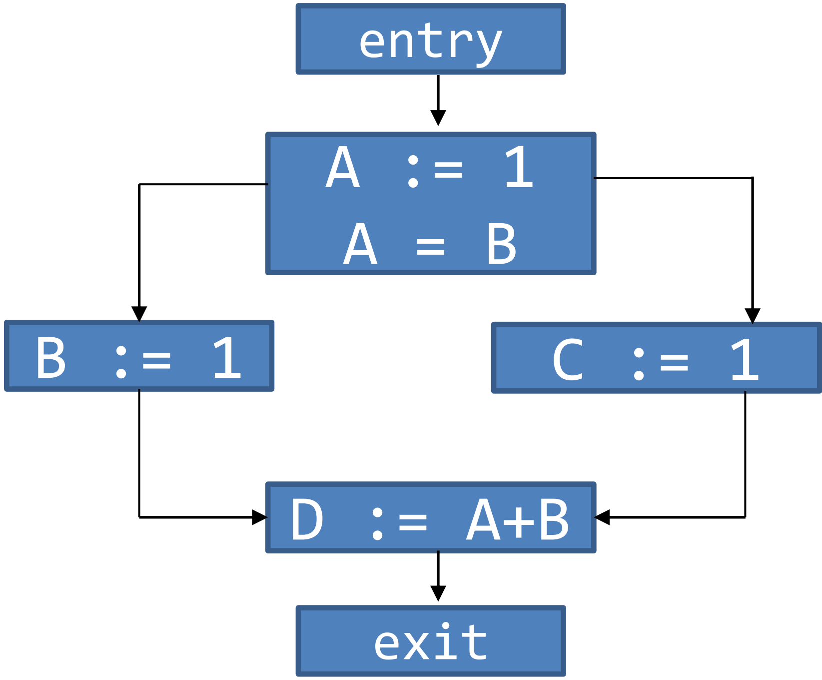
CS406:Compilers

Spring 2021

Week 14: Dataflow Analysis (Recap.) and Higher-level Loop Optimizations

Recap: Liveness

- Variables are live if *some path* leading to its use exists
- Start from exit block and proceed *backwards* against the control flow



$$\mathrm{LiveOut}(b) = \cup_{i \in Succ(b)} \mathrm{LiveIn}(i)$$

$$\mathrm{LiveIn}(b) = \underbrace{\mathrm{LiveUse}(b)}_{gen(b)} \cup (\mathrm{LiveOut}(b) - \underbrace{\mathrm{Def}(b)}_{kill(b)})$$

gen(b)

//set that contains all variables used by block b

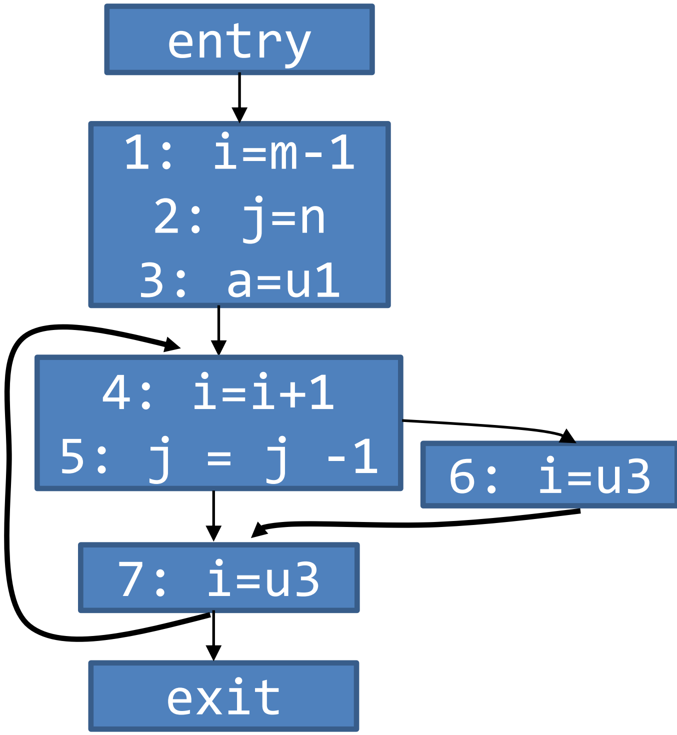
kill(b)

//set that contains all variables defined by block b

Recap: Reaching Definitions

- **Goal:** to know where in a program each variable x may have been defined when control reaches block b
- Definition d reaches block b if there is a path from point immediately following d to b, such that the variable defined in d is not redefined / killed along that path

