ECE 454 Lab 1 Report

Taylan Gocmen 1000379949
Gligor Djogo 1000884206

Q1) 5 functions to optimize are:

- place_and_route: actual functionality, has multiple loops
- ShowSetup: has a nested loop that goes through num_blocks and num_pins, can potentially take a long time
- InitArch: has a nested loop to allocate the grid
- find_type_col: has a nested loop going through num_types and num_loc
- alloc_and_load_grid: has multiple nested loops

Nested loops add complexity and are likely bottlenecks in the code. The cost of function calls and operations inside them is inflated by repetition, so they should be important to optimize.

Q2) Compile times with different OPT_FLAGS

Flags	time_1	time_2	time_3	time_4	time_5	time_avg	speedup
gprof	1.79	1.82	1.78	1.80	1.85	1.808	3.0387
gcov	2.16	2.13	2.23	2.10	2.16	2.156	2.5482
-g	1.79	1.80	1.80	1.77	1.81	1.794	3.0624
-O2	4.53	4.58	4.60	4.61	4.62	4.588	1.1975
-O3	5.44	5.50	5.50	5.51	5.52	5.494	baseline
-Os	3.98	4.01	4.06	4.02	3.96	4.006	1.3714

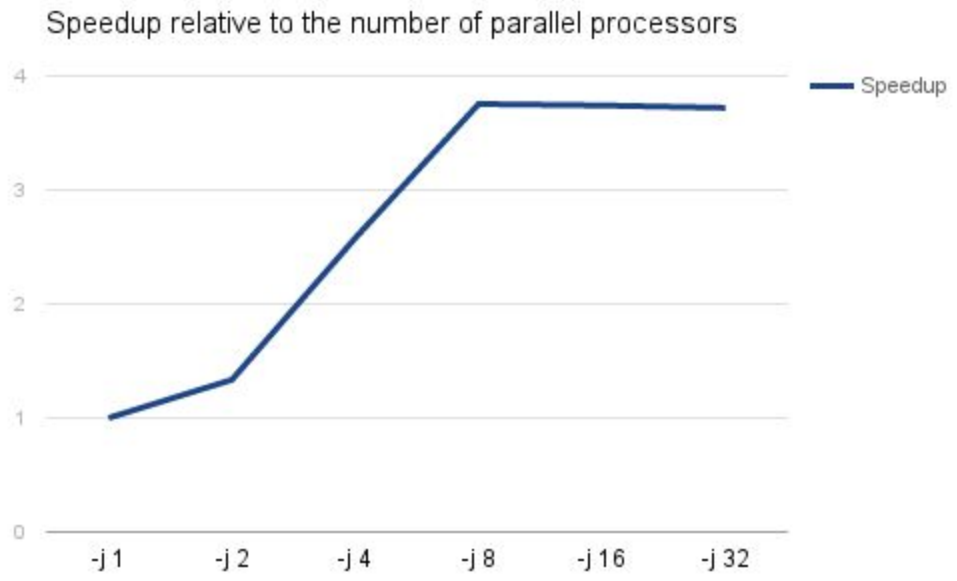
Q3) The compiler optimization -O3 was the slowest because it is doing the most aggressive optimizations such as loop unrolling in addition to quicker optimizations of -O2. To evaluate the space/speed trade-offs and perform additional optimizations takes more time for the compiler to compute.

Q4) The fastest time was for -g because the compiler only includes debug information and does no optimizations or any timing/profiling overhead. It performs none of the additional computation which the other flags specify, thus it can run faster.

Q5) gprof is faster than gcov, because gcov profiles every line and branch in the code as opposed to gprof which interrupts every 10ms. Gcov compilation needs to distinguish and keep track of all basic blocks in the code, which is more computation than adding sampling overhead.

Q6) Compile times with different number of parallel processors

Flags	time_1	time_2	time_3	time_4	time_5	time_avg	speedup
-j 1	6.66	6.69	6.63	6.73	6.74	6.690	baseline
-j 2	4.96	5.00	5.05	5.06	4.98	5.010	1.3353
-j 4	2.65	2.60	2.45	2.72	2.55	2.594	2.5790
-j 8	1.70	1.80	1.81	1.73	1.87	1.782	3.7542
-j 16	1.72	1.76	1.76	1.92	1.78	1.788	3.7416
-j 32	1.79	1.84	1.76	1.81	1.79	1.798	3.7208



As the number of parallel processes increases, the compilation time decreases because independent objects are built at the same time instead of in series. The most significant speedups were between 2 to 4 and 4 to 8 processes, where doubling the compiler's available computation power resulted in compilation time that was almost twice as fast. However, there is a limit to the maximum speedup obtained, since above 8 processes the overhead of managing the multiple processes slows down the compilation and there is no net time savings.

