

COMP 621 Program Analysis and Transformations

Assignment #1

Profiling and Traditional Flow Analysis

Due: Wednesday, January 28 2009 – 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.

Throughout this assignment you might find it useful (and we highly encourage you) to use SABS, Maximes's benchmarking framework. You can find lots of useful information on his website: <http://www.cs.mcgill.ca/~mcheva/sabs.php>.

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 Java and the AspectJ language, or MATLAB, to write the benchmarks.

If you choose to do the Java/AspectJ benchmarking, follow the directions below. If you choose to benchmark MATLAB, then you should invent your process and part of the credit you will receive is for inventing the process itself. Perhaps a good idea would be to compare the performance of MATLAB, Octave and any relevant optimizations or compilers that could be applied in those systems.

If you are using AspectJ, one key issue is to determine how the choice of AspectJ compiler affects the performance of the generated bytecode.¹

There are also tools like `Soot` that optimize the bytecode, once the benchmarks have been compiled to classfiles. Can you determine if any measurable improvement can be made with such a tool?

¹You are also encouraged to use any other base language as long as it has a compiler that produces Java bytecode. You can then use AspectJ to weave into the .class files produced by the other compiler using the `-injars` argument to `abc` and `ajc`.

Another approach is to reduce the time taken by executing the bytecode by providing more efficient interpreters or by incorporating JIT (Just-in-time) compilation into the interpreters. With JIT systems the bytecode is converted into the host machine code, and then executed.

Finally, one can profile the runtime execution of a program to discover performance and memory bottlenecks. The profiling data can then be used in feedback-directed optimization to improve the program.

In order to complete this question you should investigate the relative impact of the choice of AspectJ compiler (bonus), the use of `Soot` as a bytecode optimizer, and the use of different interpreters/JIT compilers. The simplest experiment is to use Sun's Java virtual machine, and to use it in interpreter and JIT mode (client and server). A more interesting experiment would be to use a variety of JIT compilers. You also need to gather profiling data and apply optimizations suggested by it. Particularly interesting are the JRockit tools (google JRockit) and the IBM JVMs and tools (<http://www-128.ibm.com/developerworks/java>).

What to do

- (a) Develop or find the program. Give a brief description of the program which highlights the important or interesting characteristics of the program. Your program should have a significant amount of computations (i.e. run for a reasonable length of time (at least 60 seconds on a fast machine), and you should be able to test the results of running the program. Your program should take input from a file, or from stdin, and should produce some output to a file or to stdout. **The output should be deterministic**, and the output should be detailed enough so it is clear that the optimized program is working correctly.

Your benchmark should not spend a significant amount of time doing I/O, nor should it have a GUI interface that requires human interaction.

For a bonus: You may use some feature of the AspectJ language in your benchmark. A good tutorial on AspectJ and examples of its usage can be found in the AspectJ Programming Guide at <http://eclipse.org/aspectj/> and links to several free chapters of books can be found at <http://www.sable.mcgill.ca/~hendren/AspectJ.html>.

You can either write the entire benchmark in AspectJ, or you can start with a working Java benchmark and add some AspectJ aspects to it. You should try to find interesting uses of the `cflow` pointcut and/or `around` advice.

A number of examples can be found at <http://www.sable.mcgill.ca/benchmarks> (the *Law of Demeter* and *NullCheck* aspects have been applied to a simulator program in these benchmarks, but these aspects can be applied to any Java program). However, it is better if you can write your own aspect, even if it is quite small.

- (b) Describe why you picked or wrote this benchmark. What makes it interesting to optimize? Is it single-threaded or multi-threaded. Does it allocate a lot of objects? Is it very object-oriented. Does it use a lot of arrays? And so on...

If you are doing the bonus AspectJ part: How have you used aspects? What makes your use of aspects interesting?

- (c) Download a benchmark description form from <http://www.sable.mcgill.ca/~hendren/621/benchmark.txt> and fill it out.
- (d) Describe the input (if any) and the expected output of your program.
- (e) In this part of the question, you will determine the effect of compilation and runtime environment choices. This only applies if your benchmark uses AspectJ. Compile three versions of your program on any machine using:
 - `ajc`
 - `abc -O0`
 - `abc -O1` (this is the default for `abc`)

Report any errors in the `ajc` or `abc` compilers. If a compiler cannot compile your program, report it to the appropriate news group so that the bug can be fixed and report the issue in your assignment writeup.

- (f) Run each compiled version using three or more different JVM's, and with different settings for those VM's. Settings to explore include:
 - Client JIT vs. Server JIT vs. Interpreter modes
 - Varying heap sizes

Note that each JVM will have some flags for setting the mode and memory. For the Sun JVM the mode is selected by `-client` (which is the default), `-server` or `-Xint`. Settings for different heap sizes and gc settings are `-X` options - to see details use `java -X`.

