ECE454 Lab 1

Connor Smith – 1000421411

Fan Guo – 1000626539

1.

Functions to Optimize:

- pathfinder_update_one_cost
- pathfinder_update_cost
- try_place
- try_route
- save_routing

These functions are all called often in the core routing algorithms. Therefore, improvements in these functions are most likely to produce meaningful speedups.

2.

Compilation Options	Average Compilation Time	Speedup relative to Slowest
gprof	1.81	3.19
gcov	2.08	2.78
-g	1.83	3.16
-O2	4.67	1.24
-O3	5.77	1.00
-Os	4.17	1.38

3. -O3 was the slowest because compilation with optimization takes more time, and its optimizations extend beyond what O2 and Os have.
4. Based on the average times we measured, gprof was the fastest on average. However, gprof and –g had similar compilation times, which implies that gprof adds very little compilation overhead, compared to compilation with only the –g option.
5. Gprof is faster because it only inserts code at the head and tail of each function to profile each function at execution. Gcov is then slower because the compiler generates additional information and a program flow graph for each function of the program, as well as including additional code in the object files for generating the extra information needed by gcov.

6.

Number of Processes	Average Compilation Time	Speedup relative to Slowest
1	7.08s	1.00
2	3.83s	1.85
4	2.15s	3.29
8	1.82s	3.89

7. The speedup is less than perfect, especially between 4 and 8 processes. This is because the overhead of coordinating and dividing the different source code file compilations starts to become more significant when compared to the actual work being done. Also, once the number of threads exceeds the total number of available cores on the machine, we would not see any speedup from further parallelizing the work (assuming compilation does not need significant I/O operations) as the physical machine would have to incur a context switching penalty while not actually accomplishing any faster compilation times. Even with infinite cores, after a certain number of threads compilation cannot be parallelized any further and we would see no speed-up from increasing the number of processes. This is because once there are enough processes such that every source code file has a dedicated thread, any additional processes would sit idle and not be able to contribute to compilation in any significant way.

8.

Compilation Options	Compiled Binary Size (bytes)	Size relative to Smallest
gprof	860,848	2.98
gcov	1,131,552	3.92
-g	836,056	2.89
-O2	341,896	1.18
-O3	393,888	1.36
-Os	288,976	1.00

9. -Os is smallest, as it includes all code optimizations in -O2 except for optimizations that would increase binary size, such as some function in-lining. It also includes specific optimizations to reduce binary size.

10. Gcov is largest, as it in-lines a great deal of profiling code used to count how often the lines of original code are executed. It also makes no attempt to reduce binary size versus -O2, -O3 and -Os.
11. Gprof adds code to the head and tail of the function, including a call to mcount as one of its first operations. Gcov will have additional code for every branch. It is more likely that a program will have more branches than it has methods, which would cause code compiled with the gcov options to be much larger.
12.

Compilation Options	Average Run-Time	Speedup Relative to Slowest
gprof	3.55	1.00
gcov	2.95	1.20
-g	2.84	1.25
-O2	1.32	2.68
-O3	1.23	2.87
-Os	1.44	2.46

13. Gprof was the slowest, followed by gcov, as the profiling code it injects adds overhead which increases runtime of the program. The executables that were compiled with optimization options would have seen an increase in speed that gprof did not benefit from.
14. -O3 was the fastest because it performed the greatest number of optimization transforms compared to the other optimization options. Other options, such as –g, gprof and gcov do not make optimizations, or add additional profiling instructions.
15. Gprof is slower as it uses a sampling process to gather information during runtime, which interrupts the program periodically and adds more overhead. Gprof's use of instrumentation in procedures increases the execution time of programs that make frequent function calls, and usually runs slower by a factor of 2 compared to normal.

16.