Q7) Program sizes with different OPT_FLAGS

Flags	size	Relative size increase
gprof	751 160	2.6682
gcov	1 011 872	3.5942
-g	747 168	2.6540
-O2	333 952	1.1862
-O3	379 648	1.3485
-Os	281 528	baseline

Q8) The smallest program is given by -Os, because this flag specifically optimizes the compilation to minimize the size. It also does not include the debugging or profiling flags, so the program does not need to contain extra references or overhead code. It is smaller than the speed optimization flags, because it will not do certain optimizations that could improve performance but sacrifice size.

Q9) The largest program is given by gcov (-g -fprofile-arcs -ftest-coverage). This is mostly due to the overhead required to profile the code. It can profile line by line inside functions, which is more demanding of overhead compared to the sampling of gprof. It also contains the -g debug flag, which requires certain references to be kept in the code and further inflate its size.

Q10) gprof is smaller than gcov because of the line by line profiling caused by the -ftest-coverage flag and the profiling of the arcs (branch paths in the program's control flow) caused by -fprofile-arcs flag. The overhead of these two is the difference between gcov and gprof, which only profiles in set time intervals. Sampling requires less code to interrupt the program and gather data, while line by line analysis requires more annotations around the basic blocks throughout the program, resulting in larger program size.

Q11) Performance times with different OPT_FLAGS

Flags	time_1	time_2	time_3	time_4	time_5	time_avg	speedup
gprof	3.43	3.40	3.52	3.50	3.49	3.468	baseline
gcov	3.10	2.95	2.97	2.98	3.04	3.008	1.1529
-g	2.88	2.84	2.82	2.81	2.85	2.840	1.2211
-O2	1.41	1.27	1.29	1.32	1.26	1.310	2.6473
-O3	1.34	1.20	1.20	1.19	1.22	1.230	2.8195
-Os	1.55	1.41	1.42	1.40	1.42	1.448	2.3950

Q12) gprof is the slowest because it provides no optimization to the code and only adds a time consuming overhead for profiling. During runtime, interrupts occur every 10ms and they are very costly because the code is stopped and restarted every time and significant time could be lost on system calls, context switches, or jumping to profiling code and losing instruction locality. The relatively short compilation time compared to the other flags, except -g, is a tradeoff to the longer running time.

Q13) O3 is the fastest because it provides the highest amount of optimization, which is aggressively geared towards performance improvement (like loop unrolling). It also excludes any excessive debugging or profiling functionality, which makes for a more compact and efficient code. As a result, it has the longest compilation time but provides the shortest running time.

Q14) gcov is faster than gprof. They both include -g debug info, so the difference is due to the type of profiling they provide. Gprof samples with 10ms interrupts and gcov tracks the usage of every line. Interrupts can be achieved with costly system calls and there would probably need to be a context switch or a branch to run code that updates the gprof statistics. On the other hand, gcov can annotate basic blocks of code and integrate its statistic collection with the source code. Thus, the time to stop and start the gprof code is more costly than gcov annotations and that makes it run slower.

Q15) Top 5 functions based on percentage of time used for different OPT_FLAGS

-g -pg	%
comp_td_point_to_point_delay	18.53
comp_delta_td_cost	15.04
get_non_updateable_bb	13.81
find_affected_nets	11.89
try_swap	10.14

-O2 -pg	%
try_swap	34.59
get_non_updateable_bb	18.05
comp_td_point_to_point_delay	14.29
get_seg_start	12.03
get_net_cost	9.02

-O3 -pg	%
try_swap	65.58
comp_td_point_to_point_delay	14.75
label_wire_muxes	7.38
update_bb	5.33
get_bb_from_scratch	2.87

Q16) The percentage execution time of try_swap rose from 10.14% (with -g -pg) to 65.58% (with -O3 -pg). Also, the total time spent in the function increased from 0.31s to 0.80s. The jump in percentage and time is due to the compiler performing inlining optimizations which appear to worsen the performance of that specific function, but improve overall runtime. Many functions that were computation intensive were inlined and absorbed into try_swap, and so were their profiling times. The compiler decided it was better to lose the additional function calls and sacrifice possible code locality in the instruction cache.
The inlined functions that did not appear in the -O3 profiling list (but were in -g) are find_to, find_affected_nets, get_non_updateable_bb, get_net_cost, comp_delta_td_cost, assess_swap, and update_td_cost. Some of these were top five in % time with -g, so it is not surprising that upon inlining try_swap jumped to first place.

