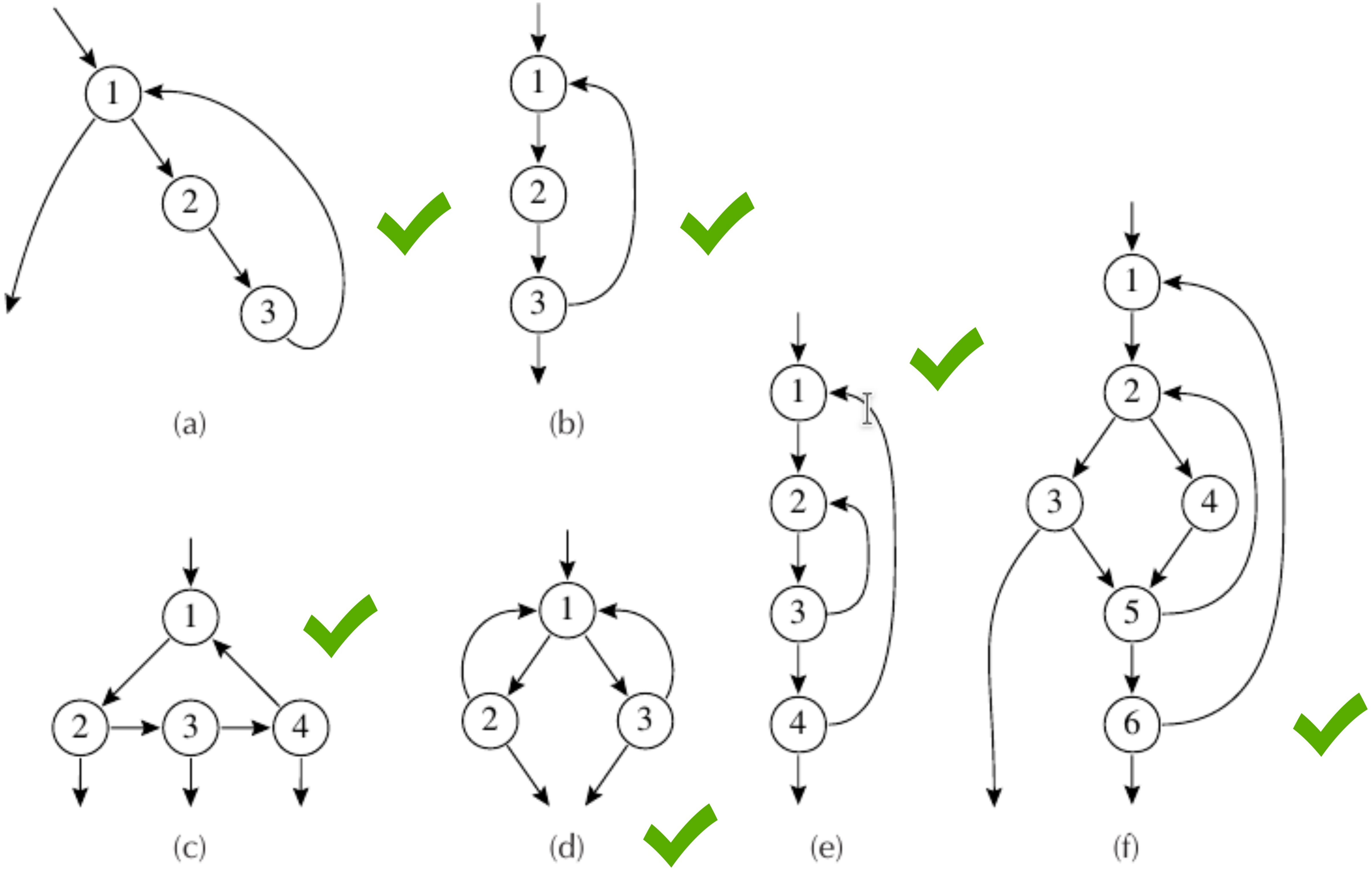
```
for (i=0; i<N; i++) {
   S1;
   S2;
}
```

What is a loop?

