Benny (Xuan) Fang - 999159887
Md. Rafiur Rashid - 998544240

ECE 454 Lab 1 Assignment

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.

- ReadOption (reading large object from external xml sources could be potentially time consuming)
- SetupVPR (large initialization for various objects)
- ShowSetup (2 for loops for i and j which goes to O(n^2))
- set_graphic_state (writes to the display pixel by pixel, which involves O(n^2) for x and y coordinates)
- place_and_route (large setups for the objects)

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.

Compilat-ion Flag	Trial 1 (s)	Trial 2 (s)	Trial 3 (s)	Trial 4 (s)	Trial 5 (s)	Average (s)	Speedup Factor
-g	1.81	1.82	1.80	1.94	1.78	1.83	3.02
gprof	1.82	1.86	1.78	1.76	1.80	1.80	3.07
gcov	2.15	2.08	2.10	2.17	2.09	2.10	2.63
-O2	4.57	4.57	4.60	4.69	4.53	4.59	1.20
-O3	5.54	5.59	5.53	5.40	5.53	5.52	1.00
-Os	3.96	3.93	3.95	4.02	3.93	3.96	1.39

Q3. (1 marks) Which is the slowest and why?
-O3 is the slowest in terms of compilation time because this compilation flag provides the highest level of optimization of 3, thus performing the most number of optimization steps to achieve this.

Benny (Xuan) Fang - 999159887
Md. Rafiur Rashid - 998544240

Q4. (1 marks) Which is the fastest and why?
gprof is the fastest (although -g could be argued too due to too small of a difference) because it performs the minimum optimization at the compilation stage with just the debug -g flag. Instead, it just inserts codes in between function calls and calculate the runtime based on the 10ms interval. The optimization level of such flag is equivalent of -O0 with no optimization such as constant propagation and constant folding.

Q5 (1 marks) Which of gprof and gcov is faster and why?
gprof is faster than gcov. Although both flags inject codes during the compilation stage and does not utilize the optimization flag, gcov injects code line by line and calculate how many times each line is executed instead of between functions like gprof. As a result, gcov does more work at this stage and takes longer to compile.

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.

Parallel Make	Trial 1 (s)	Trial 2 (s)	Trial 3 (s)	Trial 4 (s)	Trial 5 (s)	Average (s)	Speedup Factor
-j 1	6.87	6.79	6.97	6.68	6.67	6.796	1.00
-j 2	4.79	5.04	4.61	5.03	4.49	4.792	1.42
-j 4	2.57	2.75	2.90	3.06	2.44	2.744	2.48
-j 8	1.87	1.84	1.80	1.89	1.91	1.862	3.65

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?

- There is existing delay time in between task performance splitted among parallel processes. As a result, the use of more processes produce more delays which further deviates from the expecting speedups.
- *Amdahl's Law:* If parts of the code is sequential, no matter how many processors are available, it limits parallel efficiency.

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.

Compilation Flag	Size (kB)	Size Factor
-g	747.2	2.65
gprof	751.2	2.67
gcov	1011.9	3.59
-O2	334.0	1.19
-O3	379.6	1.35
-Os	281.5	1.00

Q9 (1 marks) Which is the smallest and why?
-Os is the smallest in size because this optimization utilizes many aspect of -O2 optimization but puts the emphasis on size over speed. For example, it excludes alignment optimizations which skip space to align functions, loops, jumps and labels to an address in a multiple of a power of two but adds size to the resulting code. All other optimizations aim to minimize size is enabled, resulting in taking the least space.

Q10 (1 marks) Which is the largest and why?
gcov is the largest because as mentioned in Q5, it injects line in between every line of code to check for number of time the code is executed, hence inserting much more content at the compilation stage. Also, since optimization is not utilized, the code remains long and therefore takes more space.

Q11 (1 marks) Which of gprof and gcov is smaller and why?
gprof is smaller than gcov. As mentioned in Q5, although both flags inject codes during the compilation stage and does not utilize the optimization flag, gcov injects code line by line and calculate how many times each line is executed instead of between functions like gprof. As a result, gcov inserts much more content at the compilation stage and takes up a larger size.

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.

