CSE110A: Compilers

May 31, 2024

Topics:

- Advanced LVN
- Advanced Loop Optimizations

Announcements

- HW 5 is out
 - Due in 1 week; get started early, another big one
- We are working on grading HW 3, hoping to have grades by monday
- Come see us in office hours for homework help!

Announcements

- Final Exam
 - Monday Jan 10: Noon 3 PM
 - 3 pages of notes (front and back)
 - Like the midterm
 - Designed to be 2x as long, but final has 3x time.
 - 4 questions instead of 3
 - Comprehensive, slightly more weight to last part of class

Topics to study for final

- **Module 1:** Token definitions, Regular expressions, Scanner API, Scanner implementations.
- Module 2: Grammars (BNF Form), parse trees, ambiguous grammars (and how to fix them). Precedence, associativity (of the operators in your homework), Top down parsers
- **Module 3:** ASTs how to create them, node types and members, modifications. Simple type systems, linearizing ASTs into 3 address code.
- Module 4: basic blocks, local value numbering, for loop analysis (loop unrolling). Control flow graphs (if time)

Quiz

Quiz

It's the parser's job to perform local value numbering

○ True

○ False

- Local value numbering operates over 3 address code
- The parser produces 3 address code
- In some cases, the parser might use LVN, but it is independent

```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;
```

```
H = {
      "b0 + c1" : "a2",
}
```

Quiz

Local value numbering can only work in just one basic block.

O True

○ False

• Reminder on a basic block

- Programs can be split into **Basic Blocks**:
 - A sequence of 3 address instructions such that:
 - There is a single entry, single exit

• Important property: an instruction in a basic block can assume that all preceding instructions will execute

How might they appear in a high-level language?

How many basic blocks?

```
...
if (expr) {
    ...
}
else {
    ...
}
...
```

Two Basic Blocks

Single Basic Block

```
Label_x:
op1;
op2;
op3;
br label_z;
```

```
Label_x:
op1;
op2;
op3;
Label_y:
op4;
op5;
```

Label_0:

$$x = a + b;$$
 $y = a + b;$

optimized
to

Label_0:
 $x = a + b;$

$$y = x;$$

Quiz

After perform local value numbering on the following program, how many operations can you save?

```
a = b + c;
```

$$d = e * f;$$

$$b = b + c$$
;

$$c = c + b$$
;

$$g = f * e;$$

```
a = b + c;
d = e * f;
b = b + c;
c = c + b;
g = f * e;
```

```
H = {
```

Quiz

What is a good order of performing the following optimizations (left to right):

- 1) Local value numbering
- 2) Loop unrolling
- 3) Constant propagation

- O 1-2-3
- O 3-1-2
- 3-2-1
- O 2-1-3
- O 2-3-1

```
for (int i = 0; i < 10; i++) {
  x = y + z;
}</pre>
```

loop unrolling

```
for (int i = 0; i < 10; i++) {
    x = y + z;
    i++;
    x = y + z;
}</pre>
```

how might this influence other optimizations?

```
a = 16;
b = a + c;
d = 16;
e = d + c;
```

How might constant propagation change this program

```
a = 16;
b = a + c;
d = 16;
e = d + c;
```

How might constant propagation change this program

```
a = 16;
b = 16 + c;
d = 16;
e = 16 + c;
```

```
a = 16;
b = a + c;
d = 16;
e = d + c;
```

How might constant propagation change this program

LVN can now replace the bottom one

It's a little more difficult to apply to Class IR. Do people have any ideas?

Quiz

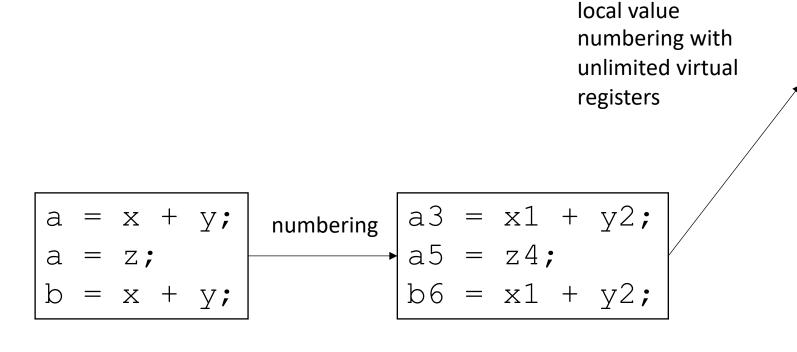
Briefly describe why local value numbering is easiest applied to a single basic block. Think about the structure of an if/else statement. Knowing that structure could you do some form of local value numbering? Briefly describe how you might do it.

