CS412/413

Introduction to Compilers
Radu Rugina

Lecture 26: Loop Optimizations
31 Mar 03

Loop optimizations

- Now we know which are the loops
- Next: optimize these loops
  - Loop invariant code motion
  - Strength reduction of induction variables
  - Induction variable elimination

Loop Invariant Code Motion

- Idea: if a computation produces same result in all loop iterations, move it out of the loop
- Example: for (i=0; i<10; i++)
  a[i] = 10*i + x*x;
- Expression x*x produces the same result in each iteration; move it of the loop:
  t = x*x;
  for (i=0; i<10; i++)
  a[i] = 10*i + t;

Loop Invariant Computation

- An instruction a = b OP c is loop-invariant if each operand is:
  - Constant, or
  - Has all definitions outside the loop, or
  - Has exactly one definition, and that is a loop-invariant computation
- Reaching definitions analysis computes all the definitions of x and y which may reach t = x OP y

Algorithm

INV = Ø
Repeat
  for each instruction i \in INV
    if operands are constants, or
    have definitions outside the loop, or
    have exactly one definition d \in INV
      then INV = INV U {i}
Until no changes in INV

Code Motion

- Next: move loop-invariant code out of the loop
- Suppose a = b OP c is loop-invariant
- We want to hoist it out of the loop
- Code motion of a definition d: a = b OP c in pre-header is valid if:
  1. Definition d dominates all loop exits where a is live
  2. There is no other definition of a in loop
  3. All uses of a in loop can only be reached from definition d
Other Issues

- **Preserve dependencies** between loop-invariant instructions when hoisting code out of the loop

```c
for (i=0; i<N; i++) {  
    x = y+z;  
    x = y+z;  
    a[i] = 10*i + x*x;  
}  
```

- **Nested loops**: apply loop invariant code motion algorithm multiple times

```c
for (i=0; i<N; i++)  
    for (j=0; j<M; j++)  
        a[i][j] = x*x + 10*i + 100*j;  
```

Induction Variables

- **An induction variable** is a variable in a loop, whose value is a function of the loop iteration number \( v = f(i) \)

- In compilers, this a linear function:

\[
f(i) = c^*i + d
\]

- **Observation**: linear combinations of linear functions are linear functions

  - Consequence: linear combinations of induction variables are induction variables

Families of Induction Variables

- Each basic induction variable defines a family of induction variables

  - Each variable in the family of \( i \) is a linear function of \( i \)

- A variable \( k \) is in the family of basic variable \( i \) if:

  1. \( k = i^j \) (the basic variable itself)
  2. \( k \) is a linear function of other variables in the family of \( i \):

\[
k = c^*i^j + d,
\]

where \( j = \text{Family}(i) \)

- A triple \( <i,a,b> \) denotes an induction variable \( k \) in the family of \( i \) such that:

\[
k = i^a + b
\]

- Triple for basic variable \( i \) is \( <i,1,0> \)

Dataflow Analysis Formulation

- Detection of induction variables: can formulate problem using the dataflow analysis framework

  - Analyze loop sub-graph, except the back edge

  - Analysis is similar to constant folding

- **Dataflow information**: a function \( F \) that assigns a triple to each variable:

  \[
  F(k) = \langle i,a,b \rangle,
  \]

where \( k \) is an induction variable in family of \( i \)

\[
F(k) = 1 \quad : k \text{ is not an induction variable}
\]

\[
F(k) = 0 \quad : \text{don't know if } k \text{ is an induction variable}
\]

Dataflow Analysis Formulation

- **Meet operation**: If \( F_1 \) and \( F_2 \) are two functions, then:

\[
(F_1 \sqcap F_2)(i) = \langle i,a,b \rangle \quad \text{if } F_1(k) = F_2(k) = \langle i,a,b \rangle
\]

\[
F_1(k) \sqcap F_2(k) = 1, \text{ otherwise}
\]

(in other words, use a flat lattice)

- **Initialization**:

  - Detect all basic induction variables

  - At loop header: \( F(i) = \langle i,1,0 \rangle \) for each basic variable \( i \)

- **Transfer function**:

  - \( F \) is information before instruction \( I \)

