Local Optimizations

15-745 Optimizing Compilers
Spring 2006

Peter Lee

Reminders

* Task 0 due on Thursday
* Find a partner
* Read up on
  * intermediate representations (4.1-4.4)
  * local optimizations (12.1, 12.3-4)

Extent of optimizations

* Local optimizations work entirely on individual basic blocks, in isolation
* Global optimizations consider whole procedure bodies
* Interprocedural optimizations work across procedure call boundaries
* Whole program optimizations, ...

Constant Folding

Constant folding

- Evaluation of integer and boolean expressions whose operands are constants
  - in some languages, must not remove division by zero or integer overflow
- Some compilers will also treat floats
- Easily applied to linearized code, in conjunction with constant propagation, copy propagation, and value numbering

Algebraic Simplifications

Integer/boolean simplifications

- i+0 = 0+i = i-0 = i
- 0-i = -i
- i*1 = 1*i = i/1 = i
- i*0 = 0*i = 0
- -(-i) = i
- i+(-j) = i-j
- b or true = true or b = true
- b or false = false or b = b
- b and true = true and b = b
- b and false = false and b = false

Address Arithmetic Simplifications

Associativity

- Generally speaking, re-associating expressions is not safe with respect to overflows
  - E.g., (i-j) + (i-j) = 2*i - 2*j
- But address arithmetic is assumed to be safe with respect to overflow, and so re-association can be applied
- Furthermore, the compiler can control the form of address arithmetic that appears

Address arithmetic example

The classic example (in Pascal):

```
var a : array[lo1..hi1, lo2..hi2] of eltype;
i, j : integer;

 ...

do j = lo2 to hi2 begin
  a[i,j] := b + a[i,j]
end
```

Address arithmetic

- Compiler generates the following address computation for a[i,j]:
  - base_a + ((i-lo1) * (hi2-lo2+1) + j-lo2) * w
- First, re-associate to get:
  - -(lo1 * (hi2-lo2+1) - lo2) * w + base_a + all constant!
  - (hi2-lo2+1) * i * w + loop invariant!
  - j * w
    w is a power of 2 (hence shift)

Simplifying address arithmetic

- Because the compiler uses a specific idiom for most address computations, it is usually possible to find a set of simplification rules (and an order for applying them) that will successfully simply them
- Usually expressed as IR tree transform rules
  - try each rule, in order
  - when one applies, apply it, and then start over

From Muchnick

1. c1+c2 ⇒ sum(c1,c2)
2. t+c ⇒ c+t
3. c1*c2 ⇒ prod(c1,c2)
4. t*c ⇒ c*t
5. c1-c2 ⇒ diff(c1,c2)
6. t-c ⇒ -c + t
7. t1+(t2+t3) ⇒ (t1+t2)+t3
8. t1*(t2*t3) ⇒ (t1*t2)*t3
9. (c1+t)+c2 ⇒ sum(c1,c2)+t
10. (c1*t)*c2 ⇒ prod(c1,c2)*t
11. (c1+t)*c2 ⇒ prod(c1,c2)+(c2*t)
12. c1*(c2+t) ⇒ prod(c1,c2)+(c1*t)
13. (t1+t2)*c ⇒ (c*t1)+(c*t2)
14. c* (t1+t2) ⇒ (c*t1)+(c*t2)
15. (t1-t2)*c ⇒ (c*t1)-(c*t2)
16. c* (t1-t2) ⇒ (c*t1)-(c*t2)
17. (t1+t2)*t3 ⇒ (t1*t3)+(t2*t3)
18. t1*(t2+t3) ⇒ (t1*t2)+(t1*t3)
19. (t1-t2)*t3 ⇒ (t1*t3)-(t2*t3)
20. t1*(t2-t3) ⇒ (t1*t2)-(t1*t3)

- sum and prod are the standard ops on ints
- t, t1, t2, and t3 are arbitrary sub-trees
- rules are attempted in order

Value Numbering

Value numbering

* A classical local optimization
* John Cocke and Jack Schwartz, "Programming Languages and their Compilers", unpublished manuscript, 1970
* A very simple idea (reinvented many times):
  * each value gets its own "number", which is essentially a hash value
  * arrange for common subexpressions to evaluate to the same number

Terminology alert!

* Classical compiler optimization literature uses the words value and variable almost interchangeably
* This is because in the IR, we generally arrange for all values of interest to be assigned to a variable (i.e., a temp)
* So, one could say "variable numbering" instead of "value numbering"

The hash

- Given a statement of the form
  - t = a ⊕ b
- we compute a value number for t by hashing on a, ⊕, and b.
  - The value numbers of a and b are used to compute the value number of t
  - if ⊕ is commutative, the same hash should be returned for either argument order

Example

```
a = x + y
b = x + y
if !z goto L1
x = !z
c = x * y
if x * y goto L2
```

Hash on (+,x,y)
No hit, so store a=x+y in table

Hash on (+,x,y)
Hit, so replace x+y with lhs of hash table entry

```
a = x + y
b = a
if !z goto L1
x = !z
c = x * y
if x * y goto L2
```

Example

```
a = x + y
b = a
if !z goto L1
x = !z
c = x * y
if x * y goto L2
```

Ensure all values are named...

```
a = x + y
b = a
t1 = !z
if t1 goto L1
x = !z
c = x * y
t2 = x * y
if t2 goto L2
```

eliminate !z

```
a = x + y
b = a
t1 = !z
if t1 goto L1
x = t1
c = x * y
t2 = c
if t2 goto L2
```

elim. x*y

```
a = x + y
b = a
t1 = !z
if t1 goto L1
x = t1
c = x * y
t2 = x * y
if t2 goto L2
```

Another example

```
i = read
j = i + 1
k = i
l = k + 1
```

For this copy, k gets i's value number...

...so the value numbers for j and l are also the same!

```
i = read
j = i + 1
k = i
l = j
```

We'll see later that common subexpression elim. fails here.

CSE and value numbering are different!

Another example

```
i = read
j = i + 1
k = i
k = k * 2
l = k + 1
```

For this copy, k gets i's value number...

...but here k gets a new value number...

...so the value numbers for j and l are different, and hence no optimization

No aliasing!

- Even for simple local optimizations we must be careful about aliasing
- Temps are "safe" in that they are non-addressable; unlike values in the heap

In this C program, this "a + 5" is redundant only if the previous two statements leave a and k unmodified

```
exam1 (int *q) {
  int a, k;
 …
  k = a + 5;
  f(a, &k);
  *q = 13;
  m = a + 5;
 …
}
```

Value numbering is local

Simple value numbering is valid only a a local optimization

We'll be able to extend it to global after applying the SSA transformation (later in term)