COMP 621 Program Analysis and Transformations
Assignment #1

Profiling and Traditional Flow Analysis

Due: Wednesday, January 29 – beginning of class

Overview:

The purpose of this assignment is two-fold. The first purpose is to participate in the collection or creation of benchmarks that will be used later in the course for testing our optimization, parallelization, and analysis methods. The second purpose is to give you some experience with stating and solving dataflow problems.

Question 1: *Benchmark Programs*

Benchmark programs form an important part of the development of new optimization, analysis, and parallelization techniques. Without some sort of quantitative measurement, we cannot really evaluate how well the new techniques work. In order to build up a comprehensive benchmark suite, each person will provide one program that can be incorporated into the suite.

This year, you will be using MATLAB and AspectMatlab. You learned about MATLAB in Assignment #0, and you will be given an introduction to AspectMatlab in class. If you get stuck using the tools, please consult with the TAs and share your expertise with other class mates.

When choosing your benchmark, consider the following:

- The benchmark should compute something interesting. Real applications from different branches of Science and/or Engineering would be excellent. Applications solving toy CS problems are **not** interesting.
- You can write your own benchmark or you can use code that someone else has written, but we need to have permission for all of us to use it as a benchmark. It is even better if we are allowed to distribute it as part of our Matlab benchmark set. Something with an open source license is ideal.
- The benchmark should not require proprietary libraries. We need to be able to access and optimize all of the code.
- The benchmark should spend most of its time computing (and not doing I/O).
- The benchmark should not use dynamic features that make the program hard to analyze. For example, it should not make use of functions like `evalin`.

- The benchmark should have deterministic output that can be checked for correctness. We need to be able to run optimized versions of the program and automatically check that the output matches the output of the original unoptimized version. The benchmark should not have a GUI interface that requires human interaction to run, should not produce graphical output, nor should it require any terminal input (we need to be able to run the benchmarks from a script). If you need external files for input (i.e. if you compute on some large data set), please use ASCII files which contain one 2d array each, together with the MATLAB command `load`. Do not use .mat files, because it may make programs hard to analyze.
- Your benchmark needs to have different inputs that allow it to run for different amounts of time. Provide several inputs to correspond to a micro(around 1 second) short (around 10 seconds), medium (around 30 seconds) and long (around 5 minutes or longer) run-time. Create a driver file which takes scale as function parameter and executes the benchmark based on following scale parameter.
  - 1: micro run-time
  - 10: small run-time
  - 50: medium run-time
  - 100: larger run-time.

Where to look for a benchmark

There are many places to look. Here are a few possibilities:

- a research group at McGill using Matlab.
- a program you have written in another language, translated to Matlab (this is interesting, because you can compare the performance of the original implementation to your Matlab implementation).
- `http://www.mathworks.com/matlabcentral/fileexchange/`
- `http://web.mst.edu/~gosavia/mrrl_website.html`
- `https://github.com/trending?l=matlab`

Steps you must do

(a) Give a brief description of your benchmark which highlights the important or interesting characteristics of the program. Describe it both from the scientist/engineering point of view (why this is an interesting/important problem) and from the compiler optimizer point of view (what features make it interesting to optimize).

(b) Describe how to run the benchmark, and describe the required formats for inputs and which different inputs you have provided. Some examples of already packaged benchmarks will be provided for examples.

(c) Run your benchmark using the following two scenarios:

- Using Mathworks production version of MATLAB
- Using Octave. If your benchmark cannot run on Octave, then instead use Mathworks MATLAB with the JIT turned off.

For each scenario run the benchmark 10 times, then report the min, max, average and standard deviation of the 10 runs. You will probably want to write a script to do this task.

Report the versions of MATLAB and Octave that you used.

Report the architecture and OS on which you did the experiments.

(d) Analyze and discuss the results you obtained in part (c).

(e) Define an aspect using AspectMatlab which can be used to count or profile some feature of your benchmark. This could be a very simple aspect which counts the total number of array accesses and/or the total number of function calls. However, it could also be something more interesting. Give the source code for your aspect and explain its purpose.

(f) Compile your original benchmark, along with your aspect using the AspectMatlab compiler. Report on any problems which could be improved by better static checks or better error messages.

(g) Run the compiled/woven Matlab code using Mathworks MATLAB and report on runtimes as in (c).

(h) Discuss the results of (g). Did introducing your aspect slow down the program substantially? If so, why? Did your aspect provide you with interesting profiling results. If so, what?

(i) Starting with the original benchmark code, profile the execution using the Matlab profile function as documented at `http://www.mathworks.com/help/techdoc/ref/profile.html`, or the IDE profiler as documented `http://blogs.mathworks.com/community/2010/02/01/speeding-up-your-program-through-profiling/`.

Summarize your profile results. Based on this profile discuss which are the most important parts of the program to optimize?

(j) Based on your observations from (i), find some part of your program that you can optimize by hand (i.e. rewrite the Matlab code for that part to be faster). Perform the rewriting and explain why you think it will improve performance. Using your hand-optimized code, rerun your timings for Mathworks MATLAB and Octave and report on the results and the speedup/slowdown as compared to the unoptimized version. Discuss your results.

What to hand in

- A file called `yourLastName.tar.gz` file containing a directory called benchmarkName/. Of course, replace yourLastName with your real last name!