Q17) The function comp_td_point_to_point_delay was not be inlined because it is not called directly from try_swap and it would be inefficient for the compiler to do so. The function is called inside a for loop (line 2168) from comp_delta_td_cost, which is itself inlined into try_swap. We believe the compiler decided not to further inline the sub-function of comp_delta_td_cost so that the loop would not get blown up and make branching more costly, as well as have a negative impact on the code locality.

Q18) For update_bb(), there are 551 instructions for -g and 214 instructions for -O3, the reduction is 2.575x.

Q19) For update_bb(), the profiling shows 0.06 secs for -g and 0.04 secs for -O3, the speedup is 1.5x. When optimized by -O3 flag the update_bb() function has increased in speed and reduced in size. The speedup is smaller than size reduction, they can't be expected to be the exact same amount but they correlate.

Q20) The number 1 function in -O3 is try_swap, the functions to optimize and the order they should be optimized is as below:

Loop line in place.c	Total number of evaluations of loop condition	% of 0 taken (condition evaluated to true)	Number of executions of the loop
For - 1377	17,855,734	95%	17,007,588
For - 1512	13,649,026	97%	13,187,845
For - 1473	4,206,708	91%	3,819,743
For - 1279	4,349,990	80%	3,479,992
While - 1312	869,998(based on the preceding if statement)	0.0%	Never executed

All the loops have similar high fallthrough rates, meaning that the times they loop versus the times they exit the loop is high. We then examine the number of total executions inside the loop to determine which one is the most important to optimize. That is why we ordered them by the number of executions.

Q21) The number 1 function in terms of self seconds was try_swap when “-O3 -pg” flags were used. The loop that was the most important in this function was the for loop starting at line 1377. To optimize this function we did the following changes:

Change lines 1278-1283 in place.c to:

```
max_pins_per_fb = max(0,
   max(type_descriptors[0].num_pins,
      max(type_descriptors[1].num_pins,
         max(type_descriptors[2].num_pins,
            type_descriptors[3].num_pins))));
```

Add to line 1381 in place.c:

```
int net_block_moved_k = net_block_moved[k];
```

Change line 1385 (was 1384 previously) in place.c to:

```
if(net_block_moved_k == FROM_AND_TO)
```

Change line 1394 (was 1393 previously) in place.c to:

```
if(net_block_moved_k == FROM)
```

Change lines 1512-1515 in place.c to:

```
int nna_unrolled = num_nets_affected - (num_nets_affected%5);
      for(k = 0; k < nna_unrolled; k=k+5)
            {
               temp_net_cost[nets_to_update[k]] = -1;
               temp_net_cost[nets_to_update[k+1]] = -1;
               temp_net_cost[nets_to_update[k+2]] = -1;
               temp_net_cost[nets_to_update[k+3]] = -1;
               temp_net_cost[nets_to_update[k+4]] = -1;
            }

   for(; k < num_nets_affected; k++)
      {
         temp_net_cost[nets_to_update[k]] = -1;
      }
```

The first and last modifications are examples of loop unrolling, the other modifications are part of the same change to the code and are an example of Loop Invariant Code Motion.

With these changes the gprof results for unmodified and modified codes were as following for the try_swap function and the overall execution:

Before modification	time_1	time_2	time_3	time_4	time_5	time_avg
try_swap self seconds	0.77	0.80	0.75	0.77	0.78	0.774
Total execution time	1.22	1.26	1.14	1.18	1.18	1.196
After modification						
try_swap self seconds	0.73	0.70	0.71	0.68	0.63	0.690
Total execution time	1.12	1.15	1.12	1.11	1.16	1.132

We can see that try_swap had a speedup of 1.1217 and total execution time had a speedup of 1.0565.