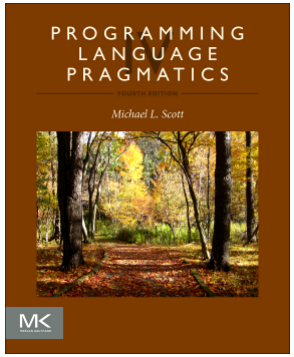
Code Optimization, Day 4: Scheduling and Loop Optimizations

17-363/17-663: Programming Language Pragmatics



Reading: PLP chapter 17

Ben Titzer Prof. Jonathan Aldrich

Machine-Dependent Optimization

- We've covered a few machine-independent optimizations
  - Dead-code elimination, strength reduction, code motion
- These (almost) always pay off, regardless of the machine
- Machine-Dependent optimizations take into account machine resources and reorganize or reschedule code to make best use of them

Scheduling

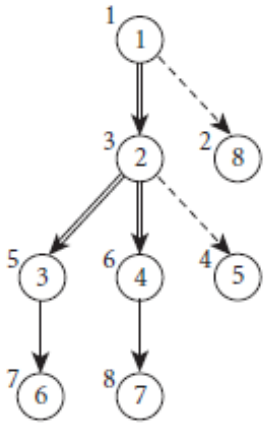
- Execution of any instruction takes machine resources (e.g. an arithmetic-logic unit) and time
  - Hardware resources: execution units (ports)
  - Time: pipeline stages such as fetch, decode, execute
- Instruction scheduling makes best use of machine resources to make the *same instructions* run in less time
- Local instruction scheduling: within a basic block
- Rearrange instructions to avoid pipeline stalls, within the constraints of data dependencies

Instruction Scheduling

- To schedule instructions to make better use of the pipeline, we first arrange them into a directed acyclic graph (DAG), in which each node represents an instruction, and each arc represents a *dependence*
  - Most arcs will represent *flow* dependences, in which one instruction uses a value produced by a previous instruction
  - A few will represent *anti*-dependences, in which a later instruction overwrites a value read by a previous instruction
  - In our example, these will correspond to updates of induction variables

Instruction Scheduling

Block 2:	Scheduled:
1. v19 := v13 × v18	v19 := v13 × v18
—	v18 := v18 – 1
—	—
—	—
—	—
2. v13 := v19 div v17	v13 := v19 div v17
—	v17 := v17 + 1
—	—
—	—
—	—
3. *v26 := v13	*v26 := v13
4. *v34 := v13	*v34 := v13
5. v17 := v17 + 1	v26 := v26 + 4
6. v26 := v26 + 4	v34 := v34 – 4
7. v34 := v34 – 4	
8. v18 := v18 – 1	
–– fall through to Block 3	



Block 3: (same)

v43 := v17 ≤ v42
if v43 goto Block 2
–– else fall through to Block 4

Figure 17.13 **Dependence DAG for Block 2 of Figure C-17.12, together with pseudocode for the entire loop, both before (left) and after (right) instruction scheduling.** Circled numbers in the DAG correspond to instructions in the original version of the loop. Smaller adjacent numbers give the schedule order in the new loop. Solid arcs indicate flow dependences; dashed arcs indicate anti-dependences. Double arcs indicate pairs of instructions that must be separated by four additional instructions in order to avoid pipeline delays on our hypothetical machine. Delays are shown explicitly in Block 2. Unless we modify the array indexing code (Exercise C-17.20), only two instructions can be moved.

Instruction Scheduling Algorithms

- Considerations:
  - Dependencies between instructions
  - Every instruction takes multiple cycles
  - Resource (machine units) usage of each instruction
- Most algorithms are based on list scheduling
  - Model each cycle of execution, assign instruction(s) to cycles in a forward or backward pass
  - An instruction is ready as soon as its input operands are ready, accounting for the delay of inputs
  - An instruction may not be able to be scheduled at a particular cycle because a unit it needs is in use by another instruction
- Instruction scheduling and register allocation are fundamentally in tension with each other

Loop Optimization Buffet

- Loops are extremely important because most programs spend a lot of time in loops
- Most optimizing compilers include some of:
  - *Loop invariant code motion* moves computations out of the body of a loop and into its header
  - *Induction variable analysis* rewrites expressions that accomplish iteration and can remove bounds checks
  - *Loop peeling* copies the body of a loop for the first iteration, which can set up other optimizations
  - *Loop unrolling* duplicates the body of a loop many times to reduce iteration overhead and unlock other optimizations
  - *Software pipelining* allows instruction scheduling across loop iterations
  - *Loop reordering* targets the iteration order of nested loops to improve cache locality