$\texttt{In(b)} = \bigcup_{i \in Pred(b)} Out(i)$



$\texttt{Out(b) = gen(b)} \cup \texttt{(In(b) - kill(b))}$

```
//set that contains all statements
that may define some variable x in
b
gen(1:a=3;2:a=4)={2}
```

```
//set that contains all statements
that define a variable x that is
also defined in b
kill(1:a=3; 2:a=4)={1,2}
```

Recap: Dataflow Analysis

- Any-path problem
  - The previous two analysis (liveness and reaching definitions) determine if some property holds true along *some path* (no guarantees)
    - a variable is used/live along some path starting from its definition (use-def chain)
    - a definition reaches a block b along some path (without intervening redefinition of the variable involved along that path) (def-use chain)

Recap: Dataflow Analysis

- Forward-flow vs. Backward-flow
  - The previous two analysis (liveness and reaching definitions) determine the properties by computing IN and OUT sets backward and forward to the control flow resp.

Recap: Dataflow Analysis

- Applications of RD (reaching definitions)

  By building def-use chains:

  – RD helps us to analyze if a variable is defined in the program before it is used (think: “uninitialized variable” warnings)

  – RD helps us know what all definitions of ‘x’ reaching a block b. This can enable constant folding if all those definitions assign the same constant to x

Recap: Dataflow Analysis

- Applications of Liveness analysis

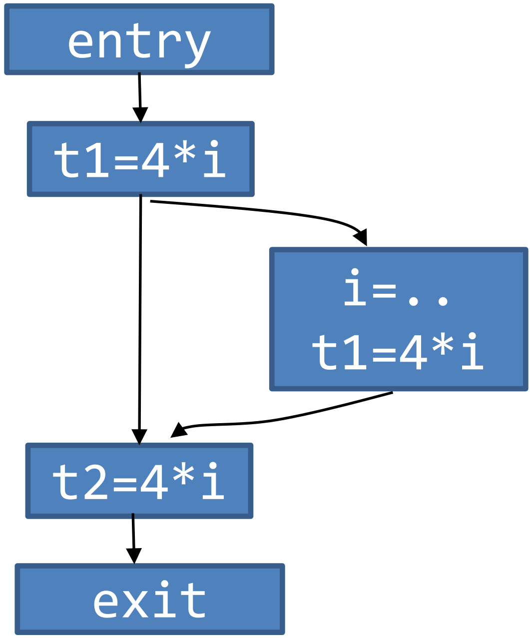
  By building use-def chains:

  – “Undefined variable” warnings

  – Register allocation

Recap: Available Expressions

- Expression "x+y" is available at block b if *every* path from entry node to b computes "x+y" (and x and y are not assigned to/ defined after initial computation of "x+y"



$$In(b) = \bigcap_{i \in Pred(b)} Out(i)$$

$$Out(b) = gen(b) \cup (In(b) - kill(b))$$

```
//set that contains all statements
that may define some variable x in
b e.g. gen(2:j=n)={2}
```

```
//set that contains all statements
that are killed by b's statements
e.g. kill(2:j=n)={2}
```

Recap: Dataflow Analysis

- RD vs. Available Expressions
  - Meet operator: Union vs. Intersection
  - Expression is available at the beginning of a block only if it is available at the end of *all* the predecessor blocks

    vs.

  - A definition reaches the beginning of a block whenever it reaches the end of any one or more of its predecessors

Recap: Dataflow Analysis

- All-path, Forward-flow problem
  - The previous analysis (Available expressions) determines if some property holds true along *all paths* (guarantees exist)
    - Whether an expression is computed along all paths

What is an all-path backward-flow problem?

Summary: Dataflow Analysis

	Any-path	**All-path**
Forward-flow	Reaching Definitions	Available Expressions
Backward-flow	Liveness Analysis	Very-busy expressions*

*also called *anticipated expressions.*

- Avoid recomputing and save space
- Can be used to identify candidates in loop-invariant code motion (see: slide 65, week 12)

Recap: Optimize Loops

- Low level optimization
  - Moving code around in a single loop
  - Examples: loop invariant code motion, strength reduction, loop unrolling
- High level optimization
  - Restructuring loops, often affects multiple loops
  - Examples: loop fusion, loop interchange, loop tiling