  - Compute information \( F' \) after \( I \)
Dataflow Analysis Formulation

- For a definition \( k = j + c \), where \( k \) is not basic induction variable
  \[ F(v) = \begin{cases} a, b+c, & \text{if } v = k \text{ and } F(j) = a, b \\ F(v), & \text{otherwise} \end{cases} \]
- For a definition \( k = j^c \), where \( k \) is not basic induction variable
  \[ F(v) = \begin{cases} a, b^c, & \text{if } v = k \text{ and } F(j) = a, b \\ F(v), & \text{otherwise} \end{cases} \]
- For any other instruction and any variable \( k \) in def[1] :
  \[ F(v) = \ldots, \text{if } F(v) = a, b \]
  \[ F(v), \text{otherwise} \]

Strength Reduction

- Basic idea: replace expensive operations (multiplications) with cheaper ones (additions) in definitions of induction variables

  ```
  while (i<10) {
    j = ...; // \langle i, j, k \rangle
    a[i] = a[i] - 2;
    i = i+2;
  }
  s = s+6;
  }
  ```

  ```
  while (i<10) {
    j = s;
    a[i] = a[i] - 2;
    i = i+2;
    s = s+6;
  }
  ```

- Benefit: cheaper to compute \( s = s+6 \) than \( j = 3^i \)
  - \( s = s+6 \) requires an addition
  - \( j = 3^i \) requires a multiplication

General Algorithm

- Algorithm:
  For each induction variable \( j \) with triple \( \langle i, a, b \rangle \)
  whose definition involves multiplication:
  1. create a new variable \( s \)
  2. replace definition of \( j \) with \( j=s \)
  3. immediately after \( i=i+c \), insert \( s = s+a^c \)
     (here \( a^c \) is constant)
  4. insert \( s = a^i+b \) into preheader

  Correctness: this transformation maintains the invariant that \( s = a^i+b \)

Strength Reduction

- Gives opportunities for copy propagation, dead code elimination

  ```
  s = 3^i+1;
  while (i<10) {
    j = s;
    a[i] = a[i] - 2;
    i = i+2;
    s = s+6;
  }
  ```

  ```
  s = 3^i+1;
  while (i<10) {
    j = s;
    a[i] = a[i] - 2;
    i = i+2;
    s = s+6;
  }
  ```

Induction Variable Elimination

- Idea: eliminate each basic induction variable whose only uses
  are in loop test conditions and in their own definitions \( i = i+c \)
  - rewrite loop test to eliminate induction variable

  ```
  s = 3^i+1;
  while (i<10) {
    a[s] = a[s] - 2;
    i = i+2;
    s = s+6;
  }
  ```

  - When are induction variables used only in loop tests?
    - Usually, after strength reduction
    - Use algorithm from strength reduction even if definitions
      of induction variables don't involve multiplications

Induction Variable Elimination

- Rewrite test condition using derived induction variables
- Remove definition of basic induction variables (if not used after the loop)

  ```
  s = 3^i+1;
  while (i<10) {
    a[s] = a[s] - 2;
    i = i+2;
    s = s+6;
  }
  ```

  ```
  s = 3^i+1;
  while (s<31) {
    a[s] = a[s] - 2;
    i = i+2;
    s = s+6;
  }
  ```
Induction Variable Elimination
For each basic induction variable \( i \) whose only uses are
- The test condition \( i < u \)
- The definition of \( i: i = i + c \)
For each derived induction variable \( k \) in its family,
with triple \(<i,c,d>\)
Replace test condition \( i < u \) with \( k < c^*u + d \)
Remove definition \( i = i + c \) if \( i \) is not live on loop exit

Where We Are
- Defined dataflow analysis framework
- Used it for several analyses
  - Live variables
  - Available expressions
  - Reaching definitions
  - Constant folding
- Loop transformations
  - Loop invariant code motion
  - Induction variables
- Next:
  - Pointer alias analysis

Pointer Alias Analysis
- Most programs use variables containing addresses
  - E.g. pointers (C++, C++, references (C, Python), call-by-reference parameters (F ortran)
- Pointer alias:
  - Multiple names for the same memory location, which occur when dereferencing variables that hold
  - memory addresses
- Problem:
  - Don’t know what variables read and written by accesses
  - via pointer aliases (e.g. \( *p = y \) may write any memory location
  - \( x = *p \) may read any memory location)
- Such assumptions may affect the precision of other analyses
- Example 1: Live variables
  - before any instruction \( x = *p \), all the variables may be live
- Example 2: Constant folding
  - \( a = 1; b = 2; *p = 6; c = a+b; \)
  - \( c = 3 \) at the end of code only if \( *p \) is not an alias for \( a \) or \( b \)
- Conclusion: precision of result for all other analyses depends
  - on the amount of alias information available
  - hence, it is a fundamental analysis