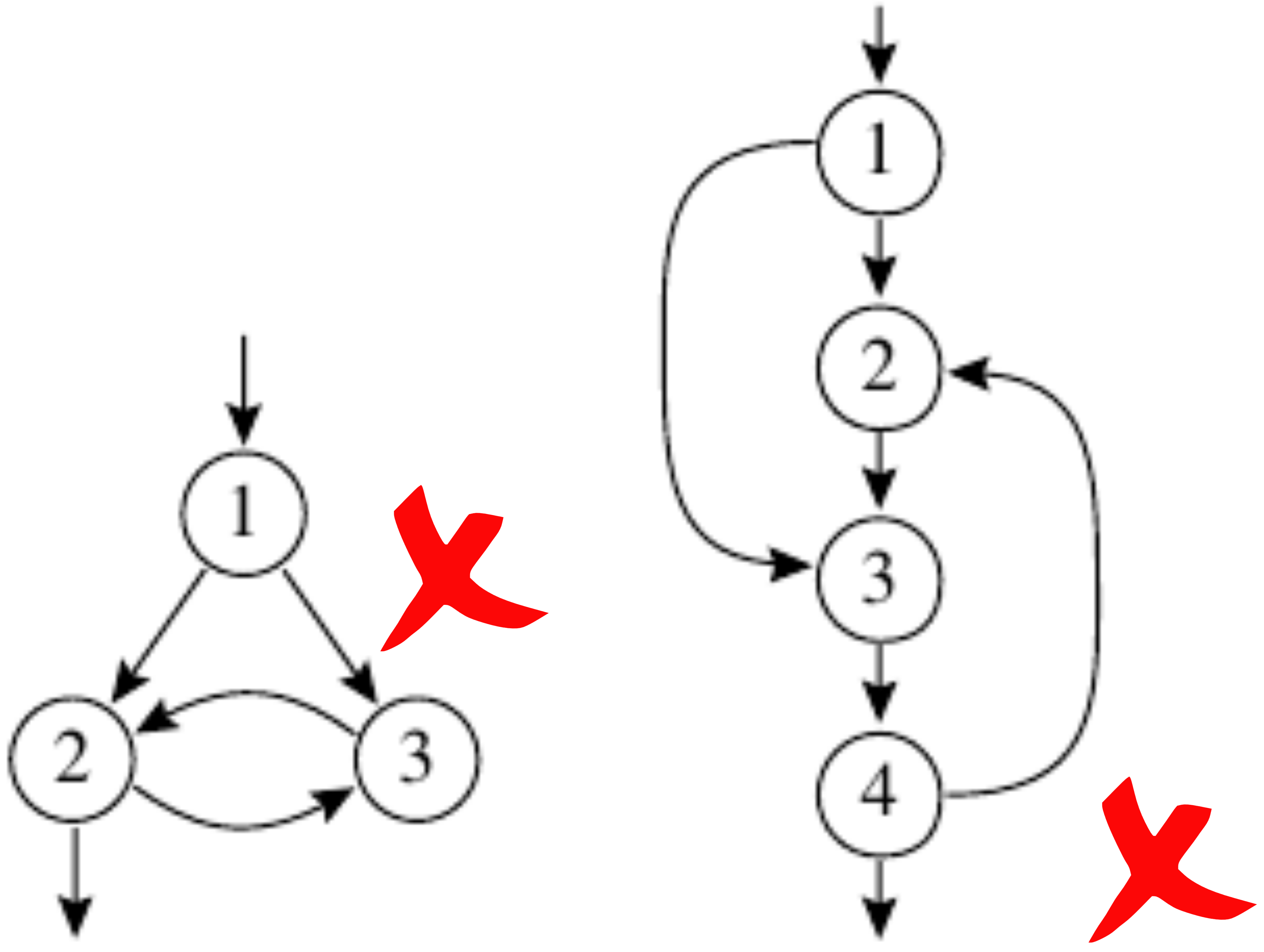
- If a loop exists as an explicit `for/while` loop, well and good. Else:
  - A loop in a CFG is a set of nodes S such that:
    - There is a designated header node h in S
    - There is a path from each node in S to h
    - There is a path from h to each node in S
    - h is the only node in S with an incoming edge from outside S



Are all these loops?



