CS412/CS413

Introduction to Compilers

Tim Teitelbaum

Lecture 30: Loop Optimizations and Pointer Analysis

07 Apr 08

Loop optimizations

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

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

- Basic induction variable: a variable whose only definition in the loop body is of the form

  i = i + c

  where c is a loop-invariant value

- Derived induction variables: Each basic induction variable i defines a family of induction variables Family(i)
  - i $\in$ Family(i)
  - k $\in$ Family(i) if there is only one definition of k in the loop body , and it has the form k = c*j or k=j+c, where

    (a) j $\in$ Family(i)

    (b) c is loop invariant

    (c) The only definition of j that reaches the definition of k is in the loop

    (d) There is no definition of i between the definitions of j and k

Representation

- Representation of induction variables in family i by triples:
  - Denote basic induction variable i by <i, 1, 0>
  - Denote induction variable k=i*a+b by triple <i, a, b>

Finding Induction Variables

Scan loop body to find all basic induction variables

do

Scan loop to find all variables k with one assignment of form k = j*b, where j is an induction variable <i,c,d>, and make k an induction variable with triple <i,c*b,d>

Scan loop to find all variables k with one assignment of form k = j±b where j is an induction variable with triple <i,c,d>, and make k an induction variable with triple <i,c,b±d>

until no more induction variables found

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

General Algorithm

- Algorithm:

For each induction variable j with triple <i,a,b>
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: transformation maintains invariant s = a*i+b

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;
}
```

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;
   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

  - Take a derived induction variable k in family i, 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 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

- Example1: 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 Ptr(v) of possible targets of v
  - Ptr(v) is a set of variables (or objects)
  - Ptr(v) includes stack- and heap-allocated variables (objects)

- Is a "may" analysis: if $x \in Ptr(v)$, then v may hold the address of x in some execution of the program

- No alias information: for each variable v, Ptr(v) = V, where V is the set of all variables in the program

Simple Alias Analyses

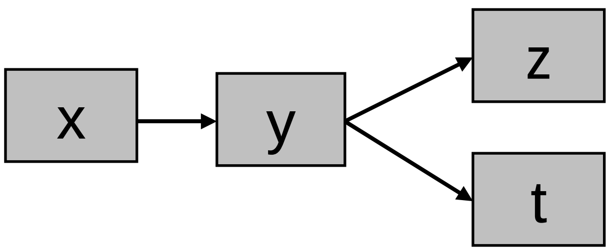
- Address-taken analysis:
  - Consider AT = set of variables whose addresses are taken
  - Then, Ptr(v) = AT, for each pointer variable v
  - Addresses of heap variables are always taken at allocation sites (e.g., x = new int[2]; x=malloc(8);)
  - Hence AT includes all heap variables
- Type-based alias analysis:
  - If v is a pointer (or reference) to type T, then 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 Ptr(v) at each program point

- Dataflow information: set 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 $u \in Ptr(v)$

Ptr(x) = {y}
Ptr(y) = {z,t}



Dataflow Alias Analysis

- Dataflow Lattice: $(2^{V \times V}, \supseteq)$
  - V x V represents “every variable may point to every var.”
  - “may” analysis: top element is $\emptyset$, meet operation is $\cup$

- Transfer functions: use standard dataflow transfer functions: out[I] = (in[I]-kill[I]) ∪ gen[I]

p = addr q	kill[I]={p} x V	gen[I]={<p,q>}
p = q	kill[I]={p} x V	gen[I]={p} x Ptr(q)
p = *q	kill[I]={p} x V	gen[I]={p} x Ptr(Ptr(q))
*p = q	kill[I]= ...	gen[I]=Ptr(p) x Ptr(q)

For all other instruction, kill[I] = {}, gen[I] = {}

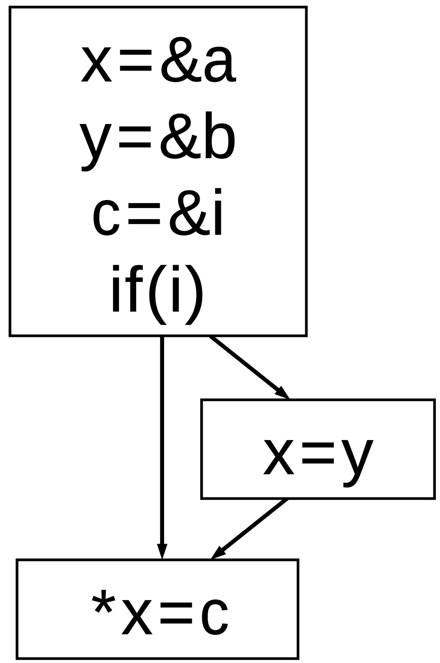
- Transfer functions are monotonic, but not distributive!

Alias Analysis Example

Program

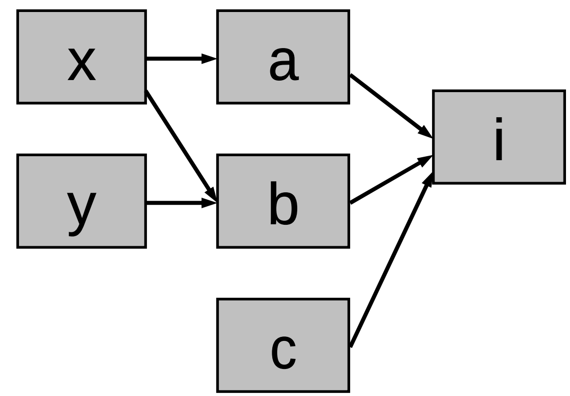
```
x=&a;
y=&b;
c=&i;
if(i) x=y;
*x=c;
```

CFG



Points-to Graph

(at the end of program)



Alias Analysis Uses

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

- Example: Live variable analysis

  Use alias information to compute use[I] and def[I] for load and store statements:

x = *y	use[I] = {y} ∪ Ptr(y)	def[I]={x}
*x = y	use[I] = {x,y}	def[I]=Ptr(x)