Compilation Options	Top 5 Functions	Percentage of Total Execution Time
-g -pg	comp_delta_td_cost	18.93
	comp_td_point_to_point_delay	17.08
	get_non_updateable_bb	16.87
	try_swap	9.47
	find_affected_nets	9.05
-O2 -pg	try_swap	49.48
	get_non_updateable_bb	14.47
	comp_td_point_to_point_delay	8.42
	get_net_cost	5.79
	get_seg_start	3.16
-O3 -pg	try_swap	66.67
	comp_td_point_to_point_delay	15.15
	label_wire_muxes	7.07
	update_bb	4.04
	get_bb_from_scratch	3.03

17. Try_swap is a much larger bottleneck (66.67% vs 18.93% of total execution time) in the -O3 compiled version when compared to the -g compiled version. This is because the three functions with greater execution time in the -g version (comp_delta_td_cost, comp_td_point_to_point_delay, and get_non_updateable_bb) benefitted from the optimizations found in -O2 and -O3 while try_swap did not, meaning its overall execution time did not decrease as much as the other functions, resulting in it taking up a much larger portion of the overall execution time percentage.

More specifically, comp_delta_td_cost benefitted from refactoring the memory accesses on the temp_point_to_point_delay_cost out of the loop using Loop Invariant Code Optimization (LICM). Get_non_updateable_bb also benefitted from LICM by no longer evaluating net[inet].num_sinks in every iteration of its main for-loop. Comp_td_point_to_point_delay benefitted from a similar optimization around the net[inet] function and also may have been inlined to further improve speed.

18. LICM optimizations did not apply to comp_td_point_to_point_delay because there was no loop to optimize. Therefore, its speed decreased mainly from the other more minor optimizations which applied to the majority of the program binary, resulting in no significant change to its overall runtime percentage.
19.

Compilation Options	Number of Instructions in update_bb()
-g	551
-O3	211

This translates to a reduction of 61.7% by compiling with -O3 compared to -g

20.

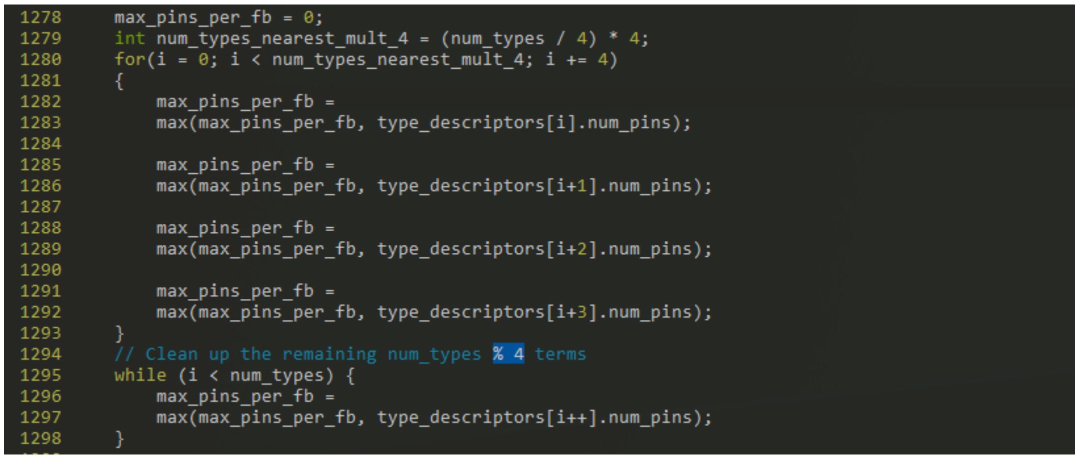
Compilation Options	Average gprof Self Seconds (s)
-g	0.09
-O3	0.03

This directly correlates with the reduction in the number of instructions measured by objdump in the section above. This is sensible, as the time a fixed clock CPU takes to execute a set of instructions increases with the size of that set. This is only true in the case where there aren't significant loops, as loops repeat the same instructions multiple times and largely eliminate this correlation between instruction count and execution time.