Loop Invariant Code Motion (LICM)

- A *loop invariant* is an instruction (i.e., a load or calculation) in a loop whose result is guaranteed to be the same in every iteration
  - If a loop is executed *n* times and we are able to move an invariant instruction out of the body and into the header (saving its result in a register for use within the body), then we will eliminate *n* -1 calculations from the program
    - a potentially significant savings
- In order to tell whether an instruction is invariant, we need to identify the bodies of loops, and we need to track the locations at which operand values are defined

LICM Example

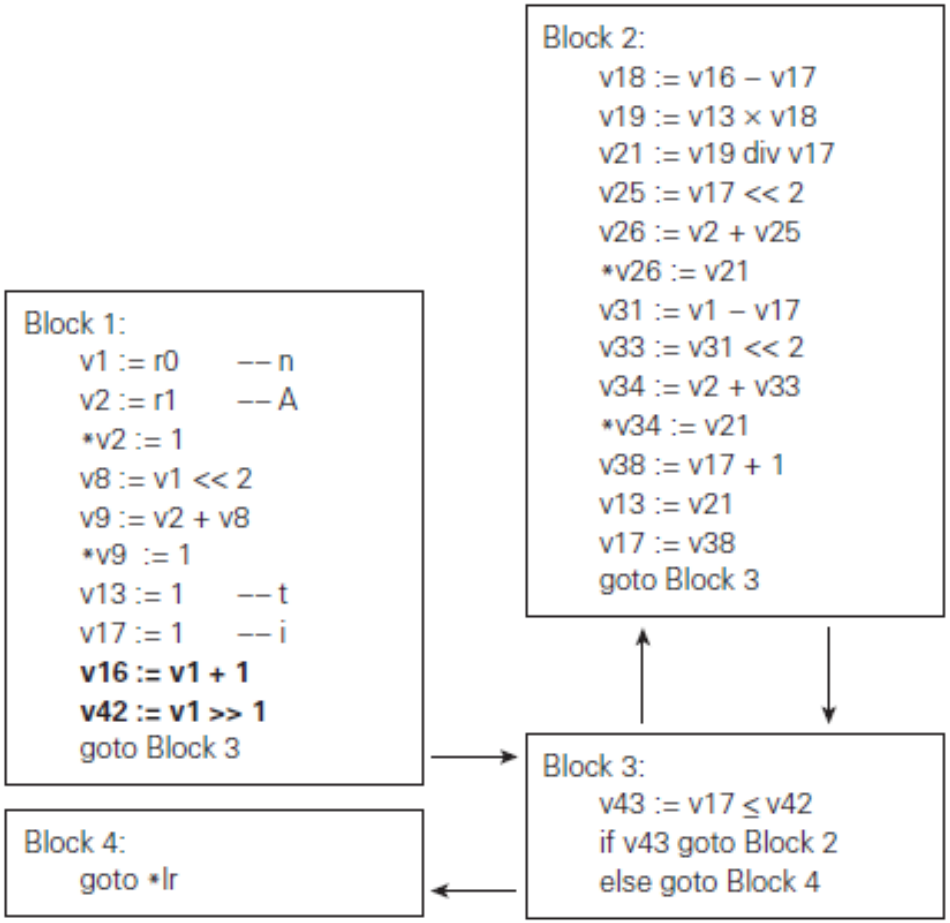


Figure 17.10 Control flow graph for the `combinations` subroutine after moving the invariant calculations of v16 and v42 (shown in boldface type) out of the loop. We have also dropped the dead stores of Figure C-17.7, and have eliminated the stack space for t and i, which now reside entirely in registers.