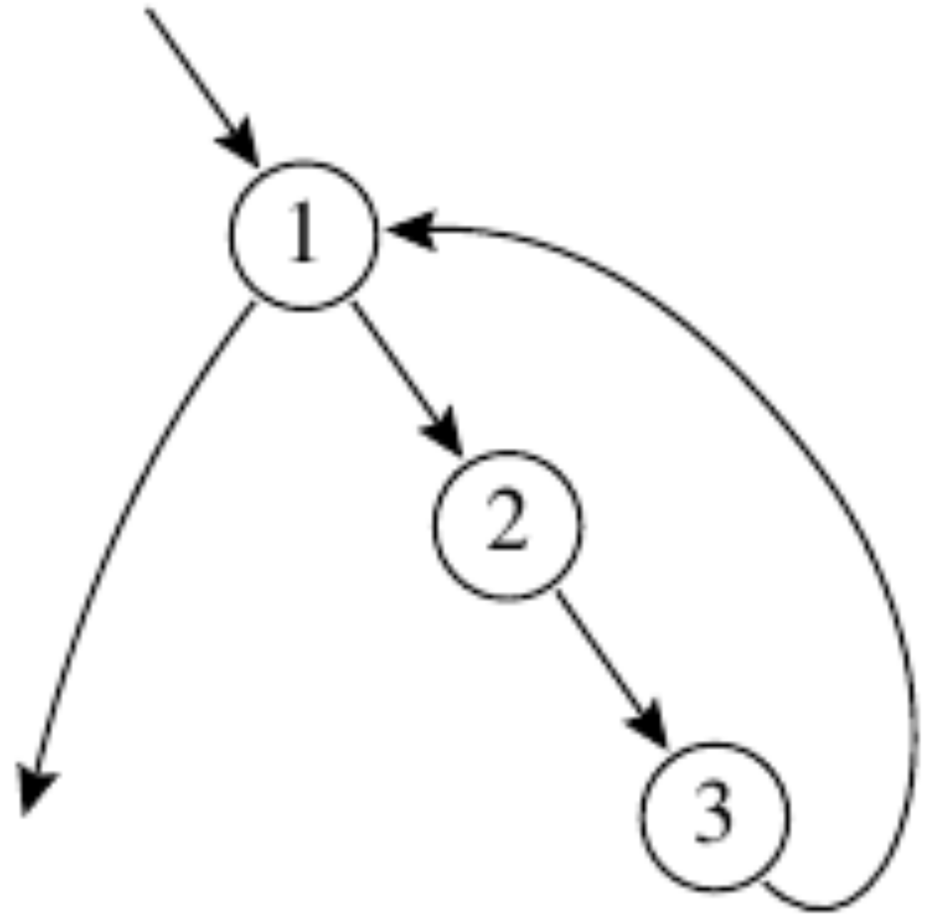
What about these?

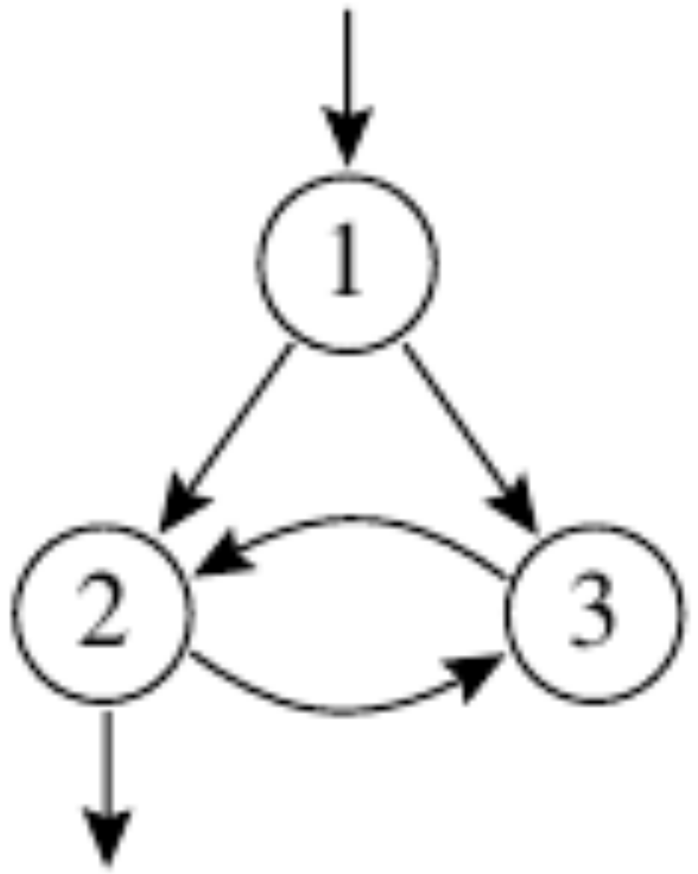


(a) (b)