Slide Courtesy: Milind Kulkarni

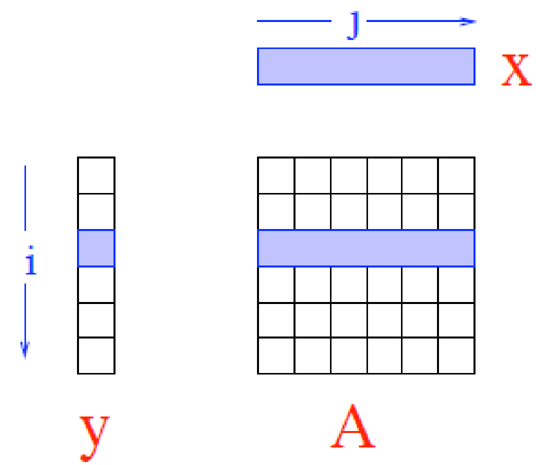
High level loop optimizations

- Many useful compiler optimizations require *restructuring* loops or sets of loops
    - Combining two loops together (*loop fusion*)
    - Switching the order of a nested loop (*loop interchange*)
    - Completely changing the traversal order of a loop (*loop tiling*)
- These sorts of high level loop optimizations usually take place at the AST level (where loop structure is obvious)

Slide Courtesy: Milind Kulkarni

Cache behavior

- Most loop transformations target cache performance
  - Attempt to increase *spatial* or *temporal* locality
  - Locality can be exploited when there is *reuse* of data (for temporal locality) or recent access of nearby data (for spatial locality)
- Loops are a good opportunity for this: many loops iterate through matrices or arrays
- Consider matrix-vector multiply example
  - Multiple traversals of vector: opportunity for spatial and temporal locality
  - Regular access to array: opportunity for spatial locality



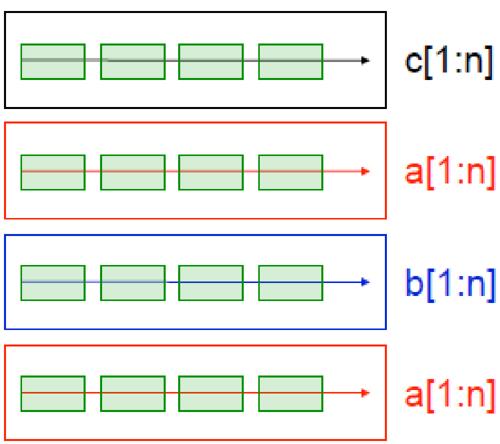
$$y = Ax$$

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    y[i] += A[i][j] * x[j]
```


Slide Courtesy: Milind Kulkarni

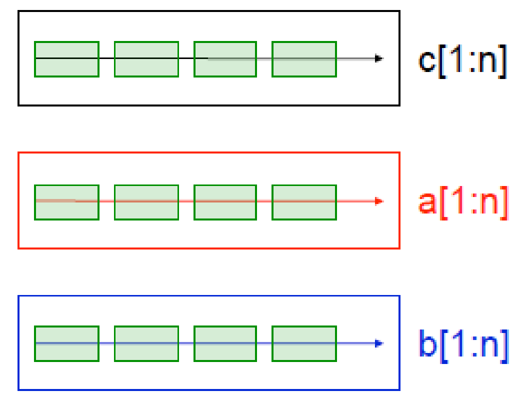
Loop fusion

```
do I = 1, n
  c[i] = a[i]
end do
do I = 1, n
  b[i] = a[i]
end do
```



- Combine two loops together into a single loop
- Why is this useful?
- Is this always legal?

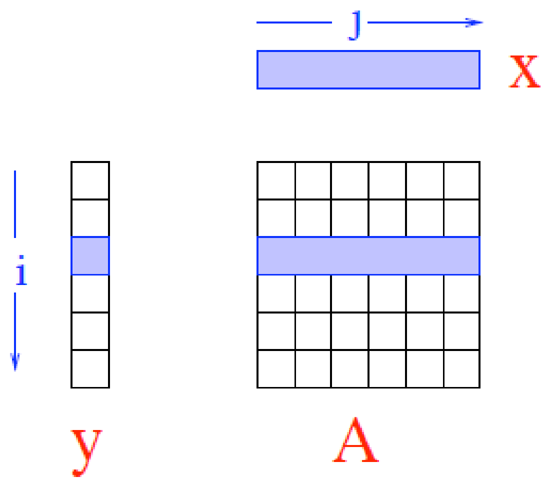
```
do I = 1, n
  c[i] = a[i]
  b[i] = a[i]
end do
```



Slide Courtesy: Milind Kulkarni

Loop interchange

- Change the order of a nested loop
- This is not always legal – it changes the order that elements are accessed!
- Why is this useful?
  - Consider matrix-matrix multiply when A is stored in column-major order (i.e., each column is stored in contiguous memory)

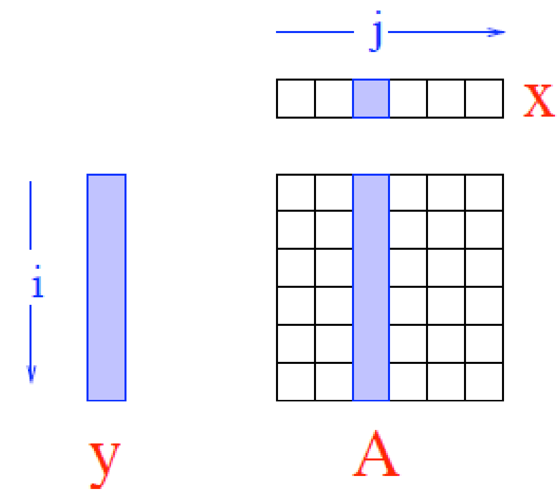



