# CSE110A: Compilers

May 23, 2022

**Topics**:

- *Finish local value numbering*
- *Loop transformations*

# Announcements

- New grades:
  - Midterm grades will be posted by end of the day
- HW 3 is out
  - Due tomorrow
  - Double check piazza for hints and discussions
- Homework 4
  - Will be released either tonight or tomorrow by midnight

# Announcements

- Schedule:
  - We'll finish up local value numberings and talk about loop transformations
  - I want to spend time on a homework overview
  - I want to talk about a global optimization
    - Either undefined variable analysis
    - Or code slicing
  - For backends, I want to talk about register allocation
- We will see what we have time for...

# Quiz

- Thank you for taking the time to fill it out;
  - Its very helpful, and especially for newly designed classes like this.

- Speaking of helpful things:
  - SETs are out!
  - Those are the official feedback forms for classes
  - It is incredibly useful for new faculty and especially for new classes
  - CSE113 example
  - I'd really appreciate it if you could fill it out

# Review

- Local value numbering
  - Local optimization
  - Simple algorithm that can be built on:
    - initial version just used string comparison
    - next we added commutativity
    - lastly we extended the algorithm to not add any new registers
  - Second lecture
    - we added constant propagation and folding
    - we talked about copy propagation and folding
    - we talked about memory and functions

# LVN with constant prop/folding

```
b = 5;
c = 3;
e = 8;

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

Work through the example

```
H = {
}
```

```
Known_values = {

}
```

# How to do it for Classier?

```
void foo(int &x) {
b = int2vr(5);
c = int2vr(3);

a = addi(b,c);
d = a;
x = vr2int(a);
}
```

```
H = {
}
```

```
Known_values = {

}
```

# Local value numbering: Memory

- 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];
```

# Local value numbering: Memory

- 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];
```

*If we want to perform this optimization we need to ensure that a does not alias x or y.*

*How can we do that?*

# Local value numbering: functions

Is this optimization allowed?

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

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

# Local value numbering: functions

Is this optimization allowed?

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

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

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

functions might have side effects! how can we tell the compiler it doesn't?

# New material

- How to stitch optimized code back into the whole program

# How to stitch optimized code back into the program

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

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

# How to stitch optimized code back into the program

*split into basic blocks*

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

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

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

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

# How to stitch optimized code back into the program

number

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

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

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

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

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

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

# How to stitch optimized code back into the program

move code on slide to make room

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

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

# How to stitch optimized code back into the program

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;
```

# How to stitch optimized code back into the program

optimize

put together?

```
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;
```

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

# How to stitch optimized code back into the program

original code

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

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

put together?

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

*What are the issues?*

# How to stitch optimized code back into the program

original code

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

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

put together?

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

undefined!

*What are the issues?*

# How to stitch optimized code back into the program

stitch
part 1: *assign original variables their latest values*

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

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

```
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;
```

# How to stitch optimized code back into the program

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?

# How to stitch optimized code back into the program

stitch part 2: drop numbers from first use of variables

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

```
a2 = b + c;
d5 = e + f;
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;
```

# How to stitch optimized code back into the program

```
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;
```

Now they can be combined

```
a2 = b + c;
d5 = e + f;
g6 = a2;
g = g6;
d = d5
a = a2;
```

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

```
a2 = b + c;
d5 = e + f;
g6 = a2;
g = g6;
d = d5
a = a2;
label_0:
h2 = g + a;
k3 = h2;
h = h2;
k = k3;
```

# How to stitch optimized code back into the program

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;
g6 = a2;
g = g6;
d = d5
a = a2;
label_0:
h2 = g + a;
k3 = h2;
h = h2;
k = k3;
```

*is it really optimized?*

*It looks a lot longer...*

# How to stitch optimized code back into the program

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;
g6 = a2;
g = g6;
d = d5
a = a2;
label_0:
h2 = g + 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*

# New material

- Loop transformations

# Loop optimizations

- Regional optimization
  - We can handle multiple basic blocks
  - but only if they fit a certain pattern

# For loops

- How do they look in different languages
  - C/C++
  - Python
  - Numpy

- The more constrained the for loops are, the more assumptions the compiler can make, but less flexibility for the programmer

# For loops

- The compiler can optimize For loops if they fit a certain pattern

- When developing a regional optimization, we start with strict constraints and then slowly relax them and make the optimization more general.
  - Sometimes it is not worth relaxing the constraints (optimization gets too complicated. Its not the compilers job to catch every pattern!)
  - If a programmer knows the pattern, then often you can write code such that the compiler can recognize the pattern and it will do better at optimizing!
  - Thus you can write more efficient code if you write it in such a way that the compiler can recognize patterns

# For loops terminology

- Loop body:
  - A series of statements that are executed each loop iteration
- Loop condition:
  - the condition that decides whether the loop body is executed
- Iteration variable:
  - A variable that is updated exactly once during the loop
  - The loop condition depends on the iteration variable
  - The loop condition is only updated through the iteration variable

# Examples

*iteration variable*
*loop body*
*loop condition*

```
for (int i = 0; i < 1024; i++) {
    counter += 1;
  }
```

```
for (; i < 1024; i+=counter) {
    counter += 1;
  }
```

```
  while (1) {
    i++;
    counter += 1;
    if (i < 1024) {
      break;
    }
  }
```

*In general, is it possible to determine if an iteration variable exists or not?*

# Examples

What about these?