Identifying loops using dominators

- A node `d` dominates a node `n` if every path from `entry` to `n` goes through `d`.
- Compute dominators of each node:

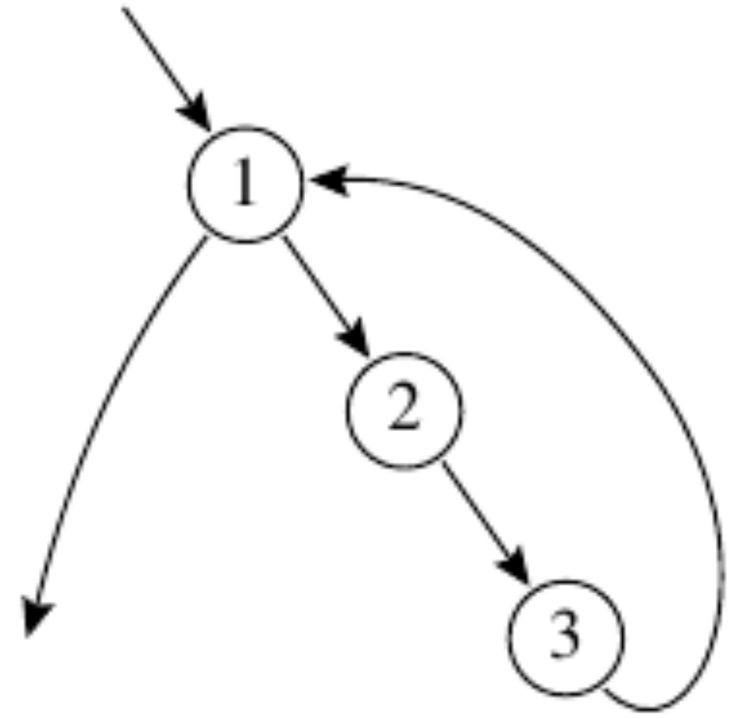


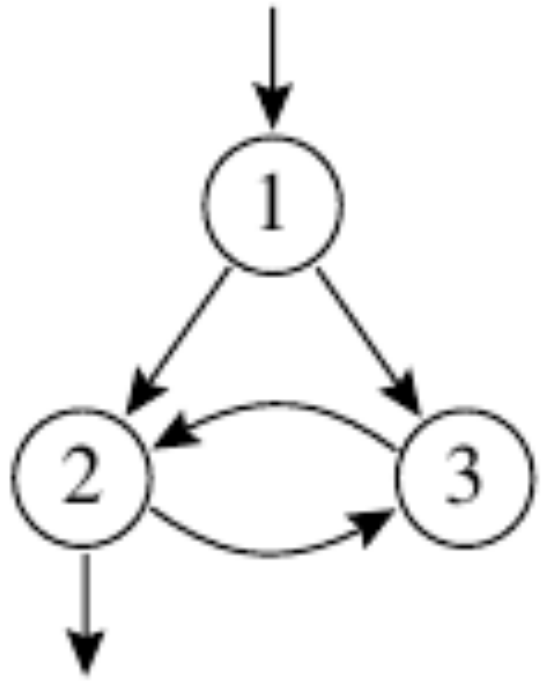


- **FFT:** Dominators are also useful in constructing SSA form of a CFG.

Identifying loops using dominators (Cont.)