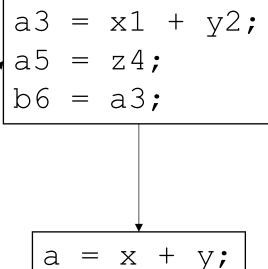
More Local Value Numbering

 We've assumed we have access to an unlimited number of virtual registers.

- In some cases we may not be able to add virtual registers
 - If an expensive register allocation pass has already occurred.
- New constraint:
 - We need to produce a program such that variables without the numbers is still valid.

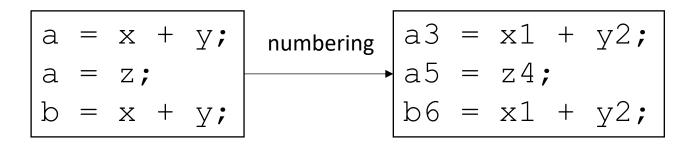
• Example:





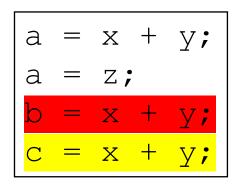
if we drop the numbers, the optimization is invalid.

Solutions?



```
a = x + y;
a = z;
b = x + y;
c = x + y;
```

Keep another hash table to keep the current variable number



We cannot optimize the first line, but we can optimize the second

```
a = x + y;
a = z;
b = x + y;
c = x + y;
```

Keep another hash table to keep the current variable number

First we number

```
a3 = x1 + y2;

a5 = z4;

b6 = x1 + y2;

c7 = x1 + y2;
```

```
Current_val = {
}

H = {
}
```

```
a3 = x1 + y2;

a5 = z4;

b6 = x1 + y2;

c7 = x1 + y2;
```

Local value numbering w/out adding registers

Keep another hash table to keep the current variable number

Local value numbering w/out adding registers

Keep another hash table to keep the current variable number

```
a3 = x1 + y2;

a5 = z4;

b6 = x1 + y2;

c7 = b6;
```

```
a = b + c;
d = e + f;
g = b + c;

label_0:
h = g + a;
k = a + g;
```

split into basic blocks

$$a = b + c;$$
 $d = e + f;$
 $g = b + c;$

```
label_0:
h = g + a;
k = a + g;
```

number

$$a2 = b0 + c1;$$

 $d5 = e3 + f4;$
 $g6 = b0 + c1;$

```
label_0:
h2 = g0 + a1;
k3 = a1 + g0;
```

move code on slide to make room

$$a2 = b0 + c1;$$

 $d5 = e3 + f4;$
 $g6 = b0 + c1;$

```
label_0:
h2 = g0 + a1;
k3 = a1 + g0;
```

optimize

$$a2 = b0 + c1;$$

 $d5 = e3 + f4;$
 $g6 = b0 + c1;$

$$a2 = b0 + c1;$$

 $d5 = e3 + f4;$
 $g6 = a2;$

```
label_0:
h2 = g0 + a1;
k3 = a1 + g0;
```

```
label_0:
h2 = g0 + a1;
k3 = h2;
```

optimize

a2 = b0 + c1; d5 = e3 + f4;g6 = b0 + c1;

label_0:

$$h2 = g0 + a1;$$

 $k3 = a1 + g0;$

$$a2 = b0 + c1;$$

 $d5 = e3 + f4;$
 $g6 = a2;$

```
label_0:
h2 = g0 + a1;
k3 = h2;
```

put together?

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
label_0:
h2 = g0 + a1;
k3 = h2;
```

What are the issues?

optimize

a2 = b0 + c1; d5 = e3 + f4;g6 = b0 + c1;

label_0:

$$h2 = g0 + a1;$$

 $k3 = a1 + g0;$

$$a2 = b0 + c1;$$

 $d5 = e3 + f4;$
 $g6 = a2;$

```
label_0:
h2 = g0 + a1;
k3 = h2;
```

put together?

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
label_0:
h2 = g0 + a1;
k3 = h2;
```

undefined!