```
for (; i < 1024; i++) {
    counter += 1;
    foo();
  }
```

```
for (; i < j; i++) {
    counter += 1;
    j = rand();
  }
```

# Loop unrolling

# Loop unrolling

- Executing multiple instances of the loop body without checking the loop condition.

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

unrolled by a **factor** of 2

```
for (int i = 0; i < 128; i++) {
   // body
}
```

```
for (int i = 0; i < 128; i++) {
   // body
   i++
   // body
}
```

*could we unroll more?*

# Loop unrolling conditions

- Under what conditions can we unroll?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 128; i++) {
   // body
}
```

```
for (int i = 0; i < 128; i++) {
   // body
   i++
   // body
}
```

What can go wrong?

# Loop unrolling conditions

- Under what conditions can we unroll?

**FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement**

```
for (int i = 0; i < 128; i++) {
  // body
}
```

**Validate that we actually have an iteration variable**

# Loop unrolling conditions

- Under what conditions can we unroll?

**FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement**

```
for (int i = 0; i < 128; i++) {
  // body
}
```

**Validate that we actually have an iteration variable**

1. **find** candidate on lhs of assignment statement

# Loop unrolling conditions

- Under what conditions can we unroll?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 128; i++) {
  // body
}
```

**Validate that we actually have an iteration variable**

1. **find** candidate on lhs of assignment statement
2. **check** no assignments to candidate in body

# Loop unrolling conditions

- Under what conditions can we unroll?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 128; i++) {
  // body
}
```

**Validate that we actually have an iteration variable**

1. **find** candidate on lhs of assignment statement
2. **check** no assignments to candidate in body
3. **check** that it matches lhs of assignment_statement

# Loop unrolling conditions

- Under what conditions can we unroll?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 128; i++) {
  // body
}
```

**Validate that we actually have an iteration variable**

1. **find** candidate on lhs of assignment statement
2. **check** no assignments to candidate in body
3. **check** that it matches lhs of assignment_statement
4. **check** loop condition
   * check that candidate variable is on lhs
   * check that the rhs is a variable or literal (cond)
   * check that cond is not assigned in body

# Loop unrolling conditions

- Under what conditions can we unroll?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 128; i++) {
  // body
}
```

**Validate that we actually have an iteration variable**

1. **find** candidate on lhs of assignment statement
2. **check** no assignments to candidate in body
3. **check** that it matches lhs of assignment_statement
4. **check** loop condition
   * check that candidate variable is on lhs
   * check that the rhs is a variable or literal (cond)
   * check that cond is not assigned in body

*Do these guarantee we will find an iteration variable?*
*What happens if we don't find one?*

*how does C-simple help us here?*

# Loop unrolling conditions

- Several ways to unroll
  - More constraints: Simpler to unroll in code generation
  - Less constraints: Harder to unroll in code generation

***Base constraints (required for any unrolling):***

**Validate that we actually have an iteration variable**

1. **find** candidate on lhs of assignment statement
2. **check** no assignments to candidate in body
3. **check** that it matches lhs of assignment_statement
4. **check** loop condition
   * check that candidate variable is on lhs
   * check that the rhs is a variable or literal (cond)
   * check that cond is not assigned in body

# Loop unrolling conditions

- Simple unroll
  - Most constraints
  - Easiest code generation

For unroll factor F

**Simple unroll constraints:**

- Loop update increments by 1
- Find the concrete number of loop iterations, LI
- F must divide LI evenly

**Simple unroll code generation:**

- create a new body = body + update + body
- perform codegen

# Loop unrolling conditions

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 128; i++) {
  // body
}
```

how to do these steps?

For unroll factor F

**Simple unroll constraints:**

- Loop update increments by 1
- Find the concrete number of loop iterations, LI
- F must divide LI evenly

**Simple unroll code generation:**

- create a new body = body + update + body
- perform codegen

# Loop unrolling conditions

**FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement**