```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    y[i] += A[i][j] * x[j]
```


Slide Courtesy: Milind Kulkarni

Loop interchange

- Change the order of a nested loop
- This is not always legal – it changes the order that elements are accessed!
- Why is this useful?
    - Consider matrix-matrix multiply when A is stored in column-major order (i.e., each column is stored in contiguous memory)




```
for (j = 0; j < N; j++)
  for (i = 0; i < N; i++)
    y[i] += A[i][j] * x[j]
```

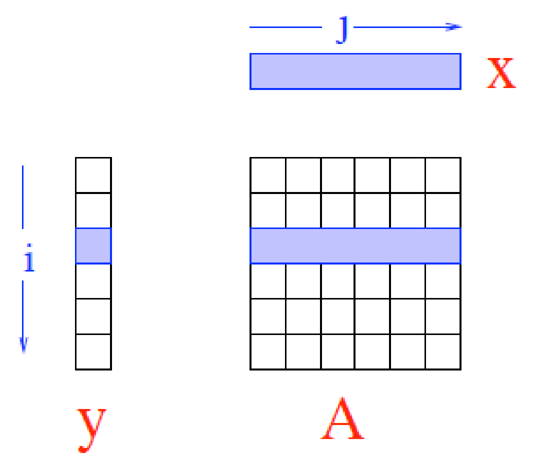

Slide Courtesy: Milind Kulkarni

Loop tiling

- Also called "loop blocking"
- One of the more complex loop transformations
- Goal: break loop up into smaller pieces to get spatial and temporal locality
  - Create new inner loops so that data accessed in inner loops fit in cache
- Also changes iteration order, so may not be legal

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    y[i] += A[i][j] * x[j]
```

```
for (ii = 0; ii < N; ii += B)
  for (jj = 0; jj < N; jj += B)
    for (i = ii; i < ii+B; i++)
      for (j = jj; j < jj+B; j++)
        y[i] += A[i][j] * x[j]
```



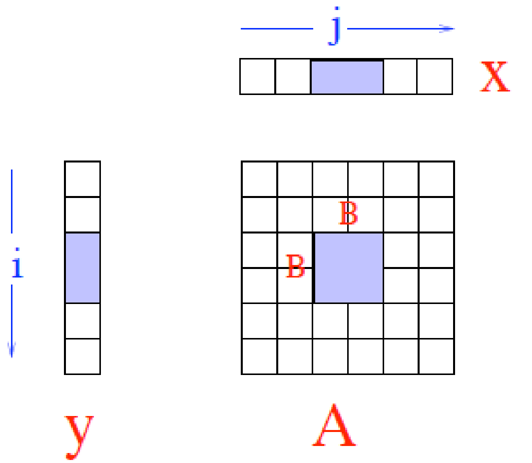
Slide Courtesy: Milind Kulkarni

Loop tiling

- Also called "loop blocking"
- One of the more complex loop transformations
- Goal: break loop up into smaller pieces to get spatial and temporal locality
  - Create new inner loops so that data accessed in inner loops fit in cache
- Also changes iteration order, so may not be legal

