Control flow graphs and loop optimizations

Agenda
- Building control flow graphs
- Low level loop optimizations
  - Code motion
  - Strength reduction
  - Unrolling
- High level loop optimizations
  - Loop fusion
  - Loop interchange
  - Loop tiling

Moving beyond basic blocks
- Up until now, we have focused on single basic blocks
- What do we do if we want to consider larger units of computation
  - Whole procedures?
  - Whole program?
- Idea: capture control flow of a program
- How control transfers between basic blocks due to:
  - Conditionals
  - Loops

Representation
- Use standard three-address code
- Jump targets are labeled
- Also label beginning/end of functions
- Want to keep track of targets of jump statements
- Any statement whose execution may immediately follow execution of jump statement
  - *Explicit* targets: targets mentioned in jump statement
  - *Implicit* targets: statements that follow conditional jump statements
  - The statement that gets executed if the branch is not taken

Running example

```
A = 4
repeat {
  t1 = A * B
  t2 = t1 / C
  if (t2 ≥ W) {
    M = t1 * k
    t3 = M + I
  }
  H = I
  M = t3 - H
} until (T3 ≥ 0)
```

```
1 A = 4
2 t1 = A * B
3 L1: t2 = t1 / C
4   if t2 < W goto L2
5   M = t1 * k
6   t3 = M + I
7 L2: H = I
8   M = t3 - H
9   if t3 ≥ 0 goto L3
10 goto L1
11 L3: halt
```
Control flow graphs

- Divides statements into basic blocks
- Basic block: a maximal sequence of statements \( I_0, I_1, I_2, \ldots, I_n \) such that if \( I_j \) and \( I_{j+1} \) are two adjacent statements in this sequence, then
  - The execution of \( I_j \) is always immediately followed by the execution of \( I_{j+1} \)
  - The execution of \( I_{j+1} \) is always immediately preceded by the execution of \( I_j \)
- Edges between basic blocks represent potential flow of control

Constructing a CFG

- To construct a CFG where each node is a basic block
- Identify leaders: first statement of a basic block
- In program order, construct a block by appending subsequent statements up to, but not including, the next leader
- Identifying leaders
  - First statement in the program
  - Explicit target of any conditional or unconditional branch
  - Implicit target of any branch

Partitioning algorithm

- Input: set of statements, \( \text{stat}(i) = i^{th} \) statement in input
- Output: set of leaders, set of basic blocks where \( \text{block}(x) \) is the set of statements in the block with leader \( x \)
- Algorithm

Running example

\[
\begin{align*}
1 & \quad A = 4 \\
2 & \quad t1 = A * B \\
3 & \quad L1: t2 = t1 / C \\
4 & \quad \text{if} \ t2 < W \text{ goto } L2 \\
5 & \quad M = t1 * k \\
6 & \quad t3 = M + I \\
7 & \quad L2: H = I \\
8 & \quad M = t3 - H \\
9 & \quad \text{if} \ t3 \geq 0 \text{ goto } L3 \\
10 & \quad \text{goto } L1 \\
11 & \quad L3: \text{halt}
\end{align*}
\]

Leaders = \{1, 3, 5, 7, 10, 11\}
Basic blocks = \{ \{1, 2\}, \{3, 4\}, \{5, 6\}, \{7, 8, 9\}, \{10\}, \{11\} \}
Putting edges in CFG

- There is a directed edge from $B_1$ to $B_2$ if
  - There is a branch from the last statement of $B_1$ to the first statement (leader) of $B_2$
  - $B_2$ immediately follows $B_1$ in program order and $B_1$ does not end with an unconditional branch
- Input: block, a sequence of basic blocks
- Output: The CFG

```plaintext
for i = 1 to |block|
  s = last statement of block(i)
  if stat(s) is a branch, then
    for each explicit target y of stat(s)
      create edge from block i to block y
  end for
  if stat(s) is not unconditional then
    create edge from block i to block i+1
  end for
```

Result

```
A = 4
l1 = A * B
j1: t2 = t1/c
if t2 < W goto l2
j3: M = l1 * k
j4: t3 = M + I
l2: H = I
j5: M = t3 - H
if t3 ≥ 0 goto l3
j6: goto l1
l3: halt
```

Discussion

- Some times we will also consider the statement-level CFG, where each node is a statement rather than a basic block
  - Either kind of graph is referred to as a CFG
  - In statement-level CFG, we often use a node to explicitly represent merging of control
  - Control merges when two different CFG nodes point to the same node
- Note: if input language is structured, front-end can generate basic block directly
  - “GOTO considered harmful”

Statement level CFG

```
K = 4
l1 = A + B
j1: t2 = t1/c
if t2 < W goto l2
j3: M = l1 * k
j4: t3 = M + I
l2: H = I
j5: M = t3 - H
if t3 ≥ 0 goto l3
j6: goto l1
l3: halt
```

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 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
  - for (...) if (*) a = 5; else a = 6
  - for (...) if (*) a = 4 + c
- 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;
for (i = 0; i >= 100) goto L1
j = 4 * i + 8A
*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

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

```c
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 \( j \)
  - whose only definition within the loop is an assignment of the form \( j = j \pm c \), where \( c \) is loop invariant
  - Intuition: the variable which determines number of iterations is usually an induction variable
- A mutual induction variable \( i \) may be
  - defined once within the loop, and its value is a linear function of some other induction variable \( j \) such that
    \[
    i = c_1 * j \pm c_2 \text{ or } i = j/c_1 \pm c_2
    \]
    where \( c_1, c_2 \) are loop invariant
- A family of induction variables include a basic induction variable and any related mutual induction variables

**Strength reduction algorithm**

- Let \( i \) be an induction variable in the family of the basic induction variable \( j \), such that \( i = c_1 * j + c_2 \)
- Create a new variable \( i' \)
- Initialize in preheader
  \( i' = c_1 * j + c_2 \)
- Track value of \( j \). After \( j = j + c_3 \), perform
  \( i' = i' + (c_1 * c_3) \)
- Replace definition of \( i \) with \( i' \)
  - Key: \( c_1, c_2, c_3 \) are all loop invariant (or constant), so computations like \( (c_1 * c_3) \) 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

**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?

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

```c
for (i = 0; i < N; i += 4)
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

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

Regular access to array: opportunity for locality

### Why is this useful?
- Elements can be accessed more efficiently

### Loop fusion

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

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

```
for (i = ii; i < ii+B; i++)
  y = A*x
```

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++)
  y[i] += A[i][j] * x[j]
```

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

```
for (i = ii; i < ii+B; i++)
  y = A*x
```

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++)
  y[i] += A[i][j] * x[j]
```

```
for (i = ii; i < ii+B; i++)
  y = A*x
```

```
for (i = ii; i < ii+B; i++)
  y = A*x
```

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++)
  y[i] += A[i][j] * x[j]
```

```
for (i = ii; i < ii+B; i++)
  y = A*x
```
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