

CS412/413

Introduction to Compilers
Radu Rugina

Lecture 29: Loop Optimizations
12 Apr 04

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

CS 412/413 Spring 2004

Introduction to Compilers

2

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;
```

CS 412/413 Spring 2004

Introduction to Compilers

3

Loop Invariant Computation

- An instruction $a = b \text{ OP } c$ 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 $t = x \text{ OP } y$

CS 412/413 Spring 2004

Introduction to Compilers

4

Algorithm

$INV = \emptyset$

Repeat

for each instruction $i \notin INV$

if operands are constants, or

have definitions outside the loop, or

have exactly one definition $d \in INV$

then $INV = INV \cup \{i\}$

Until no changes in INV

CS 412/413 Spring 2004

Introduction to Compilers

5

Code Motion

- Next: move loop-invariant code out of the loop
- Suppose $a = b \text{ OP } c$ is loop-invariant
- We want to hoist it out of the loop
- Code motion of a definition $d: a = b \text{ OP } c$ 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

CS 412/413 Spring 2004

Introduction to Compilers

6

Other Issues

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

```

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

```

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

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

```

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

```

t1 = x*x;
for (j=0; j<M; j++) {
    t2 = t1 + 10*j;
    for (i=0; i<N; i++)
        a[i][j] = t2 + 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

Induction Variables

- Two categories of induction variables
- Basic induction variables:** only incremented in loop body
 $i = i + c$
where c is a constant (positive or negative)
- Derived induction variables:** expressed as a linear function of an induction variable
 $k = c*j + d$
where:
 - either j is basic induction variable
 - or j is derived induction variable in the family of i and:
 - No definition of j outside the loop reaches definition of k
 - i is not defined between the definitions of j and k

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:
 - $k = i$ (the basic variable itself)
 - k is a linear function of other variables in the family of i :
 $k = c*j + d$, where $j \in \text{Family}(i)$
- A triple $\langle i, a, b \rangle$ denotes an induction variable k in the family of i such that: $k = i*a + b$
– Triple for basic variable i is $\langle i, 1, 0 \rangle$

Dataflow Analysis Formulation

- Detection of induction variables:** can formulate problem using the dataflow analysis framework
 - Analyze loop body sub-graph, except the back edge
 - Analysis is similar to constant folding
- Dataflow information:** a function F that assigns a triple to each variable:
 - $F(k) = \langle i, a, b \rangle$, if k is an induction variable in family of i
 - $F(k) = \perp$: k is not an induction variable
 - $F(k) = \top$: don't know if k is an induction variable

Dataflow Analysis Formulation

- Meet operation:** if $F1$ and $F2$ are two functions, then:
 $(F1 \sqcap F2)(v) = \begin{cases} \langle i, a, b \rangle & \text{if } F1(k)=F2(k)=\langle i, a, b \rangle \\ \perp & \text{otherwise} \end{cases}$
(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:**
 - consider 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
 $F'(v) = \langle i, a, b + c \rangle$, if $v = k$ and $F(j) = \langle i, a, b \rangle$
 $F'(v) = F(v)$, otherwise
- For a definition $k = j * c$, where k is not basic induction variable
 $F'(v) = \langle i, a * c, b * c \rangle$, if $v = k$ and $F(j) = \langle i, a, b \rangle$
 $F'(v) = F(v)$, otherwise
- For any other instruction and any variable k in $\text{def}[I]$:
 $F'(v) = \perp$, if $F(v) = \langle k, a, b \rangle$
 $F'(v) = F(v)$, otherwise

CS 412/413 Spring 2004

Introduction to Compilers

13


Strength Reduction

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

```

while (i < 10) {
  j = ...; // <i, 3, 1>
  a[j] = a[j] - 2;
  i = i + 2;
}

```



```

s = 3*i + 1;
while (i < 10) {
  j = s;
  a[j] = a[j] - 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

CS 412/413 Spring 2004

Introduction to Compilers

14

General Algorithm

- Algorithm:

For each induction variable j with triple $\langle i, a, b \rangle$ whose definition involves multiplication:

- create a new variable s
- replace definition of j with $j = s$
- immediately after $i = i + c$, insert $s = s + a * c$ (here $a * c$ is constant)
- insert $s = a * i + b$ into preheader

- Correctness:
this transformation maintains the invariant that $s = a * i + b$

CS 412/413 Spring 2004

Introduction to Compilers

15


Strength Reduction

- Gives opportunities for copy propagation, dead code elimination

```

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

```



```

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

```

CS 412/413 Spring 2004

Introduction to Compilers

16

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

CS 412/413 Spring 2004

Introduction to Compilers

17


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;
  s = s + 6;
}

```

CS 412/413 Spring 2004

Introduction to Compilers

18

Induction Variable Elimination

For each basic induction variable i whose only uses are

- The test condition $i < u$
- The definition of i : $i = i + c$

Take a derived induction variable k in its family,
with triple $\langle i, c, d \rangle$

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 languages use variables containing addresses
 - E.g. pointers (C,C++), references (Java), call-by-reference parameters (Pascal, C++, Fortran)
- **Pointer aliases:** 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. $*p=y$, $x=*p$, $p.f=y$, $x=p.f$, etc.)
 - Need to know accessed variables to compute dataflow information after each instruction

Pointer Alias Analysis

- **Worst case scenarios**
 - $*p = y$ 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
 $a = 1; b = 2; *p = 0; c = a+b;$
 $c = 3$ 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

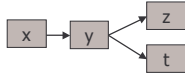
Simple Alias Analyses

- **Address-taken analysis:**
 - Consider $AT =$ set of variables whose addresses are taken
 - Then, $\text{Ptr}(v) = AT$, 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 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: $p.f$ and $q.f$ 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 $u \in \text{Ptr}(v)$

$\text{Ptr}(x) = \{y\}$
 $\text{Ptr}(y) = \{z, t\}$



CS 412/413 Spring 2004

Introduction to Compilers

25

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:
 $\text{out}[I] = (\text{in}[I] - \text{kill}[I]) \cup \text{gen}[I]$

$p = \text{addr } q$ $\text{kill}[I] = \{p\} \times V$ $\text{gen}[I] = \{(p, q)\}$
 $p = q$ $\text{kill}[I] = \{p\} \times V$ $\text{gen}[I] = \{p\} \times \text{Ptr}(q)$
 $p = *q$ $\text{kill}[I] = \{p\} \times V$ $\text{gen}[I] = \{p\} \times \text{Ptr}(\text{Ptr}(q))$
 $*p = q$ $\text{kill}[I] = \dots$ $\text{gen}[I] = \text{Ptr}(p) \times \text{Ptr}(q)$
 For all other instruction, $\text{kill}[I] = \{\}$, $\text{gen}[I] = \{\}$

- Transfer functions are monotonic, but not distributive!

CS 412/413 Spring 2004

Introduction to Compilers

26

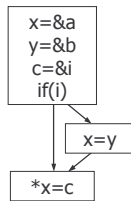
Alias Analysis Example

Program

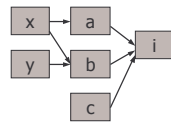
```

x=&a;
y=&b;
c=&i;
if(i) x=y;
*x=c;
  
```

CFG



Points-to Graph (at the end of program)



CS 412/413 Spring 2004

Introduction to Compilers

27

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:

$x = *y$ $\text{use}[I] = \{y\} \cup \text{Ptr}(y)$ $\text{def}[I] = \{x\}$
 $*x = y$ $\text{use}[I] = \{x, y\}$ $\text{def}[I] = \text{Ptr}(x)$

CS 412/413 Spring 2004

Introduction to Compilers

28