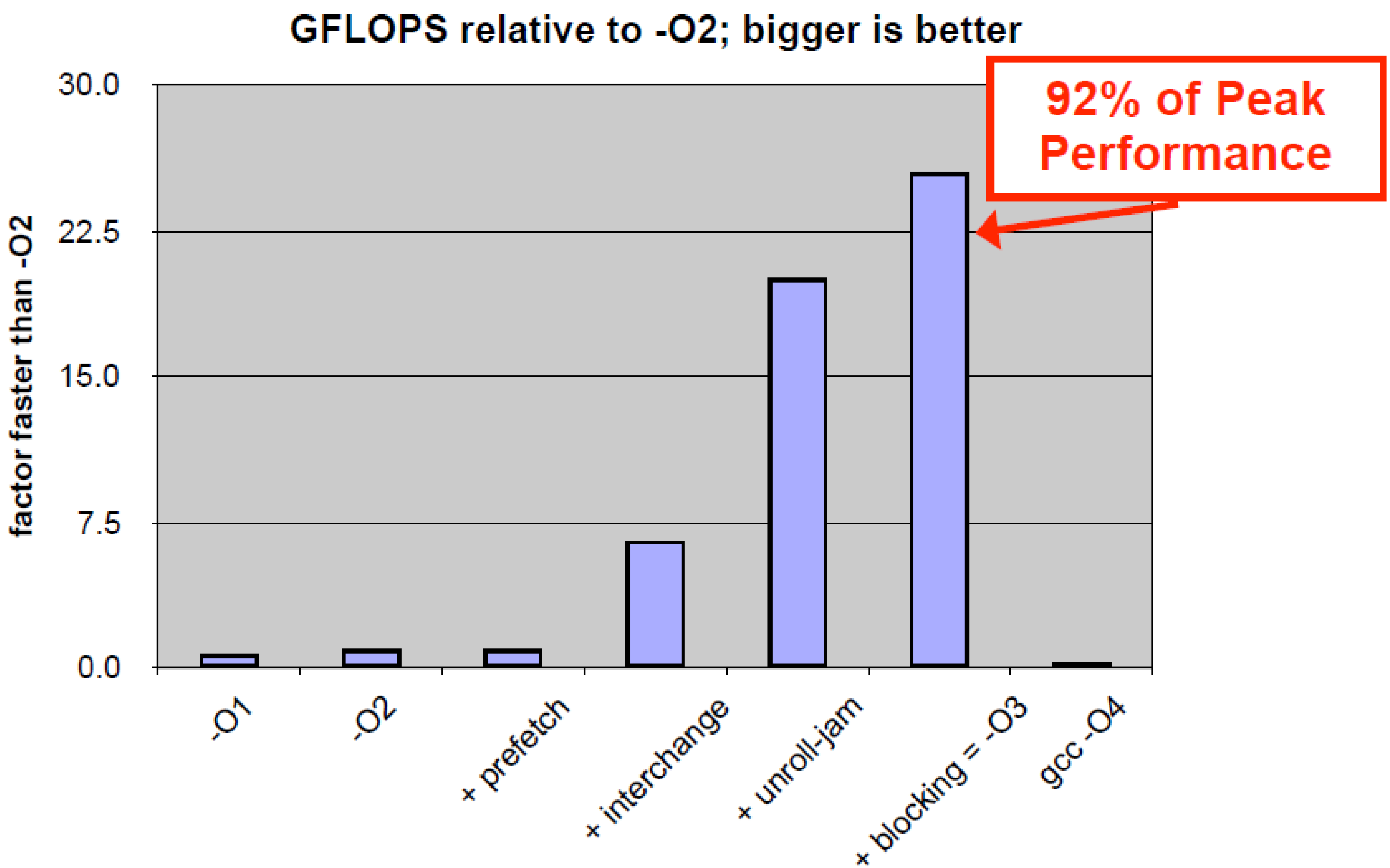
```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    y[i] += A[i][j] * x[j]
```

```
for (ii = 0; ii < N; ii += B)
  for (jj = 0; jj < N; jj += B)
    for (i = ii; i < ii+B; i++)
      for (j = jj; j < jj+B; j++)
        y[i] += A[i][j] * x[j]
```



Slide Courtesy: Milind Kulkarni

In a real (Itanium) compiler



Slide Courtesy: Milind Kulkarni

Loop transformations

- Loop transformations can have dramatic effects on performance
- Doing this legally and automatically is very difficult!
- Researchers have developed techniques to determine legality of loop transformations and automatically transform the loop
    - Techniques like *unimodular transform framework* and *polyhedral framework*
    - These approaches will get covered in more detail in advanced compilers course

Slide Courtesy: Milind Kulkarni

Dependence Analysis

Motivating question

- Can the loops on the right be run in parallel?
  - *i.e.*, can different processors run different iterations in parallel?
- What needs to be true for a loop to be parallelizable?
  - Iterations cannot interfere with each other
  - No *dependence* between iterations

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  c[i] = a[i - 1];
}
```

```
for (i = 1; i < N; i++) {
  a[i] = b[i];
  c[i] = a[i] + b[i - 1];
}
```