Alias Analysis Problem
- Goal: for each variable \( v \) that may hold an address,
  - compute the set \( \text{Ptr}(v) \) of possible targets of \( v \)
  - \( \text{Ptr}(v) \) is a set of variables (or objects)
  - \( \text{Ptr}(v) \) includes stack- and heap-allocated variables (objects)
- Is a “may” analysis: if \( x \in \text{Ptr}(v) \), then \( v \) may hold the
  - address of \( x \) in some execution of the program
- No alias information: for each variable \( v \), \( \text{Ptr}(v) = \{ v \} \)
  - where \( V \) is the set of all variables in the program

Simple Alias Analyses
- Address-taken analysis:
  - Consider \( \text{AT} = \text{set of variables whose addresses are taken}
  - Then, \( \text{Ptr}(v) = \text{AT} \), for each pointer variable \( v \)
  - Addresses of heap variables are always taken at allocation
  - sites (e.g. \( x = \text{new int}[2]; x = \text{malloc}(8) \))
  - Hence \( \text{AT} \) includes all heap variables
- Type-based alias analysis:
  - If \( v \) is a pointer (or reference) to type \( T \), then \( \text{Ptr}(v) \) is the
  - set of all variables of type \( T \)
  - Example: \( p \) and \( q \) can be aliases only if \( p \) and \( q \) are
    - references to objects of the same type
    - Works only for strongly-typed languages
Dataflow Alias Analysis

- Dataflow analysis: for each variable $v$, compute points-to set $\text{Ptr}(v)$ at each program point
- Dataflow information: set $\text{Ptr}(v)$ for each variable $v$
  - Can be represented as a graph $G \subseteq 2^{V \times V}$
  - Nodes = $V$ (program variables)
  - There is an edge $v \rightarrow u$ if $u \in \text{Ptr}(v)$

$$\begin{align*}
    \text{Ptr}(x) &= (y) \\
    \text{Ptr}(y) &= (z, t)
\end{align*}$$

Dataflow Alias Analysis

- Dataflow Lattice: $(2^{V \times V}, \supseteq)$
  - $V \times V$ is set of all possible points-to relations
  - "may" analysis: top element is $\emptyset$, meet operation is $\cup$
- Transfer functions: use standard dataflow transfer functions:
  - $\text{out}[i] = ((n[i]\cdot\text{kill}[i]) \cup \text{gen}[i])$
  - $\text{p} = \text{addr q}$ \quad $\text{kill}[i] = (p) \times V$
  - $\text{gen}[i] = ((p,q))$
  - $\text{p} = q$ \quad $\text{kill}[i] = (p) \times V$
  - $\text{gen}[i] = (p) \times \text{Ptr}(q)$
  - $\ast q$ \quad $\text{kill}[i] = \cdots$
  - $\text{gen}[i] = \text{Ptr}(p) \times \text{Ptr}(q)$

For all other instruction, $\text{kill}[i] = \emptyset$, $\text{gen}[i] = \emptyset$

- Transfer functions are monotonic, but not distributive!

Alias Analysis Example

Program:

- $x = 8a$
- $y = 8b$
- $c = 8i$
- if($i$) $x = y$
- $\ast x = c$

CFG:

- $x = 8a$
- $y = 8b$
- $c = 8i$
- if($i$) $x = y$
- $\ast x = c$

Points-to Graph (at the end of program):

- $x \rightarrow a$
- $y \rightarrow b$
- $c \rightarrow i$

- $\ast x \rightarrow c$

Alias Analysis Uses

- Once alias information is available, use it in other dataflow analyses

- Example: Live variable analysis
  - Use alias information to compute $\text{use}[i]$ and $\text{def}[i]$ for load and store statements:
    - $x = [y]$ \quad $\text{use}[i] = \{y\} \cup \text{Ptr}(y)$ \quad $\text{def}[i] = (x)$
    - $[x] = y$ \quad $\text{use}[i] = (x, y)$ \quad $\text{def}[i] = \text{Ptr}(x)$

CS 412/413 Spring 2003 Introduction to Compilers 25

CS 412/413 Spring 2003 Introduction to Compilers 26