ECE454, Fall 2015
Homework1: Profiling and Compiler Optimizations
Assigned: Mon, Sept 14th, Due: Fri, Sept 25th, 11:59PM

The TA for this assignment is Peter Sun (peteryiping.sun@mail.utoronto.ca or peteryipingsun@gmail.com).

1 Introduction

Upon graduation from Skule, and having become a high-performance program-optimizing guru, you decided to open your own high-performance code optimization consulting firm, called "OptsRus". OptsRus' first client is a reconfigurable computing company that is using the open source CAD software program VPR (developed at UofT) to map customer designs to their reconfigurable FPGA-like hardware. However, since the demise of single-threaded CPU performance, their customers have been complaining that VPR is not fast enough for their needs, especially since their hardware chips keep getting bigger and hence so do the designs VPR has to handle.

The company cannot afford to parallelize the software at this point, so they instead want you to analyze the VPR code as well as their compilation process to see if they can do better. As detailed below, you are to experiment with both, collect measurements and profiles, and write a report for the company (by answering the specific questions below).

2 VPR

VPR is a FPGA placement and routing tool. In this assignment we will focus only on placement. VPR performs *simulated annealing based placement*, as described in the lecture slides on VPR. For more info on VPR see this site (but please only use the specific version of VPR that is distributed for 454 for this assignment, not the version from this site):

`http://www.eecg.toronto.edu/vpr`

3 Procedure

In your report, please answer all of the **Q*** questions below, with a few sentences each and any requested measurements.

HINT: you will likely find the use of a shell or perl/python script to automate your collection very helpful; this should be very easy to learn, I highly recommend that you take the opportunity to do so if you do not already.

3.1 Set up

Get a copy of and extract the homework tarball through the following commands:

```
mkdir ~/hw1
cd ~/hw1
cp /cad2/ece454f/hw1/vpr454.tar.gz .
tar -xvvf vpr454.tar.gz
```

Browse the VPR source code: starting at main.c (and knowing that we are focusing on placement), follow the chain of calls through the source code for a while to become familiar with it.

Q1 (1 marks) list 5 functions (by name, other than main) that you think might be important to optimize. Please do this honestly, before you do profiling. Explain in a few sentences total how you chose them.

3.2 Build and Test

In the Makefile modify the OPT_FLAGS field (look for "454") to compile the -g version (debug) of VPR (eg., change it to "OPT_FLAGS = -g"). Type make (which should compile successfully). Then run VPR with the following command line:

```
vpr iir1.map4.latren.net k4-n10.xml place.out route.out -nodisp -place_only -seed 0
```

This will result in a lot of output, the end of which should say "Completed placement consistency check successfully". Run VPR using these arguments for all experiments below.

3.3 Measuring Compilation Time

In this assignment you will use /usr/bin/time to measure compilation and performance. In the output, note that the number that ends in "user" is runtime in seconds for "user-mode", the time to use for this report, except Section 3.4. When measuring parallel compilation (or execution) time, you should take note of the "elapsed" time instead of the "user" time. The "elapsed" time represents the wall clock time. Note that since you are measuring real systems, measurements are a little different each time due to system variability. Try to measure on an unloaded machine. For every timing measurement always do 5 runs and average them (please only report the final average).

To build the gprof version of vpr, use the flags:

```
-g -pg
```

To build the gcov version of vpr, use the flags:

```
-g -fprofile-arcs -ftest-coverage
```

Measure the compilation time of the gprof, gcov, -g, -O2, -O3, and -Os compilation flags. Be sure to run "make clean" in between each build to ensure that all files are rebuilt.

Q2 (1 marks) Report the six measurements in a table; in the same table, using the slowest method of compilation as a baseline, report the speedup for each of the five measurements. Eg., if gcov was the slowest, and -g was twice as fast as gcov, then the speedup for -g relative to gcov would be 2.0.

Q3 (1 marks) Which is the slowest and why?

Q4 (1 marks) Which is the fastest and why?

Q5 (1 marks) Which of gprof and gcov is faster and why?

3.4 Measuring Parallel Compilation Time

Make can be configured to do a "parallel make", which means it will build independent objects using multiple processes when possible. Using the -j option you can specify the max number of processes to use. Measure the compilation time ("elapsed time") of '-j X' where X has the value of 1,2,4, and 8. In each case, the OPT_FLAGS variable should be set to '-O3' in the Makefile.