- First, identify a back edge:
  - An edge from a node n to another node h, where h dominates n
- Each back edge leads to a loop:
  - Set X of nodes such that for each x ∈ X, h dominates x and there is a path from x to n not containing h
  - h is the header
- **Verify:**





1. Loop-Invariant Code Motion

- Loop-invariant code:
  - `d: t = a OP b`, such that:
    - `a` and `b` are constants; or
    - all the definitions of `a` and `b` that reach `d` are outside the loop; or
    - only one definition each of `a` and `b` reaches `d`, and that definition is loop-invariant.
- **Example:**

```
L0: t = 0
L1: i = i + 1
    t = a * b
    M[i] = t
    if i<N goto L1
L2: x = t
```

1. LICM (Cont.)

- Can we always hoist loop-invariant code?

```
L0: t = 0
L1: i = i + 1
    t = a * b
    M[i] = t
    if i<N goto L1
L2: x = t
```

```
L0: t = 0
L1: if i>=N goto L2
    i = i + 1
    t = a * b
    M[i] = t
    goto L1
L2: x = t
```

```
L0: t = 0
L1: M[j] = t
    i = i + 1
    t = a * b
    M[i] = t
    if i<N goto L1
L2: x = t
```

- **Criteria for hoisting `d: t = a OP b`:**
  - `d` dominates all loop exits at which `t` is live-out, and
  - there is only one definition of `t` in the loop, and
  - `t` is not live-out of the loop preheader
- How can we hoist code in the orange and the blue blocks?

2. Induction-Variable Optimization

- Induction variables:
  - Variables whose value depends on the iteration variable.
- Optimization:
  - Compute them efficiently, if possible.

```
    s = 0
    i = 0
L1: if i>=N goto L2
    j = i * 4
    k = j + a
    x = M[k]
    s = s + x
    i = i + 1
    goto L1
L2:
```

→

```
    s = 0
    k' = a
    b = N * 4
    c = a + b
L1: if k'>=c goto L2
    x = M[k']
    s = s + x
    k' = k' + 4
    goto L1
L2:
```

3. Loop Unrolling

- Minimize the number of increments and condition-checks
- Be careful about the increase in code size (I-cache misses!)

Unroll by factor of 2

```
L1: x = M[i]
    s = s + x
    i = i + 4
    if i<N goto L1
L2:
```

Only even no. of iterations:

```
L1: x = M[i]
    s = s + x
    x = M[i+4]
    s = s + x
    i = i + 8
    if i<N goto L1
L2:
```

Any no. of iterations:

```
    if i<N-8 goto L1
    goto L2
L1: x = M[i]
    s = s + x
    x = M[i+4]
    s = s + x
    i = i + 8
    if i<N-8 goto L1
L2: x = M[i]
    s = s + x
    i = i + 4
    if i<N goto L2
L3:
```

4. Loop Interchange

- Consecutive accesses to the arrays far apart (row-major storage):

```
for (j=0; j<n; j++) {
   for (i=0; i<m; i++) {
      c[i][j] = a[i][j] + b[i][j];
   }
}
```

- Interchange the two loops:

```
for (i=0; i<m; i++) {
   for (j=0; j<n; j++) {
      c[i][j] = a[i][j] + b[i][j];
   }
}
```

- Can we always interchange loops?

4. Loop Interchange (Cont.)

- Consider the following nested loops:

```
for (i=1; i<m; i++) {
   for (j=1; j<n; j++) {
      a[i][j] = a[i][j-1] + a[i-1][j+1];
   }
}
```

- **Two dependences across iterations:**
  - From `a[i][j-1] to a[i][j]`: Previously computed values used later.
  - From `a[i-1][j+1] to a[i][j]`: Previously computed values used later.
- After loop interchange, the second dependence requires reads from future writes.
- Hence we cannot interchange the loops.

4. Loop Interchange (Cont.)

- If the indices depend on each other, the transformation may be non-trivial:

```
for (j=0; j<n; j++) {
   for (i=j; i<j+m; i++) {
      row_sum[i-j] += matrix[i-j][j];
   }
}
```

```
for (i=0; i < m+n-1 ; i++) {
   for (j=max(0,i-m+1); j < min(n,i+1); j++) {
      row_sum[i-j] += matrix[i-j][j];
   }
}
```

- Sometimes an interchange may not clearly improve locality:

