Assignment 2 Dataflow Analysis

Due Date: **Mar. 7th**, Total Marks: 100 pts

CSCD70 Compiler Optimizations
Department of Computer Science
University of Toronto

ABSTRACT

In class, we discussed interesting dataflow analyses such as Reaching Definitions, Liveness, and Available Expressions. Although these analyses are different in certain ways, for example they compute different program properties and analyze the program in different directions (forwards, backwards), they share some common properties such as iterative algorithms, transfer functions, and meet operators. These commonalities make it worthwhile to write a generic framework that can be parameterized appropriately for solving a specific dataflow analysis. In this assignment, you will implement such an iterative dataflow analysis framework in LLVM, and use it to implement a forward dataflow analysis (Available Expressions) and a backward dataflow analysis (Liveness). Although Liveness and Available Expressions implementations are available in some form in LLVM, they are not of the iterative flavor, and the objective of this assignment is to create a generic framework for solving iterative bitvector dataflow analysis problems, and use it to implement Liveness and Available Expressions analysis.

1 POLICY

1.1 Collaboration

You will work in groups of *two* for the assignments in this course. Please turn in a single writeup per group, indicating the names and UTORid of both group members.

1.2 Submission

Please include all your files in an archive labeled with the UTORid of both group members, and email the resulting file to bojian@cs.toronto.edu. Make sure that when this archive is extracted, the files appear as follows:

```
./a2-utorid1-utorid2/writeup.pdf

./a2-utorid1-utorid2/Dataflow/include/
    dfa/framework.h
./a2-utorid1-utorid2/Dataflow/src/
    liveness.cpp
./a2-utorid1-utorid2/Dataflow/src/
    avail_expr.cpp
./a2-utorid1-utorid2/Dataflow/Makefile
./a2-utorid1-utorid2/Dataflow/tests/
```

- A report named *writeup.pdf* that briefly describes the implementation of both passes, shows the output results on the two microbenchmarks provided, and has answers to the theoretical questions in this assignment.
- Well-commented source code for your *iterative framework* and passes (*Liveness* and *Available Expressions*), and associated `Makefile` (please write your `Makefile` in such a way that all passes can be built, integrated, and run using the command `make all`).
- A subfolder named *tests* that contains all the microbenchmarks used for verification of your code. Note that for this assignment, *you do NOT need to come up with new test cases*.

2 PROBLEM STATEMENT

2.1 Iterative Framework [40 pts]

A well written iterative dataflow analysis framework significantly reduces the burden of implementing new dataflow passes, the developer only writes pass specific details such as the meet operator, transfer function, analysis direction etc. In particular, *the framework should solve any unidirectional dataflow analysis as long the analysis supplies the following*:

(1) Domain
(2) Direction (Forward/Backward)
(3) Transfer Function
(4) Meet Operation
(5) Initial Condition
(6) Boundary Condition

To simplify the design process, the domain of values have been represented as bitvectors so that it is easy to carry out set operations (union, intersection).

2.2 Dataflow Analysis [40 pts]

2.2.1 Liveness [20 pts]. Upon convergence, your *Liveness* pass should report all variables that are *live* at each program point. For this assignment, we will only track the liveness of *instruction-defined values* and *function arguments*. That is, when determining which values are used by an instruction, you will use code like this:

```
Instruction * inst = ...
```

```
for (auto iter  = inst->op_begin();
          iter != inst->op_end(); ++iter)
{
    Value * val = *iter;

    if (isa < Instruction > (val) ||
        isa < Argument > (val))
    {
        ...
    }
}
```

The fact that there are ϕ instructions has ramifications on how your passes are implemented. Think carefully about what this means to your implementation and *briefly explain this in your assignment report*.

2.2.2 Available Expressions [20 pts]. Upon convergence, your Available Expressions pass should report all the binary expressions that are *available* at each program point. For this assignment, *we are only concerned with expressions represented by an instance of* `BinaryOperator`. Analyzing comparison instructions and unary instructions such as negation is not required.

