CPEG621

Lab 3

Michael Hutts

A Calculator Compiler Middle End - Transformation and Evaluation

Objectives

The objective of this lab was to explore middle-end optimization and transformation. The lab was split into three tasks. The first being to generate basic blocks in the three -address code, then create a means of calculating latency, and lastly perform CSE optimization. These three tasks could then be combined to build a complete middle-end solution. With the basic block generation building a means of breaking up the code into its fundamental sections, the CSE optimization to improve performance via transformation, and the latency calculation fitting in as a heuristic to determine if the transformations made offer an improvement or degradation in performance.

Strategy

Note

Upon starting the first task I quickly ran into a major issue. No projects (including those that had previously compiled perfectly fine) that used yacc would not compile on my system. After losing many hours attempting to understand the origin of the issue (including fully reinstalling all relevant packages), I decided to cut my losses and find an alternative solution. Pure C would've still worked, but given there was no longer the benefits of yacc/bison available, I chose to use Python since it offered the simplest and fastest means of building all 3 middle-end tasks. Task 1 specifically would've been much easier to implement when building off of my existing code from the previous labs, but thankfully rebuilding everything was not too difficult as I had already practiced somewhat in lab 2 with my attempts at writing a runtime simulator.

Task 1

Task 1 was probably the simplest task to implement. Using the basic block partitioning algorithm given in the lecture 9 slides, implementation simply required translating the steps to real code.

Data

My implementation of a basic block partitioning algorithm used 4 booleans to track its current position with in the code:

```
inside_if = False # true when inside the "if" part of a conditional statement
inside_else = False # true when inside the "else" part of a conditional statement
end_condition = False # true when reaching the end of all condition statements
first_line = True # true when writing the very first line
```

These booleans helped identify the different expected leader statements throughout the input.

Three lists were used to track the contents of the code:

```
processed_code = [] # list containing the final output
contents_if = [] # list containing the contents of the current if statement
contents_else = [] # list containing the contents of the current else statement
```

The contents of the conditional statements would need to be held until the end of all conditions.

Implementation

Once we had the overall structure, the only thing left was to parse through the code. As long as the code fit the expected structure, it was as simple as detecting the elements of conditions, marking when their hit with their relevant boolean, and then storing their contents in the right list, to be added to the list holding the final output later. A variable was used to track the current basic block labels, and the correct goto's were inserted both at the original condition contents, and at the end of both conditions to point to the code after.

Task 2

I attempted this task to some degree in lab 2, but struggled greatly. Thankfully I had much less issues with my simplified approach for lab 3. The function parse_tac() was used to build the basic structure to feed to calculate_cycles_from_code(). Later when using task 2 for task 3 I had to modify my code to handle ignoring building blocks and better handling of conditions, but overall it was not overly complex overall. The structure of the processed lines are in the format *(variable to be written to, operation, [list of dependencies])*. The list built with this structure is then iterated through, with the max cycle value calculated with each iteration, giving a total latency in the end.

Task 3

At my first impression, task 3 seemed highly complex, but like task 1, the sides provided a basic algorithm that could be worked through.

Data

```
history = [] # operations previously seen
explored = [] # operations already transformed
processed_code = [] # final resultant code
```

Using these three lists, and one additional boolean to track when reaching the end of valid code to operate on (i.e. there is no CSE transformation to perform on the contents of conditional statements after a basic block partitioning has been done). Lastly, just like task 1, a counter would be used to track the current temporary value id.

Combined

I also created a *combined.py*, which combined all 3 tasks into a single stage, which would also implement the additional requirements of performance modeling and selecting between whether to use CSE or not.

Results

Task 1

For the input files I created (infile_0.txt, infile_1.txt), all behavior worked as expected. Like previous labs, multiple levels of conditions were not supported, but I believe with some recursive changes to my code this would be possible.