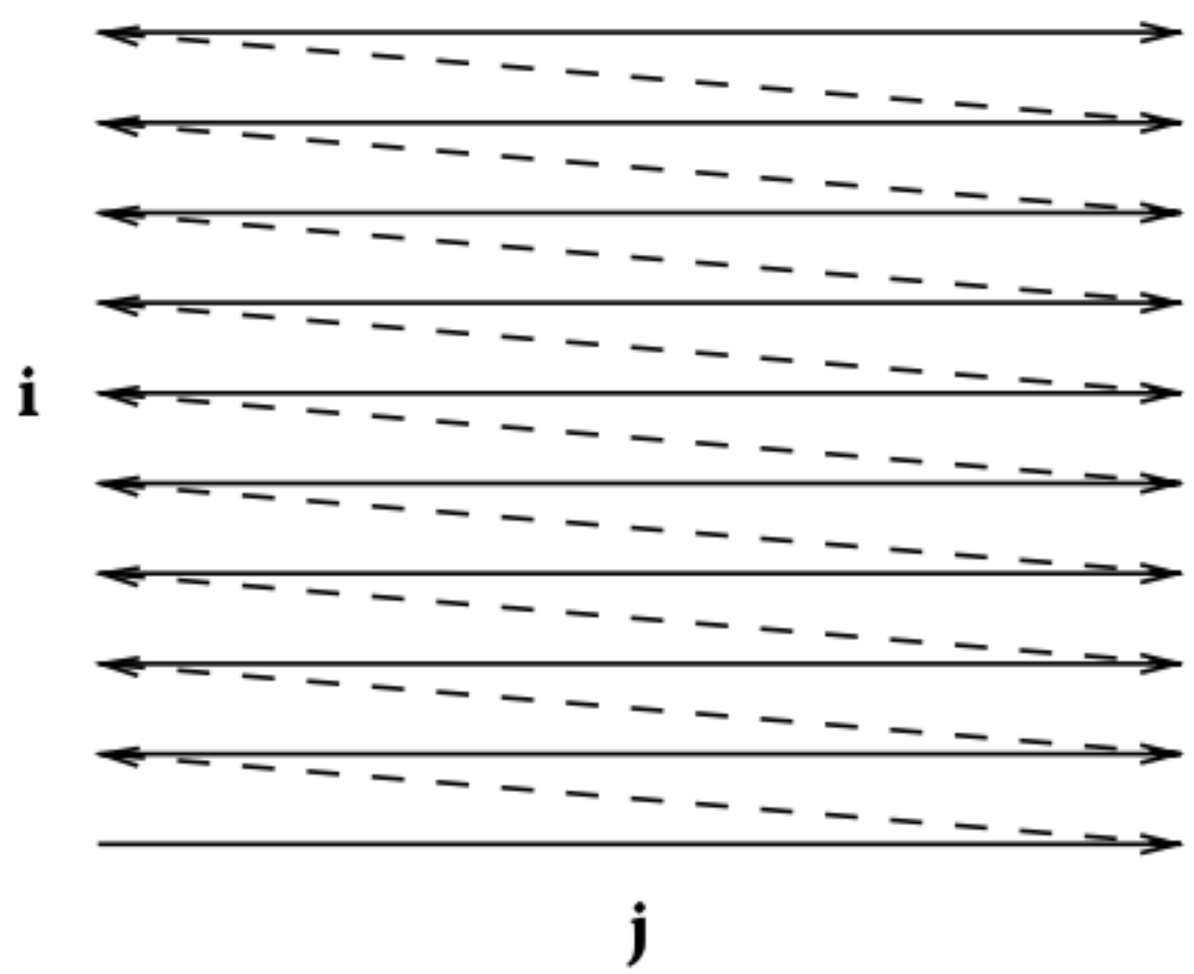
```
for (j=0; j<m; j++) {
   for (i=0; i<n; i++) {
      sum += (matrix[i][j]*row[j]);
   }
}
```

- Loop order suboptimal for `matrix`, but the best possible for `row`.