Induction Variables

- An *induction variable* takes on a simple progression of values in successive loop iterations
  - We confine our attention to arithmetic progressions
  - Induction variables appear as loop indices, subscript computations, or variables incremented or decremented explicitly within the body of the loop
- Induction variables are important for two reasons:
  - They commonly provide opportunities for strength reduction, replacing multiplication with addition
  - They are commonly redundant: instead of keeping several induction variables in registers, we can often keep a smaller number and calculate the remainder from those when needed

Identifying Induction Variables

- Tracking the locations at which an operand may have been defined amounts to the problem of *reaching definitions*
  - Formally, we say an instruction that assigns a value *v* into a location (variable or register) *l reaches* a point *p* in the code if *v* may still be in *l* at *p*
- Like the conversion to static single assignment form, considered informally earlier, the problem of reaching definitions can be structured as a set of forward, any-path data flow equations
  - We let Gen_B be the set of final assignments in block *B* (those that are not overwritten later in *B*)
  - For each assignment in *B* we also place in $Kill_B$ all *other* assignments (in any block) to the same location
- In SSA form, they are φ's that are just functions of themselves

Induction Variable Example

```
A : array [1..n] of record
        key : integer
        // other stuff
for i in 1..n
    A[i].key := 0
```

(a)

```
    v1 := 1
    v2 := n
    v3 := sizeof(record)
    v4 := &A – v3
L:  v5 := v1 × v3
    v6 := v4 + v5
    *v6 := 0
    v1 := v1 + 1
    v7 := v1 ≤ v2
    if v7 goto L
```

(b)

```
    v1 := 1
    v2 := n
    v3 := sizeof(record)
    v5 := &A
L:  *v5 := 0
    v5 := v5 + v3
    v1 := v1 + 1
    v7 := v1 ≤ v2
    if v7 goto L
```

(c)

```
    v2 := &A + (n–1) × sizeof(record)
        -- may take >1 instructions
    v3 := sizeof(record)
    v5 := &A
L:  *v5 := 0
    v5 := v5 + v3
    v7 := v5 ≤ v2
    if v7 goto L
```

(d)

Figure 17.11 **Code improvement of induction variables.** High-level pseudocode source is shown in (a). Target code prior to induction variable optimizations is shown in (b). In (c) we have performed strength reduction on v5, the array index, and eliminated v4, at which point v5 no longer depends on v1 (i). In (d) we have modified the end test to use v5 instead of v1, and have eliminated v1.

Loop Peeling and Unrolling

- Loop *peeling* copies the body of a loop once
  - The first iteration often has safety checks and other weirdness that subsequent iterations don't need
- Loop *unrolling* copies the body of a loop more than once, creating a longer loop, and allowing the scheduler to intermingle the instructions of the original iterations
- If we unroll two iterations of our combinations example we obtain the code of Figure 17.14
  - We use separate names (here starting with the letter 't') for registers written in the initial half of the loop
  - This convention minimizes anti- and output dependences, giving us more latitude in scheduling

Loop Unrolling Example

```
Block 1:
    ...                          -- code from Block 1, figure 16.11
    v44 := v42 & 01
    if !v44 goto Block 3
    -- else fall through to Block 1a
Block 1a:
    •v26 := 1
    •v34 := 1
    v17 := 2
    v26 := v26 + 4
    v34 := v34 – 4
    v18 := v18 – 1
    goto Block 3
```

Block 2:		Scheduled:
1.	t19 := v13 × v18	t19 := v13 × v18
	—	t18 := v18 – 1
	—	t17 := v17 + 1
	—	v18 := t18 – 1
	—	—
2.	t13 := t19 div v17	t13 := t19 div v17
	—	v17 := t17 + 1
	—	—
	—	—
	—	—
3.	•v26 := t13	v19 := t13 × t18
4.	•v34 := t13	•v26 := t13
5.	t17 := v17 + 1	•v34 := t13
6.	v26 := v26 + 8	v26 := v26 + 8
7.	v34 := v34 – 8	v34 := v34 – 8
8.	t18 := v18 – 1	v13 := v19 div t17
9.	v19 := t13 × t18	—
	—	—
	—	—
	—	—
	—	•(v26–4) := v13
10.	v13 := v19 div t17	•(v34+4) := v13
	—	
	—	
	—	
	—	
11.	•(v26–4) := v13	
12.	•(v34+4) := v13	
13.	v17 := t17 + 1	
14.	v18 := t18 – 1	
	-- fall through to Block 3	

```
Block 3:                                   (same)
    v43 := v17 ≤ v42
    if v43 goto Block 2
    -- else fall through to Block 4
```

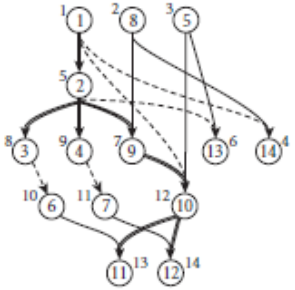


Figure 17.14 Dependence DAG for Block 2 of the `combinations` subroutine after unrolling two iterations of the body of the loop. Also shown is linearized pseudocode for the entire loop, both before (left) and after (right) instruction scheduling. New instructions added to the end of Block 1 cover the case in which the number of iterations of the original loop is not a multiple of two.