We will consider two expressions *equal* if the instructions that calculate these expressions share the same opcode, left-hand-side and right-hand-side operand. In addition to this, the expression $x \oplus y$ is equal to expression $y \oplus x$ under the condition that the operator $\oplus$ is *commutative*.

3 THEORETICAL QUESTIONS

3.1 Loop Invariant Code Motion [10 pts]

Suppose that you are optimizing the code in Fig. 1.

(1) List the *loop invariant instructions*.

(2) Indicate if each loop invariant instruction can be moved to the loop preheader, and give a brief justification.

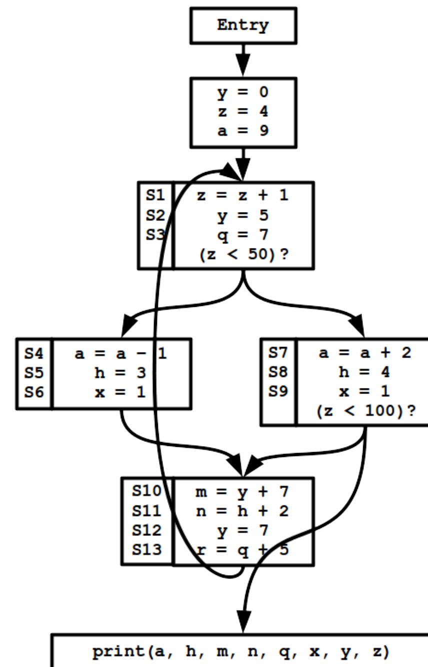

Figure 1: CFG for Analysis

3.2 Lazy Code Motion [10 pts]

Suppose that you are optimizing the code in Listing 1.

```
void foo(a, b, c)
{
    if (a > 5)
    {
        g = b + c;
    }
    else
    {
        while (b < 5)
        {
            b = b + 1;
            d = b + c;
        }
    }
    e = b + c;

    return e;
}
```

Listing 1: Source Code for Analysis

(1) Summarize, in one sentence, what is *Lazy Code Motion*?

(2) Build the CFG for this code, indicating which instructions from the original C code will be in each basic block. Using the algorithm described in class, provide *anticipated expressions* for each basic block.

(3) Provide *will-be-available expressions* for each basic block, and indicate the *earliest placement* for each expression, if applicable.

(4) Provide *postponable expressions* and *used expressions* for each basic block, and indicate the *latest placement* for each expression, if applicable.

(5) Complete the final pass of lazy code motion by inserting and replacing expressions. Provide the finalized CFG, and label each basic block with its instruction(s). Answer why it should have better performance compared with the raw CFG.

4 FAQ

Given below is the questions asked during previous offering of the class. If you do not think they fully answer your question, please open a new thread on Piazza.

Q: In class, we always dealt with the DFA by basic blocks (i.e., the transfer function operates on an entire basic block). But it seems from the starter code that we are doing transfer function on each instruction. How does this work? A: Remember that we mentioned from class that the transfer function of the entire basic blocks is just the composite of the transfer function of all the instructions within that basic block, i.e.,

$$f_{bb} = f_{i_n} \circ f_{i_{n-1}} \circ \ldots f_{i_1}{}^{1}, i_{1\ldots,n-1,n} \in bb$$

You can prove that the above equation is mathematically correct, but hopefully it makes sense by intuition.

Q: How does the domain work? What is the relationship between domain and the instruction-bitvector mapping? Specifically, suppose that we have N expressions in our program text, then for available expressions we just have a bitvector of length N? If that is the case, then how do we know which bit in the bit vector is associated with a particular expression? A: Your understanding is right. Suppose that you have N domain elements, then your bitvector length should be N as well. The reason is because there shall be a one-to-one correspondence between the domain elements and the bitvector indices.

As an example, let us suppose that we have three expressions in our program text, namely $\{a + b, a - b, a \times b\}$. When we say that the output of instruction i is $\{001\}$, that '1' can be $a + b$, $a - b$, or $a \times b$. What really matters is that *the mapping between bitvector indices and domain elements must be consistent throughout the entire analysis*, and that is the reason why we have `domain`, which plays the role of the metadata that describes the bitvectors.

[1]The order of composition depends on the direction of DFA.