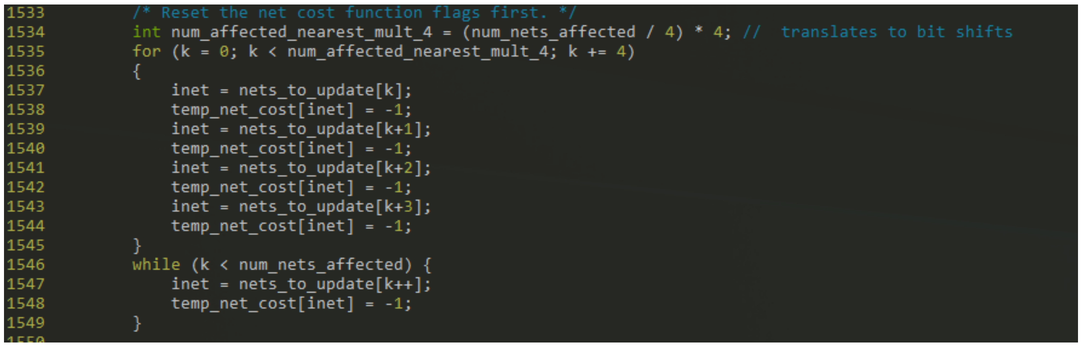
21.

Line of Code	Average Times executed per Function Call	Reasoning
1279	4	There are very few instructions executed within the body of the loop, meaning the condition check (i < num_types) and jump instruction create significant overhead
1312	0	Don't optimize. Never called with given command line arguments, and difficult to unroll or make other optimizations.
1377	19.55	Don't optimize. Nothing glaring in the code, and the loop body is complex enough to make it unlikely to benefit from loop unrolling.
1473	4.39	Don't optimize. Loop body is complex enough and not executed frequently enough to make the overhead of the loop insignificant.
1512	15.16	There are very few instructions executed within the body of the loop, meaning the condition check (k < num_nets_affected) and jump instruction create significant overhead

22. Based on the results of gcov detailed above, we would focus on the loops located on lines 1279 and 1512 in the try_place function. Through the loop unrolling outlined below, we were able to reduce the runtime of try_place from a 10-sample average of 0.71s (self-seconds) with no optimizations to a 10-sample average of 0.672s (self-seconds) with both optimizations. This reduced the overall 10-sample average runtime of the VPR command from 1.22s to 1.178s (3.44%) using the -O3 compilation flag.



```
1278     max_pins_per_fb = 0;
1279     int num_types_nearest_mult_4 = (num_types / 4) * 4;
1280     for(i = 0; i < num_types_nearest_mult_4; i += 4)
1281     {
1282         max_pins_per_fb =
1283         max(max_pins_per_fb, type_descriptors[i].num_pins);
1284
1285         max_pins_per_fb =
1286         max(max_pins_per_fb, type_descriptors[i+1].num_pins);
1287
1288         max_pins_per_fb =
1289         max(max_pins_per_fb, type_descriptors[i+2].num_pins);
1290
1291         max_pins_per_fb =
1292         max(max_pins_per_fb, type_descriptors[i+3].num_pins);
1293     }
1294     // Clean up the remaining num_types % 4 terms
1295     while (i < num_types) {
1296         max_pins_per_fb =
1297         max(max_pins_per_fb, type_descriptors[i++].num_pins);
1298     }
```



```
1533         /* Reset the net cost function flags first. */
1534         int num_affected_nearest_mult_4 = (num_nets_affected / 4) * 4; //  translates to bit shifts
1535         for (k = 0; k < num_affected_nearest_mult_4; k += 4)
1536         {
1537             inet = nets_to_update[k];
1538             temp_net_cost[inet] = -1;
1539             inet = nets_to_update[k+1];
1540             temp_net_cost[inet] = -1;
1541             inet = nets_to_update[k+2];
1542             temp_net_cost[inet] = -1;
1543             inet = nets_to_update[k+3];
1544             temp_net_cost[inet] = -1;
1545         }
1546         while (k < num_nets_affected) {
1547             inet = nets_to_update[k++];
1548             temp_net_cost[inet] = -1;
1549         }
```