Software Pipelining

- A *software-pipelined* version of our combinations subroutine appears in the bottom half of Figure 17.15 and as a control flow graph in Figure 17.16
  - The idea is to build a loop whose body comprises portions of several consecutive iterations of the original loop, with no internal start-up or shut-down cost
  - In our example, each iteration of the software-pipelined loop contributes to three separate iterations of the original loop
  - Within each new iteration (shown between vertical bars) nothing needs to wait for the divide to complete
  - To avoid delays, we have altered the code in several ways

Software Pipelining

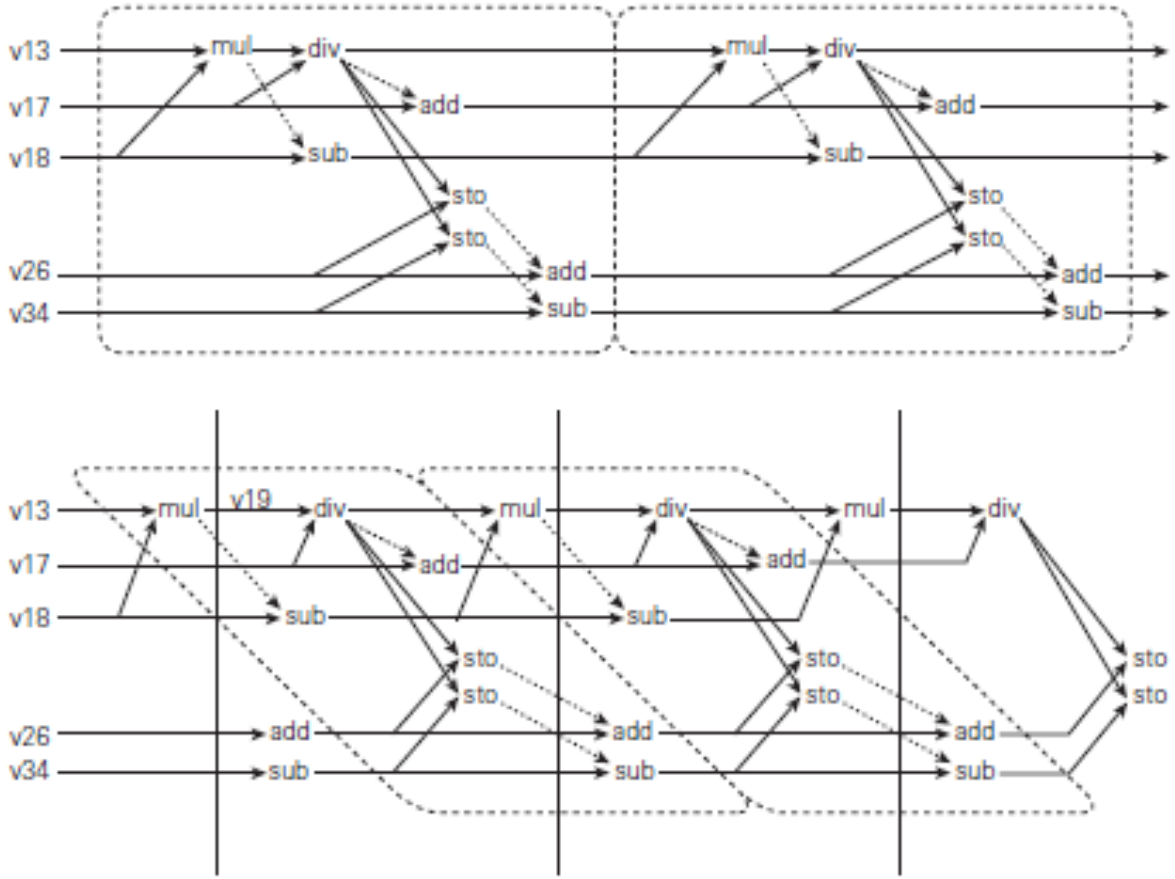


Figure 17.15 **Software pipelining.** The top diagram illustrates the execution of the original (nonpipelined) loop. In the bottom diagram, each iteration of the original loop has been spread across three iterations of the pipelined loop. Iterations of the original loop are enclosed in a dashed-line box; iterations of the pipelined loop are separated by solid vertical lines. In the bottom diagram we have also shown the code to prime the pipeline prior to the first iteration, and to flush it after the last.