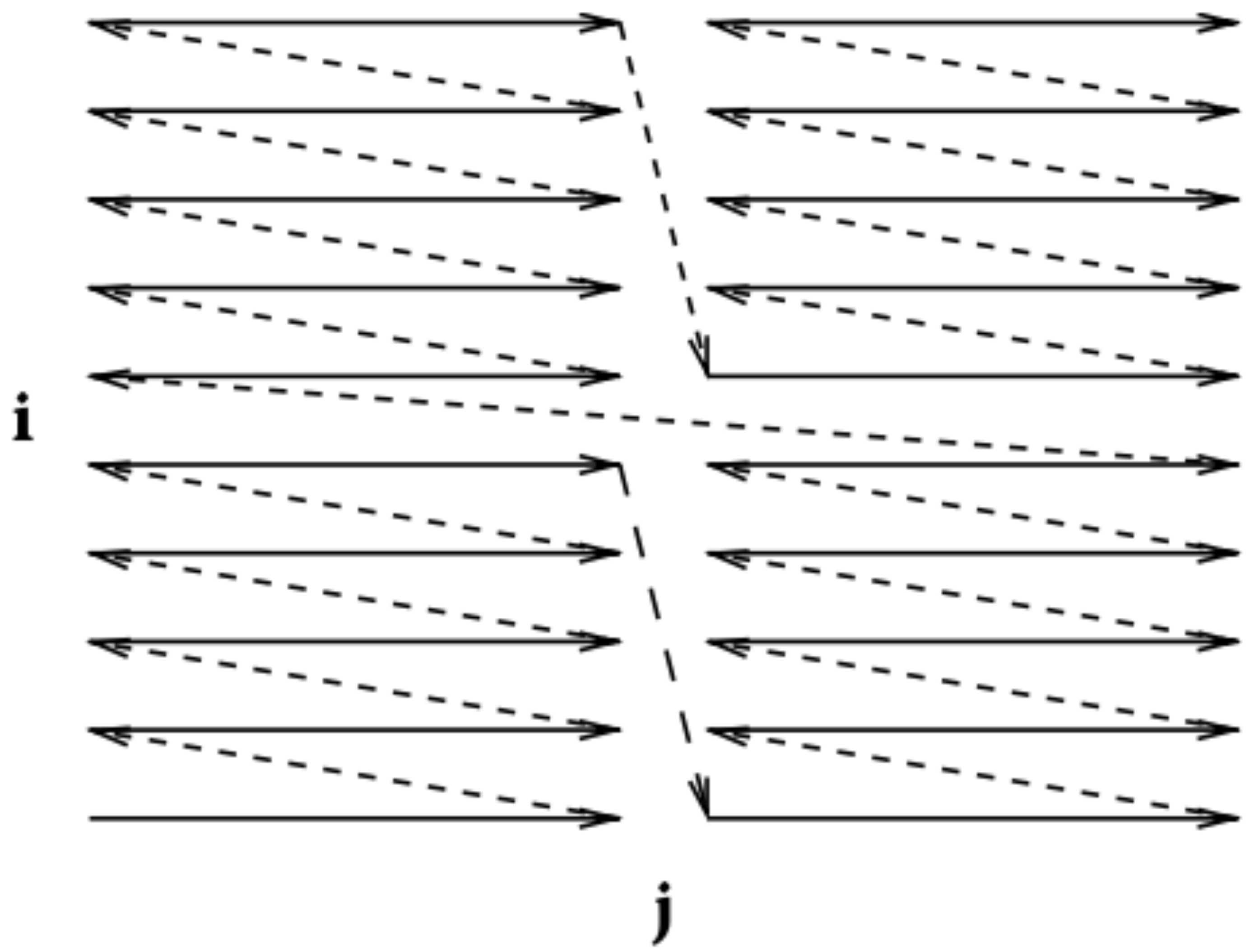
5. Loop Tiling/Blocking

- Many times computations involve reusing various parts of matrices, but the matrices might be large and may not fit into a cache block.
- Loop tiling/blocking distributes the computation in a way that each subsequent **tile** fits into a cache **block**.

Input iteration space:



Output iteration space



5. Loop Tiling (Cont.)

```
for (i=0; i<m1; i++) {
   for (j=0; j<n2; j++) {
      c[i][j] = 0;
      for (k=0; k < n1; k++) {
         c[i][j] += a[i][k]*b[k][j];
      }
   }
}
```

Input loop

```
for (i=0; i<m1; i++) {
   for (j=0; j<n2; j++) {
      c[i][j]=0;                    // Separate out initialization
   }
}
for (i1=0; i1<m1; i1 += block_size) {
   for (j1=0; j1<n2; j1 += block_size) {
      for (k1=0; k1<n1; k1 += block_size) {
         for (i=i1; i<min(m1, i1+block_size); i++) {
            for (j=j1; j<min(n2, j1+block_size); j++) {
               for (k=k1; k < min(n1, k1+block_size); k++) {
                  c[i][j] += a[i][k]*b[k][j];
               }
            }
         }
      }
   }
}
```

Output loop

Understand how we changed the iteration space.