```
for (int i = 0; i < 128; i++) {
   // body
}
```

result for a factor of 2

```
for (int i = 0; i < 128; i++) {
   // body
   i++
   // body
}
```

For unroll factor F

**Simple unroll constraints:**

- Loop update increments by 1
- Find the concrete number of loop iterations, LI
- F must divide LI evenly

**Simple unroll code generation:**

- create a new body = body + update + body
- perform codegen

# Loop unrolling conditions

what can go wrong?

**FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement**

```
for (int i = 0; i < 8; i+=3) {
  // body
}
```

For unroll factor F

**Simple unroll constraints:**

- Loop update increments by 1
- Find the concrete number of loop iterations, LI
- F must divide LI evenly

**Simple unroll code generation:**

- create a new body = body + update + body
- perform codegen

# Loop unrolling conditions

what can go wrong?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 8; i+=3) {
  // body
}
```

Actually this is fine as long as i is updated with a constant addition. but we need a more complicated formula to calculate LI:

```
ceil((end - start)/update)
```

But you may want to keep your life simpler by constraining it. We will keep it for now

For unroll factor F

**Simple unroll constraints:**

- Loop update increments by 1
- Find the concrete number of loop iterations, LI
- F must divide LI evenly

**Simple unroll code generation:**

- create a new body = body + update + body
- perform codegen

# Loop unrolling conditions

what can go wrong?

**FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement**

```
for (int i = 0; i < 4; i++) {
  // body
}
```

What if we try to unroll this by a factor of 3?

For unroll factor F

**Simple unroll constraints:**

- Loop update increments by 1
- Find the concrete number of loop iterations, LI
- F must divide LI evenly

**Simple unroll code generation:**

- create a new body = body + update + body
- perform codegen

# Loop unrolling conditions

what can go wrong?

```
FOR LPAR assignment_statement expr SEMI assignment_statement RPAR statement
```

```
for (int i = 0; i < 4; i++) {
  // body
}
```

```
for (int i = 0; i < 4; i++) {
  // body
  i++
  // body
  i++
  // body
}
```

What if we try to unroll this by a factor of 3?

How many times do we execute body?

For unroll factor F

**Simple unroll constraints:**

- Loop update increments by 1
- Find the concrete number of loop iterations, LI
- F must divide LI evenly

**Simple unroll code generation:**

- create a new body = body + update + body
- perform codegen

# Loop unrolling conditions

Let's examine this a bit closer?

```
for (int i = 0; i < 4; i++) {
   // body
}
```

What if we try to unroll this by a factor of 3?

```
for (int i = 0; i < 4; i++) {
   // body
   i++
   // body
   i++
   // body
}
```

How many times do we execute body?

# Loop unrolling conditions

Let's examine this a bit closer?

```
for (int i = 0; i < 4; i++) {
   // body
}
```

```
for (int i = 0; i < 4; i++) {
   // body
   i++
   // body
   i++
   // body
}
```

What if we try to unroll this by a factor of 3?

How many times do we execute body?

what if we executed the unrolled loop as many times as it was valid, and did the rest with a non-unrolled loop

```
for (int i = ?; i < ?; i++) {
   // body
   i++
   // body
   i++
   // body
}
```

```
for (int i = ?; i < ?; i++) {
   // body
}
```

# Loop unrolling conditions

initially the loop starts the same as the original loop

```
for (int i = 0; i < 4; i++) {
  // body
}
```

find out how many unrolled loops we can execute:
(4 / 3) * 3 = 3
This gives us the first bound

second loop is initialized with the first bound

second loop's bound is same as the original loop

what if we executed the unrolled loop as many times as it was valid, and did the rest with a non-unrolled loop

```
for (int i = ?; i < ?; i++) {
  // body
  i++
  // body
  i++
  // body
}
```

```
for (int i = ?; i < ?; i++) {
  // body
}
```

# Loop unrolling conditions

What about in the general case? For unroll factor F?

```
for (int i = x; i < y; i++) {
   // body
}
```

find out how many unrolled loops we can execute:
?
This gives us the first bound

second loop is initialized with the first bound

second loop's bound is same as the original loop

what if we executed the unrolled loop as many times as it was valid, and did the rest with a non-unrolled loop

```
for (int i = ?; i < ?; i++) {
   // body
   i++
   ...
}
```

```
for (int i = ?; i < ?; i++) {
   // body
}
```

# Loop unrolling conditions

- general unroll

For unroll factor F

**General unroll constraints:**
- Loop update increments by 1
- Find the concrete number of loop iterations, LI

**General unroll code generation:**
- Create simple unrolled loop with new bound: (LI/F)*F
- Create cleanup (basic) loop with initialization: (LI/F)*F
- perform codegen

*None of these numbers have to be concrete!*

# More loop transforms

- Loop nesting order
- Loop unroll and jam
- Tiling
- General area is called polyhedral compilation

https://en.wikipedia.org/wiki/Polytope_model

# See everyone on Wednesday

- More loop transformations