Software Pipelining Example

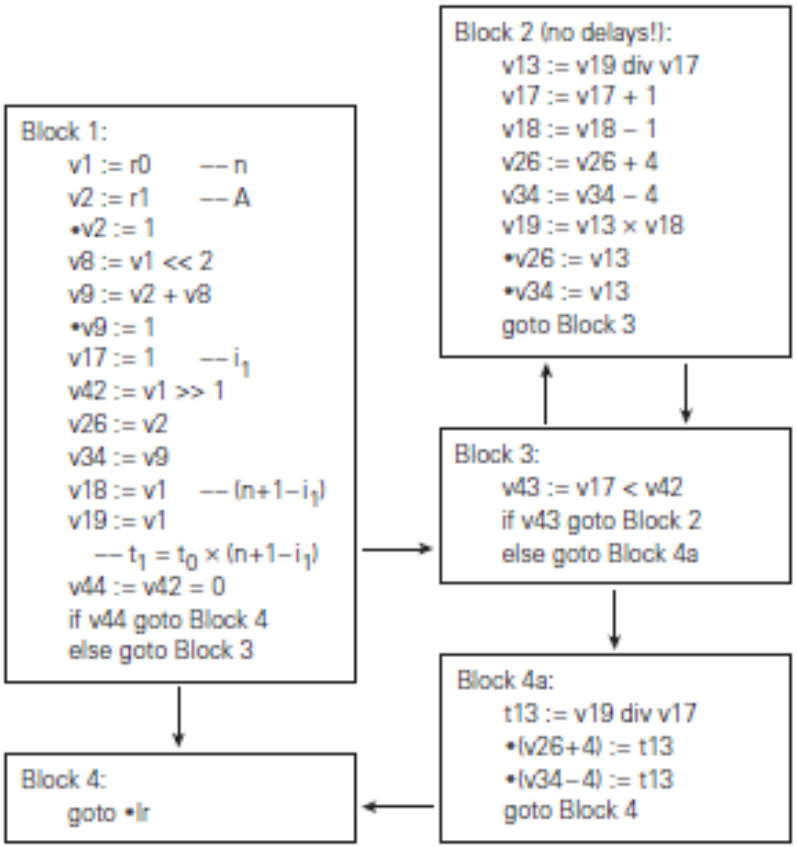


Figure 17.16 Control flow graph for the `combinations` subroutine after software pipelining. The additional code and test at the end of Block 1, the change to the test in Block 3 (< instead of ≤), and the new block (4a) make sure that there are enough iterations to accommodate the pipeline, prime it with the beginnings of the initial iteration, and flush the end of the final iteration. Suffixes on variable names in the comments in Block 1 refer to loop iterations: t_1 is the value of t in the first iteration of the loop; t_0 is a "zero-th" value used to prime the pipeline.

Loop Reordering

- The code optimization techniques that we have considered thus far have served two principal purposes
  - eliminate redundant or unnecessary instructions
  - minimize stalls on a pipelined machine
- Two other goals have become increasingly important in recent years
  - it has become increasingly important to minimize cache misses (processor speed outstrips memory latency)
  - it has become important to identify sections of code that can execute concurrently (parallel machines)
- As with other optimizations, the largest benefits come from changing the behavior of loops

Loop Reordering

- A loop-reordering compiler can improve this code by *interchanging* the nested loops:

```
for j := 1 to n
    for i := 1 to n
        A[i, j] := 0
```