5. Loop Tiling Step 1: Strip Mining

```
for (i=0; i<m1; i++) {
   for (j=0; j<n2; j++) {
      for (k=0; k < n1; k++) {
         c[i][j] += a[i][k]*b[k][j];
      }
   }
}
```

Strip mining transforms a loop into two nested loops, with the inner loop iterating over a small strip of the original loop and the outer loop iterating across strips.

Assuming block-size is smaller than n1, the inner loop now has fewer iterations.

```
for (i=0; i<m1; i++) {
   for (j=0; j<n2; j++) {
      for (k1=0; k1 < n1; k1 += block_size) {
         for (k=k1; k < min(n1, k1+block_size); k++) {
            c[i][j] += a[i][k]*b[k][j];
         }
      }
   }
}
```

5. Loop Tiling Step 2: Interchange

```
for (i=0; i<m1; i++) {
   for (j=0; j<n2; j++) {
      for (k1=0; k1 < n1; k1 += block_size) {
         for (k=k1; k < min(n1, k1+block_size); k++) {
            c[i][j] += a[i][k]*b[k][j];
         }
      }
   }
}
```

Interchange the k1 loop with the two outer loops.

```
for (k1=0; k1 < n1; k1 += block_size) {
   for (i=0; i<m1; i++) {
      for (j=0; j<n2; j++) {
         for (k=k1; k < min(n1, k1+block_size); k++) {
            c[i][j] += a[i][k]*b[k][j];
         }
      }
   }
}
```

We got tiling in one dimension. Repeat the same in other dimensions.

6. Loop Fusion

- When there is reuse of data across two independent loops, the loops can be fused together, provided their indices are compatible.

```
max = a[0];
for (i=1; i<N; i++) {
    if (a[i] > max)
        max = a[i];
}
min = a[0];
for (i=1; i<N; i++) {
    if (a[i] < min)
        min = a[i];
}
```

→

```
max = a[0];
min = a[0];
for (i=1; i<N; i++) {
    if (a[i] > max)
        max = a[i];
    if (a[i] < min)
        min = a[i];
}
```

- If one loop has a different bound than the other, but we know a relationship between the two bounds, we can *split* the longer loop and *align* one split to perform fusion.

7. Loop Fission/Distribution

- We can also split a loop body, with independent sub-parts, into two loops.

```
for (i=0; i<n; i++) {
   a[i] = a[i-1]+b[i-1];
   b[i] = k*a[i];
   c[i] = c[i-1]+1;
}
```

→

```
for (i=0; i<n; i++) {
   a[i] = a[i-1]+b[i-1];
   b[i] = k*a[i];
}
for (i=0; i<n; i++) {
   c[i] = c[i-1]+1;
}
```

- **Advantage?**
  - May reduce the memory footprint of the first loop.
  - It might be possible to parallelize the two loops separately.
- Requires *dependence analysis* across statements inside the loop body.

8. Loop Peeling

```
p = 10;
for (i=0; i<10; i++) {
   y[i] = x[i] + x[p];
   p = i;
}
```

- `p` is `10` only in the first iteration; later it is `i-1` every time.

```
y[0] = x[0] + x[10];
for (i=1; i<10; i++) {
   y[i] = x[i] + x[i-1];
}
```

- Now there is no need of referencing `p` inside the loop.
- Quite often helps in LICM.

Loop optimizations in typical compilers

- GCC and CLang perform loop-invariant code motion, loop unrolling, induction-variable optimization, loop inversion (`while` to `if+do-while`), etc.
- HotSpot C2 performs loop splitting, loop predication (move range- and null-checks out), vectorization, unrolling, *loop beautification*, etc.
- Usually performed in initial phases, typically after a few passes constant propagation.

- **New ones at CompL:**
  - Loop replacement (higher-order functions)
  - Loop $\boxed{\text{parallelization}}$

Next week!