  Inside this directory you should have a sub-directory called src/ which contains your source code (the original version), all the inputs, all the outputs, a README that describes how to compile and run your benchmark. Also include a sub-directory called aspect/ that contains the source code for your aspect, also with a README describing your aspect. Finally, include a sub-directory called improved/ that contains the source code of your hand-optimized benchmark, along with a README summarizing the differences between this version and the orig/ version. Also include an info.json file which contains following keys:

```
{
"name": "benchmark name",
"version": "1.0",
"sources": "benchmarkName/src",
"runPath": "benchmarkName/src/driverFileName.m",
"tags": ["Winter2013_COMP621",array_growing]
}
```

  Please change name, sources, runPath and tags with appropriate values. Please include the tag "Winter2013_COMP621", and then other tags can have following values which describe the characteristics of the benchmark. For example, if the benchmark has a recursive function, tags will contain a recursion entry.

```
feval
eval
array_growing
structure
function_handle
lambda
end
nested_function
subfunction
global_variable
persistent_variable
```

```
recursion
Complex
```

Thus, your file should be something like `hendren.tar.gz` and when unzipped it should create a directory structure like:

```
/fibonacci
    /src
       README
       <otherfiles>
    /aspect
       README
       <otherfiles>
    /improved
       README
       <otherfiles>
    /info.json
```

Send a link to this file, or as an attachment, to `hendren@cs.mcgill.ca`. Indicate on the title of the message `cs621 Matlab Benchmark`. Please make sure to send *just one* copy but if you must update what you've already handed in, indicate this in the subject line with the word **UPDATE**.

- A hardcopy of your answers to all parts of this question. Please try to print in a paper-saving fashion. (You do not need to print your benchmark code)

Question 2: *Putting analysis to practice*

Assume the following simplified C-like language which also has some Matlab-like features.

```
<stmt_seq>   ::= <empty_stmt> | <stmt> <stmt_seq>

<stmt>       ::= <basic_stmt> ;  |
                 if ( <cond_expr> ) <stmt> else <stmt>  |
                 while ( <cond_expr> ) do <stmt>  |
                 do <stmt> while ( <cond_expr> ) ; |
                 { <stmt_seq> }

<basic_stmt> ::=  <id> = <id_const> <bin_op> <id_const> |
                  <id> = <id_const>         |
                  <id> = read_int()         |
                  <id> = read_double()      |
                  <id> = read_array( <id_const>, <id_const> ) |
                  <id> = <id_const>  |
                  int( <int_const> ) |
                  sum( <id> ) |
                  transpose ( <id> ) |
                  write( <id> )      |
                  break

<cond_expr>       ::=  <id_const> <relop> <id_const>

<id_const>   ::=  <id> | <int_const> | <double_const> | <array_const>

<bin_op>     ::=  + | * | - | /  | mod

<rel_op>     ::=  < | > | <= | >= | == | !=
```

This language has three types of variables, scalar integers, scalar doubles and two-dimensional arrays of doubles.

Scalar integers are created via a call to an integer creator, (i.e. `int(3)`), or via `read_int`.

Scalar doubles are created via a real constant (i.e. 3, 3.0 or 3e10), or via `read_double`. Note that `3` denotes the same double value as `3.0`.

Arrays of doubles are created via the array constant (i.e. `[3.0, 4.0; 5.0, 6.0]` would create a 2x2 array), or via `read_array`.

The `rel_op` operators must get two scalar arguments, and the result is always *true* or *false*, these values are used to determine control flow. The `bin_op` operators can have arguments of any of three data types.

The typing rules are:

```
int bin_op int -> int
int bin_op double -> int
double bin_op int -> int
double bin_op double -> double
array bin_op array -> array
sum(array) -> double
transpose(array) -> array
int rel_op int -> bool
double rel_op double -> bool
```

Any other combination of argument types gives a runtime error.

Consider the following example program.

```
 1:  n = read_int();
 2:  m = read_int();
 3:  x = read_array(m,n);
 4:  x_tr = transpose(x);
 5:  if (n > m)
 6:    { s = 0;
 7:      do {
 8:        t1 = x * x_tr;
 9:        t2 = sum(t1);
10:        t3 = s + t2;
11:        n = n - 1;
12:      } while ( n > 0);
13:     write(s);
14:    }
15:   else
16:    { p = 1;
17:      i = 0;
18:      while (i < n)
19:        { t1 = sum(x);
20:          x = x + t1;
21:          t2 = sum(x_tr);
22:          if (t2 < 0)
23:            break;
24:          p = p * t1;
25:          p = p * t2;
26:          i = i + 1;
27:        }
28:      write(p);
29:    }
30: write(n);
```

(a) Draw a typical control flow graph (CFG) that could be used to represent the entire program fragment, putting the statements in basic blocks where possible.

(b) Indicate the extended basic blocks for the CFG you created in part (a). Draw the Dominator Tree for the CFG from part (a).

(c) Assuming that we can represent a definition by a pair *(varname,lineno)*, (i.e. the definition at line 4 would be (x_tr,4)), what are the set of definitions that **may** reach the input of lines 5, 7, 8, 12, 16, 18, 24, 28 and 30 in the example program? As an example, the set of definitions reaching the input of line 2 is { *(n,1)* } .

(d) Based on reaching definitions, what optimizaton(s) could be done on the example program?

(e) What variables are live at the input of lines 30, 28, 26, 22, 18, 16, 13, 10, 7, 5 and 1 in the example program?

(f) We can define a variable **x** to be *really live* at program point p if **x** will be used (before being redefined) and all paths from p to the end. Give the definition of *really live* analysis for the example language, using the steps for defining an analysis you learned in class.

(g) Even though a variable may have double type, it may be possible to store the variable in an integer, and to replace double operations with integer ones. This can lead to performance improvements, both in space and in time.

For example, in the following code snippet, all of x, y and z could be stored as integers, even though the type system would indicate that y and z have double type.

```
x = 3;
y = read_int();
z = x + y;
```

Design and formalize an analysis to identify which program may be stored as integers. Give the rules for your analysis, and the types it would compute for the example program.

What to hand in

- A hardcopy of your answers to all parts of this question.