What are the issues?

optimize

stitch
part 1: assign original
variables their latest values

```
a2 = b0 + c1;

d5 = e3 + f4;

g6 = b0 + c1;
```

$$a2 = b0 + c1;$$

 $d5 = e3 + f4;$
 $g6 = a2;$

```
label_0:

h2 = g0 + a1;

k3 = a1 + g0;
```

```
label_0:
h2 = g0 + a1;
k3 = h2;
```

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
g = g6;
d = d5
a = a2;
```

```
label_0:
h2 = g0 + a1;
k3 = h2;
h = h2;
k = k3;
```

make room on slide

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
g = g6;
d = d5
a = a2;
```

```
label_0:
h2 = g0 + a1;
k3 = h2;
h = h2;
k = k3;
```

what else needs to be done?

stitch part 2: drop numbers from first use of variables

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
g = g6;
d = d5
a = a2;
```

```
label_0:
h2 = g0 + a1;
k3 = h2;
h = h2;
k = k3;
```

```
label_0:
h2 = g + a;
k3 = h2;
h = h2;
k = k3;
```

Now they can be combined

```
a2 = b0 + c1;
d5 = e3 + f4;
g6 = a2;
g = g6;
d = d5
a = a2;
```

```
label_0:
h2 = g0 + a1;
k3 = h2;
h = h2;
k = k3;
```

```
label_0:
h2 = g + a;
k3 = h2;
h = h2;
k = k3;
```

```
a2 = b + c;
d5 = e + f;
g6 = a2;
q = q6;
d = d5
a = a2;
label 0:
h2 = q + a;
k3 = h2;
h = h2;
k = k3;
```

original

```
a = b + c;
d = e + f;
g = b + c;

label_0:
h = g + a;
k = a + g;
```

new

```
a2 = b + c;
d5 = e + f;
q6 = a2;
q = q6;
d = d5
a = a2;
label 0:
h2 = q + a;
k3 = h2;
h = h2;
k = k3;
```

is it really optimized?

It looks a lot longer...

original

```
a = b + c;
d = e + f;
g = b + c;

label_0:
h = g + a;
k = a + g;
```

new

```
a2 = b + c;
d5 = e + f;
q6 = a2;
q = q6;
d = d5
a = a2;
label 0:
h2 = q + a;
k3 = h2;
h = h2;
k = k3;
```

is it really optimized?

Common pattern for code to get larger, but it will contain patterns that are easier optimize away

later passes will minimize copies

- Colloquially, they are often used interchangeably
- Technically (e.g. according to the books)
 - Constant propagation is replacing variables with constants
 - Constant folding is compile-time evaluation when constants are known

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

constant propagation

```
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

constant propagation

```
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

constant folding

```
int x = 14;
int y = 0;
return y * (4);
```

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

```
int x = 14;
int y = 0;
return 0 * (4);
```

constant propagation

```
constant propagation
```

```
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

constant folding

```
int x = 14;
int y = 0;
return y * (4);
```

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

```
int x = 14;
int y = 0;
return 0 * (4);
```

constant propagation

```
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

constant folding

```
int x = 14;
int y = 0;
return y * (4);
```

constant

propagation

constant folding

```
int x = 14;
int y = 0;
return 0;
```

Typically performed at the same time

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

constant propagation and folding second line

```
int x = 14;
int y = 0;
return y * (28 / x + 2);
```

```
int x = 14;
int y = 0;
return 0;
```

constant propagation and folding third line

```
b = 5;
c = 3;
a = b + c;
b = a - d;
c = a + c;
d = a - d;
```

```
H = {
}

Known_values = {
}
```

numbering

```
b0 = 5;

c1 = 3;

a2 = b0 + c1;

b4 = a2 - d3;

c5 = a2 + c1;

d6 = a2 - d3;
```

```
H = {
}

Known_values = {
```

As you are iterating through code, add any constant mappings to Known_values:

```
b0 = 5;
c1 = 3;
a2 = b0 + c1;
b4 = a2 - d3;
c5 = a2 + c1;
d6 = a2 - d3;
```

```
H = {
}

Known_values = {
}
```

As you are iterating through code, add any constant mappings to Known_values:

