Loop optimizations

Agenda

- Low level loop optimizations
  - Code motion
  - Strength reduction
  - Unrolling
- High level loop optimizations
  - Loop fusion
  - Loop interchange
  - Loop tiling

Loop optimization

- 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

Low level loop optimizations

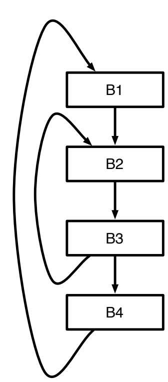
- Affect a single loop
- Usually performed at three-address code stage or later in compiler
- First problem: identifying loops
  - Low level representation doesn't have loop statements!

Identifying loops

- First, we must identify *dominators*
  - Node a dominates node b if every possible execution path that gets to b *must* pass through a
- Many different algorithms to calculate dominators – we will not cover how this is calculated
- A *back edge* is an edge from b to a when a dominates b
- The target of a back edge is a *loop header*

Natural loops

- Will focus on *natural loops* – loops that arise in structured programs
- For a node n to be in a loop with header h
  - n must be dominated by h
  - There must be a path in the CFG from n to h through a back-edge to h
- What are the back edges in the example to the right? The loop headers? The natural loops?



Loop invariant code motion

- Idea: some expressions evaluated in a loop never change; they are *loop invariant*
  - Can move loop invariant expressions outside the loop, store result in temporary and just use the temporary in each iteration
  - Why is this useful?

Identifying loop invariant code

- To determine if a statement

  s: a = b op c

  is loop invariant, find all definitions of b and c that *reach* s
  - A statement t defining b reaches s if there is a path from t to s where b is not re-defined
- s is loop invariant if both b and c satisfy one of the following
  - it is constant
  - all definitions that reach it are from outside the loop
  - only one definition reaches it and that definition is also loop invariant

Moving loop invariant code

- Just because code is loop invariant doesn't mean we can move it!

```
for (...)
   a = b + c
```

```
do
  if (*)
    break
  a = 5
while (*)
c = a;
```

```
for (...)
  if (*)
    a = 5
  else
    a = 6
```

```
a = 5;
for (...)
  if (*)
    a = 4 + c
  b = a
```

- We can move a loop invariant statement a = b op c if
  - The statement dominates all loop exits where a is live
  - There is only one definition of a in the loop
  - a is not live before the loop
- Move instruction to a *preheader*, a new block put right before loop header

Strength reduction

- Like strength reduction peephole optimization
  - Peephole: replace expensive instruction like a * 2 with a << 1
- Replace expensive instruction, multiply, with a cheap one, addition
  - Applies to uses of an *induction variable*
  - Opportunity: array indexing

```
for (i = 0; i < 100; i++)
  A[i] = 0;
```

↓

```
    i = 0;
L2:if (i >= 100) goto L1
    j = 4 * i + &A
    *j = 0;
    i = i + 1;
    goto L2
L1:
```


Strength reduction

- Like strength reduction peephole optimization
  - Peephole: replace expensive instruction like a * 2 with a << 1
- Replace expensive instruction, multiply, with a cheap one, addition
  - Applies to uses of an *induction variable*
  - Opportunity: array indexing

```
for (i = 0; i < 100; i++)
  A[i] = 0;
```

↓

```
    i = 0; k = &A;
L2:if (i >= 100) goto L1
    j = k;
    *j = 0;
    i = i + 1; k = k + 4;
    goto L2
L1:
```


Induction variables

- A *basic induction variable* is a variable i
  - whose only definition within the loop is an assignment of the form i = i ± c, where c is loop invariant
  - Intuition: the variable which determines number of iterations is usually an induction variable
- A *mutual induction variable* j may be
  - defined once within the loop, and its value is a linear function of some other induction variable i such that

    j = c1 * i ± c2 or j = i/c1 ± c2

    where c1, c2 are loop invariant
- A *family* of induction variables include a basic induction variable and any related mutual induction variables