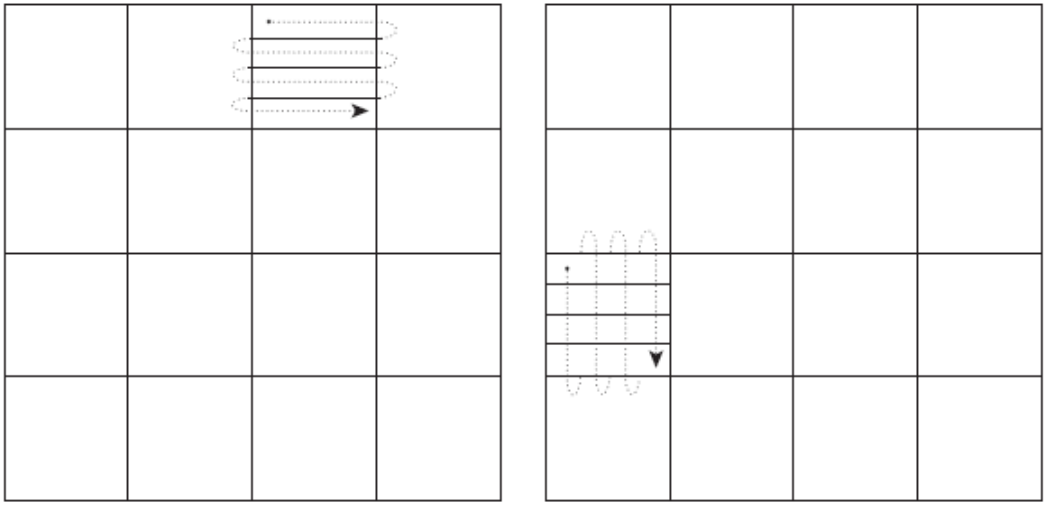


Figure 17.17 Tiling (blocking) of a matrix operation

Loop Reordering

```
for i := 1 to n
    for j := 1 to n
        A[i, j] := 0
```

- If A is laid out in row-major order, and if each cache line contains m elements of A, then this code will suffer n^2/m cache misses
- If A is laid out in column major order, and if the cache is too small to hold n lines of A, then the code will suffer n^2 misses, fetching the entire array from memory m times

Loop Dependences

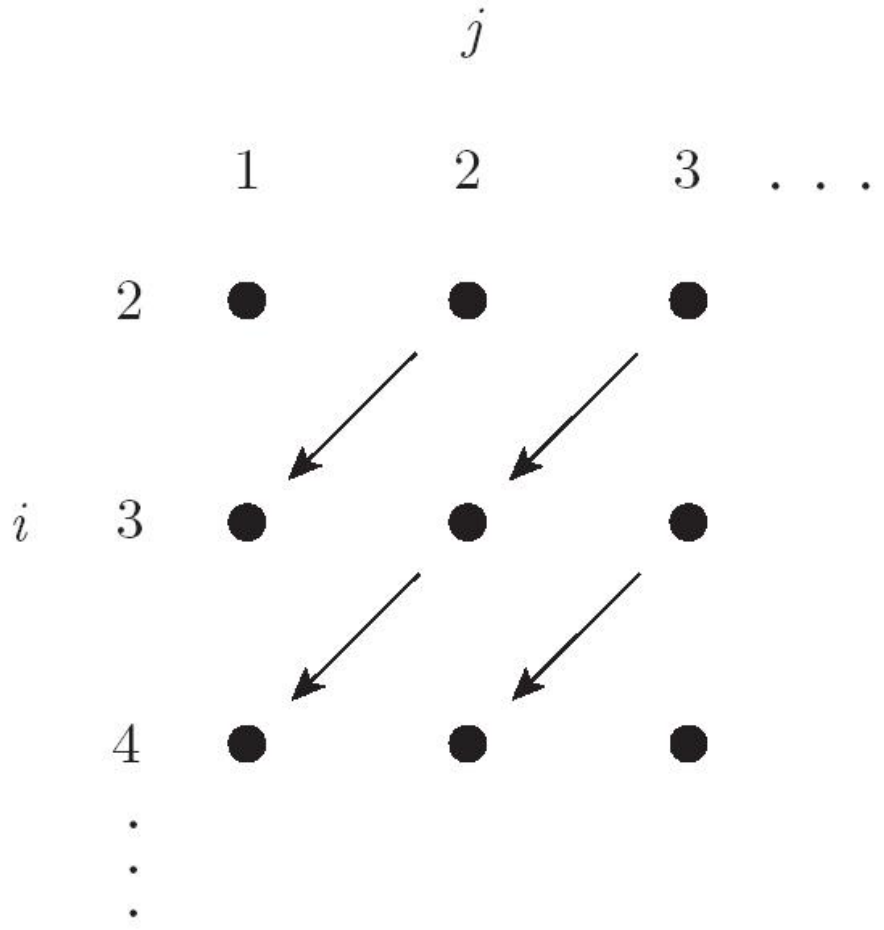
- When reordering loops, we must to respect all data dependences (*loop-carried* dependences)

```
    i := 2 to n
        for j := 1 to n1
            A[i, j] := A[i, j] + A[i - 1, j+1]
```