```
b0 = 5;

c1 = 3;

a2 = b0 + c1;

b4 = a2 - d3;

c5 = a2 + c1;

d6 = a2 - d3;
```

```
H = {
}

Known_values = {
"b0": 5
"c1": 3
```

When you find an arithmetic operation, first check if operands are known

```
b0 = 5;
c1 = 3;
a2 = b0 + c1;
b4 = a2 - d3;
c5 = a2 + c1;
d6 = a2 - d3;
```

```
H = {
}

Known_values = {
"b0": 5
"c1": 3
}
```

When you find an arithmetic operation, first check if operands are known

```
b0 = 5;
c1 = 3;
a2 = b0 + c1;
b4 = a2 - d3;
c5 = a2 + c1;
d6 = a2 - d3;
```

```
H = \{5+3\} evaluate and add to known values
```

```
Known_values = {
"b0": 5
"c1": 3
```

When you find an arithmetic operation, first check if operands are known

```
b0 = 5;
c1 = 3;
a2 = 8;
b4 = a2 - d3;
c5 = a2 + c1;
d6 = a2 - d3;
```

```
H = {
5+3

evaluate and add to known values
```

```
Known_values = {
"b0" : 5
"c1" : 3
"a2" : 8
```

When you find an arithmetic operation, first check if operands are known

```
b0 = 5;

c1 = 3;

a2 = 8;

b4 = 8 - d3;

c5 = a2 + c1;

d6 = a2 - d3;
```

propagate constant (if IR allows it)

```
Known_values = {
"b0": 5
"c1": 3
"a2": 8
```

 $H = {$

When you find an arithmetic operation, first check if operands are known

add to H

H = {
"8 - d3": "b4"

continue on.

why do we want to store 8 here rather than a2?

```
Known_values = {
"b0" : 5
"c1" : 3
"a2" : 8
```

```
b0 = 0;
d3 = 1;
f7 = 4;
a2 = b0 + c1;
b4 = a2 * d3;
d6 = e5 * f7;
```

```
H = {
}

Known_values = {
}
```

what can we do here?

```
b0 = 0;
d3 = 1;
f7 = 4;
a2 = b0 + c1;
b4 = a2 * d3;
d6 = e5 * f7;
```

```
H = {
}

Known_values = {
"b0":0, "d3":1, "f7":4
}
```

```
b0 = 0;
d3 = 1;
f7 = 4;
a2 = b0 + c1;
b4 = a2 * d3;
d6 = e5 * f7;
```

what can we do here? add a special rule for + that if any side is 0, you can just drop the 0.

```
H = {
}

Known_values = {
"b0":0, "d3":1, "f7":4
}
```

```
b0 = 0;
d3 = 1;
f7 = 4;
a2 = c1;
b4 = a2 * d3;
d6 = e5 * f7;
```

what can we do here? add a special rule for + that if any side is 0, you can just drop the 0.

```
H = {
}

Known_values = {
"b0":0, "d3":1, "f7":4
}
```

What other rules could we have?

Other considerations in LVN

Memory and functions

• Consider a 3 address code that allows memory accesses

```
a[i] = x[j] + y[k];
b[i] = x[j] + y[k];
is this transformation allowed?
a[i] = x[j] + y[k];
b[i] = a[i];
```

Consider a 3 address code that allows memory accesses

```
a[i] = x[j] + y[k];

b[i] = x[j] + y[k];
```

is this transformation allowed? No!

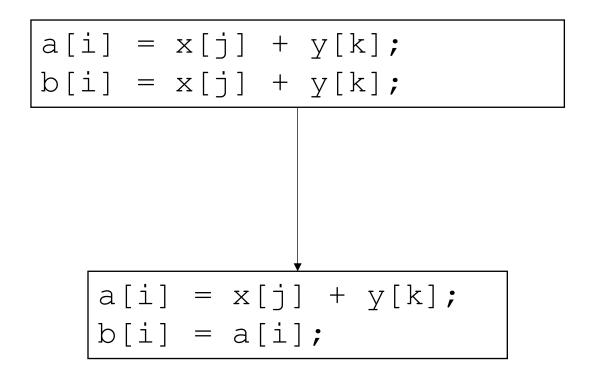
only if the compiler can prove that a does not alias \boldsymbol{x} and \boldsymbol{y}

$$a[i] = x[j] + y[k];$$

 $b[i] = a[i];$

In the worst case, every time a memory location is updated, the compiler must update the value for all pointers.

Consider a 3 address code that allows memory accesses



Example, initially:

What does b[i] equal at the end of each computation?

- How to number:
 - Number each pointer/index pair