Strength reduction algorithm

- Let j be an induction variable in the family of the basic induction variable i, such that j = c1 * i + c2
  - Create a new variable j'
  - Initialize in preheader

    j' = c1 * i + c2
  - Track value of i. After i = i + c3, perform

    j' = j' + (c1 * c3)
  - Replace definition of i with

    j = j'
- Key: c1, c2, c3 are all loop invariant (or constant), so computations like (c1 * c3) can be moved outside loop

Linear test replacement

- After strength reduction, the loop test may be the only use of the basic induction variable
- Can now eliminate induction variable altogether
- Algorithm
  - If only use of an induction variable is the loop test and its increment, and if the test is always computed
  - Can replace the test with an equivalent one using one of the mutual induction variables

```
i = 2
for (; i < k; i++)
  j = 50*i
  ... = j
```

Strength reduction

```
i = 2; j' = 50 * i
for (; i < k; i++, j' += 50)
  ... = j'
```

Linear test replacement

```
i = 2; j' = 50 * i
for (; j' < 50*k; j' += 50)
  ... = j'
```


Loop unrolling

- Modifying induction variable in each iteration can be expensive
- Can instead *unroll* loops and perform multiple iterations for each increment of the induction variable
- What are the advantages and disadvantages?

```
for (i = 0; i < N; i++)
  A[i] = ...
```

Unroll by factor of 4

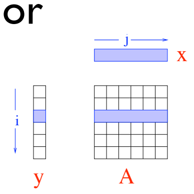
```
for (i = 0; i < N; i += 4)
  A[i] = ...
  A[i+1] = ...
  A[i+2] = ...
  A[i+3] = ...
```


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)

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

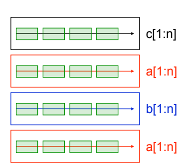


y = Ax

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

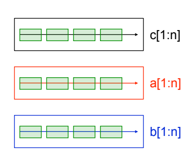

Loop fusion

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



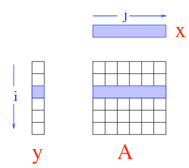
- Combine two loops together into a single loop
- Why is this useful?
- Is this always legal?

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



Loop interchange

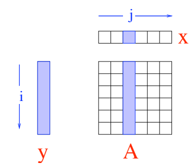
- Change the order of a nested loop
- This is not always legal – it changes the order that elements are accessed!
- Why is this useful?
  - Consider matrix-matrix multiply when A is stored in column-major order (i.e., each column is stored in contiguous memory)




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


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)



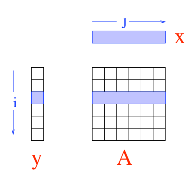

```
for (j = 0; j < N; j++)
  for (i = 0; i < N; i++)
    y[i] += A[i][j] * x[j]
```


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]
```

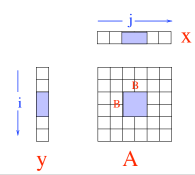


Loop tiling

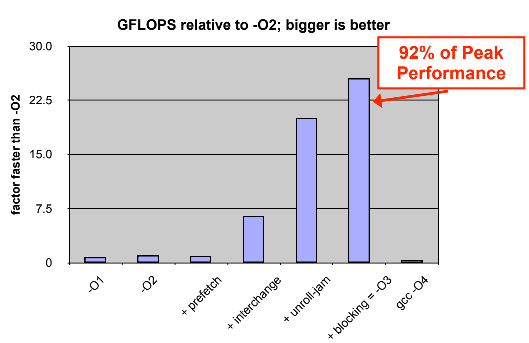
- Also called "loop blocking"
- One of the more complex loop transformations
- Goal: break loop up into smaller pieces to get spatial and temporal locality
  - Create new inner loops so that data accessed in inner loops fit in cache
- Also changes iteration order, so may not be legal

```
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]
```



In a real (Itanium) compiler



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
  - In this class, we will see some simple techniques to reason about high-level loop optimizations