Report the execution times taken for each compiled version with each VM configuration. The more VM's you can test the better – don't be afraid to download, compile, and install software. You should report exact settings, version numbers, and hardware used. Also, you should summarize your findings and order the various factors from highest to lowest importance.

- (g) Read the first half of the Soot tutorial (the section “Using Soot as a Command Line Tool”) found at <http://www.sable.mcgill.ca/soot/tutorial/>.

Starting with the `.class` files produced by step (e), optimize your program using `Soot`. Report on the success or failure of the `Soot` optimizer.

Specifically, try at least the following:

- (i) Compile your benchmark with `Soot`. Do not turn on any optimization flags. Compare the speed of the original version of your benchmark with the “sootified” version.

- (ii) Compile your benchmark with the intraprocedural Soot optimizations turned on (`-O`). Compare the speed of the version to the original benchmark and the unoptimized Soot version.
- (iii) Compile your benchmark with whatever optimization flags give you the best performance. You can start with the `-W` flag. However, you could try other options that you discover while reading the Soot documentation. Describe the Soot options that you used to get the best performance and compare this performance with the versions from parts (i) and (ii).

- (h) Java programs can be profiled. Use a profiler to profile and find the 10 most important methods. You can get a summary of the options of the standard Java profiler using `java -Xrunhprof :help`. Using the profiler, report on the top 10 methods and the proportion of total execution time they take. You can also investigate memory allocation and monitor contention in your program. Finally, you might want to look for other Java profiling tools, graphical or not – many of these will provide more detailed information than `hprof`, the built-in profiler.

- (i) Based on your profiling, try to improve the performance of your benchmark further by rewriting the Java/AspectJ code. (If you are really keen, then you can actually modify the bytecode by using Soot to produce Jasmin assembler code, modify the Jasmin assembler code by hand, and then use Jasmin to assemble it back into bytecode).

Report on how you tried to improve performance, why this worked (or didn't work), and how effective your improvements were.

- (j) Choose at least one `.class` file, try out the Dava decompiler that is a part of Soot (use the `-f dava` option to Soot). By examining the decompiled output you should be able to see the effect of the weaving done by the AspectJ compilers. Answer the following:
 - Is the decompiled program human-readable? Suggest ways in which the decompiler could be improved.
 - Does the decompiled program recompile correctly? If not, please give a short description of what does not recompile. If you can easily fix the decompiled code to recompile, do so.
 - If you recompile the decompiled code, does the recompiled code produce the correct results when executed?
 - **If you are doing the AspectJ bonus:** decompile a class with some aspect woven into it. Are there any noticeable differences in what `ajc` and `abc` did in the weaving process. If so, briefly describe the differences.

What to hand in

- A `tar.gz` file containing your source code (the original version, before you improved it), all the inputs, all the outputs, a README that describes how to compile and run

your benchmark, and the completed benchmark description form. Mail this file as an attachment to reehan.shaikh@cs.mcgill.ca. Indicate on the title of the message cs621 Java 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.

```

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

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

<basic_stmt> ::= <id> = <id_const> <bin_op> <id_const> |
                <id> = <id_const> |
                <id> = read() |
                write(<id_const>)

<expr> ::= <id_const> <relop> <id_const>

<id_const> ::= <id> | <const>

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

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

```

Use the following example program.

```

1 { n = read();
2   if (n > 0)
3     { i = n;
4       sum = 0;

```

```

5     while (i != 0)
6       { sum = sum + i;
7         i = i - 1;
8       }
9     write(sum);
10    }
11   else
12     { i = n;
13       prod = 1;
14       while (i != 0)
15         { prod = prod * i;
16           i = i + 1;
17         }
18       if (prod < 0)
19         prod = prod * -1;
20       write(prod);
21     }
22   write(i);
23   write(sum);
24   write(n);
25 }

```

- Draw a typical abstract syntax tree (AST) that could be used to represent the statement sequence from lines 12-21 of the example program.
- Draw a typical control flow graph (CFG) that could be used to represent the program fragment from lines 12-21 of the example program.
- Assuming that we can represent a definition by a pair $(varname, lineno)$, what are the set of definitions that **may** reach the input of lines 7, 15, 18 and 20 in the example program? As an example, the set of definitions reaching the input of line 2 is $\{ (n, 1) \}$.
- Give the data flow analysis for the **must** reaching definition problem. In this problem a definition reaches, only if it reaches along **all** paths. Follow all the steps outlined in class for defining the problem. Show the result of your must reaching definition analysis for lines 7,15, 18 and 20.
- What variables are live at the input of lines 2, 7, 20 and 24 in the example program?
- Some compilers warn the programmer when a use of a variable may not have a valid definition. In Java, the language definition says that each use must have a valid definition, and it issues a syntax error if there exists some path on which there is no definition of a use.

Formalize this problem for the language given in this problem. Give a flow analysis to solve the problem, and give example programs to show: (i) where your analysis correctly warns the programmer, and (ii) where your analysis warns the programmer unnecessarily (i.e. the analysis is conservative).

What to hand in

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