- Here the calculation of `A[i, j]` in iteration `(i, j)` depends on the value of `A[i-1, j+1]`, which was calculated in iteration `(i-1, j+1)`.
- This dependence is often represented by a diagram of the *iteration space* (see next slide):

Loop Interchange

- The *i* and *j* dimensions in this diagram represent loop indices, *not* array subscripts.
  - The arcs represent the loop-carried flow dependence
- If we wish to interchange the *i* and *j* loops of this code (e.g., to improve cache locality), we find that we cannot do it, because of the dependence: we would end up trying to write `A[i, j]` before we had written `A[i-1, j+1]`

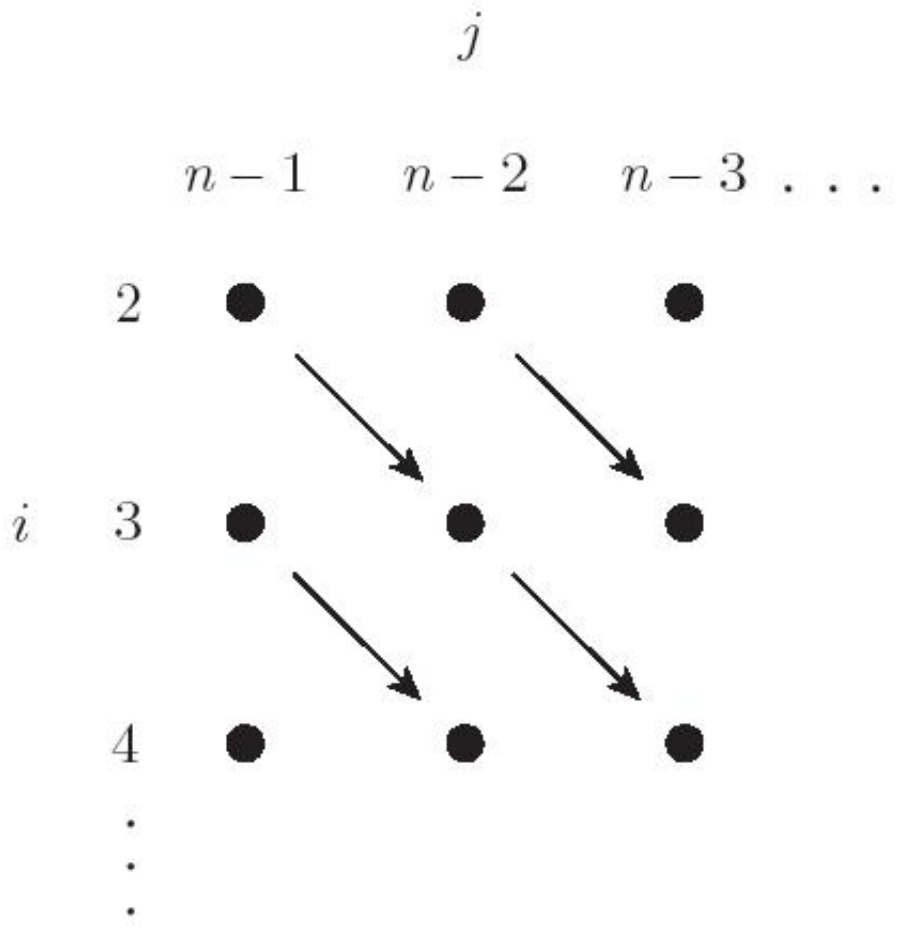


Loop Reversal

- By analyzing the loop dependence, we note that we can *reverse* the order of the *j* loop without violating the dependence:

```
for i := 2 to n
  for j := n1 to 1 by 1
     A[i, j] := A[i, j]
           A[i1, j+1]
```