```
a[i] = x[j] + y[k];

b = x[j] + y[k];
```

- How to number:
 - Number each pointer/index pair
 - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);

(b,6) = (x[j],?) + (y[k],?);
```

- How to number:
 - Number each pointer/index pair
 - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);

(b,6) = (x[j],4) + (y[k],5);
```

Does this help at all?

- How to number:
 - Number each pointer/index pair
 - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);

(b,6) = (x[j],4) + (y[k],5);

(c,7) = (x[j],4) + (y[k],5);
```

Does this help at all?

If there is no memory writes between an assignment to a variable then we can do a replacement

- How to number:
 - Number each pointer/index pair
 - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);

(b,6) = (x[j],4) + (y[k],5);

(c,7) = (b,6);
```

Does this help at all?

If there is no memory writes between an assignment to a variable then we can do a replacement

- How to number:
 - Number each pointer/index pair
 - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);

(b[i],6) = (x[j],4) + (y[k],5);
```

A compiler analysis might try to determine that addresses can't alias

```
can we trace a, x, y to
a = malloc(...);
x = malloc(...);
y = malloc(...);
// a, x, y are never overwritten
```

- How to number:
 - Number each pointer/index pair
 - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);

(b[i],6) = (x[j],1) + (y[k],2);
```

in this case we do not have to update the number

A compiler analysis might try to determine that addresses can't alias

```
can we trace a, x, y to
a = malloc(...);
x = malloc(...);
y = malloc(...);
// a, x, y are never overwritten
```

- How to number:
 - Number each pointer/index pair
 - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);

(b[i],6) = (x[j],4) + (y[k],5);
```

programmer annotations can also tell the compiler that no other pointer can access the memory pointed to by a

- How to number:
 - Number each pointer/index pair
 - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);

(b[i],6) = (x[j],4) + (y[k],5);
```

in this case we do not have to update the number

restrict a

programmer annotations can also tell the compiler that no other pointer can access the memory pointed to by a

Warning: the compiler does not enforce this!

- How to number:
 - Number each pointer/index pair
 - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);

(b[i],6) = (a[i],3);
```

How to number?

```
a = foo(x);
x = b;
c = foo(x);
```

How to number?

a = foo(x); x = b; c = foo(x);

the same way

```
a1 = foo(x0);

x3 = b2;

c4 = foo(x3);
```

What if you had first class functions?

How to number?

a = foo(x); x = b; c = foo(x);

the same way

```
a1 = foo(x0);
x3 = b2;
c4 = foo(x3);
```

Can we replace?

How to number?

the same way

$$a = foo(x);$$

 $c = foo(x);$

a1 =
$$foo(x0)$$
;
c2 = $foo(x0)$;

How about now?

How to number?

a = foo(x);c = foo(x);

the same way

```
a1 = foo(x0);

c2 = foo(x0);
```

How about now?

```
int count = 0;
int foo(int x) {
  count += 1;
  return 0;
};
```

What if foo had this implementation?

How to number?

the same way

```
a = foo(x);

c = foo(x);
```

```
a1 = foo(x0);

c2 = foo(x0);
```

How about now?

side effects!

```
int count = 0;
int foo(int x) {
  count += 1;
  return 0;
};
```

What if foo had this implementation?

```
a = foo(x);
c = foo(x);
print(count);
```

```
a1 = foo(x0);
c2 = a1;
print(count);
```

are these two programs the same?

```
int count = 0;
int foo(int x) {
  count += 1;
  return 0;
};
```

• In C/++, functions are assumed to have side effects

- A function that does not have side effects is called "pure"
 - You can annotate a function as pure
 - __attribute__((pure))
 - warning: compiler does not check this and you can introduce subtle bugs
- Functional languages tend to have a pure-by-default design. Allows more compiler optimizations, but less control to the programmer.

See everyone on Monday!