Q6 (1 marks) Report the measurements in a table; in the same table, using '-j 1' as a baseline, report the speedup for each of possibilities.

Q7 (1 marks) List some reasons why speedup is less than "perfect speedup" in this case. Why is the gain from 4 to 8 processes so small?

3.5 Measuring Program Size

Use "ls -l" to measure the size of each version of vpr from the previous section.

Q8 (1 marks) Report the six measurements in a table; in the same table, using the smallest method of compilation as a baseline, report the relative size increase for each of the six measurements. Eg., if -g was the smallest, and gprof was twice the size of -g, then the relative size increase for gprof relative to -g would be 2.0.

Q9 (1 marks) Which is the smallest and why?

Q10 (1 marks) Which is the largest and why?

Q11 (1 marks) Which of gprof and gcov is smaller and why?

3.6 Measuring Performance

Measure the run-time of VPR for all six versions compiled in the previous section.

Q12 (1 marks) Report the six measurements in a table. Again, using the slowest measurement as a baseline, also report the speedup for each version in the same table.

Q13 (1 marks) Which is the slowest and why?

Q14 (1 marks) Which is the fastest and why?

Q15 (1 marks) Which of gprof and gcov is faster and why?

3.7 Profiling with gprof

Compile gprof support for the -g, -O2, and -O3 versions, by using flags "-g -pg", "-O2 -pg" and "-O3 -pg" respectively; run each of these versions to collect the gprof result; you don't have to time any of this.

Q16 (1 marks) For each version, list the top 5 functions (give function name and percentage execution time).

Q17 (2 marks) For the "number-one" function for -O3 (the one with the greatest percentage execution time), how does its percentage execution time compare with the percentage execution time for the same function in the -g version? How is this possible? What transformation did the compiler do and to which functions?

Q18 (2 marks) For the transformation that the compiler did in the previous question, why didn't the compiler do

the same transformation to the number-two-ranked function from the -O3 version? *HINT: look for support for your argument in the VPR code, and explain what that is.*

3.8 Inspecting Assembly

Use objdump to list the assembly for the -g and -O3 versions of vpr (eg., run "objdump -d OBJ/main.o" to examine the assembly instructions for the file main.c).

Q19 (1 marks) Count the instructions for the function update_bb() in both versions (-g and -O3) and report those counts, as well as the reduction (reported as a ratio) in number of instructions for the -O3 version (ie., if the -O3 version has half as many instructions as the -g version, the reduction is 2.0x).

Q20 (1 marks) Using the gprof output from the previous section, compute and report the speedup in execution time of the -O3 version of update_bb() over the -g version (i.e., the "self seconds" from the gprof output). How does the speedup compare to the size reduction, and how can this be the case?

3.9 Profiling with gcov

Use gcov to get the per-line execution counts of the number-one function from the -O3 version (but use the -g version to gather the gcov profile). After running the gcov version of VPR, execute the gcov program to generate a profile of the appropriate file (eg., run "gcov -o OBJ -b main.c" to profile the file main.c). Running gcov will create main.c.gcov (for main.c). *NOTE: if you run the gcov program multiple times it will add to the counts in main.c.gcov; you have to remove the .gcda and .gcno files in OBJ/ to start counting from zero.*

Q21 (2 marks) Based only on the gcov results (ie., don't think too much about what the code says) list the loops in this function in the order that you would focus on optimizing them and why. Identify each loop by its line number in the original source file. If appropriate for any loop describe why you would not optimize it at all

3.10 Optional BONUS

Can you modify the source code of the number-one function (from the -O3 version), without changing the algorithm/output, and improve the average performance (when still compiling with -O3)? Note: do not eliminate code that checks for errors or prints—i.e., your code should still work the same for any input, even untested ones.

Q22 (3 marks) Clearly describe your modification (so the TA can repeat and test it), and why it improves performance. Measure it and report the speedup of your modification (relative to the original -O3 version of VPR).

4 Groups

You are asked to work in teams of two people for solving this assignment. Ensure that your report clearly lists the names and student numbers of all group members. You are allowed to work individually only by permission from the instructor.

5 Submission

Please submit your report to the company (your concise answers to the above questions), with the following command:

```
submitece454f 1 report.pdf
```

Suitable file formats include pdf and txt. If you wish to change your report, you may overwrite your submitted file by executing the above script again. To view your submission, enter

```
submitece454f -l 1
```