- This change transforms the iteration space as shown here



Loop Skewing

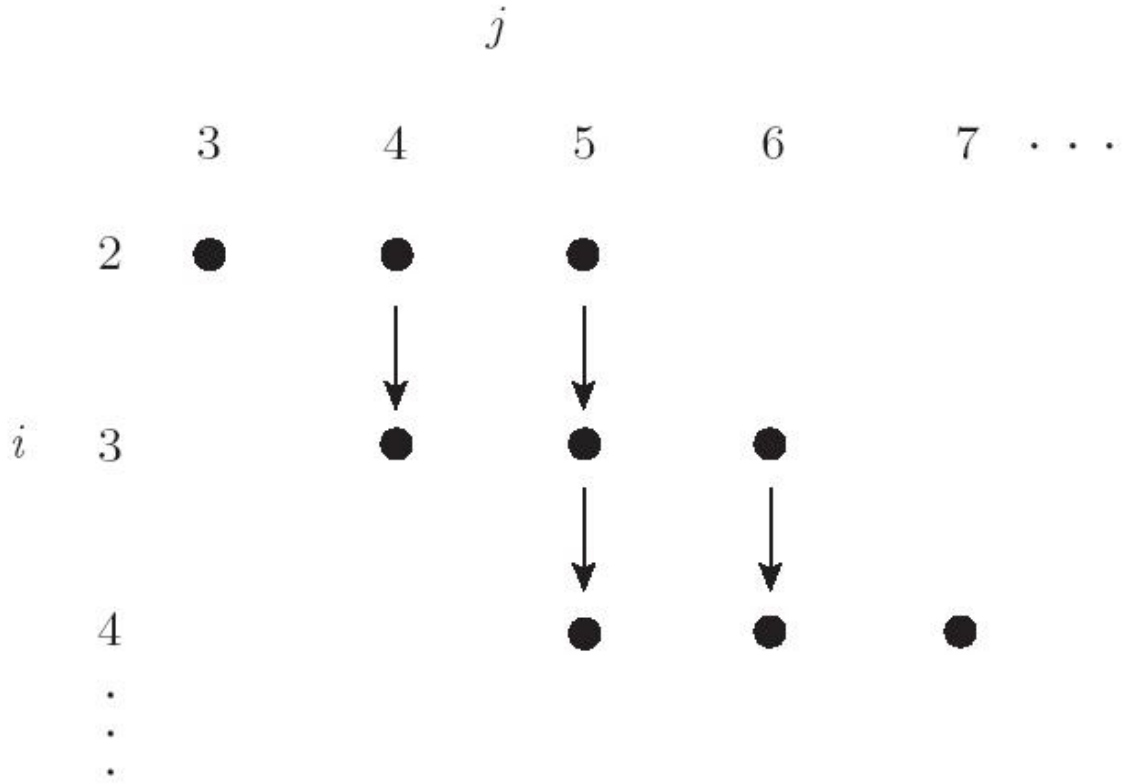
- Another transformation that sometimes serves to eliminate a dependence is known as *loop skewing*
  - it reshapes a rectangular iteration space into a parallelogram, by adding the outer loop index to the inner one, and then subtracting from the appropriate subscripts:

```
for i := 2 to n
   for j := i+1 to i+n-1
    A[i,j-i]:= A[i,j-i]-A[i1,j+1-i]
```

- A moment's consideration will reveal that this code accesses the exact same elements as before, in the exact same order

Loop Skewing

- Its iteration space, however, looks like here (the loops can be safely be interchanged)



Loop Parallelization

- Loop iterations (at least in non-recursive programs) constitute the principal source of operations that can execute in parallel
  - one needs to find *independent* loop iterations - with *no loop-carried dependences*
- Even in languages without special constructs, a compiler can often *parallelize* code by identifying—or creating—loops with as few loop-carried dependences as possible
  - These transformations are valuable tools in this endeavor

Loop Parallelization

- Consider the problem of "zero-ing out" a two-dimensional array (row-major order):

```
for i := 0 to n-1
        for j := 0 to n-1
                A[i, j] := 0
```

- On a machine containing several general purpose processors, we parallelize the outer loop:

```
-- on processor pid:
for i := (n/p × pid) to (n/p ×(pid + 1)-1)
        for j := 1 to n
                A[i, j] := 0
```

Loop Parallelization

- Other issues of importance in parallelizing compilers include *communication* and *load balance*
- Locality in parallel programs reduces communication among processors and between the processors and memory
  - Optimizations similar to those employed to reduce the number of cache misses on a uniprocessor can be used to reduce communication trafic on a multiprocessor.
- Load balance refers to the division of labor among processors on a parallel machine
  - dividing a program among 16 processors, we shall obtain a speedup of close to 16 only if each processor takes the same amount of time to do its work
  - assigning 5% of the work to each of 15 processors and 25% of the work to the sixteenth, we are likely to see a speedup of no more than four