Files (Before Basic Block Partitioning)

Infile_0

```
x = 1+2;
y = 3-x;
x = 3;
a = 4*5
b = 6/7
if (x) {
z = 6;
}
else {
z = 0;
}
c = z*3;
```

Infile_1

```
a = 120;
b = 20*2;
if (a) {
c = a*b;
}
else {
c = 0;
}
d = a-20;
```

```
if (c) {
e = d*2
}
else {
e = d/2
}
f = e+d
```

Files (After Basic Block Partitioning)

Outfile_0

```
BB1:
x = 1+2;
y = 3-x;
x = 3;
a = 4*5
b = 6/7
if (x) {
goto BB2
}
else {
goto BB3
}
BB2:
z = 6;
goto BB4
BB3:
z = 0;
goto BB4
BB4:
c = z*3;
```

Outfile_1

```
BB1:
a = 120;
b = 20*2;
if (a) {
goto BB2
}
else {
goto BB3
```

```
}
BB2:
c = a*b;
goto BB4
BB3:
c = 0;
goto BB4
BB4:
d = a-20;
if (c) {
goto BB5
}
else {
goto BB6
}
BB5:
e = d*2
goto BB7
BB6:
e = d/2
goto BB7
BB7:
f = e+d
```

Task 2

My code was able to simulate the example correctly, and would later work great as a heuristic for task 3.

Infile_0

```
A = B*1
C = D+E
F = A+C
```

```
21:37:33 mike ~/documents/cpeg621/homework3/task2 $ python3 task2.py infile_0.txt
Total cycles needed: 5
```

Task 3

My code worked as expected for the input files I generated. The more complex examples will be tested in the combined code.

Infile_0

```
A = C*3;
X = B+A;
D = C*3;
C = D/2;
```

Infile_1

```
A = X+3;
B = A*3;
C = A*B;
C = X+3;
B = A*3;
```

Outfile_0

```
tmp1 =  C*3;
A  = tmp1
X = B+A;
D  = tmp1
C = D/2;
```

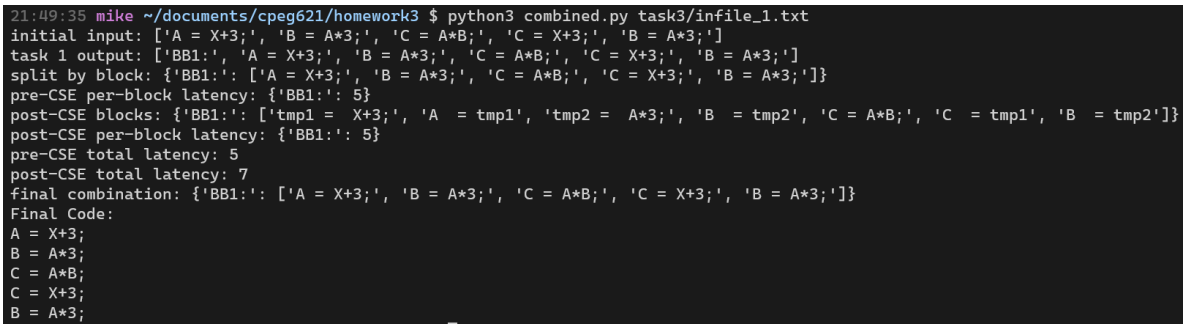
Outfile_1

```
tmp1 =  X+3;
A  = tmp1
tmp2 =  A*3;
B  = tmp2
C = A*B;
C  = tmp1
B  = tmp2
```

Combined

My combined code tested the complete top to bottom results of task 1-3 and worked as expected. It was also able to detect examples, like infile_1, where CSE would be harmful to performance.

The input files are repeats of previous examples, but the results are different due to their combined performance.



Conclusion

In conclusion, I was able to successfully implement all 3 tasks, even if not as originally planned. Combining them into a single stage, which could be hooked into my existing lab 1 and 2 code, would provide a solid middle-end for the compiler that's been written throughout this course. Optimization is a critical part of any compiler, and an underappreciated part at that. Creating this project was satisfying and enlightening. In comparison to previous labs, I was thankfully left with much less on the table to fix or solve. I had some additional more complex features that might not work, but as they weren't explicitly described in the lab instructions, I did not test or implement them.

Usage

Using the argparse library, I was able to add usage descriptions to all 3 tasks, plus combined.py. Simply run them with the -h flag to see. I wrote and tested this with Python 3.10.6, but any version of Python 3 should work.

Examples

Task 1

python3 task1.py infile_0.txt --output outfile_0.txt

Task 2

python3 task2.py infile_0.txt

Task 3

python3 task3.py infile_0.txt --output outfile_0.txt

Combined

python3 combined.py infile_0.txt --output outfile_0.txt

Personal Note

I didn't get the chance to say this in our last lecture, but your classes have been most of the most interesting and rewarding I have taken during my masters program. There's a lot of knowledge related to performance optimization that I would know nothing about without having taken your courses. When I run into applications of this knowledge in my personal life, I feel a genuine excitement I don't get from most other things I've learned of the past year. For that, I just wanted to thank you.

- Michael