CSE110A: Compilers

May 29, 2024

Topics:

- *Loop optimizations*

Announcements

- HW 5 is out
  - Due on next Friday; get started early, another big one
  - Come see us in office hours for help with HW 5; plenty of time at the beginning of the assignment!
- We are working on grading HW 3, hoping for grades by Monday

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

Write a simple grammar to parse functions like follows:

int main() { return 1 + 2; }

void foo(int a, double b) { a = a + 1; b = b + 1; return; }

You may use the format from your HW2 (like part 1.1), don't worry about left recursions, just the simple grammar. **You don't need to write the entire grammar**, just fill in the blanks.

You may use the same Tokens from HW2, and here are some new tokens

- RETURN - keyword 'return'
- VOID - keyword 'void'

Obviously, a function should have a **return type**, a **function name**, followed by a **list of arguments** enclosed by **parens.** Then followed by a **block of statements**. And for our case, assume the list of arguments is "0 or more declaration statements". You should extend the 'statement' to contain a **return_statement,** which may return an expression or may not.

Write a simple grammar to parse functions like follows:

int main() { return 1 + 2; }

void foo(int a, double b) { a = a + 1; b = b + 1; return; }

Here is a template, **fill in the blanks:**

... <assume you have the reset of your grammar from HW2>...

function_decl := return_type ID LPARAN arg_list RPARAN block_stmt

arg_list := _____

return_type := _____

block_stmt := _____

statements := assign_stmt | var_decl_stmt | if_else_stmt | for_stmt | block_stmt | return_stmt

return_stmt := _____

Quiz

Discuss some compiler optimizations that could be done on the following program? Do you think modern compilers do the optimizations that you are proposing?

```
int a = 30;
int b = 9 - (a / 5);
int c;

c = b * 4;
if (c > 10) {
   c = c - 10;
}
return c * (60 / a);
```

Quiz

Describe some compiler optimizations you know of. Write one (or more) small example program on Godbolt and look at the llvm IR (using -emit-llvm on a clang compiler) or ISA code. You can also play with optimization flags (-O0, -O3, etc). Did the compiler do the optimization you thought of?

Describe your program and the optimization below. Feel free to share your experiment on piazza!

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

```
for (i = 0; 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 literal

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 literal

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

Loop unrolling conditions

- Several ways to unroll
  - More constraints: Simpler to unroll in code gen, more things to check
  - Less constraints: less things to check, harder to unroll in code gen

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 literal

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)*(F-1)
- 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)*(F-1)
- 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 simple

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!

Homework review

Back to Local Value Numbering

How to stitch optimized code back into the 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

```
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?

How to stitch optimized code back into the program

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?

How to stitch optimized code back into the program

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;
```

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

```
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

See everyone on Friday!