ECE454, Fall 2014
Homework 1: Profiling and Compiler Optimizations

Timmy Rong Tian Tse (998182657) and Yuan Xue (998275851)

1) `SetupVPR`: this seems important to optimize because it performs many setup operations (i.e. `SetupOperation`, `SetupPlacerOpts`, `SetupAnnealSched`, `SetupRouterOpts`). Also, some of these setups perform memory allocations, which we know, are some of the most costly operations (takes the most time). Lastly, setup will always run during start-up. Per Amdahl's Law, optimizing the general case would be wise.

`place_and_route`: to start, we can probably infer that the command "Do the actual operation" above the function suggests that it is quite important. Furthermore, when we look into this function, it calls many other functions including a binary search function. Lastly, it performs memory allocations to "major routing structure", and because we know that memory operations are time consuming, optimizing them may be a good idea.

`CheckSetup`: we choose to optimize this function primarily because this function will get always get executed on start up and that it contains many `if` statements. We know that `if` statements are causes of branch instructions and we know that branch instructions are what causes hazards in pipelining. Optimizing this section of code (perhaps through a method like branch prediction) would be good.

`init_graphics`: Although this function is contained within an `if` statement (sometimes it won't get executed at all), the operations within this function seem costly. Graphics in general are expensive due to their heavy dependence on floating point operations. Also, graphics need a memory map, (in this case `private_cmap`, `cmap`), and of course, memory needs to be allocated for this.

`do_constant_net_delay_timing_analysis`: Again, although this function is contained within an `if` statement, we feel like we should optimize this because timing analysis seems like an integral part of FPGA analysis. Furthermore, when we look into the function, there are many memory allocations, which as mentioned before, seem like a judicious operation to optimize.

2)

Table 1: Compilation speedup relative to `-O3`

gprof	**gcov**	**-g**	**-O2**	**-O3**	**-Os**
2.87	2.49	2.88	1.26	1.00	1.49

3) -O3 is the slowest because this flag optimizes the most aggressively (more than -O2 and Os) and therefore, it spends more time during compilation to do that. gprof, gpcov and -g needs to insert code to profile but it does not restructure code like optimization does and therefore, their compilation times are much faster than any of the compilations that require optimizing.

4) -g is the fastest. The -g flag enables debugging symbols for the code. It is fast because it only needs to insert code and does not have to restructure code like optimizations. With respect to gprof and gcov, we can observe that they already have the -g flag in their command and thus, it only makes sense for -g alone to be the fastest.

5) gprof is faster because it only has to count and time the functions; it does not have to trace the execution like gcov. Furthermore, gprof profiles the code by interrupting the program every 10ms whereas in comparison, gcov needs profiling code to trace the program line by line and thus, generates more profiling code than gprof. In this sense, gcov is more involved and thus requires more compilation time.

6)

Table 2: Compilation speedup relative to `-j1`

-j1	**-j2**	**-j4**	**-j8**
1.00	1.79	3.69	3.57

7) There are a few reasons that this may be the case: (1) the number of physical cores on the ug machines are limited to four and so when we run `-j4` we expect one thread to run on each core thereby reducing the compilation time by a factor of four. However, this is not the case because per Amdahl's Law, there is portion of code that cannot be sped up with multi-cores because this portion of code must be executed serially. In the case of compilation, linking must be done serially since linking is dependent on the object files that are currently available. (2) When we go on and use `-j8`, we create many threads on a few cores which in turn, creates a lot of overhead that slows down the overall process.

8)

Table 3: Executable size increase relative to `-Os`

gprof	**gcov**	**-g**	**-O2**	**-O3**	**-Os**
2.89	3.83	2.88	1.18	1.33	1.00

9) `-Os` is the smallest because this compilation optimizes for size (smallest possible).

10) gcov is the largest. This is probably because the profiling that gcov performs generates more profiling code. gcov profiles programs by tracing through the execution and counting the number of times each line was executed. It makes sense that this extensive operation will bloat the executable size.

11) gprof is smaller. gprof counts and traces function calls using interrupts every 10ms. gcov on the other hand, needs to trace every line of code. Perhaps more profiling code is needed to carry out gcov's function thus leading to the smaller executable size for gprof.

12)

Table 4: Run-time speedup relative to gprof

gprof	**gcov**	**-g**	**-O2**	**-O3**	**-Os**
1.00	1.14	1.22	2.79	2.90	2.39

13) Slowest is gprof. This is probably because of the fact that gprof performs code profiling. Code profiling is an invasive operation; it needs to monitor each function and gather statistics. This process slows down the overall execution time of the program.

14) O3 is the fastest. This makes sense because O3 tells the compiler to optimize the program most aggressively, which in turn, makes the program run the fastest.

15) gprof runs slower than gcov. At first glance, this does not make sense because the profiling that gcov performs is just as, if not more extensive, than the profiling that gprof performs. However, gprof is slower probably because of the way it was implemented. gprof profiles the code by interrupting every 10ms. This frequent interruption causes a lot of overhead which in turn, slows down the overall execution time of the program.

16)

Table 5: Top 5 functions for `-g -pg`

`% time`	**`name`**
`18.49`	`comp_delta_td_cost`
`15.22`	`get_non_updateable_bb`
`13.42`	`comp_td_point_to_point_delay`
`9.98`	`find_affected_nets`
`9.00`	`try_swap`

Table 6: Top 5 functions for `-g -O2`

% time	**name**
40.81	try_swap
20.22	comp_td_point_to_point_delay
11.03	get_non_updateable_bb
7.17	get_net_cost
5.51	get_seg_start

Table 7: Top 5 functions for `-g -O3`

% time	**name**
62.69	try_swap
20.52	comp_td_point_to_point_delay
6.72	label_wire_muxes
3.73	update_bb
1.87	get_bb_from_scratch

17) The number-one function for `-O3` is `try_swap` with a percent execution of 62.69 whereas it is 9.00 for the same function in `-g`. This is probably because when the compiler optimizes each function, their runtime decreased a lot and/or a lot of functions "disappeared" leading to the disproportionately large representation for `try_swap`. This "disappearing" transformation that the compiler is doing is, of course, function inlining. For example, a function that gets inlined from `-g` to `-O2` is `comp_delta_td_cost` and a function that gets inlined from `-O2` to `-O3` is `get_non_updateable_b`

18) The function that did not get inlined (transformed by the compiler) and was number-two in execution time for `-O3` was `comp_td_point_to_point_delay`. This function did not get inlined because of the fact that this function is the single most called function in the program. This makes sense because if the compiler were to inline all functions including the ones that were called very often, then the resulting executable size would be very large.

19) The instruction count was 551 for `-g` whereas it was 221 for `-O3`. The size of reduction is 2.49 times.

20) The execution time for the function `update_bb` in the `-O3` version was three times faster than the `-g` version (0.05 vs. 0.15). From the previous question, we could also see that the instruction count was less in the `-O3` version than the `-g` version. The speedup makes sense because `-O3` is compiled to run at more optimal speed. The larger instruction count for `-g` version is due to the fact that this version tells the compiler to insert debug flags into the code and thus, many of the instructions in this version are actually used for debugging purposes.

21) There are five loops in this function. The order in which we would focus on optimizing would be, by their line numbers, 1377, 1473, 1512 and 1279. We pick 1377 first because this loop is executed the most and because there is a lot of code in this loop. Next, we picked 1473 over 1512 even though 1512 was executed much more often was because 1473 looked more long and complicated which would potentially allow for more optimizations. We picked 1279 last because it was not executed often and the loop was simple. Lastly, we would not optimize the `while` loop on line 1312 because it never gets executed.