Matt Farkas

Michael Yahner

Optimization for Parallel Compilers Final Project

Professor Dr. O’Neil

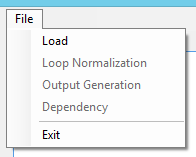
Due: April 18, 2013

**Program Overview**

The final project was to implement loop normalization if required based off the by clause of a for loop. From the normalized loop, we were to accomplish the generating the same output format as the output that was on the midterm. Starting with arrays, they are displayed by the first letter of their name in alphabetic order starting from zero, displaying their array type. This is followed by the upper bounds on new lines in order from outer most to inner with a zero delimiter after them. Next, comes all of the writes with line number, array name (position in alphabet), loop depth, and then constant with coefficient per each subscript. This is followed by another delimited zero and the reads with the same information.

**Instructions**

You will see below the File menu to operate this program. (In the README is the instructions on how to build and run this program.) From the file menu you can start off by selecting either “Load” or “Exit”. “Load” will open a dialog box to select the O’Neil code to work with. You then select “Loop Normalization” to see the normalized code in addition to being able to generate the output. Finally, click “Output Generation” to get the output that can be used in the midterm program.



**Assumptions**

Some assumptions in this final project are that there will not be any back to back for loops. There are not any constant values used as indexes in the arrays. There are not any induction variables or auxiliary variables.

**Overview of Work**

There is the LoopNormailzation.cs class that completely takes care of the loop normalizing. In this class there are two private classes (because C# doesn’t really like lists and structures with getters/setters). The first, LoopNameValue, holds the name of the index variable and the value that it is going to replace all none loop instances of it. The next class is EndFor class. This class holds the location of the endfor, and the for loop that it belongs too, index. In the main class there is a static method called *Normalized*. The number of for loops are counted. This is done to tell the user how many out of total for loops were already normalized. When a for loop is found in the code, it is checked to see if there is a by statement and if it’s value is greater than one and if the lower bound is greater than one. If so then it does the normalization.

There is another class in there, OutputGenerator, that handles all of the output generation. There are three classes. The first being Variable which holds the variables name, the type (list, box, table), typeint which is the type in int form, and position which is the position in the total number of variable list. The second bing LineInformation which holds the read and write information such as line number, array name, the depth, string of constant and coefficient, and whether or not it is a write. The last class is just like the LoopNameValue class in that it holds loop information. There is one static method called *Generate*. This method creates a list that just holds all of the for loop statements. It separates out the reads and writes in the let statements by the equal sign, though it does the same work on both. First a helper method is called, *LineGeneration*, that calls another helper method, *ReadWriteLoopInformation* which calls two helper methods, *ValueExtraxtion* which just takes the current subscript string and spilts it into a list and *ElementChange* which changes the order of the loop variables to smallest first, that fills the LineInformation string for constant and coefficient. The last helper method that is called in the main function is, *InformationOutput*, which just fills the string array with the output that was just created to be displayed to the user.

**Tests Performed**

All of the midterm tests were used. I also built one unnormalized code that is included with the rest of the test cases. I used others but this is one that I kept. I also built a test case that tests for just about everything. It is shown below and included in the test case bank.

title

var

box[] a

box[] b

box[] c

int j

int k

int i

begin

for j = 1 to 100

for k = 1 to 100

for i = 1 to 100

let a[1 \* k, 1 - j, 1 + i] = a[10 + 2, 4 \* 3 + 2 - 5 \* 2, 5]

let b[k + j, i + j, 2 \* 3 - i] = b[1, 2, 3]

let b[k + j, i + j + 2, 2 \* 3 - i] = b[1, 2, 3]

endfor

endfor

endfor

end

**Unexpected Behavior**

Some unexpected behavior was from having I, j, k loops and array subscript of k + I or pretty much anything not in order. This kept throwing my code off or throwing the answer off. It would output something like 1 0 1 or 0 1 1 or 0 0 1. I finally broke down and had to create a method that swapped the order of the index values. This was a nifty feat due to the fact that you had to take into consideration the fact that there may be *n* number of values between the two. The only other unexpected behavior I saw, because I wasn’t thinking, was the number of values left after the coefficients had been taken care of and the order of operations. Part of the order of operations from part two of the term project was used here to solve the problem.

**Outcomes**

From what I can tell of the testing that I have done is that loop normalization works without a flaw and will work for any O’Neil code. This means that you can have O’Neil code with statements other than just the for loops. I believe as long as there are not back to back for loops that this program can work on any O’Neil code.