Compilat-ion Flag	Trial 1 (s)	Trial 2 (s)	Trial 3 (s)	Trial 4 (s)	Trial 5 (s)	Average (s)	Speedup Factor
-g	2.90	2.86	2.87	2.86	2.87	2.872	1.23
gprof	3.59	3.64	3.49	3.45	3.49	3.532	1.00
gcov	3.50	3.33	3.34	3.47	3.30	3.388	1.04
-O2	1.50	1.29	1.31	1.31	1.30	1.342	2.63
-O3	1.18	1.22	1.20	1.21	1.19	1.200	2.94
-Os	1.41	1.44	1.40	1.41	1.41	1.414	2.50

Q13 (1 marks) Which is the slowest and why?
gprof is the slowest compilation flag because its profiling takes a long time to run, add a big overhead runtime cost with checking every 10ms and generate gmon.out, which also consists of a large portion of time to the program's run time. In addition, the program is also not optimized by the compiler with the -O flags.

Q14 (1 marks) Which is the fastest and why?
-O3 is the fastest because it has the highest optimization level, so most of the work are already done at the compilation stage. During program runs, the executable file is already optimized in the best form possible compared to all other compilation flags.

Q15 (1 marks) Which of gprof and gcov is faster and why?
gcov is faster slightly, although only to a factor of 1.04 which too small to account for a difference. Both compilation are slow without any optimization. In addition, gprof, as mentioned in Q13, check performance every 10ms and generate gmon.out. On the other hand, gcov also injects a large portion of code via checking for number of execution of every line as well as generate gcda and gcno files after execution. As a result, both has a huge overhead cost and take a long time executing this program.

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

Compilation Flag	Function 1	Function 2	Function 3	Function 4	Function 5
-g -pg	comp_delta_td_cost (15.97%)	get_non_updateable_bb (14.24%)	comp_td_point_to_point_delay (14.24%)	find_affected_nets (11.11%)	try_swap (10.76%)
-O2 -pg	try_swap (38.64%)	get_non_updateable_bb (17.05%)	comp_td_point_to_point_delay (14.77%)	get_seg_start (8.33%)	get_net_cost (5.30%)
-O3 -pg	try_swap (73.99%)	comp_td_point_to_point_delay (11.38%)	label_wire_muxes (5.69%)	update_bb (3.25%)	get_bb_from_scratch (1.63%)

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?
The try_swap function consists of way more portion in -O3 compared to -g. The reason for this is because the try_swap function would consists of a large portion of runtime even in -O3 to due to it being called in multiple loops and itself utilizing multiple loops to be executed. On the other hand, other functions that takes a long time to execute in the -g flag are only called in one or less loop, thus is not called as often. After optimization, it no longer consist of big runtime to take a large percentage. As a result, although try_swap did not reduce its runtime drastically after optimization, the reduction in time of other major function causes its runtime percentage to rise significantly.

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.
The second function, comp_td_point_to_point_delay, also consists of several loops to be evaluated. However, the function is only called once within another function, therefore not consisting as much runtime as try_swap. The existence of loop inside the function, however, maintained its percentage after the -O3 flag.

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).

Compilation Flag	Number of Instructions	Reduction Factor
-g	550	1.00
-O3	213	2.58

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?

Compilation Flag	Execution Time (s)	Reduction Factor
-g -pg	0.07	1.00
-O3 -pg	0.04	1.75

The speed up in this case is less compared to the reduction of the number of instructions ran with different flag. This discrepancy is mainly due to each instruction not taking the same time to execute. For example, an instruction to access memory from an I/O command may take 10 times as much as a simple **mov** command in the assembly. Thus, this correlation factor is not exactly similar and the reduction of number of instructions do not directly reflect the same reduction in time.

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

```
/*
Function 'try_swap'
Lines executed: 91.89% of 111
Branches executed: 96.15% of 52
Taken at least once: 78.85% of 52
Calls executed: 71.43% of 21
*/
```

Loops:
Line 1279 for loop: would optimize because it executes 4349990 times with 80% going into the loop
Line 1312 while loop: would not optimize because it is never executed in the code, thus would be removed by the compiler optimization anyways

Line 1377 for loop: would optimize because it executes 17855734 times with 95% going into the loop
Line 1473 for loop: would optimize because it executes 4206708 times with 91% going into the loop
Line 1512 for loop: would optimize because it executes 13649026 times with 97% going into the loop

***Optional BONUS 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).**